Information Flow Inference for ML

FRANÇOIS POTTIER and VINCENT SIMONET
INRIA

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 noninterference proof is reasonably light-weight, 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, noninterference is reduced to *subject reduction* for a nonstandard language extension. Lastly, a *semi-syntactic* approach to type soundness allows dealing with constraint-based polymorphism separately.

Categories and Subject Descriptors: F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages—*Operational semantics; Program analysis*; F.3.3 [**Logics and Meanings of Programs**]: Studies of Program Constructs—*Control primitives; Functional constructs; Type structure*; D.4.6 [**Operating systems**]: Security and Protection—*Information flow controls*

General Terms: Languages, Security, Theory

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* or *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 *noninterference* [Goguen and Meseguer 1982]: 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

Authors' address: INRIA, B.P. 105, 78153 Le Chesnay Cedex, France.

Permission to make digital/hard copy of all or part of this material without fee for personal or classroom use provided that the copies are not made or distributed for profit or commercial advantage, the ACM copyright/server notice, the title of the publication, and its date appear, and notice is given that copying is by permission of the ACM, Inc. To copy otherwise, to republish, to post on servers, or to redistribute to lists requires prior specific permission and/or a fee.

area concern extremely simplified programming languages. Several papers address pure λ-calculi [Heintze and Riecke 1998; Abadi et al. 1999; Pottier and Conchon 2000]. Volpano *et al.* [Volpano et al. 1996; Volpano and Smith 1997b] study a core imperative programming language, where all variables store integers. Volpano and Smith [Volpano and Smith 1997a] also study a language equipped with a fatal (non-catchable) exception, corresponding to failure of arithmetic operations. Banerjee and Naumann [Banerjee and Naumann 2002] deal with a fragment of Java, which includes classes and methods. Standing in sharp contrast, Myers [Myers 1999a; 1999b] 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, which are described in the conference version of this paper [Pottier and Simonet 2002a].

In an attempt to bridge the gap, we consider a call-by-value λ-calculus equipped with let-polymorphism, products and sums, references, exceptions, and generic primitive operations. (These last appear only in Section 7.) We refer to it as Core ML, because of its similarity with Wright and Felleisen's Core ML [Wright and Felleisen 1994]. In our version, however, exception names have global scope, and neither exception names nor exceptions are first-class values. Our calculus is very close to the core of the functional programming language Caml-Light [Leroy et al. 1997]. We endow it with a polymorphic, constraint-based type system, called MLIF, which has decidable type inference and guarantees noninterference.

A (monomorphic) treatment of references in a higher-order language can be found in [Zdancewic and Myers 2001; 2002]. Exceptions have been studied by Myers [Myers 1999a; 1999b] 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 noninterference, are novel. Our use of unannotated product types and our treatment of generic primitive operations (such as polymorphic equality), which require custom constraint forms, are also original contributions of this paper.

This paper is a revised and extended version of [Pottier and Simonet 2002a]. The main novelty with respect to the conference version resides in our decision to make exceptions second-class entities, rather than first-class values. This simplifies the type system, by allowing several notions to be suppressed; namely, exception types, alternatives, and conditional constraints. (More explanations are given in Section 5.4.) Eliminating conditional constraints, in particular, makes it more straightforward to design an efficient constraint solving procedure, and helps infer more readable types. We believe that the loss of expressiveness associated with this design decision remains minimal. Another improvement consists in a more detailed description of constraint solving, including a correctness proof.

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, noninterference [Goguen and Meseguer 1982] 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.* [Abadi et al. 1996] devised a *labeled* 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 [Pottier and Conchon 2000] later showed how *static*, type-based dependency analyses could be systematically derived, and proven safe, from such a labeled semantics.

Unfortunately, in a programming language with side effects, it is possible to leak information through the *absence* of a certain effect. Indeed, consider the program fragment "if $x = 1$ 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 labeled semantics to account for the effect of code that is *not* executed; so, the approach must be reconsidered.

Direct noninterference proofs, although straightforward for simple programming languages [Volpano et al. 1996], become increasingly complex in the presence of advanced features such as dynamic memory allocation, higher-order functions, and type polymorphism. A noninterference proof can be viewed as a *bisimulation* proof. For this reason, it requires manipulating a large, and often cumbersome, invariant: see e.g. [Zdancewic and Myers 2001]. 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 noninterference 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 bisimulation invariant 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 indispensable, because our types are verbose, and because information flow often occurs in unexpected ways. Because we employ subtyping (as well as other forms of constraints), our type inference system is constraint-based. Yet, if type 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 [Pottier 2001], 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 unintrusive. Then, we build a constraint-based system in the style of HM(X) [Odersky et al. 1999], which we prove correct with respect to the former.

We now proceed as follows. We first present the syntax of Core ML (Section 3). Then, we introduce our extension of Core ML, which we refer to as "Core ML^2", give

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

Fig. 1. The syntax of Core ML

an operational semantics for 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 noninterference property for Core ML (Section 6). In Section 7, we extend the language with generic primitive operations. Culminating our development, Section 8 presents MLIF, a constraint-based type system which we prove correct with respect to MLIF_0. We show that constraint solving is decidable, allowing type inference. Section 9 lists some example programs with their types. Lastly, we discuss a few design alternatives in Section 10.

3. CORE ML

Let k range over integers; let x, m range over disjoint denumerable sets of *program variables*, and *memory locations*, respectively; let j range over $\{1, 2\}$. Let ε range over a denumerable set $\mathcal{E}$ of *exception names*. Then, *values*, *answers*, *expressions* and *evaluation contexts* are defined as in figure 1.

Values include variables, a unit constant, integers, λ-abstractions, memory locations, pairs, and applications of an injection. An abstraction $\mathsf{fix}\, f.\lambda x.e$ may recursively refer to itself through the program variable f. (This is done merely to avoid introducing a separate fix-point combinator. We write $\lambda x.e$ when f does not appear free in e.) Answers represent completed computations; they are either values or unhandled exceptions of the form $\mathsf{raise}\ \varepsilon\ v$. An expression is an answer, a so-called *basic expression*, a let construct, or another expression enclosed within an evaluation context.

Basic expressions include function applications, instances of three primitive operations, which allow allocating, updating, and dereferencing memory cells, pair projections, and sum elimination (case) constructs. 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* [Flanagan et al. 1993], offers a number of advantages. First and foremost, it enables a much lighter formulation of our type-and-effect system. Indeed, because values have no computational effect, a basic expression's components now contribute nothing 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 are then implemented by different translation schemes. We will come back to this point in section 5.6.

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 [Wright and Felleisen 1994], the let keyword directs the type checker to give x polymorphic type. Following Wright [Wright 1995], 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 bind $x = e_1$ in e_2 evaluates e_1, binds x to its value, 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. We write $e_1;\ e_2$ for bind $x = e_1$ in e_2, where x does not occur in e_2.

The remaining evaluation contexts offer a variety of ways of handling exceptions. If the expression in the hole reduces to raise $\varepsilon\, v$, then [ ] handle $\varepsilon\, x \succ e$ binds x to v and evaluates e; otherwise, it has no effect. The context [ ] handle e done is analogous, but catches every exception, regardless of its name. It does not bind a variable, because exceptions are not values. The context [ ] handle e raise also catches every exception, and executes the handler e; then, however, it lets the answer raise $\varepsilon\, v$ escape, instead of completing normally. Lastly, the context [ ] finally e always executes e, regardless of the answer produced by the expression in the hole, before proceeding; it is similar to Lisp's unwind-protect and Java's try-finally constructs.

Why do we provide so many distinct ways of handling exceptions? The explanation lies in our decision to make exceptions second-class entities: an exception is not a value, so a variable cannot be bound to an exception, and raising an exception requires its name ε to be statically specified. This design choice will be motivated in section 5.4. To mitigate the loss in expressiveness, we must provide enough context forms to cover all common programming idioms.

We do not yet give an operational semantics for Core ML, because we view it as a fragment of Core ML^2, which we define in the next section.

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:

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

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 brackets can appear at an arbitrary depth within a term. For instance, if v is a Core ML value, then $\langle v_1 \mid v_2 \rangle\, v$ 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, because that would not make sense given our intended interpretation; so, the sub-terms of such a construct must be Core ML terms.

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 $\{1, 2\}$. These functions satisfy $\lfloor \langle e_1 \mid e_2 \rangle \rfloor_i = e_i$ and are homomorphisms on other expression forms.

Before giving more definitions, let us give a hint of how Core ML^2 allows keeping track of the differences between two Core ML programs throughout execution. For instance, let us consider the function $\lambda x.0$. Clearly, its result does not reveal any information about its argument, since it is a constant. Indeed, the type system which we will present in the following claims that this function maps "secret" inputs to "public" outputs. Now, in order to prove that the type system is correct, we must establish a noninterference result: for all integers k_1 and k_2, the programs $e_1 = (\lambda x.0)\, k_1$ and $e_2 = (\lambda x.0)\, k_2$ yield the same value. To do so, we encode these two programs into a single Core ML^2 term, namely $e = (\lambda x.0)\, \langle k_1 \mid k_2 \rangle$. Its two projections are the original Core ML programs: for $i \in \{1, 2\}$, $\lfloor e \rfloor_i$ is e_i. Note that the "secret" inputs k_1 and k_2 appear under brackets in e, while the structure common to e_1 and e_2, namely the application of $\lambda x.0$, is shared—that is, it appears outside the brackets. According to Core ML^2's operational semantics, which we will describe further on, the composite term $(\lambda x.0)\, \langle k_1 \mid k_2 \rangle$ reduces to the Core ML^2 term 0. The fact that this term does not contain any brackets is sufficient to ensure that its two projections coincide, that is, the original programs e_1 and e_2 both produce the same result. The noninterference proof developed in this paper (Theorem 15) is based on the same approach: we will prove that, under appropriate typing hypotheses, the result of a Core ML^2 reduction sequence does not contain any brackets.

The reduction sequence $(\lambda x.0)\, \langle k_1 \mid k_2 \rangle \to 0$, which we described above, is extremely simple. In general, however, reductions in Core ML^2 can be much more complex: several of its reduction rules must lift brackets when they block reduction. For instance, because the application $\langle \lambda x.x \mid \lambda x.0 \rangle\, 1$ is not a β-redex, it must be taken care of by a reduction rule other than (β). We introduce a new rule, (lift-app), which reduces it to $\langle (\lambda x.x)\, 1 \mid (\lambda x.0)\, 1 \rangle$. Note that this step affects neither projection, so it has no computational content: by moving brackets, it only keeps track of information flow. Each side of the new term is now a β-redex, allowing reduction to proceed: we obtain $\langle (\lambda x.x)\, 1 \mid (\lambda x.0)\, 1 \rangle \to^\star \langle 1 \mid 0 \rangle$.

4.2 Stores and configurations

The meaning of memory locations 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 $m \notin \mathrm{dom}(\mu)$. 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* $e \mathbin{/_i} \mu$ is a triple of an expression e, a store μ, and an index $i \in \{\bullet, 1, 2\}$, whose purpose is explained in Section 4.3. It is *stuck* if it is irreducible and e isn't an answer. It is *successful* if e is an answer. We write $e \mathbin{/} \mu$ for $e \mathbin{/_\bullet} \mu$. To guarantee that brackets cannot become nested during reduction and that void is

used exclusively in store bindings, as described above, we must introduce a couple of technical notions, whose definitions one may wish to skip upon first reading. A configuration $e \mid_i \mu$ 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 every $m \in \mathrm{dom}(\mu)$, $\mu(m)$ 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 $\langle v \mid \mathsf{void} \rangle$ (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;

—otherwise, m is in scope everywhere within e and μ.

A configuration $e \mid_i \mu$ is *closed* if all occurrences of memory locations in it are in scope. We restrict our attention to well-formed, closed configurations. (We let the interested reader check that this subset of configurations is stable under the reduction rules introduced in Section 4.3.) We identify configurations up to consistent renamings of memory locations.

The projection functions 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 $\lfloor e \mid \mu \rfloor_i = \lfloor e \rfloor_i \mid \lfloor \mu \rfloor_i$.

4.3 Semantics

The small-step operational semantics of Core ML^2 is given in Figure 2. 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 progress 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 $\langle e_1[x \Leftarrow \lfloor v \rfloor_1] \mid e_2[x \Leftarrow \lfloor v \rfloor_2] \rangle$.

Roughly speaking, the rules in the first two groups are 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: reductions 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

Basic reductions

$$
\begin{array}{rcll}
(\mathsf{fix}\, f.\lambda x.e)\, v \,/_i\, \mu & \to & e[x \Leftarrow v][f \Leftarrow \mathsf{fix}\, f.\lambda x.e] \,/_i\, \mu & (\beta) \\
\mathsf{ref}\, v \,/_i\, \mu & \to & m \,/_i\, \mu \oplus [m \mapsto \mathrm{new}_i\, v] & (\mathrm{ref}) \\
m := v \,/_i\, \mu & \to & () \,/_i\, \mu[m \mapsto \mathrm{update}_i\, \mu(m)\, v] & (\mathrm{assign}) \\
!\, m \,/_i\, \mu & \to & \mathrm{read}_i\, \mu(m) \,/_i\, \mu & (\mathrm{deref}) \\
\mathsf{proj}_j\, (v_1,\, v_2) \,/_i\, \mu & \to & v_j \,/_i\, \mu & (\mathrm{proj}) \\
(\mathsf{inj}_j\, v)\, \mathsf{case}\, x \succ e_1\, e_2 \,/_i\, \mu & \to & e_j[x \Leftarrow v] \,/_i\, \mu & (\mathrm{case}) \\
\mathsf{let}\, x = v\, \mathsf{in}\, e \,/_i\, \mu & \to & e[x \Leftarrow v] \,/_i\, \mu & (\mathrm{let})
\end{array}
$$

Sequencing

$$
\begin{array}{rcll}
\mathsf{bind}\, x = v\, \mathsf{in}\, e \,/_i\, \mu & \to & e[x \Leftarrow v] \,/_i\, \mu & (\mathrm{bind}) \\
\mathsf{raise}\, \varepsilon\, v\, \mathsf{handle}\, \varepsilon\, x \succ e \,/_i\, \mu & \to & e[x \Leftarrow v] \,/_i\, \mu & (\mathrm{handle}) \\
\mathsf{raise}\, \varepsilon\, v\, \mathsf{handle}\, e\, \mathsf{done} \,/_i\, \mu & \to & e \,/_i\, \mu & (\mathrm{handle\text{-}done}) \\
\mathsf{raise}\, \varepsilon\, v\, \mathsf{handle}\, e\, \mathsf{raise} \,/_i\, \mu & \to & e;\, \mathsf{raise}\, \varepsilon\, v \,/_i\, \mu & (\mathrm{handle\text{-}raise}) \\
a\, \mathsf{finally}\, e \,/_i\, \mu & \to & e;\, a \,/_i\, \mu & (\mathrm{finally}) \\
E[a] \,/_i\, \mu & \to & a \,/_i\, \mu & (\mathrm{pop}) \\
 & & \textit{if } E \textit{ handles neither } \lfloor a \rfloor_1 \textit{ nor } \lfloor a \rfloor_2 &
\end{array}
$$

Lifting

$$
\begin{array}{rcll}
\langle v_1 \mid v_2 \rangle\, v \,/\, \mu & \to & \langle v_1\, \lfloor v \rfloor_1 \mid v_2\, \lfloor v \rfloor_2 \rangle \,/\, \mu & (\mathrm{lift\text{-}app}) \\
\langle v_1 \mid v_2 \rangle := v \,/\, \mu & \to & \langle v_1 := \lfloor v \rfloor_1 \mid v_2 := \lfloor v \rfloor_2 \rangle \,/\, \mu & (\mathrm{lift\text{-}assign}) \\
!\, \langle v_1 \mid v_2 \rangle \,/\, \mu & \to & \langle\, !\, v_1 \mid !\, v_2 \rangle \,/\, \mu & (\mathrm{lift\text{-}deref}) \\
\mathsf{proj}_j\, \langle v_1 \mid v_2 \rangle \,/\, \mu & \to & \langle \mathsf{proj}_j\, v_1 \mid \mathsf{proj}_j\, v_2 \rangle \,/\, \mu & (\mathrm{lift\text{-}proj}) \\
\langle v_1 \mid v_2 \rangle\, \mathsf{case}\, x \succ e_1\, e_2 \,/\, \mu & \to & \langle v_1\, \mathsf{case}\, x \succ \lfloor e_1 \rfloor_1\, \lfloor e_2 \rfloor_1 \mid & \\
 & & \quad v_2\, \mathsf{case}\, x \succ \lfloor e_1 \rfloor_2\, \lfloor e_2 \rfloor_2 \rangle \,/\, \mu & (\mathrm{lift\text{-}case}) \\
E[\langle a_1 \mid a_2 \rangle] \,/\, \mu & \to & \langle \lfloor E \rfloor_1 [a_1] \mid \lfloor E \rfloor_2 [a_2] \rangle \,/\, \mu & (\mathrm{lift\text{-}context}) \\
 & & \textit{if none of the sequencing rules applies} &
\end{array}
$$

Reduction under a context

$$
\frac{e \,/_i\, \mu \to e' \,/_i\, \mu'}{E[e] \,/_i\, \mu \to E[e'] \,/_i\, \mu'} \qquad (\mathrm{context})
$$

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

Auxiliary functions

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

Fig. 2. Operational semantics of Core ML^2

$\langle \cdot \mid \cdot \rangle$ 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 answers (i.e. values and exceptions) are handled or propagated by evaluation contexts. We say that E *handles* a if and only if $E[a]$ can be reduced via a sequencing rule other than (pop).

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 appears in the semantics of Abadi *et al.*'s labeled λ-calculus [Abadi et al. 1996]. 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.

The $\langle \cdot \mid \cdot \rangle$ construct is reminiscent of the *fork* node introduced by Field and Teitelbaum to perform incremental reduction of λ-terms [Field and Teitelbaum 1990]. In fact, (lift-app) is one of their reduction rules. However, the details differ; in particular, we work with terms, whereas Field and Teitelbaum consider graphs, allowing a redex to be shared between two projections of a term.

Our design attempts to discard as little sharing information as possible; indeed, replacing all of the "lift" rules with the single rule $e \to \langle \lfloor e \rfloor_1 \mid \lfloor e \rfloor_2 \rangle$, while computationally correct, would cause the type system to view every expression as "secret". Yet, the reduction rules of Core ML^2 are not canonical: we have imagined a number of slight variations that work equally well. This is a common defect of purely syntactic proof techniques. This point should not be taken too seriously: Core ML^2 is a technical device, whose sole purpose is to prove a particular type system sound. One may wonder how general this syntactic approach is. We do not have a definite answer, although we have used it successfully in different settings [Pottier 2002; Simonet 2002].

4.4 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 a successful configuration, then so can the term itself.

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

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

LEMMA 2 (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 by Lemma 1. □

LEMMA 3. *If $e \mathbin{/} \mu$ is stuck, then $\lfloor e \mathbin{/} \mu \rfloor_i$ is stuck for some $i \in \{1, 2\}$.*

PROOF. By induction on the structure of e.

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

∘ *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 $\lfloor e \mathbin{/} \mu \rfloor_i$ is stuck.

∘ *Cases* $e = (v_1 := v_2)$, $e = !\, v$, $e = \mathsf{proj}_j\ v$, $e = v\ \mathsf{case}\ x \succ e_1\ e_2$ are similar to the previous case.

∘ *Case* $e = E[e_1]$. $e_1 \mathbin{/} \mu$ must be irreducible, otherwise, by (context), $e \mathbin{/} \mu$ would be reducible. Let us temporarily assume that e_1 is an answer a. Then, E does not

handle a, otherwise $E[a]$ would be reducible via one of the sequencing rules. If a were of the form $\langle a_1 \mid a_2 \rangle$, then clearly $E[a]$ would be reducible via either one of the sequencing rules or (lift-context). So, a must be of the form v or $\mathsf{raise}\ \varepsilon\ v$, which, considering that E does not handle a, implies that E handles neither $\lfloor a \rfloor_1$ nor $\lfloor a \rfloor_2$. As a result, (pop) must be applicable, a contradiction.

So, e_1 is not an answer, which implies that e_1 / μ 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 Core ML evaluation context F; in particular, so is $\lfloor E[e_1] \rfloor_i / \lfloor \mu \rfloor_i$, which is $\lfloor e \,/\, \mu \rfloor_i$.

∘ *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 answer, there exists $i \in \{1, 2\}$ such that e_i isn't an answer. As a result, $e_i /_i \mu$ is stuck. It follows that $e_i / \lfloor \mu \rfloor_i$ is stuck as well. □

LEMMA 4 (COMPLETENESS). *Assume* $\lfloor e / \mu \rfloor_i \to^\star a_i / \mu'_i$ *for all* $i \in \{1, 2\}$. *Then, there exists a configuration* a / μ' *such that* $e / \mu \to^\star a / \mu'$.

PROOF. To begin, let us establish that e / μ does not admit an infinite reduction sequence. We 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 $e \,/\, \mu$ admits an infinite reduction sequence, then Lemma 2 yields an infinite reduction sequence out of $\lfloor e \,/\, \mu \rfloor_i$, for some $i \in \{1, 2\}$. However, this is impossible, because both $\lfloor e / \mu \rfloor_1$ and $\lfloor e / \mu \rfloor_2$ can be reduced to normal forms, and the semantics of the Core ML fragment is deterministic.

So, e / μ reduces to an irreducible configuration. Let us temporarily assume that it is stuck. Then, by Lemma 3, at least one of its projections is stuck, which implies, by Lemma 2, that $\lfloor e \,/\, \mu \rfloor_i$ reduces to a stuck configuration, for some $i \in \{1, 2\}$—a contradiction. Thus, $e \,/\, \mu$ reduces to a successful configuration. □

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 nonterminating 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* noninterference result, whereby two programs can be guaranteed to yield the same result only if they both terminate. We do not aim at a *strong* noninterference result, because it would make little sense to plug information leaks related to termination without attacking timing leaks in general. Furthermore, such a result might 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 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 [Denning 1982], we typically use the meta-variable pc, rather than ℓ, when considering information obtained by observing the value of the "program counter".) We write $\bot$ and $\top$ for $\mathcal{L}$'s least and greatest elements, respectively. *Types* and *rows* are then defined as follows:

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

These are the types of ML's type system, decorated with extra security annotations. A *row* r is an infinite, quasi-constant family of security levels, indexed by $\mathcal{E}$. (A family is *quasi-constant* if all but a finite number of its entries are equal.) We write $(\varepsilon : pc; r)$ for the row whose element at index ε is pc and whose other elements are given by the sub-row r, which is indexed by $\mathcal{E} \setminus \{\varepsilon\}$. We write ∂pc for the constant row which maps every exception name to pc. We write $\sqcup r$ for $\sqcup_{\varepsilon \in \mathcal{E}}\, r(\varepsilon)$.

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

Function types carry several security annotations. The 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 of its behavior (i.e. in its effect); as a result, their security level will be made ℓ or greater. The annotation pc tells how much information is associated with the knowledge that this function gains control. 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. In other words, the annotation pc represents a lower bound on the level of the function's effects. The annotations ℓ and pc are standard, and can be found (under different names) e.g. in Heintze and Riecke's work [Heintze and Riecke 1998]. We correct a slight oversight on their part, however, by noticing that pc can be made contravariant, rather than invariant (see Section 5.2). In Section 10, we will suggest merging the annotations ℓ and pc; we keep them distinct in the bulk of the paper.

In addition, every function type carries an effect $[r]$. For every exception name ε, the security level $r(\varepsilon)$ indicates how much information is gained by observing that the function raises an exception named ε. Following Myers [Myers 1999a; 1999b], we 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; see Section 10 for more details. The reader may notice that rows do not record the type of exception arguments. Indeed, as in ML, we make exceptions monomorphic by assuming given a fixed mapping *typexn* from exception names to types. This decision makes function types much more compact.

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

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

Fig. 3. Subtyping

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

Fig. 4. Guards

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. Similarly, product types carry no security annotation, because, in the absence of a physical equality operator such as Caml-Light's ==, all of the information carried by a tuple is in fact carried by its components. Thus, we break the convention, established in a number of previous papers [Heintze and Riecke 1998; Pottier and Conchon 2000], that all types should be of the form $*^{\ell}$. This design decision, which we expect to help reduce verbosity, has implications on constraint solving, as explained in Section 5.2.

Sum types carry a security annotation ℓ, which reflects how much information the tag carries, i.e. how much information is obtained by determining whether the value was built using a left or right injection.

5.2 Subtyping and guards

We equip types and rows with a subtyping relation $\leq$, which extends the partial order $(\mathcal{L}, \leq)$. It is defined by the axioms in Figure 3. 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 last axiom extends subtyping to rows, point-wise and covariantly. The use of subtyping in information flow control is ubiquitous [Bell and LaPadula 1975; Denning 1982; Volpano and Smith 1997b; Heintze and Riecke 1998] 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.

Figure 4 defines the binary predicate $\lhd$, which relates a security level and a type. In short, the assertion $\ell \lhd t$ (read: ℓ *guards* t) requires t to have security level ℓ or greater, and is used to record a potential information flow. This is similar to Abadi *et al.*'s "t is protected at level ℓ" [Abadi et al. 1999]. In systems where every type constructor carries a security annotation [Heintze and Riecke 1998; Pottier and Conchon 2000], $\lhd$ would be syntactic sugar for $\leq$. Indeed, every instance of it would then be of the form $\ell \lhd *^{\ell'}$ and equivalent to $\ell \leq \ell'$. Here, the situation is more complex, because unit and product types carry no annotation. As a result, $\lhd$ constraints must receive a treatment of their own during constraint solving; see Section 8.5.

For any given ℓ and t, there exists a (minimal) supertype t' of t such that $\ell \lhd t'$ holds. Thus, the presence of $\ell \lhd t$ among a typing rule's premises usually cannot

irremediably prevent the application of that rule: the premise can be satisfied by first promoting t to t' using the subtyping rule. One exception is E-ASSIGN (see Figure 6), where t cannot be promoted to a supertype because it appears as an invariant argument to the ref type constructor.

The predicate $\lhd$ interacts nicely with subtyping:

LEMMA 5. *If* $\ell' \leq \ell$ *and* $\ell \lhd t$ *and* $t \leq t'$ *then* $\ell' \lhd t'$.

PROOF. $\ell \lhd t$ is equivalent to $\ell \leq \mathit{level}(t)$, for an appropriate function *level*, whose defining clauses include $\mathit{level}(\mathsf{unit}) = \top$ and $\mathit{level}(t_1 \times t_2) = \mathit{level}(t_1) \sqcap \mathit{level}(t_2)$. Furthermore, *level* is covariant in its argument. The result follows. □

5.3 Typing judgements

A *polytype* s is a nonempty set of types. By abuse of notation, a type t may be viewed as a polytype $\{t\}$. A *polytype environment* Γ is a partial mapping from program variables to polytypes. A *memory environment* M is a partial mapping from memory locations to types.

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 first judgement form is quite simple:

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

(We write $\Gamma, M \vdash v : s$ if and only if $\Gamma, M \vdash v : t$ holds for all $t \in s$.) On the other hand, expressions do produce side effects, so the second judgement form is more elaborate:

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

The assumption pc again tells how much information is associated with the knowledge that e is evaluated; it is a lower bound on the level of its effects. It is standard [Volpano and Smith 1997b; Heintze and Riecke 1998]. The row r tells how much information one obtains by observing exceptions escape out of e.

Two extra judgement forms are employed to reason about stores: $M \vdash \mu$ and configurations: $\Gamma \vdash e \,/_i\, \mu : t \ [r]$. These are analogous to those found in e.g. [Pottier 2001]. We omit Γ and M in a judgement when they are empty.

Even though the security lattice $(\mathcal{L}, \leq)$ is arbitrary, we wish to establish a temporary dichotomy between "low" and "high" security levels. (This distinction will be eliminated in Section 6.) In the present section, we assume H is a fixed, upward-closed subset of $\mathcal{L}$, and view levels inside (resp. outside) H as "high" (resp. "low"). Because noninterference is about two expressions that differ only in "high"-level sub-terms, our type system will require 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 Figures 5 and 6.) This will be our only use of H in this section.

5.4 Typing rules

We now comment on the typing rules, given in Figures 5 and 6. 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

$$\textsc{v-Unit}\ \ \Gamma, M \vdash () : \mathsf{unit} \qquad \textsc{v-Int}\ \ \Gamma, M \vdash k : \mathsf{int}^* \qquad \textsc{v-Void}\ \ \Gamma, M \vdash \mathsf{void} : * \qquad \textsc{v-Loc}\ \ \Gamma, M \vdash m : M(m)\ \mathsf{ref}^* \qquad \textsc{v-Var}\ \ \frac{t \in \Gamma(x)}{\Gamma, M \vdash x : t}$$

$$\textsc{v-Abs}\ \ \frac{pc, \Gamma[x \mapsto t'][f \mapsto (t' \xrightarrow{pc\ [r]} t)^\ell], M \vdash e : t\ [r]}{\Gamma, M \vdash \mathsf{fix}\, f.\lambda x.e : (t' \xrightarrow{pc\ [r]} t)^\ell} \qquad \textsc{v-Pair}\ \ \frac{\Gamma, M \vdash v_1 : t_1 \qquad \Gamma, M \vdash v_2 : t_2}{\Gamma, M \vdash (v_1,\ v_2) : t_1 \times t_2}$$

$$\textsc{v-Inj}\ \ \frac{\Gamma, M \vdash v : t}{\Gamma, M \vdash \mathsf{inj}_j\ v : (t +^j *)^*} \qquad \textsc{v-Bracket}\ \ \frac{\Gamma, M \vdash v_1 : t \qquad \Gamma, M \vdash v_2 : t \qquad pc' \in H \qquad pc' \lhd t}{\Gamma, M \vdash \langle v_1 \mid v_2 \rangle : t} \qquad \textsc{v-Sub}\ \ \frac{\Gamma, M \vdash v : t' \qquad t' \leq t}{\Gamma, M \vdash v : t}$$

Fig. 5. The type system MLIF$_0$ (values)

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 the function's side effects. V-PAIR is entirely standard. In V-INJ, $(t_1 +^j t_2)^\ell$ stands for $(t_j + t_i)^\ell$, where i and j are 1 and 2, not necessarily in that order, i.e. $\{i, j\} = \{1, 2\}$. 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-RAISE's premise checks that the exception's argument v has an appropriate type, as determined by the fixed mapping *typexn* and the exception name ε. Its conclusion ensures that the expression's effect is a row that maps ε to pc. In conjunction with E-BIND, E-HANDLE, E-HANDLEDONE and E-HANDLERAISE, this guarantees that any code fragment which observes this exception must run at level pc or greater.

E-APP governs function application. The security level pc, which is an assumption in the conclusion, appears on top of the $\rightarrow$ type constructor in the premise. It represents information that flows from caller to callee, as a result of the invocation itself. Furthermore, 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 be typechecked at level $pc \sqcup \ell$. Lastly, the function's result, too, may reveal information about its identity, so 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.

In E-PROJ, both pc and r are unconstrained, because pair projection has no side effect. In E-CASE, the branch e_j, by being executed, gains information about the sum's tag, whose security level is ℓ. As a result, it must be typechecked under the stricter security assumption $pc \sqcup \ell$, and its result type t must be guarded by ℓ. This rule is a straightforward generalization of the treatment of if constructs in previous

$$\textsc{e-Value}\ \frac{\Gamma, M \vdash v : t}{*, \Gamma, M \vdash v : t\ [*]} \qquad \textsc{e-Raise}\ \frac{\Gamma, M \vdash v : \mathit{typexn}(\varepsilon)}{pc, \Gamma, M \vdash \mathsf{raise}\ \varepsilon\, v : *\ [\varepsilon : pc; *]} \qquad \textsc{e-App}\ \frac{\Gamma, M \vdash v_1 : (t' \xrightarrow{pc \sqcup \ell\ [r]} t)^{\ell} \quad \Gamma, M \vdash v_2 : t' \quad \ell \lhd t}{pc, \Gamma, M \vdash v_1\, v_2 : t\ [r]}$$

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

$$\textsc{e-Deref}\ \frac{\Gamma, M \vdash v : t'\ \mathsf{ref}^{\ell} \quad t' \leq t \quad \ell \lhd t}{pc, \Gamma, M \vdash\ !\, v : t\ [*]} \qquad \textsc{e-Proj}\ \frac{\Gamma, M \vdash v : t_1 \times t_2}{*, \Gamma, M \vdash \mathsf{proj}_j\ v : t_j\ [*]}$$

$$\textsc{e-Case}\ \frac{\Gamma, M \vdash v : (t_1 + t_2)^{\ell} \quad \forall j \in \{1,2\} \quad pc \sqcup \ell, \Gamma[x \mapsto t_j], M \vdash e_j : t\ [r] \quad \ell \lhd t}{pc, \Gamma, M \vdash v\ \mathsf{case}\ x \succ e_1\ e_2 : t\ [r]}$$

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

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

$$\textsc{e-HandleDone}\ \frac{pc, \Gamma, M \vdash e_1 : t\ [r_1] \quad pc \sqcup (\sqcup r_1), \Gamma, M \vdash e_2 : t\ [r_2] \quad (\sqcup r_1) \lhd t}{pc, \Gamma, M \vdash e_1\ \mathsf{handle}\ e_2\ \mathsf{done} : t\ [r_2]} \qquad \textsc{e-HandleRaise}\ \frac{pc, \Gamma, M \vdash e_1 : t\ [r] \quad pc \sqcup (\sqcup r), \Gamma, M \vdash e_2 : *\ [\partial\bot]}{pc, \Gamma, M \vdash e_1\ \mathsf{handle}\ e_2\ \mathsf{raise} : t\ [r]}$$

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

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

$$\textsc{Store}\ \frac{\mathrm{dom}(M) = \mathrm{dom}(\mu) \quad \forall m \in \mathrm{dom}(\mu) \quad M \vdash \mu(m) : M(m)}{M \vdash \mu} \qquad \textsc{Conf}\ \frac{pc, \Gamma, M \vdash e : t\ [r] \quad M \vdash \mu}{\Gamma \vdash e\ /\ \mu : t\ [r]}$$

Fig. 6. The type system MLIF$_0$ (expressions and configurations)

information flow analyses for imperative languages [Denning 1982; Volpano and Smith 1997b].

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 bind $x = e_1$ 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 more details.

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_ε, 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 $pc_\varepsilon \lhd t$. E-HANDLEDONE is analogous; however, because this 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 [Myers 1999a; 1999b] performs the same approximation. Like E-HANDLEDONE, E-HANDLERAISE typechecks the handler e_2 at an increased level. E-FINALLY, on the other hand, typechecks e_1 and e_2 at the same level pc. Indeed, because e_2's invocation *must* occur, regardless of the answer produced by e_1, no information is associated with it.

Both E-HANDLERAISE and E-FINALLY require e_2 not to leak any information through exceptions. (This is done by requiring its effect to be the constant row $\partial\bot$.) This design choice may seem restrictive, but we believe it strikes a good balance between expressiveness and simplicity. In the conference version of this paper [Pottier and Simonet 2002a; 2002b], we presented more general versions of these rules, whereby e_2 was allowed to raise arbitrary exceptions. This, however, required adding a third premise, of the form $\sqcup r_2 \leq \sqcap r_1$, reflecting the fact that, if an exception raised by e_1 escapes, then e_2 must have completed successfully. This additional premise involved a form of conditional constraint, making constraint solving more intricate and yielding more complex inferred types, which is why we propose simpler versions of these rules here.

In E-HANDLERAISE, the effect of the whole expression, namely r, is exactly e_1's effect, because it is known that any informative exception that escapes out of e_1 handle e_2 raise was originally raised by e_1. (By "informative" exception, we mean one whose security level is strictly greater than $\bot$.) This is more precise than we could hope to achieve if this idiom was emulated in a language with first-class exceptions, by writing, say, e_1 handle $x \succ (e_2;$ raise $x)$. Indeed, in the type system given in the conference version of this paper [Pottier and Simonet 2002a], the sub-expression raise x would be typechecked at an increased security level $pc \sqcup (\sqcup r)$. As a result, every exception ε liable to escape out of e_1 would be re-raised at level $\sqcup r$, instead of its original level $r(\varepsilon)$, which would defeat the purpose of discriminating between exception names. This explains why, in this paper, we rely solely on special-purpose constructs, such as handle – raise, and abandon first-class exceptions. (Again, in the conference paper, E-RAISE involved a conditional constraint, which is no longer necessary here, because every raise form explicitly

specifies an exception name ε.)

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 fourth premise, however, is slightly more general than that of V-BRACKET. By definition, the auxiliary predicate $e\Uparrow$ holds if and only if the Core ML expression e is of the form raise $\varepsilon\, v$ or bind $x =$ raise $\varepsilon\, v$ in e' or raise $\varepsilon\, v$ handle e' raise or e'; raise $\varepsilon\, v$. This syntactic criterion, which is preserved by substitution and by reduction, ensures that e cannot reduce to a value, that is, e must diverge or reduce to a raise form. There is no way, in the syntax of typing judgements, to express the knowledge that the expression at hand cannot possibly return a value; yet, the ability to keep track of such knowledge is needed, in a small number of places, for subject reduction to hold. The use of the predicate $\cdot\Uparrow$ in E-BRACKET's last premise can be viewed as a cheap way of affording this expressiveness. In short, E-BRACKET's last premise requires t to have a "high" security level, unless it is known that one of the expressions at hand will never produce a value. This is in accordance with the fact that our noninterference result, to be given in Section 6, requires *both* expressions to produce values.

Rules E-SUB, STORE and CONF are standard.

5.5 Subject reduction

We now give a subject reduction proof for Core ML^2.

LEMMA 6 (WEAKENING). *$pc' \leq pc$ 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, and the induction hypothesis, it is easy to check that every premise remains valid when pc decreases. The result follows. □

LEMMA 7 (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 6 yields $pc, \Gamma, M \vdash e_i : t\ [r]$, as required. □

LEMMA 8 (GUARD). *If $\Gamma, M \vdash \langle v_1 \mid v_2 \rangle : t$ then there exists $pc' \in H$ s.t. $pc' \lhd t$.*

PROOF. Thanks to Lemma 5, 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 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 9 (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 2), by Lemma 7, by V-VOID and V-BRACKET. □

LEMMA 10 (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. Both statements are proved simultaneously, by induction.

∘ *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 7, 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 4.3), was our goal.

∘ *Case* E-BRACKET. Similar to the case of V-BRACKET. We use the fact that $\cdot\Uparrow$ is preserved by substitution, i.e. $e\Uparrow$ implies $e[x \Leftarrow v]\Uparrow$.

The other cases are immediate or analogous to one of those above. □

LEMMA 11 (VALUE). $pc, M \vdash v : t \ [r]$ *implies* $M \vdash v : t$.

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

∘ *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. □

LEMMA 12 (SUBJECT REDUCTION). *Let* $e \,/_i\, \mu \to e' \,/_i\, \mu'$. *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 $e \,/_i\, \mu \to e' \,/_i\, \mu'$. 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 θ stand for $(t' \xrightarrow{pc \sqcup \ell \ [r]} t)^\ell$. In E-APP's premises, 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 $\to$ is contravariant (resp. covariant) in its first and second (resp. third and fourth) parameters, applying Lemma 6 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 10 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 $\mu \oplus [m \mapsto \mathrm{new}_i\, v]$. E-REF's premises are $M \vdash v : t'$ and $pc \lhd t'$, provided $t = t'\ \mathsf{ref}^*$. By Lemma 9, these imply $M \vdash \mathrm{new}_i\, v : t'$. Define $M' = M[m \mapsto t']$. According to STORE, $M \vdash \mu$ implies $\mathrm{dom}(M) = \mathrm{dom}(\mu)$. Because $\mu \oplus [m \mapsto \mathrm{new}_i\, v]$ 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 $()$. E-ASSIGN's premises are $M \vdash m : t'\ \mathsf{ref}^*$ 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 9, we have $M \vdash \mathrm{update}_i\, \mu(m)\, v : t'$, which yields $M \vdash \mu'$.

∘ *Case* (deref). e is $!\, m$. E-DEREF's first two premises are $M \vdash m : t'\ \mathsf{ref}^*$ and $t' \leq t$. As above, the former entails $M \vdash \mu(m) : t'$. By Lemma 9, $M \vdash \mathrm{read}_i\, \mu(m) : t'$ follows. Conclude with V-SUB and E-VALUE.

∘ *Case* (proj). e is $\mathsf{proj}_j\ (v_1,\ v_2)$ and e' is v_j. E-PROJ's premise is $M \vdash (v_1,\ v_2) : t_1 \times t_2$, where t_j is t. According to V-PAIR and V-SUB, this implies $M \vdash v_j : t_j$.

∘ *Case* (case). e is $(\mathsf{inj}_j\ v)\ \mathsf{case}\ x \succ e_1\ e_2$ and e' is $e_j[x \Leftarrow v]$. E-CASE's first premise is $M \vdash \mathsf{inj}_j\ v : (t_1 + t_2)^\ell$. According to V-INJ and V-SUB, this implies $M \vdash v : t_j$. This allows applying Lemma 10 to E-CASE's second premise, yielding $pc \sqcup \ell, M \vdash e_j[x \Leftarrow v] : t\ [r]$. The result follows by Lemma 6.

∘ *Case* (let). By E-LET and Lemma 10.

∘ *Case* (bind). e is $\mathsf{bind}\ x = v\ \mathsf{in}\ e_2$ and e' is $e_2[x \Leftarrow v]$. E-BIND's premises are $pc, M \vdash v : t'\ [r_1]$ and $pc \sqcup (\sqcup r_1), (x \mapsto t'), M \vdash e_2 : t\ [r_2]$, where $r_2 \leq r$. By Lemma 11, the former implies $M \vdash v : t'$. By Lemma 6, the latter implies $pc, (x \mapsto t'), M \vdash e_2 : t\ [r_2]$. By Lemma 10 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 $e_2[x \Leftarrow v]$. E-HANDLE's first two premises are of the form $pc, M \vdash \mathsf{raise}\ \varepsilon\, v : t\ [*]$ and $pc \sqcup *, (x \mapsto \mathit{typexn}(\varepsilon)), M \vdash e_2 : t\ [r]$. According to E-SUB and E-RAISE, the former implies $M \vdash v : \mathit{typexn}(\varepsilon)$. By Lemmas 10 and 6, this yields $pc, M \vdash e_2[x \Leftarrow v] : t\ [r]$.

∘ *Case* (handle-done). e is $a\ \mathsf{handle}\ e_2\ \mathsf{done}$ and e' is e_2. E-HANDLEDONE's second premise is $pc \sqcup *, M \vdash e_2 : t\ [r]$. Lemma 6 yields $pc, M \vdash e_2 : t\ [r]$.

∘ *Cases* (handle-raise), (finally). e is of the form $a\ \mathsf{handle}\ e_2\ \mathsf{raise}$ or $a\ \mathsf{finally}\ e_2$, while e' is $(e_2;\ a)$. E-HANDLERAISE or E-FINALLY's first premise is $pc, M \vdash a : t\ [r]$. Its second premise, modulo an application of Lemma 6, is $pc, M \vdash e_2 : *\ [\partial\bot]$. Given the identities $pc \sqcup (\sqcup(\partial\bot)) = pc \sqcup \bot = pc$ and $r \sqcup (\partial\bot) = r$, E-BIND yields $pc, M \vdash (e_2;\ a) : t\ [r]$.

∘ *Case* (pop). e is $E[a]$ and e' is a. Several sub-cases arise.

Sub-case $E = \mathsf{bind}\ x = [\,]\ \mathsf{in}\ e_2$. E-BIND's first premise is $pc, M \vdash a : t'\ [r_1]$, where $r_1 \leq r$. Because E does not handle a, a must be of the form $\mathsf{raise}\ \varepsilon\, v$ or $\langle \mathsf{raise}\ \varepsilon_1\, v_1 \mid \mathsf{raise}\ \varepsilon_2\, v_2 \rangle$. So, this judgement must be a consequence of E-RAISE, E-BRACKET and E-SUB. A derivation of identical shape can be built to establish $pc, M \vdash a : t\ [r_1]$. (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 $E = [\,]\ \mathsf{handle}\ \varepsilon\, x \succ e_2$. E-HANDLE's first premise is $pc, M \vdash a : t\ [\varepsilon :$

$*; r']$. a 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 $pc, M \vdash a : t\ [\varepsilon : pc'; r']$, that is, $pc, M \vdash a : t\ [r]$.

Sub-case $E = [\,]\ \mathsf{handle}\ e_2\ \mathsf{done}$. E-HandleDone's first premise is of the form $pc, M \vdash a : t\ [*]$. Because a must be a value, Lemma 11 and E-Value yield $pc, M \vdash a : t\ [r]$.

Sub-case $E = [\,]\ \mathsf{handle}\ e_2\ \mathsf{raise}$. E-HandleRaise's first premise is the goal.

∘ *Case* (lift-app). e is $\langle v_1 \mid v_2 \rangle\, v$. Let θ stand for $(t' \xrightarrow{pc \sqcup \ell\ [r]} t)^\ell$. 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 7 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 8 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 7 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 8 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 7 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 8 yields $\ell \in H$. Lastly, by E-Bracket, we obtain $pc, M \vdash \langle !\, v_1 \mid !\, v_2 \rangle : t\ [r]$.

∘ *Case* (lift-proj). e is $\mathsf{proj}_j\ \langle v_1 \mid v_2 \rangle$. E-Proj's premise is $M \vdash \langle v_1 \mid v_2 \rangle : t_1 \times t_2$, where t_j is t. By Lemma 8, there exists $pc' \in H$ such that $pc' \lhd t_1 \times t_2$, which implies, in particular, $pc' \lhd t_j$. Furthermore, by Lemma 7, we have $M \vdash v_i : t_1 \times t_2$, for all $i \in \{1, 2\}$. By E-Proj, this implies $pc \sqcup pc', M \vdash \mathsf{proj}_j\ v_i : t_j\ [r]$. Lastly, by E-Bracket, we obtain $pc, M \vdash \langle \mathsf{proj}_j\ v_1 \mid \mathsf{proj}_j\ v_2 \rangle : t_j\ [r]$.

∘ *Case* (lift-case). e is $\langle v_1 \mid v_2 \rangle\ \mathsf{case}\ x \succ e_1\ e_2$. Lemma 8, applied to E-Case's first premise, yields $\ell \in H$. By applying Lemma 7 to E-Case's first two premises and re-building new instances of E-Case, we obtain $pc \sqcup \ell, M \vdash v_i\ \mathsf{case}\ x \succ \lfloor e_1 \rfloor_i\ \lfloor e_2 \rfloor_i : t\ [r]$, for all $i \in \{1, 2\}$. E-Case's third premise is $\ell \lhd t$, which allows applying E-Bracket, yielding the goal.

∘ *Case* (lift-context). e is $E[\langle a_1 \mid a_2 \rangle]$. If E is a bind context, then, because e cannot be reduced by (bind), $\langle a_1 \mid a_2 \rangle$ cannot be a value. If, on the other hand, E is a handle context, then, because (pop) isn't applicable, E must handle a_1 or a_2. In either case, we conclude that a_j is of the form $\mathsf{raise}\ \varepsilon\ v$, for some $j \in \{1, 2\}$. Now, e's typing derivation must end with an instance of E-Bind, E-Handle, E-HandleDone or E-HandleRaise, whose first premise is of the form $pc, M \vdash \langle a_1 \mid a_2 \rangle : t'\ [r_1]$. Because $\langle a_1 \mid a_2 \rangle$ isn't a value, this must be a consequence of E-Sub and E-Bracket, which yields $pc \sqcup \ell, M \vdash a_i : t'\ [r_1]$, for some $\ell \in H$ and for all $i \in \{1, 2\}$. In particular, taking $i = j$ and according to E-Sub and E-Raise, this implies $\ell \leq r_1(\varepsilon)$, whence $\ell \leq \sqcup r_1$. Thus, the security assumption in E-Bind, E-Handle, E-HandleDone or E-HandleRaise's second premise is greater than or equal to ℓ. As a result, by applying Lemma 7 to that premise, then building new instances of E-Bind, E-Handle, E-HandleDone or E-HandleRaise, we obtain

$pc \sqcup \ell, M \vdash \lfloor E \rfloor_i[a_i] : t\ [r]$, for all $i \in \{1,2\}$. There remains to apply E-BRACKET. If E is a bind or handle − raise context, then $\lfloor E \rfloor_j[a_j]\Uparrow$ holds. If, on the other hand, E is some other handle context, then $\ell \lhd t$ holds, according to E-HANDLE or E-HANDLEDONE'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 $e_i \,/_i\, \mu \to e_i' \,/_i\, \mu'$ 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, the judgement $pc \sqcup pc', M' \vdash e_j : t\ [r]$ holds as well. The result follows by E-BRACKET, whose fourth premise is preserved because $\cdot\Uparrow$ is preserved by reduction, i.e. for all $i \in \{1,2\}$, $e_i\Uparrow$ implies $e_i'\Uparrow$.

∘ *Case* (context). e is $E[e_0]$ and e' is $E[e_0']$, where $e_0 \,/_i\, \mu \to e_0' \,/_i\, \mu'$. Applying the induction hypothesis to E-BIND, E-HANDLE, E-HANDLEDONE, E-HANDLERAISE or E-FINALLY'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, E-HANDLEDONE, E-HANDLERAISE or E-FINALLY to conclude. □

The previous lemma entails the following, more abstract statement:

THEOREM 13 (SUBJECT REDUCTION). *If* $\vdash e \,/\, \mu : t\ [r]$ *and* $e \,/\, \mu \to e' \,/\, \mu'$ *then* $\vdash e' \,/\, \mu' : t\ [r]$.

PROOF. By CONF and Lemma 12. □

We do not give a *progress* statement (i.e. "no well-typed configuration is stuck") because it is unrelated to our concerns; that is, it would be of no use in the noninterference proof. If desired, progress for Core ML can be established via a straightforward case analysis.

5.6 On evaluation order

As explained in Section 3, our restricted syntax is fully explicit about evaluation order. In practice, it is possible to allow a 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:

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

Conversely, under a right-to-left evaluation strategy, the application $e_1\, e_2$ is encoded as bind $x_2 = e_2$ in bind $x_1 = e_1$ in $x_1\, x_2$, yielding another derived rule, that differs

in the security assumptions of the premises:

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

In either case, the expression that is evaluated last is typechecked at an increased security level, reflecting the fact that, if it receives control, then the other expression must have completed normally.

Some variants of ML, such as Caml-Light [Leroy et al. 1997] and Objective Caml [Leroy et al. 2002], leave the evaluation order unspecified. 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 all $\{i, j\} = \{1, 2\}$. However, there is a catch. Let us assume that e_i (resp. e_j) potentially raises an exception ε_i (resp. ε_j). Then, because E-RAISE annotates every exception with the current pc, and because pc can only increase within subexpressions, we must have $\sqcup r_i \leq r_j(\varepsilon_j)$ and $\sqcup r_j \leq r_i(\varepsilon_i)$. Of course, by definition, we also have $r_j(\varepsilon_j) \leq \sqcup r_j$ and $r_i(\varepsilon_i) \leq \sqcup r_i$. As a result, all four inequalities must be equalities. In other words, if both e_i and e_j are liable to raise at least one exception, then *all* exceptions in r_i and r_j must receive 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

In this section, we omit pc and r in typing judgements when they are unspecified, i.e. when they could be written $*$.

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 the judgements that involve Core ML expressions, because H is used only in V-BRACKET and E-BRACKET.) We write $e \rightarrow^\star a$ if there exists a store μ such that $e \,/\, \varnothing \rightarrow^\star a \,/\, \mu$, where $\varnothing$ is the empty store.

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

LEMMA 14. *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 13, by CONF and Lemma 11, 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, 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 established in Section 4.4 to reformulate this result in a Core ML setting:

THEOREM 15 (NON-INTERFERENCE). *Choose $\ell, h \in \mathcal{L}$ such that $h \not\sqsubseteq \ell$. Let $h \lhd t$. Assume $(x \mapsto t) \vdash e : \mathsf{int}^\ell$, where e is a Core ML expression. If, for all $i \in \{1,2\}$, $\vdash v_i : t$ and $e[x \Leftarrow v_i] \rightarrow^\star v_i'$ hold, then $v_1' = v_2'$.*

PROOF. Let H be the upward closure of $\{h\}$. Define $v = \langle v_1 \mid v_2 \rangle$. By V-BRACKET, $\vdash_H v : t$ holds. Lemma 10 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, there exists an answer a such that $e[x \Leftarrow v] \rightarrow^\star a$. Then, Lemma 2 yields $\lfloor a \rfloor_i = v_i'$ for all $i \in \{1,2\}$, which implies that a is a value. Lastly, $h \not\sqsubseteq \ell$ yields $\ell \notin H$. The result follows by Lemma 14. □

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 in the expression e has a "high"-level type t, e admits 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). Because both programs are assumed to terminate, this is a *weak* noninterference statement; see the discussion in Section 4.4. For simplicity, we have restricted our attention to the case of integer results, which may be compared using equality. It would be possible to give a more general statement, expressed in terms of a notion of observational equivalence, as a corollary of Theorem 15. As another corollary, one may allow several holes, instead of the single hole x. This essentially amounts to specializing Theorem 15 to the case where t is a tuple type.

7. GENERIC PRIMITIVE OPERATIONS

Practical programming languages usually provide many primitive operations, such as integer arithmetic operators. Some languages, such as Caml-Light [Leroy et al. 1997], Objective Caml [Leroy et al. 2002] or SML [Milner et al. 1997], provide generic (i.e. polymorphic) comparison, hashing or marshalling functions. In the following, we present a way of assigning types to such generic 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.

7.1 Semantics

Let o range over a set of operation names, and extend the syntax of expressions as follows:

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

We assume that the semantics of every operation o is given as a partial function $[\![o]\!]$ which maps closed Core ML configurations $v \,/\, \mu$ to closed Core ML answers. As a result, operations may access the store and raise exceptions; however, we do not allow them to modify the store. For simplicity, we only consider unary operations; multiple arguments must be passed in a tuple.

Let *accessibility* with respect to a (Core ML) store μ be the smallest transitive relation between (Core ML) values such that, for every value v, every sub-term of v that is not found under a λ-abstraction is accessible through v and, for every memory location m, $\mu(m)$ is accessible through m. If v' is accessible through v with respect to μ, we also say that v' is accessible through $v \,/\, \mu$. If no λ-abstraction is accessible through $v \,/\, \mu$, then let μ_v be the restriction of μ to the memory locations

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

Fig. 7. Collecting security annotations

accessible through $v \,/\, \mu$, and let $|v \,/\, \mu|$ stand for $v \,/\, \mu_v$. In words, when $|v \,/\, \mu|$ is defined, it represents the whole data structure accessible through v within the store μ. Below, we use this notion to force operations to access data only through v. This prevents them from being stateful, e.g. by maintaining a pointer to a private, mutable data structure, and is necessary for our typing rule to be sound.

The semantics of Core ML^2 is extended as follows:

$$\begin{array}{rcll} o\,v \,/_i\, \mu & \rightarrow & [\![o]\!](|v \,/\, \lfloor\mu\rfloor_i|) \,/_i\, \mu & \text{(op)} \\ & & \textit{if } i \in \{1,2\} & \\ o\,v \,/\, \mu & \rightarrow & [\![o]\!](|\lfloor v \,/\, \mu\rfloor_1|) \,/\, \mu & (\text{op}_\bullet) \\ & & \textit{if } |\lfloor v \,/\, \mu\rfloor_1| = |\lfloor v \,/\, \mu\rfloor_2| & \\ o\,v \,/\, \mu & \rightarrow & \langle o\,\lfloor v\rfloor_1 \mid o\,\lfloor v\rfloor_2\rangle \,/\, \mu & \text{(lift-op)} \\ & & \textit{if } |\lfloor v \,/\, \mu\rfloor_1| \neq |\lfloor v \,/\, \mu\rfloor_2| & \end{array}$$

(op) and (op$_\bullet$) are the basic reduction rules associated with primitive operations. It would be possible to merge them in a single rule, but we believe this formulation is somewhat clearer. (op) accesses the store through its projection $\lfloor\mu\rfloor_i$, as done e.g. by (deref) in Section 4.3. As explained above, the semantic function $[\![o]\!]$ is applied to $|v \,/\, \lfloor\mu\rfloor_i|$, rather than $v \,/\, \lfloor\mu\rfloor_i$, which makes it impossible for the operation to use any data but that accessible through v itself. The rule is inapplicable if $|v \,/\, \lfloor\mu\rfloor_i|$ is undefined, i.e. if a λ-abstraction is accessible through v. Indeed, it is illegal to apply a generic primitive operation to a data structure that contains a λ-abstraction; our typing rule will prevent this situation from arising.

(op) applies only when $i \in \{1,2\}$, i.e. when performing reduction under brackets. When i is $\bullet$, i.e. when reducing outside brackets, exactly one of (op$_\bullet$) and (lift-op) applies. If the argument to o is the same under both projections, then (op$_\bullet$) is applicable, and performs a shared reduction step. Otherwise, (lift-op) applies, and introduces brackets at the top level, so as to allow reduction via (op). We let the reader check that the results of Section 4.4 are preserved by this extension.

7.2 Typing

We introduce a two-place predicate $\blacktriangleleft$, which relates a type and a security level, and whose definition appears in Figure 7. In short, $t \blacktriangleleft \ell$ holds if and only if *all* of the security annotations which appear within t, including its sub-terms, are less than or equal to ℓ. It also requires t to have no function type as a sub-term. This definition mimics the behavior of generic primitive operations, such as Caml-Light's generic comparison or hashing operations, which traverse data structures recursively, and fail upon encountering a closure. The predicate $\blacktriangleleft$ enjoys the following property:

LEMMA 16. *Assume $\vdash_H v \,/\, \mu : t$ and $t \blacktriangleleft \ell$. If a sub-term of the form $\langle v_1 \mid v_2\rangle$ is accessible through $v \,/\, \mu$, with $v_1 \neq v_2$, then $\ell \in H$.*

PROOF. By induction on the path that leads to $\langle v_1 \mid v_2\rangle$.

∘ *Case* v is $\langle v_1 \mid v_2 \rangle$. Then, $\vdash_H v \,/\, \mu : t$ implies $pc \lhd t$, for some $pc \in H$. By Lemma 7, both v_1 and v_2 have type t; so, considering that these values differ, t cannot be built solely out of products and unit. (Types that are built solely out of products and unit are inhabited by a single value.) In that case, $pc \lhd t$ and $t \blacktriangleleft \ell$ imply $pc \leq \ell$, whence $\ell \in H$.

∘ *Case* v is $(v_1',\ v_2')$, and $\langle v_1 \mid v_2 \rangle$ is accessible through $v_j' \,/\, \mu$, for some $j \in \{1, 2\}$. Then, we must have $\vdash_H v_j' \,/\, \mu : t_j$, where $t = t_1 \times t_2$, and $t_j \blacktriangleleft \ell$. The result follows by induction hypothesis.

∘ *Case* v is $\mathsf{inj}_j\ v'$, and $\langle v_1 \mid v_2 \rangle$ is accessible through $v' \,/\, \mu$, for some $j \in \{1, 2\}$. Then, we must have $\vdash_H v' \,/\, \mu : t'$, where $t = (t' +^j *)^*$, and $t' \blacktriangleleft \ell$. The result follows by induction hypothesis.

∘ *Case* v is m, and $\langle v_1 \mid v_2 \rangle$ is accessible through $\mu(m) \,/\, \mu$. Then, we must have $\vdash_H \mu(m) \,/\, \mu : t'$, where $t = t'\ \mathsf{ref}^*$ and $t' \blacktriangleleft \ell$. The result follows by induction hypothesis. □

We wish to give a typing rule for primitive operations that is independent of their semantics. To achieve this, we will assume that every primitive operation comes with a typing rule which is sufficient to ensure type safety in the usual sense, and we will show how to refine it with information flow analysis in mind. In the following, R denotes a finite set of exception names. We write $R : pc$ for the row which maps ε to pc if $\varepsilon \in R$ and to $\bot$ otherwise. For every operation o, we assume a ternary relation *typeof*(o) such that, if $(t', t, R) \in$ *typeof*(o), then $M \vdash v : t'$ and $M \vdash \mu$ imply $pc, M \vdash [\![o]\!](|v \,/\, \mu|) : t\ \ [R : pc]$ for all $pc \in \mathcal{L}$. Roughly speaking, this amounts to assuming subject reduction for (op) and (op$_\bullet$). Then, we augment the type system with the following rule:

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

E-Primitive requires the security level of the result type t to dominate *all* of the security levels which appear in the argument type t'. Indeed, because nothing is known about the semantics of o, no better approximation can be given: the result may depend on any value accessible through v within the current store. Any exception that is liable to be raised by o is marked similarly.

In short, given a typing rule for o that does not know about security, encoded by the ternary relation *typeof*(o), our approach produces a refined version, which guarantees noninterference, regardless of o's semantics. Of course, we must check that the new reduction rules satisfy subject reduction under the extended type system. This is done by adding new cases to the proof of Lemma 12, as follows:

∘ *Case* (op). According to Conf and E-Primitive, we have $(t', t, R) \in$ *typeof*(o) and $M \vdash v : t'$ and $M \vdash \mu$. The latter implies $M \vdash \lfloor \mu \rfloor_i$. According to our assumption about *typeof*$(\cdot)$, this implies $pc, M \vdash [\![o]\!](|v \,/\, \lfloor \mu \rfloor_i|) : t\ \ [R : pc]$. The result follows by E-Sub and Conf.

∘ *Case* (op$_\bullet$). Analogous; the rule's side-condition is unused.

∘ *Case* (lift-op). Because $|\lfloor v \,/\, \mu \rfloor_1|$ and $|\lfloor v \,/\, \mu \rfloor_2|$ differ, a sub-term of the form $\langle v_1 \mid v_2 \rangle$, where $v_1 \neq v_2$, must be accessible through $v \,/\, \mu$. According to Conf and

E-PRIMITIVE, we may then apply Lemma 16, yielding $\ell \in H$. Applying Lemma 7 and building a new instance of E-PRIMITIVE, we obtain $pc \sqcup \ell, M \vdash o\ \lfloor v \rfloor_i : t\ [R : pc \sqcup \ell]$ for all $i \in \{1, 2\}$. Recalling $\ell \lhd t$, we conclude with E-BRACKET. □

7.3 Applications

Let us now illustrate the use of this general mechanism. We give typing rules for several concrete primitive operations and prove that they are instances of E-PRIMITIVE.

To begin, let us consider a binary integer arithmetic operation, such as addition. The meaning of addition is, of course, given by $[\![+]\!]((k_1,\ k_2)\ /\ \mu) = k_1 + k_2$. Its treatment is in fact quite simple, because it is monomorphic: it maps a pair of integers to an integer. For this reason, it would be easy to deal with it directly. Nevertheless, let us proceed. Define *typeof* $(+)$ by setting $(\mathsf{int}^* \times \mathsf{int}^*, \mathsf{int}^*, \varnothing) \in$ *typeof* $(+)$. We let the reader check that this definition satisfies the requirement stated in Section 7.2. Then, E-PRIMITIVE may be specialized as follows:

$$\text{E-ADD}\quad \frac{\Gamma, M \vdash v : \mathsf{int}^\ell \times \mathsf{int}^\ell}{*, \Gamma, M \vdash +v : \mathsf{int}^\ell\ [*]}$$

This rule effectively makes the sum's security the least upper bound of the operands' levels. Most operations on primitive data can be dealt with in a similar manner. In some cases, a direct treatment is preferable; in the case of division, for instance, an exception is raised only if the second argument is zero, so the security level associated with the exception should be that of the second argument alone, not the union of both arguments' levels, as we would obtain by specializing E-PRIMITIVE.

The treatment of Caml-Light's generic (i.e. polymorphic) comparison operators is more interesting, and is the true motivation for developing our generic approach. Let bool^ℓ stand for $(\mathsf{unit} + \mathsf{unit})^\ell$. Define *typeof* $(=)$ by setting $(t \times t, \mathsf{bool}^*, \varnothing) \in$ *typeof* $(=)$ for every type t. We do not define $[\![=]\!]$, because that would require a somewhat lengthy co-inductive definition, which is irrelevant here; let us simply say that it always produces a Boolean value, so the requirement stated in Section 7.2 is satisfied. Then, specializing E-PRIMITIVE yields

$$\frac{\Gamma, M \vdash v : t \times t \qquad t \blacktriangleleft \ell}{*, \Gamma, M \vdash =v : \mathsf{bool}^\ell\ [*]}$$

All of Caml-Light's generic comparison operators (namely, `=`, `<>`, `<`, `>`) can be dealt with in the same manner. (One exception is physical equality `==`, which cannot be defined in our framework, since only mutable values have addresses in our semantics. Anyway, it would be difficult to give it a precise type, since products do not carry a security annotation.) 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 fact by requiring ℓ to dominate all security annotations which appear in t.

Lastly, generic hashing and marshalling operations can be dealt with similarly, yielding the following typing rules:

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

By contrast, in Myers' Java-based framework [Myers 1999a; 1999b], 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 labeled this. The parametric class Vector[L], for instance, must compute a hash code 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.

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 variables, constraints, and using them to form universally quantified, constrained type schemes, in the style of HM(X) [Odersky et al. 1999]. 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. For more details about the proof of correspondence between MLIF_0 and MLIF, the reader is referred to [Pottier 2001]. For information about deriving a set of type inference rules from the typing rules given in this section, see [Odersky et al. 1999; Sulzmann et al. 1999; Sulzmann 2000]. In this section, we will concentrate mainly on constraint solving, because $\lhd$ and $\blacktriangleleft$ constraints are new.

8.1 Types and constraints

In MLIF, the grammar of types, rows and levels is extended with *type*, *row* and *level variables*, written β, γ and δ, respectively. We write α for a variable of arbitrary kind. Furthermore, Rémy's [Rémy 1993] row syntax is introduced, turning rows into finite lists of bindings from exception names to levels, terminated with a row variable or with a uniform row $\partial\lambda$.

$$\begin{array}{rcl} \tau & ::= & \beta \mid \mathsf{unit} \mid \mathsf{int}^\lambda \mid (\tau \xrightarrow{\pi\ [\rho]} \tau)^\lambda \mid \tau\ \mathsf{ref}^\lambda \mid \tau \times \tau \mid (\tau + \tau)^\lambda \\ \rho & ::= & \gamma \mid (\varepsilon : \lambda; \rho) \mid \partial\lambda \\ \lambda, \pi & ::= & \delta \mid \ell \end{array}$$

(λ and π are level meta-variables, just as ℓ and pc were ground level meta-variables.) The variable-free types (resp. rows, levels) of MLIF are isomorphic to the types (resp. rows, levels) of MLIF_0; we identify them and refer to them as *ground*. Then, *constraints* are defined as follows:

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

The constraint forms on the first line are standard [Odersky et al. 1999]. Those on the second line are subtyping constraints. We will use $\tau_1 = \tau_2$ as syntactic sugar for $\tau_1 \leq \tau_2 \wedge \tau_2 \leq \tau_1$. The third line lists custom constraint forms, which correspond to the notions developed in Sections 5 and 7. We will say that a constraint C *involves types* (resp. *rows*, resp. *levels*) if it is of the form $\tau \leq \tau$, $\lambda \lhd \tau$ or $\tau \blacktriangleleft \lambda$ (resp. $\rho \leq \rho$, resp. $\lambda \leq \lambda$). We omit the sorting rules necessary to ensure that terms and constraints which contain rows are well-formed; see [Rémy 1993]. Let us simply recall that these rules associate a cofinite subset of $\mathcal{E}$, written $\text{dom}(\rho)$ and called the *domain* of ρ, with every row ρ.

Let a *ground assignment* ϕ map every variable α to a ground type, row or level, according to its kind. The meaning of terms and constraints under an assignment ϕ is defined in the obvious way; we write $\phi \vdash C$ if and only if ϕ *satisfies* C. A constraint C is *satisfiable* if and only if there exists an assignment ϕ such that $\phi \vdash C$. We write $C \Vdash C'$ (read: C *entails* C') if and only if every assignment ϕ which satisfies C satisfies C' as well. We write $C \equiv C'$ if and only if satisfiability of C and satisfiability of C' are equivalent.

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 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 Figures 8 and 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 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. In particular, every term of the form $\sqcup\rho$ may be replaced with a fresh level variable δ, together with the constraint $\rho \leq \partial\delta$.

8.3 Non-interference

We prove the following statement by induction on type derivations, along the lines of [Pottier 2001].

LEMMA 17 (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 [Pottier 2001].) In particular, every ground typing judgement in MLIF is also a valid judgement in MLIF_0. This allows us to lift our noninterference result to MLIF. That is, the statement of Theorem 15 remains valid if $(x \mapsto t) \vdash e : \mathsf{int}^\ell$ and $\vdash v_i : t$ are read as MLIF typing judgements.

8.4 Type inference

It is easy to check that there exists a type inference algorithm which computes principal types for MLIF. Sulzmann [Sulzmann 2000] shows 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.

$$\textsc{v-Unit}\quad C, \Gamma \vdash () : \mathsf{unit} \qquad \textsc{v-Int}\quad C, \Gamma \vdash k : \mathsf{int}^* \qquad \textsc{v-Var}\quad \frac{\Gamma(x) = \forall \bar{\alpha}[D].\tau \qquad C \Vdash \exists \bar{\alpha}.D}{C \wedge D, \Gamma \vdash x : \tau}$$

$$\textsc{v-Abs}\quad \frac{C, \pi, \Gamma[x \mapsto \tau'][f \mapsto (\tau' \xrightarrow{\pi\ [\rho]} \tau)^\lambda] \vdash e : \tau\ [\rho]}{C, \Gamma \vdash \mathsf{fix}\, f.\lambda x.e : (\tau' \xrightarrow{\pi\ [\rho]} \tau)^\lambda} \qquad \textsc{v-Pair}\quad \frac{C, \Gamma \vdash v_1 : \tau_1 \qquad C, \Gamma \vdash v_2 : \tau_2}{C, \Gamma \vdash (v_1,\ v_2) : \tau_1 \times \tau_2}$$

$$\textsc{v-Inj}\quad \frac{C, \Gamma \vdash v : \tau}{C, \Gamma \vdash \mathsf{inj}_j\ v : (\tau +^j *)^*} \qquad \textsc{v-Sub}\quad \frac{C, \Gamma \vdash v : \tau' \qquad C \Vdash \tau' \leq \tau}{C, \Gamma \vdash v : \tau}$$

Fig. 8. The type system MLIF (values)

$$\textsc{e-Value}\quad \frac{C, \Gamma \vdash v : \tau}{C, *, \Gamma \vdash v : \tau\ [*]} \qquad \textsc{e-Raise}\quad \frac{C, \Gamma \vdash v : \mathit{typexn}(\varepsilon)}{C, \pi, \Gamma \vdash \mathsf{raise}\ \varepsilon\ v : *\ [\varepsilon : \pi; *]} \qquad \textsc{e-App}\quad \frac{C, \Gamma \vdash v_1 : (\tau' \xrightarrow{\pi \sqcup \lambda\ [\rho]} \tau)^\lambda \quad C, \Gamma \vdash v_2 : \tau' \qquad C \Vdash \lambda \lhd \tau}{C, \pi, \Gamma \vdash v_1\ v_2 : \tau\ [\rho]}$$

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

$$\textsc{e-Deref}\quad \frac{C, \Gamma \vdash v : \tau'\ \mathsf{ref}^\lambda \qquad C \Vdash \tau' \leq \tau \qquad C \Vdash \lambda \lhd \tau}{C, \pi, \Gamma \vdash\ !v : \tau\ [*]} \qquad \textsc{e-Proj}\quad \frac{C, \Gamma \vdash v : \tau_1 \times \tau_2}{C, *, \Gamma \vdash \mathsf{proj}_j\ v : \tau_j\ [*]}$$

$$\textsc{e-Case}\quad \frac{C, \Gamma \vdash v : (\tau_1 + \tau_2)^\lambda \qquad \forall j \in \{1, 2\} \quad C, \pi \sqcup \lambda, \Gamma[x \mapsto \tau_j] \vdash e_j : \tau\ [\rho] \qquad C \Vdash \lambda \lhd \tau}{C, \pi, \Gamma \vdash v\ \mathsf{case}\ x \succ e_1\ e_2 : \tau\ [\rho]}$$

$$\textsc{e-Let}\quad \frac{C \wedge D, \Gamma \vdash v : \tau' \qquad \bar{\alpha} \cap \mathrm{fv}(C, \Gamma) = \varnothing \quad C, \pi, \Gamma[x \mapsto \forall \bar{\alpha}[D].\tau'] \vdash e : \tau\ [\rho]}{C \wedge \exists \bar{\alpha}.D, \pi, \Gamma \vdash \mathsf{let}\ x = v\ \mathsf{in}\ e : \tau\ [\rho]} \qquad \textsc{e-Bind}\quad \frac{C, \pi, \Gamma \vdash e_1 : \tau'\ [\rho_1] \quad C, \pi \sqcup (\sqcup \rho_1), \Gamma[x \mapsto \tau'] \vdash e_2 : \tau\ [\rho_2]}{C, \pi, \Gamma \vdash \mathsf{bind}\ x = e_1\ \mathsf{in}\ e_2 : \tau\ [\rho_1 \sqcup \rho_2]}$$

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

$$\textsc{e-HandleDone}\quad \frac{C, \pi, \Gamma \vdash e_1 : \tau\ [\rho_1] \quad C, \pi \sqcup (\sqcup \rho_1), \Gamma \vdash e_2 : \tau\ [\rho_2] \qquad C \Vdash \sqcup \rho_1 \lhd \tau}{C, \pi, \Gamma \vdash e_1\ \mathsf{handle}\ e_2\ \mathsf{done} : \tau\ [\rho_2]} \qquad \textsc{e-HandleRaise}\quad \frac{C, \pi, \Gamma \vdash e_1 : \tau\ [\rho] \quad C, \pi \sqcup (\sqcup \rho), \Gamma \vdash e_2 : *\ [\partial\bot]}{C, \pi, \Gamma \vdash e_1\ \mathsf{handle}\ e_2\ \mathsf{raise} : \tau\ [\rho]}$$

$$\textsc{e-Finally}\quad \frac{C, \pi, \Gamma \vdash e_1 : \tau\ [\rho] \quad C, \pi, \Gamma \vdash e_2 : *\ [\partial\bot]}{C, \pi, \Gamma \vdash e_1\ \mathsf{finally}\ e_2 : \tau\ [\rho]} \qquad \textsc{e-Sub}\quad \frac{C, \pi, \Gamma \vdash e : \tau'\ [\rho'] \qquad C \Vdash \tau' \leq \tau \qquad C \Vdash \rho' \leq \rho}{C, \pi, \Gamma \vdash e : \tau\ [\rho]}$$

Fig. 9. The type system MLIF (expressions)

$$\tau \approx_C \tau \qquad \frac{\tau_1 \approx_C \tau_2}{\tau_2 \approx_C \tau_1} \qquad \frac{\tau_1 \approx_C \tau_2 \qquad \tau_2 \approx_C \tau_3}{\tau_1 \approx_C \tau_3} \qquad \frac{\tau_1 \leq \tau_2 \in C}{\tau_1 \approx_C \tau_2}$$

$$\frac{(\tau_1' \xrightarrow{*\ [*]} \tau_1)^* \approx_C (\tau_2' \xrightarrow{*\ [*]} \tau_2)^*}{\tau_1' \approx_C \tau_2' \qquad \tau_1 \approx_C \tau_2} \qquad \frac{\tau_1\ \mathsf{ref}^* \approx_C \tau_2\ \mathsf{ref}^*}{\tau_1 \approx_C \tau_2} \qquad \frac{\tau_1 \times \tau_1' \approx_C \tau_2 \times \tau_2'}{\tau_1 \approx_C \tau_2 \qquad \tau_1' \approx_C \tau_2'}$$

$$\frac{(\tau_1 + \tau_1')^* \approx_C (\tau_2 + \tau_2')^*}{\tau_1 \approx_C \tau_2 \qquad \tau_1' \approx_C \tau_2'}$$

Fig. 10. Structural equivalence induced by a constraint

8.5 Constraint solving

Our subtyping relation is structural (a.k.a. atomic); that is, two ground types which are in the subtyping relation must have the same structure, and may differ only in their security annotations. Constraint solving for atomic subtyping is decidable and well understood [Rehof 1997]. The introduction of rows is essentially orthogonal to other constraint solving issues [Fähndrich 1999; Pottier 2000]. There mainly remains to show that the the custom constraint forms employed by MLIF preserve the decidability of constraint solving. In the following, we do so by giving a simple algorithm that determines whether a constraint is satisfiable. We do not aim at efficiency, because that would require more advanced rewriting strategies and constraint simplification techniques, which we will study in a later paper.

By α-conversion and scope extrusion, any constraint C can be written $\exists \bar{\alpha}.C'$, where C' does not employ existential quantification. Moreover, the constraint C is satisfiable if and only if C' is satisfiable. Thus, from here on, we will consider solely constraints that do not make use of existential quantitication. As a result, every constraint C can be viewed as a conjunction $c_1 \wedge \ldots \wedge c_n$, where every *elementary constraint* c_i is of the form **true**, **false**, $\cdot \leq \cdot$, $\cdot \lhd \cdot$ or $\cdot \blacktriangleleft \cdot$. We identify such conjunctions modulo permutations and repetitions of elementary constraints. We write $C' \in C$ (read: C' *appears in* C) if and only if $C = * \wedge C'$.

Because ground types are finite, our algorithm must perform an *occur check* in order to verify that the constraint at hand does not impose a cycle on type structure. For this purpose, given a constraint C, we introduce an equivalence relation $\approx_C$ between types, defined in Figure 10. The rightmost rule in the figure states that any two types which are related by a subtyping constraint should be structurally equivalent; the next rules propagate structural equivalence from terms to sub-terms. Let us write $\tau_1 \prec \tau$ if and only if τ_1 is a strict subterm of τ. Then, we define the *domination* relation induced by C as follows: $\tau_1 \prec_C \tau$ holds if and only if there exist τ_1' and τ' such that $\tau_1 \approx_C \tau_1'$ and $\tau_1' \prec \tau'$ and $\tau' \approx_C \tau$. Let $\prec_C^+$ denote the transitive closure of $\prec_C$. A constraint C *satisfies the occur check* if and only if there exists no type τ such that $\tau \prec_C^+ \tau$.

LEMMA 18 (OCCUR CHECK). *If C fails the occur check, then C is not satisfiable.*

PROOF. Define the *height* of a ground type by $h((t' \xrightarrow{*\ [*]} t)^*) = h(t \times t') = h((t + t')^*) = 1 + \max(h(t), h(t'))$, $h(t\ \mathsf{ref}^*) = 1 + h(t)$ and $h(\mathsf{unit}) = h(\mathsf{int}^*) = 0$. If $\phi \vdash C$, then $\tau \approx_C \tau'$ implies $h(\phi(\tau)) = h(\phi(\tau'))$ and $\tau \prec \tau'$ implies $\phi(\tau) < \phi(\tau')$. Thus, $\tau \prec_C^+ \tau$ implies $h(\tau) < h(\tau)$. The result follows. □

Types

$$\begin{array}{rcll}
\mathsf{int}^{\lambda_1} \leq \mathsf{int}^{\lambda_2} & \vDash & \lambda_1 \leq \lambda_2 & \\
(\tau_1' \xrightarrow{\pi_1\ [\rho_1]} \tau_1)^{\lambda_1} \leq (\tau_2' \xrightarrow{\pi_2\ [\rho_2]} \tau_2)^{\lambda_2} & \vDash & \tau_2' \leq \tau_1' \wedge \pi_2 \leq \pi_1 \wedge \rho_1 \leq \rho_2 \wedge \tau_1 \leq \tau_2 \wedge \lambda_1 \leq \lambda_2 & \\
\tau_1\ \mathsf{ref}^{\lambda_1} \leq \tau_2\ \mathsf{ref}^{\lambda_2} & \vDash & \tau_1 = \tau_2 \wedge \lambda_1 \leq \lambda_2 & \\
\tau_1 \times \tau_1' \leq \tau_2 \times \tau_2' & \vDash & \tau_1 \leq \tau_2 \wedge \tau_1' \leq \tau_2' & \\
(\tau_1 + \tau_1')^{\lambda_1} \leq (\tau_2 + \tau_2')^{\lambda_2} & \vDash & \tau_1 \leq \tau_2 \wedge \tau_1' \leq \tau_2' \wedge \lambda_1 \leq \lambda_2 &
\end{array}$$

Rows

$$\begin{array}{rcl}
(\varepsilon : \lambda_1; \rho_1) \leq (\varepsilon : \lambda_2; \rho_2) & \vDash & \lambda_1 \leq \lambda_2 \wedge \rho_1 \leq \rho_2 \\
(\varepsilon : \lambda; \rho) \leq \partial\lambda' & \vDash & \lambda \leq \lambda' \wedge \rho \leq \partial\lambda' \\
\partial\lambda' \leq (\varepsilon : \lambda; \rho) & \vDash & \lambda' \leq \lambda \wedge \partial\lambda' \leq \rho \\
\partial\lambda_1 \leq \gamma_1 \leq \cdots \leq \gamma_n \leq \partial\lambda_2 & \vDash & \lambda_1 \leq \lambda_2
\end{array}$$

Guards

$$\begin{array}{rcl}
\lambda' \lhd \mathsf{int}^{\lambda} & \vDash & \lambda' \leq \lambda \\
\lambda' \lhd (* \xrightarrow{*\ [*]} *)^{\lambda} & \vDash & \lambda' \leq \lambda \\
\lambda' \lhd *\ \mathsf{ref}^{\lambda} & \vDash & \lambda' \leq \lambda \\
\lambda' \lhd \tau_1 \times \tau_2 & \vDash & \lambda' \lhd \tau_1 \wedge \lambda' \lhd \tau_2 \\
\lambda' \lhd (* + *)^{\lambda} & \vDash & \lambda' \leq \lambda
\end{array}$$

$$\begin{array}{rcl}
\mathsf{int}^{\lambda} \blacktriangleleft \lambda' & \vDash & \lambda \leq \lambda' \\
\tau\ \mathsf{ref}^{\lambda} \blacktriangleleft \lambda' & \vDash & \tau \blacktriangleleft \lambda' \wedge \lambda \leq \lambda' \\
\tau \times \tau' \blacktriangleleft \lambda' & \vDash & \tau \blacktriangleleft \lambda' \wedge \tau' \blacktriangleleft \lambda' \\
(\tau + \tau')^{\lambda} \blacktriangleleft \lambda' & \vDash & \tau \blacktriangleleft \lambda' \wedge \tau' \blacktriangleleft \lambda' \wedge \lambda \leq \lambda'
\end{array}$$

Errors

$$\begin{array}{rcll}
\ell_1 \leq \delta_1 \leq \cdots \leq \delta_n \leq \ell_2 & \vDash & \mathbf{false} & \text{if } \ell_1 \not\leq_{\mathcal{L}} \ell_2 \\
\tau_1 \leq \tau_2 & \vDash & \mathbf{false} & \text{if } \tau_1 \not\sim \tau_2 \\
(* \xrightarrow{*\ [*]} *)^{*} \blacktriangleleft * & \vDash & \mathbf{false} &
\end{array}$$

Context

$$*\wedge C \ \vDash\ C' \qquad \text{if } C \vDash C'$$

Fig. 11. Syntactic constraint implication

Roughly speaking, once it is known that the constraint passes the occur check, the bulk of the constraint solving procedure consists in *expanding* types and rows and *decomposing* constraints, so as to obtain constraints that bear on variables or on atoms only. The absence of cycles in the type structure guarantees that the expansion process terminates.

Let us introduce a so-called *syntactic* implication predicate between constraints, written $C \vDash C'$, defined by the rules in Figure 11. This predicate allows deriving (a finite number of) logical consequences of a constraint C. The rules in *Types* and *Guards* (see Figure 11) concern constraints bearing on types whose structure is known, i.e. non-variable types. Such constraints are decomposed into a number of sub-constraints bearing on their sub-terms. Similarly, the rules in *Rows* decompose constraints bearing on non-variable rows. The last rule in *Rows* allows $\partial\lambda_1$ and $\partial\lambda_2$ to be linked by an arbitrarily long path of row variables. (This is made necessary by the fact that the existence of a constraint $\partial\lambda \leq \gamma$ does not allow expanding γ into $\partial\delta$, for a fresh δ. Indeed, $\partial\ell \leq r$ does *not* imply that r is a constant row.) The

Closure

$$\frac{C \vDash C' \qquad C' \notin C}{C \twoheadrightarrow C \wedge C'}$$

$$\frac{\varepsilon_1 \neq \varepsilon_2 \qquad \gamma \notin \mathrm{fv}(C) \qquad \nexists \rho \quad \rho_1 \leq (\varepsilon_2 : \lambda_2; \rho) \wedge (\varepsilon_1 : \lambda_1; \rho) \leq \rho_2 \in C}{C \wedge (\varepsilon_1 : \lambda_1; \rho_1) \leq (\varepsilon_2 : \lambda_2; \rho_2) \twoheadrightarrow C \wedge \rho_1 \leq (\varepsilon_2 : \lambda_2; \gamma) \wedge (\varepsilon_1 : \lambda_1; \gamma) \leq \rho_2}$$

Expansion

$$\frac{\beta \lessgtr \mathsf{unit} \in C}{C \twoheadrightarrow C[\mathsf{unit}/\beta]} \qquad \frac{\beta \lessgtr \mathsf{int}^* \in C \qquad \delta \notin \mathrm{fv}(C)}{C \twoheadrightarrow C[\mathsf{int}^\delta/\beta]} \qquad \frac{\begin{array}{c}\beta \lessgtr (* \xrightarrow{* \ [*]} *)^* \in C \\ \beta', \beta'', \gamma, \delta, \delta' \notin \mathrm{fv}(C)\end{array}}{C \twoheadrightarrow C[(\beta'' \xrightarrow{\delta' \ [\gamma]} \beta')^\delta/\beta]}$$

$$\frac{\beta \lessgtr * \ \mathsf{ref}^* \in C \qquad \beta', \delta \notin \mathrm{fv}(C)}{C \twoheadrightarrow C[\beta' \ \mathsf{ref}^\delta/\beta]} \qquad \frac{\beta \lessgtr * \times * \in C \qquad \beta', \beta'' \notin \mathrm{fv}(C)}{C \twoheadrightarrow C[\beta' \times \beta''/\beta]}$$

$$\frac{\beta \lessgtr (* + *)^* \in C \qquad \beta', \beta'', \delta \notin \mathrm{fv}(C)}{C \twoheadrightarrow C[(\beta' + \beta'')^\delta/\beta]} \qquad \frac{\gamma \lessgtr (\varepsilon : *; *) \in C \qquad \gamma', \delta \notin \mathrm{fv}(C)}{C \twoheadrightarrow C[(\varepsilon : \delta; \gamma')/\gamma]}$$

Fig. 12. Rewriting constraints

first rule in *Errors* discovers paths of the form $\ell_1 \leq \cdots \leq \ell_2$, which are inconsistent unless $\ell_1 \leq \ell_2$ holds in $\mathcal{L}$. The second error rule encodes the fact that any types which are in the subtyping relation must have the same head constructor. (Let $\tau_1 \sim \tau_2$ hold if and only if either one of τ_1, τ_2 is a variable or τ_1 and τ_2 are (non-variable) types with the same head constructor.) The last error rule reflects the fact that an arrow type cannot satisfy a $\blacktriangleleft$ constraint.

The final piece in the puzzle is a reduction relation $\twoheadrightarrow$ on constraints, defined in Figure 12. The first rewriting rule closes the constraint under syntactic implication. The second rule solves subtyping constraints between rows that do not exhibit the same head label; in other words, it allows row labels to commute. It introduces a new row variable γ, which must be fresh with respect to the whole constraint, as expressed by its second premise. The last premise prevents multiple applications of the rule, which would compromise termination. The remaining rules *expand* every type or row variable that is related to a non-variable term. This is achieved by substituting for the former, within the whole constraint, a fresh term whose structure mirrors the latter. $\tau_1 \lessgtr \tau_2 \in C$ is short for $\tau_1 \leq \tau_2 \in C \vee \tau_2 \leq \tau_1 \in C$. The following lemma states that reduction preserves satisfiability.

Lemma 19 (Correctness). *If $C \twoheadrightarrow C'$ then $C \equiv C'$.*

Proof. By inspection of the definitions of $\vDash$ and $\twoheadrightarrow$. □

Reduction is strongly normalizing. This property ensures that the constraint solving algorithm terminates.

Lemma 20 (Termination). *There is no infinite reduction for $\twoheadrightarrow$ out of a constraint that satisfies the occur check.*

Proof. Let us first remark that, if C passes the occur check, then so does every reduct of C.

Let C be a constraint that satisfies the occur check. Define the *height* of τ with respect to C, written $h_C(\tau)$, as $\max\{n \mid \exists \tau_1 \ldots \tau_n \quad \tau_n \prec_C \cdots \prec_C \tau_1 \prec_C \tau\}$. Because C only has a finite number of sub-terms, any infinite descending chain for $\prec_C$ must exhibit a cycle. However, because C satisfies the occur check, no such cycle exists. As a result, $h_C(\tau)$ must be finite.

A row label ε is said to be *apparent* in C if there exists a row variable γ in $\mathrm{fv}(C)$ such that $\varepsilon \notin \mathrm{dom}(\gamma)$. It is easy to check that reduction preserves the set of apparent row labels, i.e. it does not cause new labels to appear.

We extend h_C to elementary constraints that involve types: let $h_C(\tau_1 \leq \tau_2) = \max(h_C(\tau_1), h_C(\tau_2))$ and $h_C(\tau \blacktriangleleft *) = h_C(* \triangleleft \tau) = h_C(\tau)$. The *weight* of a row variable γ is the pair $(\mathrm{dom}(\gamma), 1)$. The weight of a constraint $\rho_1 \leq \rho_2$ is the pair $(\mathrm{dom}(\rho_1), 0)$. (Because the constraint is well-sorted, $\mathrm{dom}(\rho_1) = \mathrm{dom}(\rho_2)$ must hold.) Lastly, let us say that an elementary constraint c is *active* in C if and only if either (i) there exists C' such that $c \vDash C'$ and $C' \notin C$ or (ii) $c = (\varepsilon_1 : \lambda_1; \rho_1) \leq (\varepsilon_2 : \lambda_2; \rho_2)$ and $\nexists \rho \quad \rho_1 \leq (\varepsilon_2 : \lambda_2; \rho) \wedge (\varepsilon_1 : \lambda_1; \rho) \leq \rho_2 \in C$.

Let us now measure a constraint according to the following quantities, ordered lexicographically: (1) the multiset of the heights of its type variables; (2) the multiset of the heights of its active elementary constraints that involve types; (3) the multiset of the weights of its row variables and of its active elementary constraints that involve rows; (4) the number of paths $\partial\lambda_1 \leq \gamma_1 \leq \cdots \leq \gamma_n \leq \partial\lambda_2$ which appear in it, while $\lambda_1 \leq \lambda_2$ does not; (5) 1 if **false** appears in the constraint, 0 otherwise. Given that heights are finite and the set of apparent exception names is fixed, the ordering on measures has no infinite decreasing chain. We claim that this measure decreases through every reduction step. Indeed, all expansion rules but the last one eliminate a type variable, while introducing fresh type variables of lesser height, so they decrease (1). The rules in *Types* and *Guards* (Figure 11) remove an active constraint that involves types, while possibly introducing constraints on types of lesser heights, on rows or on levels, so they decrease (2). Similarly, the first three rules in *Rows*, as well as the second and last rules in Figure 12, decrease (3). The last rule in *Rows* decreases (4). Lastly, the error rules decrease (5). It follows that $\twoheadrightarrow$ terminates. □

Checking for the presence of **false** provides a complete satisfiability check for constraints that are normal forms with respect to $\twoheadrightarrow$.

LEMMA 21 (COMPLETENESS). *Let C be a normal form with respect to $\twoheadrightarrow$ which passes the occur check. C is satisfiable if and only if C does not contain* **false**.

PROOF. Clearly, if C contains **false** then C is not satisfiable. Conversely, assume C passes the occur check test and does not contain **false**.

Let us define a strict ordering $\prec$ on elementary constraints as the smallest transitive relation such that (i) $\tau_1' \prec \tau_i$ and $\tau_2' \prec \tau_j$ and $\{i, j\} = \{1, 2\}$ imply $(\tau_1' \leq \tau_2') \prec (\tau_1 \leq \tau_2)$, (ii) $\tau' \prec \tau$ implies $(\tau' \blacktriangleleft \lambda) \prec (\tau \blacktriangleleft \lambda)$ and $(\lambda \triangleleft \tau') \prec (\lambda \triangleleft \tau)$, (iii) $\mathrm{dom}(\rho_1') \subset \mathrm{dom}(\rho_1)$ and $\mathrm{dom}(\rho_2') \subset \mathrm{dom}(\rho_2)$ imply $(\rho_1' \leq \rho_2') \prec (\rho_1 \leq \rho_2)$, (iv) if c' involves rows and c involves types, then $c' \prec c$, and (v) if c' involves labels and c involves rows, then $c' \prec c$. Again, if heights are finite and the set of apparent exception names is fixed, this ordering has no infinite decreasing chain.

Define the assignment ϕ as follows:

$$\begin{aligned}\phi(\delta) &= \sqcup \{\ell \mid \ell \leq \cdots \leq \delta \in C\} \\ \phi(\gamma) &= \partial(\sqcup \{\phi(\lambda) \mid \partial\lambda \leq \cdots \leq \gamma \in C\}) \\ \phi(\beta) &= \mathsf{unit}\end{aligned}$$

We will now prove that $\phi \vdash C$; that is, every elementary constraint c such that $c \in C$ is satisfied by ϕ. The proof is by well-founded induction on $\prec$. The meta-variable $\dot{\tau}$ (resp. $\dot{\rho}$) denotes a non-variable type (resp. row).

∘ *Case* $c = \dot{\tau}_1 \leq \dot{\tau}_2$. Because C does not contain **false** and because it is a normal form, the types $\dot{\tau}_1$ and $\dot{\tau}_2$ must have the same head constructor. By inspection of the first group of rules in Figure 11, c is entailed by a number of elementary constraints, each of which appears in C and is less than c with respect to $\prec$. The result follows by the induction hypothesis.

∘ *Cases* $c = \dot{\tau} \blacktriangleleft \lambda$, $c = \lambda \triangleleft \dot{\tau}$, $c = \dot{\rho}_1 \leq \dot{\rho}_2$. Similar to the previous case.

∘ *Cases* $c = \dot{\tau} \leq \beta$, $c = \beta \leq \dot{\tau}$, $c = (\varepsilon : \lambda; \rho) \leq \gamma$ and $c = \gamma \leq (\varepsilon : \lambda; \rho)$. Because none of the expansion rules apply, these cases cannot arise.

∘ *Case* $c = \beta \leq \beta'$. Immediate.

∘ *Cases* $c = \beta \blacktriangleleft \lambda$ and $c = \lambda \triangleleft \beta$. Because $\phi(\beta) = \mathsf{unit}$, $\phi \vdash c$ holds.

∘ *Case* $c = \partial\lambda_1 \leq \partial\lambda_2$. Because C is a normal form, $\lambda_1 \leq \lambda_2 \notin C$ must hold. By induction hypothesis, $\phi(\lambda_1) \leq \phi(\lambda_2)$ follows. This yields $\phi(\partial\lambda_1) \leq \phi(\partial\lambda_2)$.

∘ *Case* $c = \partial\lambda \leq \gamma$. By construction, $\partial(\phi(\lambda)) \leq \phi(\gamma)$ holds.

∘ *Case* $c = \gamma \leq \partial\lambda$. If $\partial\lambda' \leq \cdots \leq \gamma \in C$, then, by transitivity, $\partial\lambda' \leq \cdots \leq \partial\lambda \in C$ holds as well. Because C is a normal form, $\lambda' \leq \lambda \in C$ must hold. By induction hypothesis, $\phi(\lambda') \leq \phi(\lambda)$ follows. As a result, we have $\phi(\gamma) \leq \partial(\phi(\lambda))$.

∘ *Case* $c = \gamma \leq \gamma'$. By transitivity, $\partial\lambda \leq \cdots \leq \gamma \in C$ implies $\partial\lambda \leq \cdots \leq \gamma' \in C$. It follows that $\{\partial\lambda \mid \partial\lambda \leq \cdots \leq \gamma \in C\} \subseteq \{\partial\lambda \mid \partial\lambda \leq \cdots \leq \gamma' \in C\}$. This yields $\phi(\gamma) \leq \phi(\gamma')$.

∘ *Case* $c = \ell_1 \leq \ell_2$. Because C does not contain **false** and because it is a normal form, $\ell_1 \leq \ell_2$ must hold in $\mathcal{L}$. $\phi \vdash c$ follows.

∘ *Case* $c = \ell \leq \delta$. By construction, $\ell \leq \phi(\delta)$ holds.

∘ *Case* $c = \delta \leq \ell$. If $\ell' \leq \cdots \leq \delta \in C$, then, by transitivity, $\ell' \leq \cdots \leq \ell \in C$ holds as well. Because C does not contain **false** and because it is a normal form, $\ell' \leq \ell$ must hold. As a result, we have $\sqcup\{\ell' \mid \ell' \leq \cdots \leq \delta \in C\} \leq \ell$, that is, $\phi(\delta) \leq \ell$.

∘ *Case* $c = \delta \leq \delta'$. By transitivity, $\ell \leq \cdots \leq \delta \in C$ implies $\ell \leq \cdots \leq \delta' \in C$. It follows that $\{\ell \mid \ell \leq \cdots \leq \delta \in C\} \subseteq \{\ell \mid \ell \leq \cdots \leq \delta' \in C\}$. This yields $\phi(\delta) \leq \phi(\delta')$. □

The results developed in this section may be summarized as follows.

THEOREM 22. *Constraint solving is decidable.*

PROOF. Given a constraint C, determine whether it satisfies the occur check. If it doesn't, report C is unsatisfiable. If it does, normalize it with respect to $\twoheadrightarrow$, yielding C'. If C' contains **false**, report C is unsatisfiable; otherwise, report it

is satisfiable. Lemma 20 ensures that the algorithm terminates. Lemmas 18, 19 and 21 guarantee that it is correct. □

9. EXAMPLES

We intend to integrate MLIF into a realistic programming language, such as Caml-Light [Leroy et al. 1997]. 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 $\rightarrow$ when they are unconstrained, anonymous type variables. Because none of 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 ('b, 'd) list = <'d>
  | []
  | (::) of 'b * ('b, 'd) list
```

In $\beta\,\mathsf{list}^{\delta}$, the parameter β is the type of the list's elements, as usual, while δ is a security level. The annotation `<'d>` on the right-hand side is meant to indicate that δ 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 $\forall[\delta \leq \delta'].\ * \ \mathsf{list}^{\delta} \rightarrow \mathsf{int}^{\delta'}$. As expected, the result's security annotation δ' does not depend on the type of the list's elements. The constraint $\delta \leq \delta'$ 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 $\forall[].\ * \ \mathsf{list}^{\delta} \rightarrow \mathsf{int}^{\delta}$, 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

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

Here, γ represents `f`'s effect. Because `iter` does not raise any exceptions of its own, γ is also `iter`'s effect. δ 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 $\forall[].\mathsf{int}^{\delta}\ \mathsf{ref}^{\delta} \xrightarrow{\delta\ [*]} \mathsf{unit}$ 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 $\forall[].*\ \mathsf{list}^{\delta} \xrightarrow{\delta\ [*]} \mathsf{int}^{\delta}$. 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:

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

The constraint $\beta \blacktriangleleft \delta$, which arises due to the use of polymorphic equality, specifies that δ 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:

$$\forall[\beta \blacktriangleleft \delta, \delta \lhd \beta', \delta \leq \delta'].\beta \to (\beta \times \beta')\ \mathsf{list}^{\delta} \xrightarrow{\delta'\ [\mathsf{Not_found}:\ \delta';\ *]} \beta'$$

Here, as in `mem_assoc`, δ 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 $\delta \lhd \beta'$ and $\delta \leq \delta'$.

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:

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

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

10. DISCUSSION

10.1 On exceptions

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' [Myers 1999a; 1999b] sets of path labels X, on the other hand, record the security level associated with normal termination under a special label $\underline{\mathsf{n}}$, which is then used in the sequential composition rule. It is, however, typically an upper bound for the value of *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 $X[\underline{\mathsf{n}}]$ 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 r_1 is $\partial\bot$, then $\sqcup r_1$ is $\bot$, 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 a distinction between $\underline{\emptyset}$ and $\bot$ (i.e. between expressions that do not raise exceptions and expressions that raise only low-security exceptions), which we have dropped, for simplicity. More importantly, it requires counting the number of non-$\underline{\emptyset}$ entries in a row; in the presence of row variables, this requires heavy constraint forms, which is why we avoid it. This difficulty does not arise in Myers' framework because he relies on Java's explicit, monomorphic `throws` clauses.

There exists a simple monadic encoding of exceptions into sums [Moggi 1989; Wadler 1992]. 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, we have found 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). For more details, the reader is referred to a recent paper by the second author [Simonet 2002].

10.2 Variations

In this paper, the type `unit` carries no security annotation, which is natural, and fits well with the constraint solving algorithm proposed in Section 8.5, because `unit` can be proposed as a solution for β in every constraint of the form $\lambda \lhd \beta$ or $\beta \blacktriangleleft \lambda$

$$\textsc{v-Abs}\quad \frac{pc, \Gamma[x \mapsto t'][f \mapsto t' \xrightarrow{pc\ [r]} t], M \vdash e : t\ [r]}{\Gamma, M \vdash \mathsf{fix}\, f.\lambda x.e : t' \xrightarrow{pc\ [r]} t} \qquad \textsc{e-App}\quad \frac{\Gamma, M \vdash v_1 : t' \xrightarrow{pc'\ [r]} t \quad \Gamma, M \vdash v_2 : t' \quad pc \leq pc'}{pc, \Gamma, M \vdash v_1\, v_2 : t\ [r]}$$

Fig. 13. Modifications for an invariant pc parameter

(see the proof of Lemma 21). However, as a consequence of this fact, it is *not* the case that $\ell \lhd t$ and $t \blacktriangleleft \ell'$ imply $\ell \leq \ell'$. We have found that the lack of such a law makes constraint simplification more intricate and less effective. This might be sufficient motivation to switch back to a type of the form unit^ℓ. (Product types would remain unannotated.) Then, every type t would contain at least one security annotation, restoring the above law.

As in previous work [Heintze and Riecke 1998], our arrow types carry two annotations pc and ℓ, which are respectively contravariant and covariant. These annotations are independent. Yet, the first premise of rule E-App (Figure 6), together with the subtyping rules, show that a function cannot be applied unless $\ell \leq pc$ holds. So, the current type system makes it possible to create functions that are not applicable—a rather undesirable feature. To eliminate this problem, one might wish to merge the annotations pc and ℓ, that is, to use arrow types of the form $t \xrightarrow{pc\ [r]} t'$, where pc is *invariant*. Then, by definition, $\ell \lhd * \xrightarrow{pc\ [*]} t'$ would be equivalent to $\ell \leq pc \wedge \ell \lhd t'$. The typing rules for abstraction and application would be modified as described in Figure 13. In addition to earlier detection of type errors, this modification would perhaps help infer more readable types. We have experimented with this idea, however, and have run into trouble with recursive definitions: the types inferred for some recursive functions become less precise, unless polymorphic recursion [Mycroft 1984] is added to the type system.

10.3 Future work

Our main direction for future work is to create a full implementation of the system on top of a fragment of Objective Caml, and to assess its usability through a number of case studies. We also intend to publish a more detailed account of our constraint resolution and simplification techniques. Lastly, the fact that certain distinct types appear to have the "same" meaning, illustrated in Section 9 by comparing the types ascribed to `length` and `length'`, would deserve deeper study.

REFERENCES

Abadi, M., Banerjee, A., Heintze, N., and Riecke, J. G. 1999. A core calculus of dependency. In *Conference Record of the 26th ACM Symposium on Principles of Programming Languages*. ACM Press, San Antonio, Texas, 147–160. URL: `http://www.soe.ucsc.edu/~abadi/Papers/flowpopl.ps`.

Abadi, M., Lampson, B., and Lévy, J.-J. 1996. Analysis and caching of dependencies. In *Proceedings of the 1996 ACM SIGPLAN International Conference on Functional Programming*. ACM Press, Philadelphia, Pennsylvania, 83–91. URL: `http://www.soe.ucsc.edu/~abadi/Papers/make-preprint.ps`.

Banerjee, A. and Naumann, D. 2002. Secure information flow and pointer confinement in a Java-like language. In *Proceedings of the 15th IEEE Computer Security Foundations Workshop*

(CSFW 15). Cape Breton, Nova Scotia, 253–267. URL: http://www.cs.stevens-tech.edu/~naumann/csfw15.ps.

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

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

Fähndrich, M. 1999. Bane: A library for scalable constraint-based program analysis. Ph.D. thesis, University of California at Berkeley. URL: http://research.microsoft.com/~maf/diss.ps.

Field, J. and Teitelbaum, T. 1990. Incremental reduction in the lambda calculus. In *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*. ACM Press, 307–322.

Flanagan, C., Sabry, A., Duba, B. F., and Felleisen, M. 1993. The essence of compiling with continuations. In *Proceedings of the ACM SIGPLAN'93 Conference on Programming Language Design and Implementation*. ACM Press, Albuquerque, New Mexico, 237–247. URL: http://www.cs.rice.edu/CS/PLT/Publications/pldi93-fsdf.ps.gz.

Goguen, J. and Meseguer, J. 1982. Security policies and security models. In *Proceedings of the 1982 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, Oakland, California, 11–20.

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

Leroy, X., Doligez, D., et al. 1997. The Caml Light system, release 0.74. URL: http://caml.inria.fr/.

Leroy, X., Doligez, D., Garrigue, J., Rémy, D., and Vouillon, J. 2002. The Objective Caml system, release 3.06. URL: http://caml.inria.fr/.

Milner, R., Tofte, M., Harper, R., and MacQueen, D. 1997. *The Definition of Standard ML (Revised)*. The MIT Press.

Moggi, E. 1989. An abstract view of programming languages. Tech. Rep. ECS-LFCS-90-113, University of Edinburgh. June. URL: http://www.disi.unige.it/person/MoggiE/ftp/abs-view.ps.gz.

Mycroft, A. 1984. Polymorphic type schemes and recursive definitions. In *Proceedings of the 6th International Symposium on Programming*, M. Paul and B. Robinet, Eds. Lecture Notes in Computer Science, vol. 167. Toulouse, France, 217–228.

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

Myers, A. C. 1999b. Mostly-static decentralized information flow control. Ph.D. thesis, Massachusetts Institute of Technology. Technical Report MIT/LCS/TR-783. URL: http://www.cs.cornell.edu/andru/release/tr783.ps.gz.

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

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

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

Pottier, F. 2002. A simple view of type-secure information flow in the π-calculus. In *Proceedings of the 15th IEEE Computer Security Foundations Workshop*. Cape Breton, Nova Scotia, 320–330. URL: http://pauillac.inria.fr/~fpottier/publis/fpottier-csfw15.ps.gz.

Pottier, F. and Conchon, S. 2000. Information flow inference for free. In *Proceedings of the the 5th ACM SIGPLAN International Conference on Functional Programming (ICFP'00)*. ACM Press, Montréal, Canada, 46–57. URL: http://pauillac.inria.fr/~fpottier/publis/fpottier-conchon-icfp00.ps.gz.

Pottier, F. and Simonet, V. 2002a. Information flow inference for ML. In *Proceedings of the 29th ACM Symposium on Principles of Programming Languages (POPL'02)*. ACM Press, Portland, Oregon, 319–330. URL: http://pauillac.inria.fr/~fpottier/publis/fpottier-simonet-popl02.ps.gz.

Pottier, F. and Simonet, V. 2002b. Information flow inference for ML. Full version. URL: http://pauillac.inria.fr/~fpottier/publis/fpottier-simonet-popl02-long.ps.gz.

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

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

Simonet, V. 2002. Fine-grained information flow analysis for a λ-calculus with sum types. In *Proceedings of the 15th IEEE Computer Security Foundations Workshop (CSFW 15)*. Cape Breton, Nova Scotia, 223–237. URL: http://cristal.inria.fr/~simonet/publis/simonet-csfw-02.ps.gz.

Sulzmann, M. 2000. Completenss of constraint-based inference. URL: http://www.cs.mu.oz.au/~sulzmann/publications/constraint-inference.ps.

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

Volpano, D. and Smith, G. 1997a. Eliminating covert flows with minimum typings. In *10th IEEE Computer Security Foundations Workshop*. Rockport, MA, 156–168. URL: http://www.cs.nps.navy.mil/people/faculty/volpano/papers/csfw97.ps.Z.

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

Volpano, D., Smith, G., and Irvine, C. 1996. A sound type system for secure flow analysis. *Journal of Computer Security 4,* 3, 167–187. URL: http://www.cs.nps.navy.mil/people/faculty/volpano/papers/jcs96.ps.Z.

Wadler, P. 1992. Comprehending monads. *Mathematical Structures in Computer Science 2*, 461–493. URL: http://www.research.avayalabs.com/user/wadler/papers/monads/monads.ps.gz.

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

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

Zdancewic, S. and Myers, A. C. 2001. Secure information flow and CPS. In *Proceedings of the 2001 European Symposium on Programming (ESOP'01)*, D. Sands, Ed. Lecture Notes in Computer Science. Springer Verlag, Genova, Italy. URL: http://www.cs.cornell.edu/zdance/lincont.ps.

Zdancewic, S. and Myers, A. C. 2002. Secure information flow via linear continuations. *Higher Order and Symbolic Computation.* To appear. URL: http://www.cs.cornell.edu/andru/papers/hosc01.ps.gz.

Document compiled August 22, 2002.