Fine-grained Information Flow Analysis for a λ-calculus with Sum Types

Vincent Simonet

INRIA Rocquencourt
E-mail: Vincent.Simonet@inria.fr

Abstract

This paper presents a new type system tracing information flow for a λ-calculus equipped with polymorphic "let" and with sums (a.k.a. union types or polymorphic variants). The type system allows establishing (weak) non-interference properties. Thanks to original forms of security annotations and constraints, it is more accurate than existing analyses. Through a straightforward encoding into sums, this work also provides a new type-based information flow analysis for programming languages featuring exceptions. From these systems, one may derive constraint-based formulations, in the style of HM(X), which have decidable type inference.

1 Introduction

Information flow analysis is concerned with statically determining the dependencies between the inputs and the outputs of a program. It allows establishing instances of a *non-interference* property that may address *secrecy* and *integrity* issues.

Although the first pieces of work in this area appear in the late 70's [2], such an analysis has been formulated as a type system only in the past few years [8, 18, 3, 12]. Types seem to be most suitable for static analysis of information flow. Practically speaking, they may serve as a specification langage, offer automated verification of code — provided type inference is available — and have no run-time cost. From a theoretical point of view, they can express non-interference results in simple and precise ways.

In these systems, types are usually annotated with *security levels* chosen in a suitable lattice. Each annotation gives an approximation of the information which the expression that it describes may convey. Thus type constructors for (expressions producing) *base values* — e.g. integers or enumerated constants — carry *one* security level representing all of the information attached to the value. Such an approximation may be too restrictive in some cases. Indeed, consider the program fragment

$$\begin{aligned}\mathsf{let}\ t = \mathsf{if}\ x\ &\mathsf{then}\ (\mathsf{if}\ y\ \mathsf{then}\ A\ \mathsf{else}\ B)\\ &\mathsf{else}\ \ (\mathsf{if}\ z\ \mathsf{then}\ A\ \mathsf{else}\ D)\end{aligned}$$

(Here, the language is equipped with three constants A, B, D belonging to the same datatype.) t's value directly depends on the booleans x, y and z. Previous type systems will record this potential information flow by constraining the security level attached to t to exceed those of x, y and z. As a result, they will similarly constrain the level attached to the integer u produced by

$$\mathsf{let}\ u = t\ \mathsf{case}\ [A, B \mapsto 1 \mid D \mapsto 0]$$

Yet, u does not depend on y because testing whether t's value is D rather than A or B cannot leak any information about y.

Our work is a proposal for a more sophisticated analysis (concerning information flow) of sum types. In the previous example, we will attach a (triangular) matrix of three security levels $q(A \cdot B)$, $q(B \cdot D)$ and $q(A \cdot D)$ to the identifier t: one for each (unordered) pair of constructors. Thus $q(A \cdot B)$ describes how much information one may leak by testing whether t's value is A rather than B. It must therefore be at least the union of x and y's levels. Similarly, $q(A \cdot D)$ must be the union of x and z's levels and $q(B \cdot D)$ must be x's level or greater. In the previous example, because the test allows to determine whether t's value is D rather than A or B, our system will approximate the possible information leak by $q(A \cdot D) \sqcup q(B \cdot D)$, i.e. the union of x and z's levels. Therefore, it will be able to establish the absence of dependency between y and u.

Recent studies in the area of information flow analysis concern realistic programming languages providing an exception mechanism — such as Java [5] or ML [13]. The treatment of exceptions in these systems seems relatively *ad hoc* and is not perfectly well understood: although there exists a simple monadic encoding of exceptions into sums [4, 19], these systems address exceptions in a direct manner. Indeed, both [5] and [13] try to achieve a better precision in the analysis of information flow due to exceptions than existing systems [3] dealing with sums provide.

Because we describe a very accurate type system for sums, we are able to obtain a suitable analysis for exceptions by a simple translation. There are two reasons why this approach is interesting. First, it describes a new analysis of information flow due to exceptions that is more accurate than existing ones. Second, because it may encode existing systems, it allows a better understanding of their design.

2 Overview

In the current paper, we follow a similar approach to that developed in [13] for Core ML. Section 3 introduces the language λ_+ and a technical extension, called λ_+^2, that allows us to reason about the simultaneous reduction of two expressions. We present in section 4 the type system and state the non-interference theorem. Then we address the question of type inference in section 5 and present a small set of examples in section 6. Section 7 explains how it is possible to obtain from the type system for sums another one for exceptions by a simple translation of the latter in the former. Lastly, section 8 proposes some restrictions of these systems and relates them to existing work.

By lack of space, some proofs are omitted; they can be found in the full version of this paper [16].

3 λ-calculus with sums

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

3.1 The λ_+-calculus

Let k range over integers; let x, c range over two disjoint, denumerable sets of *program variables* and *constructor names*, respectively. We denote by $\mathcal{C}$ the set of all constructor names and let C range over subsets of $\mathcal{C}$. (In the examples, we will assume $\{A, B, D\} \subseteq \mathcal{C}$.) Then, *expressions* and *handlers* are defined as follows:

$$
\begin{array}{lcll}
e & ::= & & \textit{expression} \\
 & \mid & k & \text{(integer constant)} \\
 & \mid & x & \text{(program variable)} \\
 & \mid & \lambda x.e & \text{(abstraction)} \\
 & \mid & e\,e & \text{(application)} \\
 & \mid & \mathsf{let}\ x = e\ \mathsf{in}\ e & \text{(local definition)} \\
 & \mid & (e, e) & \text{(pair construction)} \\
 & \mid & \pi_j\, e & \text{(pair projection, } j \in \{1, 2\}) \\
 & \mid & c\,e & \text{(sum construction)} \\
 & \mid & \bar{c}\,e & \text{(sum destruction)} \\
 & \mid & e\ \mathsf{case}\ [h \mid \ldots \mid h] & \text{(sum case)} \\
h & ::= & C : x \mapsto e & \textit{handler}
\end{array}
$$

Our language includes the λ-calculus with pairs and a "let" binding in the style of ML. It is extended with sum expressions built by the construction $c\,e$. For each constructor name c, λ_+ provides a destructor $\bar{c}$ such that $\bar{c}\,(c\,e)$ evaluates to e. A handler is a triple of a subset of $\mathcal{C}$, a program variable and an expression. The case construction allows to match an expression's head constructor against a list of handlers: $c\,e$ case $[C_1 : x_1 \mapsto e_1 \mid \ldots \mid C_n : x_n \mapsto e_n]$ reduces to $e_j[x_j \Leftarrow c\,e]$ if $c \in C_j$. (In the following, we will assume that case expressions are *deterministic*, i.e. that the C_j are pairwise disjoint.) The conjunction of projections and case clauses gives us the expressiveness of *à la ML* pattern matching on data constructors. We still provide specific integer constants, although they may be considered as a special case of sums, because they help state non-interference theorems in a concise manner. Lastly, we do not provide any construction allowing recursion; but this can be achieved straightforwardly, for instance by introducing an extra parameter f in the λ construct as in [13].

3.2 The λ_+^2 calculus

Because establishing a non-interference result requires reasoning about two expressions and exhibiting a bisimulation between their executions, we introduce a technical extension of λ_+. It allows to deal simultaneously with two expressions that share some sub-terms throughout a reduction.

This extension, called λ_+^2, is similar to Core ML^2 in [13]. It is as follows:

$$e \quad ::= \quad \ldots \mid \langle e \mid e \rangle$$

We do not allow nesting $\langle \cdot \mid \cdot \rangle$ constructs. A λ_+^2 term represents a pair of λ_+ terms. For instance, the λ_+^2 expression $\langle e_1 \mid e_2 \rangle$ encodes the pair (e_1, e_2). Because brackets can appear at an arbitrary depth within a λ_+^2 term, the encoding allows to keep track of sharing: assuming e, e_1 and e_2 are λ_+ expressions, both $\langle e_1 \mid e_2 \rangle\, e$ and $\langle e_1\, e \mid e_2\, e \rangle$ encode the pair $(e_1\, e, e_2\, e)$, but the former explicitly records the fact that the argument e is shared.

In order to relate a λ_+^2 term to the pair of λ_+ expressions which it encodes, we define two *projections*, $\lfloor \cdot \rfloor_1$ and $\lfloor \cdot \rfloor_2$. They are homomorphisms except at $\langle \cdot \mid \cdot \rangle$ nodes: $\lfloor \langle e_1 \mid e_2 \rangle \rfloor_i = e_i$. We extend them to handlers by $\lfloor C : x \mapsto e \rfloor_i = C : x \mapsto \lfloor e \rfloor_i$ and pointwise to lists of handlers $\vec{h}$.

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 $\langle e_1[x \Leftarrow \lfloor v \rfloor_1] \mid e_2[x \Leftarrow \lfloor v \rfloor_2] \rangle$.

$$
\begin{array}{llclll}
\textbf{Basic reductions} & (\lambda x.e_1)\, e_2 & \rightarrow & e_1[x \Leftarrow e_2] & & (\beta) \\
& \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 & \rightarrow & e_2[x \Leftarrow e_1] & & (\text{let}) \\
& \pi_j\,(e_1, e_2) & \rightarrow & e_j & & (\text{proj}) \\
& \bar{c}\,(c\, e) & \rightarrow & e & & (\text{destr}) \\
& e\ \mathsf{case}\ [C_1 : x_1 \mapsto e_1 \mid \ldots \mid C_n : x_n \mapsto e_n] & \rightarrow & e_j[x_j \Leftarrow e] & \text{if } e \downarrow C_j & (\text{case}) \\
& E[e] & \rightarrow & E[e'] & \text{if } e \rightarrow e' & (\text{context}) \\
& & & & & \\
\textbf{Lifting rules} & \langle e_1 \mid e_2 \rangle\, e & \rightarrow & \langle e_1 \lfloor e \rfloor_1 \mid e_2 \lfloor e \rfloor_2 \rangle & & (\text{lift-}\beta) \\
& \pi_j \langle e_1 \mid e_2 \rangle & \rightarrow & \langle \pi_j\, e_1 \mid \pi_j\, e_2 \rangle & & (\text{lift-proj}) \\
& \bar{c} \langle e_1 \mid e_2 \rangle & \rightarrow & \langle \bar{c}\, e_1 \mid \bar{c}\, e_2 \rangle & & (\text{lift-destr}) \\
& \langle e_1 \mid e_2 \rangle\ \mathsf{case}\ \vec{h} & \rightarrow & \langle e_1\ \mathsf{case}\ \lfloor \vec{h} \rfloor_1 \mid e_2\ \mathsf{case}\ \lfloor \vec{h} \rfloor_2 \rangle & \text{if } \langle e_1 \mid e_2 \rangle \curlywedge \vec{h} & (\text{lift-case})
\end{array}
$$

Figure 1. Semantics of λ_+^2

3.3 Semantics

The small step operational semantics of λ_+^2 is given in figure 1. The semantics of λ_+ is obtained as a fragment of that of λ_+^2. To clarify the presentation, we divide the set of reduction rules into two groups. *Basic reductions* are those of λ_+. They perform computation. When read as λ_+^2 reduction rules, they may be applied outside brackets (in this case, the two projections perfom the same reduction step) or within a bracket context (then one of the two projections remains unchanged). *Lifting rules* are specific to λ_+^2 and deal with sharing. They have no computational: they leave both projections unchanged. Their purpose is only to prevent $\langle \cdot \mid \cdot \rangle$ constructs from blocking reduction by lifting them up (and thus duplicating some sub-terms).

We choose a *call-by-name* semantics (we will deal with *call-by-value* in section 4.6). Thus, *evaluation contexts* are defined as follows:

$$
\begin{array}{lll}
E & ::= & [\,]\, e \mid \pi_j\, [\,] \mid \bar{c}\, [\,] \mid [\,]\ \mathsf{case}\ [h \mid \ldots \mid h] \\
& \mid & \langle [\,] \mid e \rangle \mid \langle e \mid [\,] \rangle
\end{array}
$$

(The two last forms of context are specific to λ_+^2, allowing to apply basic reduction rules within a bracket construct.)

The rule (case) defines the semantics of case clauses. We write $e \downarrow C$ (read: e *matches* C) if and only if $e = c\, e'$ for some $c \in C$ or $e = \langle c_1\, e'_1 \mid c_2\, e'_2 \rangle$ for some $c_1 \in C$ and $c_2 \in C$. For instance, by application of the (case) rule, the expression $c\, e'\ \mathsf{case}\ [C_1 : x_1 \mapsto e_1 \mid \ldots \mid C_n : x_n \mapsto e_n]$ reduces to $e_j[x_j \Leftarrow c\, e']$ if $c \in C_j$. Our semantics of case keeps good track of sharing since values like $\langle c_1\, e'_1 \mid c_2\, e'_2 \rangle$ may be matched without any lifting if the two head constructors fall within the same C_j. If they do not, (lift-case) must be applied before reducing the two resulting case sub-expressions separately: $\langle e_1 \mid e_2 \rangle \curlywedge [C_1 : * \mapsto * \mid \ldots \mid C_n : * \mapsto *]$ holds if and only if there exist distinct indices j_1 and j_2 such that $e_1 \downarrow C_{j_1}$ and $e_2 \downarrow C_{j_2}$.

3.4 Relating λ_+^2 to λ_+

An expression e is a *normal form* if it is irreducible. We denote by the meta-variable n any expression which is a normal form. The following lemma relates λ_+^2 normal forms to those of λ_+.

Lemma 3.1 (Normal forms) *If e is a normal form then $\lfloor e \rfloor_1$ or $\lfloor e \rfloor_2$ is a normal form.*

Proof. By induction on the structure of e.

○ *Cases* $e = k$, $e = x$, $e = \lambda x.e'$, $e = c\, e$. $\lfloor e \rfloor_1$ and $\lfloor e \rfloor_2$ are both normal forms.

○ *Case* $e = e_1\, e_2$. Because (context) is not applicable, e_1 is a normal form. By induction hypothesis, $\lfloor e_1 \rfloor_i$ is a normal form for some $i \in \{1, 2\}$. Because neither (β) nor (lift-β) is applicable, e_1 cannot be of the form $\langle * \mid * \rangle$ or $\lambda *.*$. It follows that $\lfloor e_1 \rfloor_i$ is not a λ-abstraction. We conclude that $\lfloor e \rfloor_i$ is a normal form.

○ *Cases* $e = \pi_j\, e'$, $e = \bar{c}\, e'$. Similar to the previous case.

○ *Case* $e = \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2$. e is not a normal form.

○ *Case* $e = e'\ \mathsf{case}\ \vec{h}$. Because neither (case) nor (lift-case) is applicable, $e' \not\downarrow C_j$, for all j, and $e' \not\curlywedge \vec{h}$.

If $e' = \langle * \mid * \rangle$ it follows that $\lfloor e' \rfloor_i$ is not of the form $c\, *$ with $c \in C_1 \cup \ldots \cup C_n$ for some $i \in \{1, 2\}$. Because (context) is not applicable, both $\lfloor e' \rfloor_1$ and $\lfloor e' \rfloor_2$ are normal forms. We conclude that $\lfloor e \rfloor_i$ is a normal form.

Otherwise, $\lfloor e' \rfloor_i$ is not of the form $c\, *$ with $c \in C_1 \cup \ldots \cup C_n$ for all $i \in \{1, 2\}$. Because (context) is not applicable, e' is a normal form. Then, by induction hypothesis, $\lfloor e' \rfloor_i$ is a normal form for some $i \in \{1, 2\}$. It follows that $\lfloor e \rfloor_i$ is a normal form.

○ *Case* $e = \langle e_1 \mid e_2 \rangle$. Because (context) is not applicable with $E = \langle [\,] \mid * \rangle$ or $\langle * \mid [\,] \rangle$, $\lfloor e \rfloor_1$ and $\lfloor e \rfloor_2$ are both normal forms. □

We now show that every λ^2_+ reduction correctly simulates the simultaneous reduction of two λ_+ expressions. This is expressed by two lemmas of *soundness* and *correctness*. The former states that the projections of a λ^2_+ reduction are valid λ_+ reductions. The latter ensures that if both projections of an expression can produce a normal form then so can the expression. The proof techniques are almost identical to those used in [13].

Lemma 3.2 (Soundness) *Let $i \in \{1, 2\}$. If $e \rightarrow e'$, then $\lfloor e \rfloor_i \rightarrow^= \lfloor e' \rfloor_i$.*

Proof. By inspection of the reduction rules. □

Lemma 3.3 (Completeness) *Assume $\lfloor e \rfloor_i \rightarrow^\star n_i$ for all $i \in \{1, 2\}$. Then, there exists n such that $e \rightarrow^\star n$.*

Proof. Because each of the "lift" reduction rules moves some $\langle \cdot \mid \cdot \rangle$ constructor strictly closer to the term's root, no infinite reduction sequence can consist exclusively of instances of these rules. As a consequence, if e admits an infinite reduction sequence, then lemma 3.2 yields an infinite one out of $\lfloor e \rfloor_i$ for some $i \in \{1, 2\}$. However, this is impossible, because both $\lfloor e \rfloor_1$ and $\lfloor e \rfloor_2$ can be reduced to normal forms, and the semantics of the λ_+ fragment is deterministic. We conclude that there exists a normal form n such that $e \rightarrow^\star n$. □

4 Typing

Given two distinct constructors c_1 and c_2, we denote by $c_1 \cdot c_2$ the (unordered) pair $\{c_1, c_2\}$. For any $C \subseteq \mathcal{C}$, let C^2 denotes the set of such pairs of elements of C.

Is this section, we present a type system tracing information flow in λ_+. This is a *ground* type system, in so far as it has no type variables. It handles polymorphism in an extensional way: a type scheme is represented by the set of its ground instances. This presentation is very amenable to proofs. Nevertheless, it does not describe a type inference algorithm: we will address this issue in section 5.

4.1 The type algebra

Let $(\mathcal{L}, \leq)$ be a lattice whose elements, denoted by ℓ, represent *security levels*. *Types*, *alternatives*, *rows* and *matrices* are respectively defined as follows:

$$\begin{array}{rcl} t & ::= & \mathsf{int}^\ell \mid t \rightarrow t \mid t \times t \mid r^q \\ a & ::= & \mathsf{Abs} \mid \mathsf{Pre}\, t \\ r & ::= & \{c \mapsto a\}_{c \in C} \\ q & ::= & \{c_1 \cdot c_2 \mapsto \ell\}_{c_1 \cdot c_2 \in C^2} \end{array}$$

A *row* (resp. a *matrix*) is an infinite, almost constant family of alternatives (resp. levels) indexed by a subset of $\mathcal{C}$ (resp. $\mathcal{C}^2$), its domain. (A family is *almost constant* if all but a finite number of its entries are equal.) It is worth noting that because pairs of $\mathcal{C}^2$ are unordered, matrices are in fact triangular. We write $(c : a; r)$ for the row whose element at index c is a and whose other elements are given by the sub-row r, which is indexed by $\mathcal{C} \backslash \{c\}$. We denote by $\partial_C a$ the row indexed by C that maps all its entries to a (the domain C may be omitted when it can be deduced from the context). $r_{|C}$ represents the row of the same domain as r which is equal to r on C and Abs elsewhere. Similarly, we write $\partial^2_C \ell$ for the matrix indexed by C^2 that maps all its entries to ℓ and $q_{|C}$ denotes the row of the same domain as q which is equal to q on C^2 and $\perp$ elsewhere. Lastly, $(r^q)_{|C}$ is a shorthand for $r_1{}^{q_1}$ where $r_1 = r_{|C}$ and $q_1 = q_{|C}$. Given a matrix q and a set of constructors C, we define $q(C) = \sqcup \{q(c \cdot c') \mid c \in C, c' \notin C\}$.

Our types are those of ML's type system (with rows for sum types [14]) decorated with security annotations that are simple levels ℓ and matrices q.

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

In many type systems tracing information flow [8, 3, 12, 21, 13], arrows carry a security level representing information about the function's identity. Nevertheless, because the only way to observe a function consists in applying it and examining its results, there is really no difference between a function whose *identity* is secret and a function that produces secret *results*. As a consequence, following [1], we do not equip the $\rightarrow$ type constructor with an external security level: all the security annotations related to a function may be carried by its result type. Similarly, all of the information carried by a tuple is in fact carried by its components. Thus products have no security annotations.

The main novelty resides in sum types, such as r^q, which consist of a row and a matrix. First, following [14], the row r indicates for every constructor $c \in \mathcal{C}$ if the given expression may ($\mathsf{Pre}\, t$) or may not (Abs) produce a value whose head constructor is c. The constructor Pre carries, in addition, the type of the constructor's argument. Second, for every pair of constructors $c_1 \cdot c_2 \in \mathcal{C}^2$, $q(c_1 \cdot c_2)$ gives an approximation of the level of information leaked by revealing that the expression produces a value whose head constructor is c_1 rather than c_2, or symmetrically c_2 rather than c_1. Note that if $r(c) = \mathsf{Abs}$, none of the levels $q(c \cdot *)$ carries in practice any relevant information. Thus, the type system would have the same expressiveness if we considered only sum types r^q such that $r(c) = \mathsf{Abs}$ implies $q(c \cdot *) = \perp$. Nevertheless, we prefer not to introduce such a constraint which may needlessly complicate the presentation.

As usual in type systems tracing information flow, we equip the algebra with a subtyping relation $\leq$ that ex-

$$\mathsf{int}^{\oplus} \qquad \ominus \to \oplus \qquad \oplus \times \oplus \qquad \oplus^{\oplus} \qquad \mathsf{Pre}\ \oplus$$

$$\mathsf{Abs} \leq \mathsf{Pre}\ * \qquad \{* \mapsto \oplus\} \qquad \{* \cdot * \mapsto \oplus\}$$

Figure 2. Subtyping

tends the ordering over information levels. This allows giving a directed view of the program's information flow graph. It is defined by the axioms in figure 2. $\oplus$ and $\ominus$ stand respectively for *covariant* and *contravariant* arguments. For instance, the axiom $\oplus^{\oplus}$ is an abbreviation of $r_1{}^{q_1} \leq r_2{}^{q_2} \Leftrightarrow r_1 \leq r_2 \wedge q_1 \leq q_2$. Similarly, the last two axioms extend $\leq$ pointwise on rows and matrices, respectively.

4.2 Guards

We introduce a two-place predicate $[\ell_1, \ldots, \ell_n] \trianglelefteq [t_1, \ldots, t_n]$ whose first argument is a (finite) list of security levels and whose second argument is a list of types of the same length. We also write $[t_1, \ldots, t_n] \leq t$ as a shorthand for $t_1 \leq t \wedge \ldots \wedge t_n \leq t$ and $\ell \trianglelefteq [t_1, \ldots, t_n]$ for $[\ell, \ldots, \ell] \trianglelefteq [t_1, \ldots, t_n]$.

In practice, we will use constraints of the form $[\ell_1, \ldots, \ell_n] \trianglelefteq [t_1, \ldots, t_n] \leq t$ to record potential information flow at a point of the program where the execution path may take one of n possible branches, depending on the result of a series of tests. The security level ℓ_j is intended to describe the information revealed by the test which guards the j^{th} branch, and t_j is the type of that branch's result. Last, t is the type of the whole expression. It must be at least the union of all t_j (i.e. $[t_1, \ldots, t_n] \leq t$) but it must also keep track of the information which the series of tests may leak. (Although the set of types is not a lattice, we define the *union* of a finite list of types $t_1, \ldots, t_n$ as the smallest type t such that $t_1 \leq t \wedge \ldots \wedge t_n \leq t$ if there exists one.)

We now comment the rules of figure 3 defining the predicate $[\ell_1, \ldots, \ell_n] \trianglelefteq [t_1, \ldots, t_n]$. Because we intend to compute the union t of $t_1, \ldots, t_n$, $\trianglelefteq$ first constrains them to have the same structure. If $t_1, \ldots, t_n$ are integer types $\mathsf{int}^{\ell'_1}, \ldots, \mathsf{int}^{\ell'_n}$, GUARDS-INT requires each integer type t_j to have security level ℓ_j or greater. Consequently, that of t will be constrained to be $\ell_1 \sqcup \ldots \sqcup \ell_n$ or greater and will record all potential information flows. Because $\to$ and $\times$ types carry no security annotation, rules GUARDS-FUN and GUARDS-PAIR propagate down the constraint on the result's type for $\to$ and the components types for $\times$. This reflects the fact, as explained in section 4.1, that all information about the identity of a function is given by its results and all information carried by a tuple is carried by its components. Lastly, GUARDS-SUM handles sum types. Its first premise propagates the constraint down. The second one constrains the matrix. If two different branches j_1 and j_2 may produce results with different head constructors, namely c_1 and c_2, then any further test that distinguishes these head constructors is liable to leak information of level $\ell_{j_1} \sqcup \ell_{j_2}$. As a result, we constrain the field $c_1 \cdot c_2$ in q_{j_1} (resp. q_{j_2}) to be ℓ_{j_1} (resp. ℓ_{j_2}) or greater. Thus, the same field in t must be greater than or equal to $\ell_{j_1} \sqcup \ell_{j_2}$.

Our guards allow keeping more precise information about flows than the simple guards of [13]. To illustrate this point, let us consider the following example:

$$\begin{array}{ll} \mathsf{let}\ t = x\ \mathsf{case}[& \{A\} : y \mapsto \mathsf{if}\ \bar{A}\,y\ \mathsf{then}\ A_1\ \mathsf{else}\ A_2 \\ & \{B\} : y \mapsto \mathsf{if}\ \bar{B}\,y\ \mathsf{then}\ B_1\ \mathsf{else}\ B_2 \\ & \{D\} : y \mapsto \mathsf{if}\ \bar{D}\,y\ \mathsf{then}\ D_1\ \mathsf{else}\ D_2] \end{array}$$

Because the guard constraint can consider the type of each branch of the case construct in isolation rather than only their union, the type system will be able to take into account the fact that, in this example, only the first (resp. second, third) branch can produce A_1 or A_2 (resp. B_1 or B_2, D_1 or D_2). As a result, the levels associated with the pairs $A_1 \cdot A_2$, $B_1 \cdot B_2$ and $D_1 \cdot D_2$ are not constrained to be greater than the security levels attached to the identifier x (i.e. $q(A \cdot B)$, $q(A \cdot D)$ and $q(B \cdot D)$ if x has type $*^q$.) Using the intermediate result t, we now compute an integer u:

$$\begin{array}{ll} \mathsf{let}\ u = t\ \mathsf{case}[& \{A_1\} : \_ \mapsto 0 \\ & \mathcal{C} \backslash \{A_1\} : \_ \mapsto 1] \end{array}$$

Therefore the security level of the integer u will in particular not be constrained to be greater than $q(B \cdot D)$, reflecting the fact that u's value carries no information about whether t's head constructor is B rather than D.

Lemma 4.1 (Subset) *If* $[\ell_1, \ldots, \ell_n] \trianglelefteq [t_1, \ldots, t_n]$ *then for all* $\{i_1, \ldots, i_m\} \subseteq \{1, \ldots, n\}$, $[\ell_{i_1}, \ldots, \ell_{i_m}] \trianglelefteq [t_{i_1}, \ldots, t_{i_m}]$ *holds.*

Lemma 4.2 (Transitivity) *If* $\ell'_1 \leq \ell_1, \ldots, \ell'_n \leq \ell_n$ *and* $[\ell_1, \ldots, \ell_n] \trianglelefteq [t_1, \ldots, t_n]$ *then* $[\ell'_1, \ldots, \ell'_n] \trianglelefteq [t_1, \ldots, t_n]$ *holds.*

Proof. By induction on the derivation of $[\ell_1, \ldots, \ell_n] \trianglelefteq [t_1, \ldots, t_n]$. □

4.3 Additional notations

A *polytype* s is a nonempty set of types. A *polytype environment* Γ is a partial mapping from program variables to polytypes. $\Gamma[x \mapsto s]$ denotes the environment which maps x to s and agrees with Γ otherwise. A type judgement $\Gamma \vdash e : t$ is a triple of a polytype environment, an expression and a type. (We also write $\Gamma \vdash e : s$ when $\Gamma \vdash e : t$ holds for all $t \in s$.)

Types

GUARDS-INT
$$\frac{\ell_1 \leq \ell'_1 \quad \cdots \quad \ell_n \leq \ell'_n}{[\ell_1, \ldots, \ell_n] \trianglelefteq [\mathsf{int}^{\ell'_1}, \ldots, \mathsf{int}^{\ell'_n}]}$$

GUARDS-FUN
$$\frac{[\ell_1, \ldots, \ell_n] \trianglelefteq [t_1, \ldots, t_n]}{[\ell_1, \ldots, \ell_n] \trianglelefteq [t'_1 \to t_1, \ldots, t'_n \to t_n]}$$

GUARDS-PAIR
$$\frac{[\ell_1, \ldots, \ell_n] \trianglelefteq [t_1, \ldots, t_n] \qquad [\ell_1, \ldots, \ell_n] \trianglelefteq [t'_1, \ldots, t'_n]}{[\ell_1, \ldots, \ell_n] \trianglelefteq [t_1 \times t'_1, \ldots, t_n \times t'_n]}$$

GUARDS-SUM
$$\frac{[\ell_1, \ldots, \ell_n] \trianglelefteq [r_1, \ldots, r_n] \qquad \forall j_1 \neq j_2, c_1 \neq c_2,\ \ r_{j_1}(c_1) \neq \mathsf{Abs} \wedge r_{j_2}(c_2) \neq \mathsf{Abs} \Rightarrow \ell_{j_1} \leq q_{j_1}(c_1 \cdot c_2) \wedge \ell_{j_2} \leq q_{j_2}(c_1 \cdot c_2)}{[\ell_1, \ldots, \ell_n] \trianglelefteq [{r_1}^{q_1}, \ldots {r_n}^{q_n}]}$$

Rows

GUARDS-ALT
$$\frac{[\ell_{i_1}, \ldots, \ell_{i_m}] \trianglelefteq [t_{i_1}, \ldots, t_{i_m}]}{[\ell_1, \ldots, \ell_n] \trianglelefteq [\mathsf{Abs}^\star, \mathsf{Pre}\, t_{i_1}, \mathsf{Abs}^\star, \ldots, \mathsf{Abs}^\star, \mathsf{Pre}\, t_{i_m}, \mathsf{Abs}^\star]}$$

GUARDS-ROW
$$\frac{\forall c \in \mathcal{C},\ \ [\ell_1, \ldots, \ell_n] \trianglelefteq [r_1(c), \ldots, r_n(c)]}{[\ell_1, \ldots, \ell_n] \trianglelefteq [r_1, \ldots, r_n]}$$

Figure 3. Guards

Given a row r and a set of constructors C, we say that r *cuts* C ($r \Cap C$) if and only if there exists $c \in C$ and a type t such that $r(c) = \mathsf{Pre}\, t$. Similarly, r *is included in* C ($r \Subset C$) if and only if every c such that $r(c) = \mathsf{Pre}\, *$ is in C.

4.4 Typing rules

Because the security lattice $(\mathcal{L}, \leq)$ is arbitrary, our proof technique requires to temporarily split security levels between *low* and *high* ones. That's the reason why, in the present section and the next one, we assume fixed H, an upward-closed subset of $\mathcal{L}$ whose elements will be considered as high security levels. Full generality will be recovered in section 4.6.

λ_+'s typing rules are given in figure 4. INT assigns a base type to integer constants, with an unconstrained security level. Because security annotations appear only on sum nodes and leaves in types, rules VAR, ABS, APP, LET, PAIR and PROJ involve no particular constraint and are identical to those of [11]. Polymorphism is allowed by rule LET: e_1 can be given a polytype s. Therefore, by VAR, each occurrence of x within e_2 can be typed with a different $t \in s$. Rule E-APP of [13] differs from APP (regardless of the annotations concerning side-effects) by an extra constraint $\ell \lhd t$, where ℓ is the security annotation of the $\to$ type constructor. Because the function's result may reveal information about the identity of the function itself, its type t must be guarded by ℓ. In this paper, potential information about the function's identity is directly propagated to its result type by $\trianglelefteq$. As a result, no extra constraint is needed in the premises of rule APP.

INJ associates to the expression $c\, e$ a row which maps c to $\mathsf{Pre}\, t$ (where t is e's type) and leaves other entries unconstrained, allowing them to be Abs. The matrix is alike unconstrained (and may be $\partial\bot$) since knowing the head constructor of the value cannot reveal any information at this point. DESTR requires $\bar{c}$'s argument to have type $(c : \mathsf{Pre}\, *; \partial\mathsf{Abs})^*$. This statically ensures that its head-constructor is c, allowing the matrix to be ignored. It would in particular be the case for a value matched by $\{c\}$ in a case expression.

Let us now consider rule CASE. Its first premise simply specifies the type of the matched expression, which is expected to be a sum type. The condition $r \Subset C_1 \cup \ldots \cup C_n$ ensures that the matching is exhaustive: a handler must be provided for each head constructor that e may exhibit. The third premise concerns handlers. The handler e_j is considered only if it is liable to be invoked (i.e. $r \Cap C_j$). It is typechecked in an environment where the type assigned to the program variable x_j is nothing but the "restriction" to C_j of the matched expression's type. The type given to x_j is therefore more precise than that of e, reflecting the success of the test guarding the handler. A security level $q(C_j) = \sqcup\{q(c \cdot c') \mid c \in C_j, c' \notin C_j\}$ is associated to each handler. It is an approximation of the information leaked by revealing whether the e's head constructor belongs to C_j (i.e. whether e_j will be executed). Then, the last premise computes the union t of all t_j guarded by these levels as described in section 4.2.

Rule BRACKET is specific to λ^2_+. It allows typing $\langle \cdot \mid \cdot \rangle$ constructs by computing the union t of the types of the two sub-expressions. Moreover, because brackets enclose secret parts of a computation, they must receive *high* type, i.e. t must be guarded by arbitrary levels chosen in H.

$$\textsc{Int}\quad \Gamma \vdash k : \mathsf{int}^{*} \qquad \textsc{Var}\quad \frac{t \in \Gamma(x)}{\Gamma \vdash x : t} \qquad \textsc{Abs}\quad \frac{\Gamma[x \mapsto t'] \vdash e : t}{\Gamma \vdash \lambda x.e : t' \to t} \qquad \textsc{App}\quad \frac{\Gamma \vdash e_1 : t' \to t \qquad \Gamma \vdash e_2 : t'}{\Gamma \vdash e_1\, e_2 : t}$$

$$\textsc{Let}\quad \frac{\Gamma \vdash e_1 : s \qquad \Gamma[x \mapsto s] \vdash e_2 : t}{\Gamma \vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : t} \qquad \textsc{Pair}\quad \frac{\Gamma \vdash e_1 : t_1 \qquad \Gamma \vdash e_2 : t_2}{\Gamma \vdash (e_1, e_2) : t_1 \times t_2} \qquad \textsc{Proj}\quad \frac{\Gamma \vdash e : t_1 \times t_2}{\Gamma \vdash \pi_j\, e : t_j} \qquad \textsc{Inj}\quad \frac{\Gamma \vdash e : t}{\Gamma \vdash c\, e : (c : \mathsf{Pre}\, t; *)^{*}}$$

$$\textsc{Destr}\quad \frac{\Gamma \vdash e : (c : \mathsf{Pre}\, t; \partial\mathsf{Abs})^{*}}{\Gamma \vdash \bar{c}\, e : t} \qquad \textsc{Case}\quad \frac{\begin{array}{c}\Gamma \vdash e : r^q \qquad r \Subset C_1 \cup \ldots \cup C_n \\ \forall j,\ \ r \Cap C_j \Rightarrow \Gamma[x_j \mapsto (r^q)_{|C_j}] \vdash e_j : t_j \qquad [q(C_1), \ldots, q(C_n)] \trianglelefteqslant [t_1, \ldots, t_n] \leq t\end{array}}{\Gamma \vdash e\ \mathsf{case}\ [C_1 : x_1 \mapsto e_1 \mid \ldots \mid C_n : x_n \mapsto e_n] : t}$$

$$\textsc{Bracket}\quad \frac{\Gamma \vdash e_1 : t_1 \qquad \Gamma \vdash e_2 : t_2 \qquad \ell_1, \ell_2 \in H \qquad [\ell_1, \ell_2] \trianglelefteqslant [t_1, t_2] \leq t}{\Gamma \vdash \langle e_1 \mid e_2 \rangle : t} \qquad \textsc{Sub}\quad \frac{\Gamma \vdash e : t' \qquad t' \leq t}{\Gamma \vdash e : t}$$

Figure 4. The type system for λ^2_+

4.5 Type soundness

We first state a few auxiliary lemmas whose proofs are straightforward, then establish the subject reduction theorem.

Lemma 4.3 (Projection) *Let $i \in \{1, 2\}$. If $\Gamma \vdash e : t$ then $\Gamma \vdash \lfloor e \rfloor_i : t$.*

Lemma 4.4 (Guard) *If $\Gamma \vdash \langle e_1 \mid e_2 \rangle : t$ then there exist $\ell_1, \ell_2 \in H$ and two types t_1, t_2 such that $\Gamma \vdash e_1 : t_1$, $\Gamma \vdash e_2 : t_2$ and $[\ell_1, \ell_2] \trianglelefteqslant [t_1, t_2] \leq t$.*

Lemma 4.5 (Sum) *Assume $\Gamma \vdash e : r^q$. If $e \downarrow C$ then $\Gamma \vdash e : (r^q)_{|C}$ and $r \Cap C$.*

Lemma 4.6 (Substitution) *Assume $\vdash e' : s$. Then $\Gamma[x \mapsto s] \vdash e : t$ implies $\Gamma \vdash e[x \Leftarrow e'] : t$.*

Theorem 4.1 (Subject reduction) *Let $e \to e'$. If $\vdash e : t$ then $\vdash e' : t$.*

Proof. By induction on the derivation of $e \to e'$. We assume, w.l.o.g., that the derivation of $\vdash e : t$ does not end with an instance of 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 $(\lambda x.e_1)\, e_2$ and e' is $e_1[x \Leftarrow e_2]$. In App's premises, we have $\vdash \lambda x.e_1 : t' \to t$ and $\vdash e_2 : t'$. The former must end with an instance of Abs, followed by a number of instances of Sub. Because $\to$ is contravariant (resp. covariant) in its first (resp. second) parameter, applying Sub to Abs's premise yields $(x \mapsto t'') \vdash e_2 : t$ for some t'' such that $t' \leq t''$. By Sub, $\vdash e_2 : t''$ holds. Then lemma 4.6 yields $\vdash e_1[x \Leftarrow e_2] : t$.

○ *Case* (let). By Let and lemma 4.6.

○ *Case* (proj). e is $\pi_j\, (e_1, e_2)$ and e' is e_j. In Proj's premises, we have $\vdash (e_1, e_2) : t_1 \times t_2$ with $t_j = t$. This derivation must end by an instance of Pair followed by a number of instances of Sub. It follows that $\vdash e_j : t_j$, i.e. $\vdash e' : t$.

○ *Case* (destr). e is $\bar{c}\, (c\, e')$. In Destr's premises, we have $\vdash c\, e' : (c : \mathsf{Pre}\, t; *)^{*}$. By Sub and Inj, this yields $\vdash e' : t$.

○ *Case* (case). e is e_0 case $[C_1 : x_1 \mapsto e_1 \mid \ldots \mid C_n : x_n \mapsto e_n]$ and e' is $e_j[x_j \Leftarrow e_0]$, with $e_0 \downarrow C_j$. Within Case's premises, we have $\vdash e_0 : r^q$ and $(x_j \mapsto (r^q)_{|C_j}) \vdash e_j : t_j$ with $t_j \leq t$. By lemma 4.5, the former yields $\vdash e_0 : (r^q)_{|C_j}$. Applying lemma 4.6 and Sub, we obtain $\vdash e' : t$.

○ *Case* (lift-β). e is $\langle e_1 \mid e_2 \rangle\, e_0$ and e' is $\langle e_1 \lfloor e_0 \rfloor_1 \mid e_2 \lfloor e_0 \rfloor_2 \rangle$. In App's premises, we have $\vdash \langle e_1 \mid e_2 \rangle : t' \to t$ and $\vdash e_0 : t'$. Lemma 4.3 yields $\vdash \lfloor e_0 \rfloor_1 : t'$ and $\vdash \lfloor e_0 \rfloor_2 : t'$. By lemma 4.4 and Guards-Fun, $\vdash e_1 : t'_1 \to t_1$ and $\vdash e_2 : t'_2 \to t_2$ hold for some t_1, t_2, t'_1, t'_2 such that $[\ell_1, \ell_2] \trianglelefteqslant [t_1, t_2] \leq t$ (with $\ell_1, \ell_2 \in H$), $t'_1 \geq t'$ and $t'_2 \geq t'$. By Sub and App, it follows $\vdash e_1 \lfloor e_0 \rfloor_1 : t_1$ and $\vdash e_2 \lfloor e_0 \rfloor_2 : t_2$. By Bracket, we conclude that $\vdash e' : t$.

○ *Case* (lift-proj). e is $\pi_j \langle e_1 \mid e_2 \rangle$ and e' is $\langle \pi_j\, e_1 \mid \pi_j\, e_2 \rangle$. In Proj's premises, we find $\vdash \langle e_1 \mid e_2 \rangle : t_1 \times t_2$ with $t_j = t$. By lemma 4.4 and Guards-Pair, $\vdash e_1 : t_{11} \times t_{12}$ and $\vdash e_2 : t_{21} \times t_{22}$ hold for some t_{11}, t_{12}, t_{21}, t_{22} with in particular $[\ell_1, \ell_2] \trianglelefteqslant [t_{1j}, t_{2j}] \leq t_j$ for some $\ell_1, \ell_2 \in H$. By Proj, it follows that $\vdash \pi_j\, e_1 : t_{1j}$ and $\vdash \pi_j\, e_2 : t_{2j}$. Conclude by Bracket.

○ *Case* (lift-destr). e is $\bar{c}\,\langle e_1 \mid e_2\rangle$ and e' is $\langle \bar{c}\,e_1 \mid \bar{c}\,e_2\rangle$. In DESTR's premises, we have $\vdash \langle e_1 \mid e_2\rangle : (c : \mathsf{Pre}\,t; \partial\mathsf{Abs})^*$. By lemma 4.4, we have $\vdash e_1 : (c : \mathsf{Pre}\,t_1; \partial\mathsf{Abs})^*$ and $\vdash e_2 : (c : \mathsf{Pre}\,t_2; \partial\mathsf{Abs})^*$ for some t_1, t_2 such that $[\ell_1, \ell_2] \trianglelefteq [t_1, t_2] \leq t$ with $\ell_1, \ell_2 \in H$. By DESTR, it follows that $\vdash \bar{c}\,e_1 : t_1$ and $\vdash \bar{c}\,e_2 : t_2$. Conclude by BRACKET.

○ *Case* (lift-case). e is $\langle e'_1 \mid e'_2\rangle$ case $[C_j : x_j \mapsto e_j]_j$ and e' is $\langle e'_1$ case $[C_j : x_j \mapsto \lfloor e_j\rfloor_1]_j \mid e'_2$ case $[C_j : x_j \mapsto \lfloor e_j\rfloor_2]_j\rangle$. Because $\langle e'_1 \mid e'_2\rangle \curlywedge [C_1 \mid \ldots \mid C_n]$, $e'_1 \downarrow C_{j_1}$ and $e'_2 \downarrow C_{j_2}$ hold for some $j_1 \neq j_2$.

In CASE's premises, we have $\vdash \langle e'_1 \mid e'_2\rangle : r^q$ and $r \Subset C_1 \cup \ldots C_n$ and $r \Cap C_{j_i} \Rightarrow (x \mapsto (r^q)_{|C_{j_i}}) \vdash e_{j_i} : t_{j_i}$ (for all $i \in \{1, 2\}$) and $[q(C_1), \ldots, q(C_n)] \trianglelefteq [t_1, \ldots, t_n] \leq t$.

By lemma 4.4, there exist r_1, q_1, r_2 and q_2 such that $\vdash e'_1 : {r_1}^{q_1}$ and $\vdash e'_2 : {r_2}^{q_2}$ with $[\ell_1, \ell_2] \trianglelefteq [{r_1}^{q_1}, {r_2}^{q_2}] \leq r^q$ for some $\ell_1, \ell_2 \in H$. By lemma 4.5, we obtain $\vdash e'_i : ({r_i}^{q_i})_{|C_{j_i}}$ and $r_i \Cap C_{j_i}$ (for all $i \in \{1, 2\}$). The latter and $r_{j_i} \leq r$ imply $r \Cap C_{j_i}$. Then $(x \mapsto (r^q)_{|C_{j_i}}) \vdash e_{j_i} : t_{j_i}$ holds and, by lemma 4.3, this yields $(x \mapsto (r^q)_{|C_{j_i}}) \vdash \lfloor e_{j_i}\rfloor_i : t_{j_i}$. Applying an instance of CASE, we obtain $\vdash e'_i$ case $[C_j : x_j \mapsto \lfloor e_j\rfloor_i]_j : t_{j_i}$.

$r_1 \Cap C_{j_1}$, $r_2 \Cap C_{j_2}$ and GUARDS-SUM imply $\ell_1 \leq q_1(c_1 \cdot c_2)$ and $\ell_2 \leq q_2(c_1 \cdot c_2)$. It follows $\ell_1 \leq q(C_1)$ and $\ell_2 \leq q(C_2)$. Then, because $[q(C_1), \ldots, q(C_n)] \trianglelefteq [t_1, \ldots, t_n] \leq t$, we have by lemmas 4.1 and 4.2, $[\ell_1, \ell_2] \trianglelefteq [t_{i_1}, t_{i_2}] \leq t$. Conclude by BRACKET.

○ *Case* (context). By induction hypothesis. □

4.6 Non-interference

In the following, the set H is no longer fixed. Thus, it appears as an extra parameter on λ^2_+ typing judgements (we write $\vdash_H$ instead of $\vdash$). It is still unnecessary to mention it on those judgements which involve λ_+ expressions because H is used only in the BRACKET rule.

Theorem 4.2 (Non-interference) *Choose $\ell, h \in \mathcal{L}$ such that $h \not\leq \ell$. Let $h \trianglelefteq [t_1, t_2] \leq t$. Assume $(x \mapsto t) \vdash e : \mathsf{int}^\ell$, where e is a λ_+ expression. If $\vdash e_i : t_i$ and $e[x \Leftarrow e_i] \rightarrow^\star k_i$, for $i \in \{1, 2\}$, then $k_1 = k_2$.*

Proof. Let H be the upper cone $\{h' \mid h \leq h'\}$. Define $e' = \langle e_1 \mid e_2\rangle$. By BRACKET, $\vdash_H e' : t$. Lemma 4.6 yields $\vdash_H e[x \Leftarrow e'] : \mathsf{int}^\ell$. Now, $\lfloor e[x \Leftarrow e']\rfloor_i$ is $e[x \Leftarrow e_i]$, which, by hypothesis, reduces to k_i. According to lemma 3.3, there exists a normal form n such that $e[x \Leftarrow e'] \rightarrow^\star n$. By theorem 4.1, $\vdash_H e[x \Leftarrow e'] : \mathsf{int}^\ell$ implies $\vdash_H n : \mathsf{int}^\ell$.

By lemma 3.1, $\lfloor n\rfloor_i$ is a normal form for some $i \in \{1, 2\}$. Because the semantics of the λ_+ fragment is deterministic, $\lfloor n\rfloor_i = k_i$ must hold. It follows that n is of the form k or $\langle * \mid *\rangle$. If the latter, then by lemma 4.4, there exists $\ell' \in H$ such that $\ell' \leq \ell$, which implies $\ell \in H$, a contradiction. Thus, we must have $n = k = k_i$.

By lemma 3.2, $e_1 \rightarrow^\star k$ and $e_2 \rightarrow^\star k$ hold. It follows that $n_1 = n_2 = k$. □

This theorem establishes a *weak* non-interference result in so far as it requires both expressions to converge. Indeed, in order to provide a fine-grained analysis, our type system is able to ignore some test conditions. For instance, define e as e' case $[A : \_ \mapsto D \mid B : \_ \mapsto D]$, where e' is an arbitrary expression of type $(A, B : \mathsf{Pre}\,*; \partial\mathsf{Abs})$. The type system statically detects that the result of e's evaluation does not depend on e'. Yet e's termination does depend on that of e': reducing the case clause requires e' to produce either A or B (even though it does not affect the final result). Obtaining a *strong* non-interference statement would require dropping the fine-grained treatment of sums. Anyway, it would be of little sense since we do not deal with timing leaks in general.

Because the type system satisfies a *progress* property (i.e. "no well-typed expression is stuck"), each hypothesis "$e[x \Leftarrow e_i]$ yields an integer" of the non-interference theorem can be safely weakened into "$e[x \Leftarrow e_i]$ does not diverge", i.e. $e[x \Leftarrow e_i] \rightarrow^\star n_i$, because, by progress, any λ_+ normal form of type int^* must be an integer constant.

The non-interference result still applies for λ_+^{CBV}, the λ_+-calculus equipped with a *call-by-value* semantics $\rightarrow_{\text{CBV}}$ (we omit its definition because it is standard). Indeed, if $e[x \Leftarrow e_1] \rightarrow^*_{\text{CBV}} k_1$ and $e[x \Leftarrow e_1] \rightarrow^*_{\text{CBV}} k_2$ then, by normalization, $e[x \Leftarrow e_1] \rightarrow^* k_1$ and $e[x \Leftarrow e_1] \rightarrow^* k_2$. Applying theorem 4.2 with the correct hypothesis about types, we obtain $k_1 = k_2$.

5 Type inference

We now explain how a type system with decidable type inference can be obtained from that of section 4. This raises several technical issues. By lack of space we prefer to present it in an informal manner only.

The description of an inference algorithm for a constraint based type system generally consists of two distinct parts: a set of inference rules and a constraint solving algorithm. Obtaining inference rules in the style of HM(X) [17] from a set of rules such as that of figure 4 is a well-studied issue; the reader is referred to [7, 11] for more details. It requires introducing type variables, a constraint language and universally quantified, constrained type schemes. The correctness of the system thus obtained may be proven by a simple encoding of its judgements into λ_+ judgements. This set of rules may be viewed as an algorithm which, given an input expression, returns a constraint which is satisfiable if and only if the expression is well typed.

Constraint solving for (non-atomic) subtyping is known to be decidable and reasonably efficient algorithms have been proposed in this area [9, 10]. However, our system involves non-standard forms of constraints. We claim that constraint solving remains a decidable problem.

Constraints of the form $r \Subset C$ can be encoded using simple subtyping constraints requiring fields not in C to be Abs. In CASE's premises, typing judgements concerning the handlers are subject to a condition of the form $r \Cap C$. Such a condition may be enforced in the type inference system by prefixing with it every constraint produced by the judgement. This introduces conditional constraints such as $(r \Cap C) \mathrel{?} \gamma$ where γ is an arbitrary constraint. Such a constraint may be solved by keeping it unchanged as long as none of r's fields corresponding to constructors in C are known to be $\mathsf{Pre}\,*$. When one of them is unified with $\mathsf{Pre}\,*$ then the condition is satisfied and the conditional $(r \Cap C) \mathrel{?} \gamma$ must be replaced by γ itself.

Guards $[\ell_1, \ldots, \ell_n] \trianglelefteq [t_1, \ldots, t_n] \leq t$ can be solved in a "lazy" manner. As long as nothing is known about the structure of t (or any of the t_i), the constraint is preserved intact. When t (or one of the t_i) is instantiated with a term, the constraint may be propagated throughout its structure to the leaves, generating a number of subconstraints. This propagation is straightforward on → and × nodes. At sum nodes, it requires the introduction of a form of conditional constraint: $[\ell_1, \ldots, \ell_n] \trianglelefteq [{r_1}^{q_1}, \ldots, {r_n}^{q_n}] \leq r^q$ is indeed equivalent to $[\ell_1, \ldots, \ell_n] \trianglelefteq [r_1, \ldots, r_n] \leq r$ and $\forall c_1 \neq c_2,\ (\mathsf{Pre}\,*, \mathsf{Pre}\,*) \leq (r_{j_1} \star r_{j_2})(c_1 \cdot c_2) \Rightarrow \ell_{j_1} \leq q_{j_1}(c_1 \cdot c_2)$ for every $j_1 \neq j_2$ ($r_{j_1} \star r_{j_2}$ denotes the matrix formed by the cartesian product of the rows r_{j_1} and r_{j_2}).

If the number of constructor names that are present in every sum type is finite and statically known (e.g. if sum type are used to represent finite variant types such as ML's datatypes), such a constraint may be decomposed pointwise by generating a different conditional constraint for each pair of constructors $c_1 \cdot c_2$. Otherwise, it may be viewed as a conditional constraint involving two-dimensional rows: $(\mathsf{Pre}\,*, \mathsf{Pre}\,*) \leq (r_{j_1} \star r_{j_2}) \Rightarrow \ell_{j_1} \leq q_{j_1}$. Although two-dimensional rows have never been — to the best of our knowledge — really used, they form a natural generalization of rows [15] and can be manipulated using the same techniques [10].

Lastly, constraints of the form $q(C) \leq \ell$ are equivalent to $\forall\ c_1 \in C,\ \forall\ c_2 \in \mathcal{C} \backslash C,\ q(c_1 \cdot c_2) \leq \ell$. Once again, if the number of involved constructor names is finite, this constraint may be decomposed pointwise into a number of inequalities between security levels. Otherwise, it may be viewed as a subtyping constraint between two-dimensional rows: $q' \leq \partial^2 \ell$ (we denote by $\partial^2 \ell$ the constant two-dimensional row with the same domain as q') where q' is the restriction of q to the rectangle $C \times (\mathcal{C} \backslash C)$. Assuming C is finite or cofinite, this restriction may be computed thanks to unification constraints using Rémy's row syntax [14].

6 Examples

In this section, we illustrate the expressiveness of λ_+'s type system by describing the types obtained for a small set of relevant examples. We use a Caml-like syntax, which can be easily de-sugared into λ_+. In particular, we allow constructors with no argument and booleans. Booleans can be easily encoded within sums by choosing two different constructors T and F (for the constants *true* and *false*, respectively). The test `if` e_1 `then` e_2 `else` e_3 can be translated into e_1 `case` $[\{T\} : e_2 \mid \{F\} : e_3]$. Because it involves only two constructors, the type of a boolean expression $(T : \mathsf{Pre}\,; F : \mathsf{Pre}\,; \partial\mathsf{Abs})^{(T \cdot F : \ell; *)}$ carries only one relevant security level, ℓ. Thus, it may be abbreviated into b^ℓ.

Our first examples are those of the introduction.

```
let f x y z =
  if x then (if y then A else B)
       else (if z then A else D)

let g = function
    A | B -> true
  | D -> false

let h x y z = g (f x y z)
```

The function f admits all types of the form $\mathsf{b}^\alpha \to \mathsf{b}^\beta \to \mathsf{b}^\delta \to (A, B, D : \mathsf{Pre}\,; *)^{(A \cdot B : \alpha \sqcup \beta; A \cdot D : \alpha \sqcup \delta; B \cdot D : \alpha; *)}$ (where α, β and δ are security levels). The tails of the row and the matrix describing the function's results are unconstrained, allowing them to be $\partial\mathsf{Abs}$ and $\partial\bot$, respectively. As explained in the introduction, testing whether the result of f is B rather than D can leak information only about the first argument. Therefore, the field $B \cdot D$ of the matrix is only constrained by the level α. We obtain $(A, B, D : \mathsf{Pre}\,; \partial\mathsf{Abs})^{(A \cdot D, B \cdot D : \alpha; *)} \to \mathsf{b}^\alpha$ for g. Lastly our system detects the absence of information flow from the second argument to the output of h, the composition of f and g, as reflected by its types: $\mathsf{b}^\alpha \to \mathsf{b}^\beta \to \mathsf{b}^\delta \to \mathsf{b}^{\alpha \sqcup \delta}$.

The function f' is identical to the function f but it returns its result embedded in a λ-abstraction:

```
let f' x y z =
  if x then (if y then (fun _ -> A)
                  else (fun _ -> B))
       else (if z then (fun _ -> A)
                  else (fun _ -> D))
```

For f', the system gives $\mathsf{b}^\alpha \to \mathsf{b}^\beta \to \mathsf{b}^\delta \to (* \to (A, B, D : \mathsf{Pre}\,; *)^{(A \cdot B : \alpha \sqcup \beta; A \cdot D : \alpha \sqcup \delta; B \cdot D : \alpha; *)})$. This example illustrates the interest of the absence of security level on the → type constructor: it allows the accuracy of the typing of sums to pass through it. As a result, if we re-implement f by `let f x y z = (f' x y z) 0`, we obtain exactly the same type scheme as that for the first version.

Our next example tests whether the head constructor of its argument is A or not:

Basic reductions

$$\begin{array}{rcll} (\lambda x.e)\, v & \to & e[x \leftarrow v] & (\beta) \\ \pi_j\,(v_1, v_2) & \to & v_j & \text{(proj)} \\ \mathsf{let}\ x = v\ \mathsf{in}\ e & \to & e[x \leftarrow v] & \text{(let)} \end{array}$$

Sequencing

$$\begin{array}{rcll} \mathsf{bind}\ x = v\ \mathsf{in}\ e & \to & e[x \leftarrow v] & \text{(bind)} \\ \mathsf{raise}\ (\varepsilon\, v)\ \mathsf{handle}\ \varepsilon\, x \succ e & \to & e[x \leftarrow v] & \text{(handle)} \\ \mathsf{raise}\ (\varepsilon\, v)\ \mathsf{handle}\ x \succ e & \to & e[x \leftarrow \varepsilon\, v] & \text{(handle-all)} \end{array}$$

$$\frac{o \text{ escapes } E}{E[o] \to o}\ \text{(throw-context)} \qquad \frac{e \to e'}{E[e] \to E[e']}\ \text{(context)}$$

Figure 5. Operational semantics of $\lambda_{\mathcal{E}}$

```
let test_A = function
    A _ -> true
  | _ -> false
```

It admits $r^q \to \mathsf{b}^{q(\{A\})}$ (for every q) as type: the resulting boolean is marked with the union of all security levels attached to the constructor A in the input.

The *rotate* function performs a transposition of three constructors:

```
let rotate = function
    A -> B
  | B -> D
  | D -> A
```

The same transposition arises between the corresponding fields of the types describing the input and the output of this function: $(A{:}\alpha;B{:}\beta;D{:}\delta;\partial\mathsf{Abs})^{(A\cdot B:\delta';A\cdot D:\beta';B\cdot D:\alpha';\partial\bot))} \to ((A{:}\delta;B{:}\alpha;D{:}\beta;\partial\mathsf{Abs})^{(A\cdot B:\beta';A\cdot D:\alpha';B\cdot D:\delta';\partial\bot)}$. Because our type system guards the type of each branch of the case clause before computing the type of the whole expression, it is able to exactly relate in this example the security levels associated with each pair of constructors of the output with those of the input.

7 λ-calculus with exceptions

7.1 The $\lambda_{\mathcal{E}}$-calculus

In this section, we define a λ-calculus with "let" binding and exceptions, $\lambda_{\mathcal{E}}$. Let $\mathcal{E}$ be a denumerable set whose elements are called *exception names* and denoted by ε. Then values, outcomes, expressions and evaluation contexts of $\lambda_{\mathcal{E}}$ are defined as follows:

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

In fact, $\lambda_{\mathcal{E}}$ is the language studied in [13] (where it is referred to as "*Core ML*") deprived of the constructions dealing with references. Its small-step semantics is recalled in figure 5. (o *escapes* E if and only if $E[o]$ cannot be reduced by one of the rules (bind), (handle) and (handle-all)).

Because of the presence of *effects*, our presentation of $\lambda_{\mathcal{E}}$ differs from that of λ_+ in two points. First, following [20], it is restricted to a call-by-value setting to preserve confluence. Second, we introduce a segregation between values and expressions. This syntactic restriction enables a lighter formulation of the type system and allows it to remain independent of the evaluation order. It does not reduce the expressiveness of the language because usual expressions may be encoded into our restricted syntax in a straightforward manner (see [13], section 5.7).

7.2 Encoding $\lambda_{\mathcal{E}}$ into λ_+^{CBV}

Let the constructors of λ_+ be exactly the exception names of $\lambda_{\mathcal{E}}$, with an additional one, denoted by η (i.e. $\mathcal{C} = \mathcal{E} \cup \{\eta\}$). The basic idea of the encoding introduced by figure 6 is to translate every expression e of $\lambda_{\mathcal{E}}$ into an expression $\llbracket e \rrbracket$ of λ_+ such that if e evaluates to a value v without raising an exception then $\llbracket e \rrbracket$ evaluates to $\eta\,\llparenthesis v \rrparenthesis$ in λ_+^{CBV} and if the execution of e raises an exception $\varepsilon\, v$ then $\llbracket e \rrbracket$ reduces to $\varepsilon\,\llparenthesis v \rrparenthesis$. Such an encoding may be defined in a systematic way using monads [4, 19], but here we prefer, for simplicity, a direct translation. It is stable w.r.t. substitution in the sense that $\llbracket e[x \leftarrow v] \rrbracket = \llbracket e \rrbracket[x \Leftarrow \llparenthesis v \rrparenthesis]$. The following lemmas establish the correctness of the encoding with respect to the semantics.

Lemma 7.1 (Correctness) *If* $e \to e'$ *then* $\llbracket e \rrbracket \to^{+}_{\text{CBV}} \llbracket e' \rrbracket$.

Proof. By induction on the derivation of $e \to e'$.

○ *Case* (β). e is $(\lambda x.e_0)\, v$ and e' is $e_0[x \leftarrow v]$. Then $\llbracket e \rrbracket = (\lambda x.\llbracket e_0 \rrbracket)\,\llparenthesis v \rrparenthesis$. By ($\beta$), we have $\llbracket e \rrbracket \to_{\text{CBV}} \llbracket e_0 \rrbracket[x \Leftarrow \llparenthesis v \rrparenthesis]$, i.e. $\llbracket e \rrbracket \to_{\text{CBV}} \llbracket e' \rrbracket$.

○ *Case* (proj). e is $\pi_j\,(v_1, v_2)$, e' is v_j. Then $\llbracket e \rrbracket = \eta\,(\pi_j\,(\llparenthesis v_1 \rrparenthesis, \llparenthesis v_2 \rrparenthesis))$. By (proj), we have $\llbracket e \rrbracket \to_{\text{CBV}} \eta\,\llparenthesis v_j \rrparenthesis$, i.e. $\llbracket e \rrbracket \to_{\text{CBV}} \llbracket e' \rrbracket$.

○ *Case* (bind). e is $\mathsf{bind}\ x = v\ \mathsf{in}\ e_0$ and e' is $e_0[x \leftarrow v]$. We have $\llbracket e \rrbracket = \mathsf{case}\ \{\{\eta\} : y_1 \mapsto \mathsf{let}\ x = \bar{\eta}\, y_1\ \mathsf{in}\ \llbracket e_0 \rrbracket;\quad \mathcal{E} : y_1 \mapsto y_1\}$ Then, by applying successively (case) and (bind), $\llbracket e \rrbracket \to^2_{\text{CBV}} \llbracket e_0 \rrbracket[x \leftarrow \llparenthesis v \rrparenthesis]$, i.e. $\llbracket e \rrbracket \to^2_{\text{CBV}} \llbracket e' \rrbracket$.

$$\begin{array}{llcl}
\textbf{Values} & (\!|x|\!) & = & x \\
 & (\!|k|\!) & = & k \\
 & (\!|(v_1, v_2)|\!) & = & ((\!|v_1|\!), (\!|v_2|\!)) \\
 & (\!|\lambda x.e|\!) & = & \lambda x.[\![e]\!] \\
 & (\!|\varepsilon\, v|\!) & = & \varepsilon\, (\!|v|\!)
\end{array}
\qquad
\begin{array}{llcl}
\textbf{Expressions} & [\![v]\!] & = & \eta\, (\!|v|\!) \\
 & [\![v_1\, v_2]\!] & = & (\!|v_1|\!)\, (\!|v_2|\!) \\
 & [\![\pi_j\, v]\!] & = & \eta\, (\pi_j\, (\!|v|\!)) \\
 & [\![\mathsf{raise}\ v]\!] & = & (\!|v|\!) \\
 & [\![\mathsf{let}\ x = v\ \mathsf{in}\ e]\!] & = & \mathsf{let}\ x = (\!|v|\!)\ \mathsf{in}\ [\![e]\!] \\
 & [\![E[e]]\!] & = & [\![E]\!][[\![e]\!]]
\end{array}$$

$$\begin{array}{llcl}
\textbf{Evaluation contexts} & [\![\mathsf{bind}\ x = [\,]\ \mathsf{in}\ e]\!] & = & [\,]\ \mathsf{case}\ [\eta : y \mapsto \mathsf{let}\ x = \bar{\eta}\, y\ \mathsf{in}\ [\![e]\!] \mid \mathcal{E} : y \mapsto y] \\
 & [\![[\,]\ \mathsf{handle}\ \varepsilon\, x \succ e]\!] & = & [\,]\ \mathsf{case}\ [\varepsilon : y \mapsto \mathsf{let}\ x = \bar{\varepsilon}\, y\ \mathsf{in}\ [\![e]\!] \mid \mathcal{C}\backslash\{\varepsilon\} : y \mapsto y] \\
 & [\![[\,]\ \mathsf{handle}\ x \succ e]\!] & = & [\,]\ \mathsf{case}\ [\eta : y \mapsto y \mid \mathcal{E} : x \mapsto [\![e]\!]]
\end{array}$$

Figure 6. Encoding $\lambda_{\mathcal{E}}$ into λ_+

○ *Cases* (handle), (handle-all) and (throw-context) are similar to (bind).

○ *Case* (context). e is $E[e_0]$ and e' is $E[e'_0]$ with $e_0 \to e'_0$. By induction hypothesis, we have $[\![e_0]\!] \to^+_{\text{CBV}} [\![e'_0]\!]$. Then, by (context), $[\![E]\!][[\![e_0]\!]] \to^+_{\text{CBV}} [\![E]\!][[\![e'_0]\!]]$, i.e. $[\![e]\!] \to^+_{\text{CBV}} [\![e']\!]$. □

Lemma 7.2 (Values) *If* $(\!|v_1|\!) = (\!|v_2|\!)$ *then* $v_1 = v_2$.

Proof. By induction on the definition of $(\!|\cdot|\!)$ and $[\![\cdot]\!]$. □

7.3 Typing $\lambda_{\mathcal{E}}$

We now define a type system for $\lambda_{\mathcal{E}}$ and prove its correctness by translating each $\lambda_{\mathcal{E}}$ judgement into a λ_+ one. The type algebra for $\lambda_{\mathcal{E}}$ is a simple subset of that of λ_+. Restricted *types*, *alternatives*, *rows* and *matrices* (denoted by **bold** meta-variables) are defined as follows:

$$\begin{array}{lcl}
\mathbf{t} & ::= & \mathsf{int}^{\ell} \mid \mathbf{t} \times \mathbf{t} \mid \mathbf{t} \to \mathbf{r}^{\mathbf{q}} \mid \dot{\mathbf{r}}^{\dot{\mathbf{q}}} \\
\mathbf{a} & ::= & \mathsf{Abs} \mid \mathsf{Pre}\,\mathbf{t} \\
\mathbf{r} & ::= & \{c \mapsto \mathbf{a}\} \\
\mathbf{q} & ::= & \{c \cdot c \mapsto \ell\}
\end{array}$$

The meta-variable $\dot{\mathbf{r}}$ (resp. $\dot{\mathbf{q}}$) stands for a row $\mathbf{r}$ (resp. matrix $\mathbf{q}$) carrying no information about η, i.e. such that $\mathbf{r}(\eta) = \mathsf{Abs}$ (resp. $\forall \varepsilon \in \mathcal{E}, \mathbf{q}(\eta \cdot \varepsilon) = \bot$). Types for integers and pairs remain unchanged. In this restriction, λ_+ sum types play two distinct roles. First they appear as *effects* on the right-hand-side of function types. Here, they describe the possible normal or exceptional results the function may produce, with associated security levels. Second, they are used to type expressions whose result is an exception: $\dot{\mathbf{r}}^{\dot{\mathbf{q}}}$. There, we use dotted versions of content and level rows to signify that the fields relating to the η constructor may be constrained to be Abs and $\bot$, respectively (the translation of an exception value into λ_+ is a sum whose head constructor must belong to $\mathcal{E}$). $\lambda_{\mathcal{E}}$'s polytypes (i.e. nonempty upward-closed sets of types $\mathbf{t}$) are denoted by the meta-variable $\mathbf{s}$.

The typing rules for $\lambda_{\mathcal{E}}$ are given in figure 7. We distinguish two forms of judgements: $\Gamma \vdash v : \mathbf{t}$ and $\Gamma \Vdash e : \mathbf{r}^{\mathbf{q}}$. The former deals with values and involves a simple type (because values cannot raise any exception). The latter associates to an expression an effect $\mathbf{r}^{\mathbf{q}}$, describing what kind of result (value or exceptions) it may produce: $\mathbf{r}(\eta)$ is Abs if e never reduces to a value and is $\mathsf{Pre}\,\mathbf{t}$ if e may produce a value of type $\mathbf{t}$. Analogously, for every expression name ε, $\mathbf{r}(\varepsilon)$ is Abs if e never raises an exception named ε and is $\mathsf{Pre}\,\mathbf{t}$ if it may raise such an exception with an argument of type $\mathbf{t}$. $\mathbf{q}(c_1 \cdot c_2)$ is as expected an approximation of the level of the information one leaks by revealing that e produces a result of name c_1 (value or exception) rather than c_2. $\lambda_{\mathcal{E}}$'s typing rules may be obtained by a simple combination of λ_+ rules (figure 4) and the translation of $\lambda_{\mathcal{E}}$ into λ_+ ($[\![\cdot]\!]$ for $\Vdash$ judgements on expressions and $(\!|\cdot|\!)$ for $\vdash$ judgements on values), with some straightforward constraint simplifications. First, in the second premise of E-BIND, we may assume in the right-hand-side of the $\Rightarrow$ that $\mathbf{r}_1(\eta) = \mathsf{Pre}\,\mathbf{t}$ for some $\mathbf{t}$ (we write $\overline{\mathsf{Pre}}\,\mathbf{r}_1(\eta)$ for such a $\mathbf{t}$). Second, because $\{\eta\}$ and $\mathcal{E}$ are complementary, $\mathbf{q}_1(\{\eta\}) = \mathbf{q}_1(\mathcal{E})$ holds. Therefore, the constraint $[\mathbf{q}_1(\{\eta\}), \mathbf{q}_1(\mathcal{E})] \trianglelefteq [(\mathbf{r}_1{}^{\mathbf{q}_1})_{|\mathcal{E}}, \mathbf{r}_2{}^{\mathbf{q}_2}] \leq \mathbf{r}^{\mathbf{q}}$ is simplified into $\mathbf{q}_1(\{\eta\}) \trianglelefteq [(\mathbf{r}_1{}^{\mathbf{q}_1})_{|\mathcal{E}}, \mathbf{r}_2{}^{\mathbf{q}_2}] \leq \mathbf{r}^{\mathbf{q}}$. Rules E-HANDLE and E-HANDLEALL present analogous simplifications.

The following lemma states the *soundness* of this type system by considering each of its judgements as one of λ_+.

Lemma 7.3 (Soundness) $\Gamma \vdash v : \mathbf{t}$ *implies* $\Gamma \vdash (\!|v|\!) : \mathbf{t}$. $\Gamma \Vdash e : \mathbf{r}^{\mathbf{q}}$ *implies* $\Gamma \vdash [\![e]\!] : \mathbf{r}^{\mathbf{q}}$.

From this correspondence, we immediately obtain the following non-interference result for $\lambda_{\mathcal{E}}$. For simplicity, the statement only concerns integer results, but a more general one can be obtained.

Theorem 7.1 (Non-interference) *Choose* $\ell, h \in \mathcal{L}$ *such that* $h \not\leq \ell$. *Let* $h \trianglelefteq [t_1, t_2] \leq t$. *Assume* $(x \mapsto t) \Vdash$

Values

V-INT

$$\Gamma \vdash k : \textsf{int}^{*}$$

V-VAR

$$\frac{\mathbf{t} \in \Gamma(x)}{\Gamma \vdash x : \mathbf{t}}$$

V-PAIR

$$\frac{\Gamma \vdash v_1 : \mathbf{t}_1 \qquad \Gamma \vdash v_2 : \mathbf{t}_2}{\Gamma \vdash (v_1, v_2) : \mathbf{t}_1 \times \mathbf{t}_2}$$

V-ABS

$$\frac{\Gamma[x \mapsto \mathbf{t}] \Vdash e : \mathbf{r}^{\mathbf{q}}}{\Gamma \vdash \lambda x.e : \mathbf{t} \to \mathbf{r}^{\mathbf{q}}}$$

V-EXN

$$\frac{\Gamma \vdash v : \mathbf{t}}{\Gamma \vdash \varepsilon\, v : (\varepsilon : \textsf{Pre}\,\mathbf{t}; *)^{*}}$$

V-SUB

$$\frac{\Gamma \vdash v : \mathbf{t}' \qquad \mathbf{t}' \leq \mathbf{t}}{\Gamma \vdash v : \mathbf{t}}$$

Expressions

E-VALUE

$$\frac{\Gamma \vdash v : \mathbf{t}}{\Gamma \Vdash v : (\eta : \textsf{Pre}\,\mathbf{t}; *)^{*}}$$

E-RAISE

$$\frac{\Gamma \vdash v : \dot{\mathbf{r}}^{\dot{\mathbf{q}}}}{\Gamma \Vdash \textsf{raise}\; v : \dot{\mathbf{r}}^{\dot{\mathbf{q}}}}$$

E-APP

$$\frac{\Gamma \vdash v_1 : \mathbf{t} \to \mathbf{r}^{\mathbf{q}} \qquad \Gamma \vdash v_2 : \mathbf{t}}{\Gamma \Vdash v_1\, v_2 : \mathbf{r}^{\mathbf{q}}}$$

E-PROJ

$$\frac{\Gamma \vdash v : \mathbf{t}_1 \times \mathbf{t}_2}{\Gamma \Vdash \pi_j\, v : (\eta : \textsf{Pre}\,\mathbf{t}_1; *)^{*}}$$

E-BIND

$$\frac{\Gamma \Vdash e_1 : {\mathbf{r}_1}^{\mathbf{q}_1} \qquad \mathbf{r}_1 \Cap \{\eta\} \Rightarrow \Gamma[x \mapsto \overline{\textsf{Pre}}\,\mathbf{r}_1(\eta)] \Vdash e_2 : {\mathbf{r}_2}^{\mathbf{q}_2} \qquad \mathbf{q}_1(\{\eta\}) \trianglelefteqslant [({\mathbf{r}_1}^{\mathbf{q}_1})_{|\mathcal{E}}, {\mathbf{r}_2}^{\mathbf{q}_2}] \leq \mathbf{r}^{\mathbf{q}}}{\Gamma \Vdash \textsf{bind}\; x = e_1 \;\textsf{in}\; e_2 : \mathbf{r}^{\mathbf{q}}}$$

E-HANDLE

$$\frac{\Gamma \Vdash e_1 : {\mathbf{r}_1}^{\mathbf{q}_1} \qquad \mathbf{r}_1 \Cap \{\varepsilon\} \Rightarrow \Gamma[x \mapsto \overline{\textsf{Pre}}\,\mathbf{r}_1(\varepsilon)] \Vdash e_2 : {\mathbf{r}_2}^{\mathbf{q}_2} \qquad \mathbf{q}_1(\{\varepsilon\}) \trianglelefteqslant [({\mathbf{r}_1}^{\mathbf{q}_1})_{|\mathcal{C}\setminus\{\varepsilon\}}, {\mathbf{r}_2}^{\mathbf{q}_2}] \leq \mathbf{r}^{\mathbf{q}}}{\Gamma \Vdash e_1 \;\textsf{handle}\; \varepsilon\, x \succ e_2 : \mathbf{r}^{\mathbf{q}}}$$

E-HANDLEALL

$$\frac{\Gamma \Vdash e_1 : {\mathbf{r}_1}^{\mathbf{q}_1} \qquad \mathbf{r}_1 \Cap \mathcal{E} \Rightarrow \Gamma[x \mapsto ({\mathbf{r}_1}^{\mathbf{q}_1})_{|\mathcal{E}}] \Vdash e_2 : {\mathbf{r}_2}^{\mathbf{q}_2} \qquad \mathbf{q}_1(\{\eta\}) \trianglelefteqslant [({\mathbf{r}_1}^{\mathbf{q}_1})_{|\{\eta\}}, {\mathbf{r}_2}^{\mathbf{q}_2}] \leq \mathbf{r}^{\mathbf{q}}}{\Gamma \Vdash e_1 \;\textsf{handle}\; x \succ e_2 : \mathbf{r}^{\mathbf{q}}}$$

E-LET

$$\frac{\Gamma \vdash v : \mathbf{s} \qquad \Gamma[x \mapsto \mathbf{s}] \Vdash e : \mathbf{r}^{\mathbf{q}}}{\Gamma \Vdash \textsf{let}\; x = v \;\textsf{in}\; e : \mathbf{r}^{\mathbf{q}}}$$

E-SUB

$$\frac{\Gamma \Vdash e : \mathbf{r}'^{\mathbf{q}'} \qquad \mathbf{r}' \leq \mathbf{r} \qquad \mathbf{q}' \leq \mathbf{q}}{\Gamma \Vdash e : \mathbf{r}^{\mathbf{q}}}$$

Figure 7. The type system for $\lambda_{\mathcal{E}}$

$e : (\eta : \textsf{Pre}\,\textsf{int}^{\ell}; *)^{*}$*, where* e *is a* $\lambda_{\mathcal{E}}$ *expression. If* $\vdash v_i : t_i$ *and* $e[x \Leftarrow v_i] \to^{\star} v_i$*, for* $i \in \{1, 2\}$*, then* $v_1 = v_2$*.*

Proof. By lemmas 7.1, 7.2, 7.3 and theorem 4.2. □

8 One-dimensional systems

We now present two type systems, $\lambda_{+}^{(1)}$ and $\lambda_{\mathcal{E}}^{(1)}$, derived from λ_{+} and $\lambda_{\mathcal{E}}$. Although they provide a less accurate analysis, they remain of interest because they involve simpler annotations and give us the opportunity to compare our treatment of exceptions with previous works [5, 13].

8.1 The system $\lambda_{+}^{(1)}$

In this section, we present a more lightweight type system for λ_{+} (which we will refer to as $\lambda_{+}^{(1)}$) where information carried by a sum is described by a one-dimensional row of levels indexed by constructor names. We begin by restricting the case construction of λ_{+} to have only two handlers: e case $[\{c\} : x \mapsto e_1 \mid (\mathcal{C}\backslash\{c\}) : x \mapsto e_2]$. (Note that such a restriction still allows multiple matching by nesting case clauses. Moreover, it still allows the encoding of exceptions into sums, see section 8.2.)

The point of this restriction is that the use of matrices q in the CASE rule of λ_{+} is now limited to an access of the form $q(\{c\})$ for some $c \in \mathcal{C}$ (or $q(\mathcal{C}\backslash\{c\})$ which is equal to $q(\{c\})$). The basic idea of $\lambda_{+}^{(1)}$ consists in directly storing these levels in sum types, rather than the whole matrix q. Thus, types, alternatives, rows and *vectors* of $\lambda_{+}^{(1)}$ (which are denoted $\bar{t}$, $\bar{a}$, $\bar{r}$ and $\bar{q}$, respectively) are obtained by the same grammar as that of section 4.1, where the definition of matrices is replaced by $\bar{q} ::= \{c \mapsto \ell\}$. Sum types now have the form $\bar{r}^{\bar{q}}$ where both $\bar{r}$ and $\bar{q}$ are indexed by $\mathcal{C}$. The meaning of $\bar{r}$ remains unchanged. Simultaneously, the vector $\bar{q}$ indicates for each constructor c how much information

$$\textsc{Case}^{(1)}\quad \frac{\Gamma \vdash e : \bar{r}^{\bar{q}} \qquad \Gamma[x_1 \mapsto (\bar{r}^{\bar{q}})_{|\{c\}}] \vdash e_1 : \bar{t} \qquad \Gamma[x_2 \mapsto (\bar{r}^{\bar{q}})_{|\mathcal{C}\setminus\{c\}}] \vdash e_2 : \bar{t} \qquad \bar{q}(c) \lhd \bar{t}}{\Gamma \vdash v \ \mathsf{case}\ [\{c\} : x_1 \mapsto e_1 \mid \mathcal{C}\setminus\{c\} : x_2 \mapsto e_2] : \bar{t}}$$

Figure 8. The $\textsc{Case}^{(1)}$ rule of $\lambda_+^{(1)}$

Types

$$\frac{\ell \leq \ell'}{\ell \lhd \mathsf{int}^{\ell'}} \qquad \frac{\ell \lhd \bar{t}}{\ell \lhd \bar{t}' \to \bar{t}} \qquad \frac{\ell \lhd \bar{t}_1 \qquad \ell \lhd \bar{t}_2}{\ell \lhd \bar{t}_1 \times \bar{t}_2}$$

$$\frac{\ell \lhd \bar{r} \qquad \forall c, \bar{r}(c) \neq \mathsf{Abs} \Rightarrow \ell \leq \bar{q}(c)}{\ell \lhd \bar{r}^{\bar{q}}} \qquad \frac{\ell \lhd \bar{r} \qquad \bar{r} = (* : \mathsf{Pre}\,*; \partial\mathsf{Abs})}{\ell \lhd \bar{r}^{\bar{q}}}$$

Rows

$$\ell \lhd \mathsf{Abs} \qquad \frac{\ell \lhd \bar{t}}{\ell \lhd \mathsf{Pre}\,\bar{t}} \qquad \frac{\forall c \in \mathcal{C},\ \ \ell \lhd \bar{r}(c)}{\ell \lhd \bar{r}}$$

Figure 9. One-dimensional guards

one may leak by testing whether the expression at hand produces a result whose head constructor is c. This corresponds to the level given by $q(\{c\}) = \sqcup\{q(c \cdot c') \mid c' \neq c\}$ in the previous system. In other words, a matrix q is approximated in $\lambda_+^{(1)}$ by the vector $\bar{q}$ defined by $\bar{q}(c) = q(\{c\})$.

This correspondence allows us to derive the typing rules of $\lambda_+^{(1)}$ from those of λ_+. All the rules remain syntactically unchanged, except $\textsc{Case}$ whose new version, $\textsc{Case}^{(1)}$, is given in figure 8. The main difference with the previous rule lies in the fact that the union of the resulting types of the two branches is computed *before* marking it by the level $\bar{q}(c)$: the type system is no longer able to take into account the origin of each component of the resulting type while guarding it. As a result, the predicate $\lhd$ in $\lambda_+^{(1)}$, defined in figure 9, takes only two arguments: a security level and a type. $\ell \lhd \bar{r}^{\bar{q}}$ constrains each field of $\bar{q}$ corresponding to an entry of $\bar{r}$ which is $\mathsf{Pre}\,*$ to be ℓ or greater. There is nevertheless a special case: if all but one fields of $\bar{r}$ are Abs, (that means when the possible head constructor the expression may produce is known by the type system) then it is unnecessary to constrain the row $\bar{q}$.

We now briefly prove the correctness of $\lambda_+^{(1)}$ thanks to an encoding into λ_+. We introduce a mapping $\langle\!\langle\cdot\rangle\!\rangle$ from $\lambda_+^{(1)}$ types into those of λ_+. It is a homomorphism, except on vectors which are translated into matrices using the following approximation: $\langle\!\langle\bar{q}\rangle\!\rangle(c_1 \cdot c_2) = \bar{q}(c_1) \sqcap \bar{q}(c_2)$. We extend $\langle\!\langle\cdot\rangle\!\rangle$ pointwise on polytypes and environments.

Lemma 8.1 (Subtyping) *If $\bar{t}_1 \leq \bar{t}_2$ then $\langle\!\langle\bar{t}_1\rangle\!\rangle \leq \langle\!\langle\bar{t}_2\rangle\!\rangle$.*

Proof. We first prove that $\bar{q}_1 \leq \bar{q}_2$ implies $\langle\!\langle\bar{q}_1\rangle\!\rangle \leq \langle\!\langle\bar{q}_2\rangle\!\rangle$. Assume $\bar{q}_1 \leq \bar{q}_2$. Let c and c' be two distinct constructor names. We have $\bar{q}_1(c) \leq \bar{q}_2(c)$ and $\bar{q}_1(c') \leq \bar{q}_2(c')$. This yields $\bar{q}_1(c) \sqcap \bar{q}_1(c') \leq \bar{q}_2(c) \sqcap \bar{q}_2(c')$, i.e. $\langle\!\langle\bar{q}_1\rangle\!\rangle(c \cdot c') \leq \langle\!\langle\bar{q}_2\rangle\!\rangle(c \cdot c')$.

Conclude by induction on the structure of types, alternatives and rows. □

Lemma 8.2 (Guard) *$\ell \lhd \bar{t}$ implies $\ell \trianglelefteq [\langle\!\langle\bar{t}\rangle\!\rangle, \langle\!\langle\bar{t}\rangle\!\rangle]$.*

Proof. By induction on the structure of $\bar{t}$. The only case of interest is $\bar{t} = \bar{r}^{\bar{q}}$. Let $r^q = \langle\!\langle\bar{r}\rangle\!\rangle^{\langle\!\langle\bar{q}\rangle\!\rangle} = \langle\!\langle\bar{r}^{\bar{q}}\rangle\!\rangle$. By definition, $\ell \lhd \bar{r}^{\bar{q}}$ implies $\ell \lhd \bar{r}$. By induction hypothesis and $\textsc{Guards-Row}$, it follows that $\ell \trianglelefteq [r, r]$. If only one entry of $\bar{r}$ is Pre then the same arises for r and $\ell \trianglelefteq [r^q, r^q]$. Otherwise, if $r(c_1) = \mathsf{Pre}\,*$ and $r(c_2) = \mathsf{Pre}\,*$ for some $c_1 \neq c_2$, $\ell \lhd \bar{r}^{\bar{q}}$ implies $\ell \leq \bar{q}(c_1)$ and $\ell \leq \bar{q}(c_2)$. It follows $\ell \leq \bar{q}(c_1) \sqcap \bar{q}(c_2) = q(c_1 \cdot c_2)$. Once again, we conclude that $\ell \trianglelefteq [r^q, r^q]$. □

Lemma 8.3 (Restriction) *$\langle\!\langle(\bar{r}^{\bar{q}})_{|C}\rangle\!\rangle = \langle\!\langle\bar{r}^{\bar{q}}\rangle\!\rangle_{|C}$ holds for all $C \subseteq \mathcal{C}$.*

Proof. It is clear that $\langle\!\langle\bar{r}_{|C}\rangle\!\rangle = \langle\!\langle\bar{r}\rangle\!\rangle_{|C}$. Let us consider $c_1 \neq c_2$. If $c_1 \in C$ and $c_2 \in C$ (i.e. $c_1 \cdot c_2 \in C^2$) then $\langle\!\langle\bar{q}\rangle\!\rangle_{|C^2}(c_1 \cdot c_2) = \langle\!\langle\bar{q}\rangle\!\rangle(c_1 \cdot c_2) = \bar{q}(c_1) \sqcap \bar{q}(c_2) = \bar{q}_{|C}(c_1) \sqcap \bar{q}_{|C}(c_2) = \langle\!\langle\bar{q}_{|C}\rangle\!\rangle(c_1 \cdot c_2)$. Otherwise, either $\bar{q}_{|C}(c_1)$ or $\bar{q}_{|C}(c_2)$ is $\perp$. Then $\langle\!\langle\bar{q}_{|C}\rangle\!\rangle(c_1 \cdot c_2) = \bar{q}_{|C}(c_1) \sqcap \bar{q}_{|C}(c_2) = \perp = \langle\!\langle\bar{q}\rangle\!\rangle_{|C^2}(c_1 \cdot c_2)$. We conclude that $\langle\!\langle\bar{q}\rangle\!\rangle_{|C^2} = \langle\!\langle\bar{q}_{|C}\rangle\!\rangle$. □

Lemma 8.4 (Soundness) *If $\Gamma \vdash e : \bar{t}$ then $\langle\!\langle\Gamma\rangle\!\rangle \vdash e : \langle\!\langle\bar{t}\rangle\!\rangle$.*

Proof. By induction on the derivation of $\Gamma \vdash e : \bar{t}$.

∘ *Case* $\textsc{Case}^{(1)}$. e is e' case $[\{c\} : x_1 \mapsto e_1 \mid \mathcal{C}\setminus\{c\} : x_2 \mapsto e_2]$. Among $\textsc{Case}^{(1)}$'s premises we find $\Gamma \vdash e' : \bar{r}^{\bar{q}}$, $\Gamma[x_1 \mapsto (\bar{r}^{\bar{q}})_{|\{c\}}] \vdash e_1 : \bar{t}$, $\Gamma[x_2 \mapsto (\bar{r}^{\bar{q}})_{|\mathcal{C}\setminus\{c\}}] \vdash e_2 : \bar{t}$ and $\bar{q}(c) \lhd \bar{t}$. Let $r^q = \langle\!\langle\bar{r}\rangle\!\rangle^{\langle\!\langle\bar{q}\rangle\!\rangle} = \langle\!\langle\bar{r}^{\bar{q}}\rangle\!\rangle$. By induction hypothesis and lemma 8.3, we obtain $\langle\!\langle\Gamma\rangle\!\rangle \vdash e' : \langle\!\langle\bar{r}\rangle\!\rangle^{\langle\!\langle\bar{q}\rangle\!\rangle}$, $\langle\!\langle\Gamma\rangle\!\rangle[x_1 \mapsto (\langle\!\langle\bar{r}\rangle\!\rangle^{\langle\!\langle\bar{q}\rangle\!\rangle})_{|\{c\}}] \vdash e_1 : \langle\!\langle\bar{t}\rangle\!\rangle$, $\Gamma[x_2 \mapsto (\langle\!\langle\bar{r}\rangle\!\rangle^{\langle\!\langle\bar{q}\rangle\!\rangle})_{|\mathcal{C}\setminus\{c\}}] \vdash e_2 : \langle\!\langle\bar{t}\rangle\!\rangle$. Moreover $\langle\!\langle\bar{q}\rangle\!\rangle(\{c\}) = \langle\!\langle\bar{q}\rangle\!\rangle(\mathcal{C}\setminus\{c\}) = \sqcup\{\bar{q}(c) \sqcap \bar{q}(c') \mid c' \neq c\}$. Then $\langle\!\langle\bar{q}\rangle\!\rangle(\{c\}) \leq \bar{q}(c)$ and $\langle\!\langle\bar{q}\rangle\!\rangle(\mathcal{C}\setminus\{c\}) \leq \bar{q}(c)$. By lemmas 8.2 and 4.2, $\bar{q}(c) \lhd \bar{t}$ yields $[\langle\!\langle\bar{q}\rangle\!\rangle(\{c\}), \langle\!\langle\bar{q}\rangle\!\rangle(\mathcal{C}\setminus\{c\})] \trianglelefteq [\langle\!\langle\bar{t}\rangle\!\rangle, \langle\!\langle\bar{t}\rangle\!\rangle] \leq \langle\!\langle\bar{t}\rangle\!\rangle$. Conclude by an instance of $\textsc{Case}$.

∘ *Case* $\textsc{Sub}^{(1)}$. By lemma 8.1 and the induction hypothesis.

Other cases are immediate. □

$$\textsc{e-Bind}^{(1)}\quad \frac{\Gamma \Vdash e_1 : \bar{\mathbf{r}}_1^{\bar{\mathbf{q}}_1} \qquad \Gamma[x \mapsto \overline{\mathsf{Pre}}\, \bar{\mathbf{r}}_1(\eta)] \Vdash e_2 : \bar{\mathbf{r}}_2^{\bar{\mathbf{q}}_2} \qquad (\bar{\mathbf{r}}_1^{\bar{\mathbf{q}}_1})_{|\mathcal{E}} \leq \bar{\mathbf{r}}^{\bar{\mathbf{q}}} \qquad \bar{\mathbf{q}}(\eta) \lhd \bar{\mathbf{r}}^{\bar{\mathbf{q}}}}{\Gamma \Vdash \mathsf{bind}\ x = e_1\ \mathsf{in}\ e_2 : \bar{\mathbf{r}}^{\bar{\mathbf{q}}}}$$

$$\textsc{e-Handle}^{(1)}\quad \frac{\Gamma \Vdash e_1 : \bar{\mathbf{r}}_1^{\bar{\mathbf{q}}_1} \qquad \Gamma[x \mapsto \overline{\mathsf{Pre}}\, \bar{\mathbf{r}}_1(\varepsilon)] \Vdash e_2 : \bar{\mathbf{r}}^{\bar{\mathbf{q}}} \qquad (\bar{\mathbf{r}}_1^{\bar{\mathbf{q}}_1})_{|\mathcal{C}\setminus\{\varepsilon\}} \leq \bar{\mathbf{r}}^{\bar{\mathbf{q}}} \qquad \bar{\mathbf{q}}(\varepsilon) \lhd \bar{\mathbf{r}}^{\bar{\mathbf{q}}}}{\Gamma \Vdash e_1\ \mathsf{handle}\ \varepsilon\, x \succ e_2 : \bar{\mathbf{r}}^{\bar{\mathbf{q}}}}$$

$$\textsc{e-HandleAll}^{(1)}\quad \frac{\Gamma \Vdash e_1 : \bar{\mathbf{r}}_1^{\bar{\mathbf{q}}_1} \qquad \Gamma[x \mapsto (\bar{\mathbf{r}}_1^{\bar{\mathbf{q}}_1})_{|\mathcal{E}}] \Vdash e_2 : \bar{\mathbf{r}}^{\bar{\mathbf{q}}} \qquad (\bar{\mathbf{r}}_1^{\bar{\mathbf{q}}_1})_{|\{\eta\}} \leq \bar{\mathbf{r}}^{\bar{\mathbf{q}}} \qquad \bar{\mathbf{q}}(\eta) \lhd \bar{\mathbf{r}}^{\bar{\mathbf{q}}}}{\Gamma \Vdash e_1\ \mathsf{handle}\ x \succ e_2 : \bar{\mathbf{r}}^{\bar{\mathbf{q}}}}$$

Figure 10. The type system for $\lambda_{\mathcal{E}}^{(1)}$

Theorem 8.1 (Non-interference) *Choose $\ell, h \in \mathcal{L}$ such that $h \not\leq \ell$. Let $h \lhd t$. Assume $(x \mapsto t) \vdash e : \mathsf{int}^{\ell}$, where e is a λ_+ expression. If $\vdash e_i : t$ and $e[x \Leftarrow e_i] \rightarrow^{\star} k_i$, for $i \in \{1, 2\}$, then $k_1 = k_2$.*

Proof. By lemmas 8.2, 8.4 and theorem 4.2. □

8.2 The system $\lambda_{\mathcal{E}}^{(1)}$

Using the same mechanism, it is possible to obtain the corresponding type system $\lambda_{\mathcal{E}}^{(1)}$ for $\lambda_{\mathcal{E}}$. Those of its rules that are different from $\lambda_{\mathcal{E}}$'s are given in figure 10.

$\lambda_{\mathcal{E}}^{(1)}$ provides a treatment of exceptions that is very similar to that of JFlow [6, 5], although the presentation is superficially different. Indeed, JFlow introduces a notion of *path labels*. Setting aside Java-specific features, paths in JFlow are $\underline{\mathsf{n}}$ (which represents normal termination) and names of classes that inherit from $\mathsf{Throwable}$, i.e. classes representing exceptions. This directly corresponds in our framework to η and the exception names $\varepsilon \in \mathcal{E}$, respectively. A path label X maps each path s to either the special constant $\underline{\emptyset}$, if the expression cannot terminate through the path s, or a security level approximating how much information will be obtained by observing that this path is the effective termination path. This is comparable to our alternatives Abs and $\mathsf{Pre}\,\ell$. The accuracy provided by the $\lhd$ constraint when all fields but one of a row are Abs is obtained in JFlow by a non-syntax-directed rule, called *single-path rule*, allowing $X[\underline{\mathsf{n}}]$ to be reset to $\underline{\emptyset}$ if all other paths are already mapped to $\underline{\emptyset}$ by X. (Because exception names are classes in Java and are therefore equipped with subtyping, this rule cannot be applied safely to non-$\underline{\mathsf{n}}$ paths. But, as noticed by Myers, if exceptions were not identified with classes, the single-path rule could be applied to exceptions too.)

If we constrain the field η of every row r of $\lambda_{\mathcal{E}}^{(1)}$ to be $\mathsf{Pre}\,*$, we obtain a treatment of exceptions similar to that proposed in [13]. Then, every lower-bound constraint on the η entry of a row $\bar{q}$ (generated by a $\lhd$ constraint) must also constrain some other field of the same row. As a result, it is possible to enforce the invariant that, for any row $\bar{\mathbf{q}}$, $\bar{\mathbf{q}}(\eta) = \sqcup\{\bar{\mathbf{q}}(\varepsilon) \mid \varepsilon \in \mathcal{E}\}$. The main interest of this policy lies in the fact that, because $\ell \lhd (\eta : \mathsf{Pre}\,t; \bar{\mathbf{r}})^{\bar{\mathbf{q}}}$ becomes equivalent to $\ell \lhd t \wedge \forall\varepsilon, (\bar{\mathbf{r}}(\varepsilon) \neq \mathsf{Abs} \Rightarrow \ell \leq \bar{\mathbf{q}}(\varepsilon))$, the system requires only a very simple form of conditional constraints.

9 Conclusion

It is an interesting question in what context this analysis would be useful. Because of the structure of security annotations involving matrices of levels, a type inference algorithm based on our framework is likely to produce very verbose type schemes. That is the reason why it seems difficult to use it as the basis for a generic secure programming language, as we aim at with MLIF [13]. Nevertheless such an implementation might be of interest for automated analysis of very sensitive (relatively to information flow) part of programs for which systems such as [5, 13] remain too approximative. Such a case may particularly arise in programming languages for devices with limited ressources, such as JavaCard, where integer constants are used as flags in order to represent different data in an unstructured manner.

Moreover, it seems possible to design a reasonably efficient algorithm addressing type inference and constraint solving for this system: we believe that this is mainly an implementation and proof issue.

Another topic of interest lies in adding mutable cells (a.k.a. references) to λ_+ and $\lambda_{\mathcal{E}}$. Obtaining a treatment of references similar to that of [13] (where reference types $t\ \mathsf{ref}^{\ell}$ have an invariant argument t describing the content of the cell and carry an external security annotation ℓ related to the reference's identity) is straightforward. This remains an orthogonal problem to the accurate treatment of union types. It mainly requires adding an extra security annotation pc on every typing judgement and on $\rightarrow$ types. Nevertheless, such a framework does not provide as fine-grained a treatment of information flow generated by side-effects as that for functional aspects of the language. For instance, one may rewrite our first example in an imperative style

$$\begin{array}{l} \mathsf{if}\ x\ \ \mathsf{then}\ (\mathsf{if}\ y\ \mathsf{then}\ t := A\ \mathsf{else}\ t := B) \\ \qquad\quad \mathsf{else}\ \ (\mathsf{if}\ z\ \mathsf{then}\ t := A\ \mathsf{else}\ t := D); \\ \mathsf{let}\ u = !\,t\ \mathsf{case}\ [A, B \mapsto 1 \mid D \mapsto 0] \end{array}$$

Then, the system would no longer be able to detect the absence of dependency between u and the value stored in $!\,t$

because in all the branches of this program the content of the cell t must have the same type (i.e. $(A, B, D : \mathsf{Pre}\,;*)^q$). Therefore, the three security levels of the matrix q will be constrained to exceed that of x, y and z. Thus, an interesting direction for further work consists in obtaining a fine-grained analysis of dependencies due to side-effects.

References

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

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

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

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

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

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

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

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

[9] J. Palsberg, M. Wand, and P. M. O'Keefe. Type inference with non-structural subtyping. *Formal Aspects of Computing*, 9:49–67, 1997. URL: `http://www.cs.purdue.edu/homes/palsberg/paper/fac97.ps.gz`.

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

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

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

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

[14] D. Rémy. Records and variants as a natural extension of ML. In *Proceedings of the Sixteenth Annual Symposium on Principles Of Programming Languages (POPL'89)*, pages 77–88, Austin, Texas, jan 1989.

[15] D. Rémy. *Algèbres Touffues. Application au Typage Polymorphe des Objets Enregistrements dans les Langages Fonctionnels*. Thèse de doctorat, Université de Paris 7, 1990. URL: `ftp://ftp.inria.fr/INRIA/Projects/cristal/Didier.Remy/these.ps.gz`.

[16] V. Simonet. Fine-grained information flow analysis for a λ-calculus with sum types. Full version. URL: `http://cristal.inria.fr/~simonet/publis/simonet-csfw-02-long.ps.gz`, Feb. 2002.

[17] M. Sulzmann. *A general framework for Hindley/Milner type systems with constraints*. PhD thesis, Yale University, Department of Computer Science, May 2000. URL: `http://www.cs.mu.oz.au/~sulzmann/publications/diss.ps.gz`.

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

[19] P. Wadler. Comprehending monads. *Mathematical Structures in Computer Science*, 2:461–493, 1992. URL: `http://cm.bell-labs.com/cm/cs/who/wadler/papers/monads/monads.ps.gz`.

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

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