ML Dependency Analysis for Assessors

Philippe Ayrault[1], Vincent Benayoun[2],
Catherine Dubois[3], and François Pessaux[4]

[1] Etersafe - Paris 6 - LIP6
[2] CNAM - CEDRIC
[3] ENSIIE - INRIA - CEDRIC
[4] ENSTA ParisTech - UEI

Abstract. Critical software needs to obtain an assessment before commissioning. This assessment is given after a long task of software analysis performed by assessors. They may be helped by tools, used interactively, to build models using information-flow analysis. Tools like SPARK-Ada exist for Ada subsets used for critical software. But some emergent languages such as those of the ML family lack such adapted tools. Providing similar tools for ML languages requires special attention on specific features such as higher-order functions and pattern-matching. This paper presents an information-flow analysis for such a language specifically designed according to the needs of assessors. This analysis can be parametrized to allow assessors getting a view of dependencies at several levels of abstraction and gives the basis for an efficient fault tolerance analysis.

1 Introduction

Software is used to control everyday life as well as high risk systems. While games on smartphones and recreational instant messengers can contain software failures without staking human lives or implying catastrophic financial consequences, aeronautic trajectory controllers or automatic train protections are critical software.

Critical software must pass a safety assessment before its commissioning, given by sworn assessors. They perform a precise analysis as required by standards of the involved domains (IEC-61508 [15] for general purposes, CENELEC-50128 [14] for railways, etc.). This analysis aims at discovering any lack leading to feared events. Prior to software development, a hazard analysis states all safety prescriptions and drives software specification construction. Software analysis must convince the assessor that the development satisfies all safety requirements expressed in this specification. Assessment is a huge and difficult task, which amounts between 10 and 15% of the total development costs, due to the growing number of critical functionalities assigned to software components. Assessors can be helped by software analysis tools, however they do not want to delegate the acceptance to a totally automatic and opaque tool, as they can be prosecuted in case of accident.

The assessment activity rests upon standard methodologies such as software *FMECA*[5] and *Fault Tree Analysis*, which are based on functional models. It mainly consists in exploiting some data flow analysis to build models, determine the impact of failures etc. Models construction especially relies on identification of the *real* inputs and outputs of a program or a software component (i.e. parameters, external entities – functions and constants – and side effects – accesses to values from terminals, files, sensors etc.) and *dependencies* between these inputs and outputs. Such models help recovering specifications of a software component, allowing to abstract it as a black box whose functional behaviour is simpler to manipulate. In the context of FMECA, the assessor has to study the effects resulting from the injection of failures on the functional behavior of a component. A fault injection is a modification of the value of an identifier (not necessarily erroneously), usually an input. As a consequence, we can also define a real input as an identifier "having an impact" on the component, i.e when it is modified (by a fault injection), the value of one of the outputs -at least- may change.

A last requirement in this kind of activity is to cope with abstraction. One may not be interested in the real dependencies of some components, either because we do not have their implementation or because they are considered "trusted" or out of scope. In such a case, we must be able to stop the analysis on such components, considering them as "terminal basic bricks" whose dependencies will be represented by the component's name. Such components will be "tagged", hence enabling a choice of the abstraction level. This choice is under the assessor's responsibility, taking benefits of his experience in the domain of the system (e.g. railway) and his knowledge of this particular instance of it (e.g. a subway) he got through the documentation of the system. Determining the target of analysis then consists of understanding which components are considered critical and which ones can be safely abstracted.

Programming languages of the ML family become to be used in critical software development. Hence they need tools for computer-aided certification, in particular dependency analysis tools. Some industrial projects are already written using ML languages. Jane Street Capital develops critical trading systems in OCaml [8] which deals with hundreds of millions of dollars everyday. The certified embedded-code generator SCADE is also written in OCaml [10]. Many other critical software are developed using ML languages such as the Goanna static analysis for safety critical C/C++ [6], or the LexiFi Apropos software platform for pricing and management of financial products [11].

This paper presents a static analysis for functional programming languages to compute dependencies based on assessor's needs, as recensed by one of the authors (P. Ayrault, an independent safety assessor (ISA) for railway domain) and informally presented in the previous paragraphs.

We first examine related works about information-flow analysis in functional languages in Section 2. In Section 3, the core language is introduced with its syntax and operational semantics. Then Section 4 formalizes the dependency analysis aiming to address the previously quoted needs and states its correctness.

[5] Failure Modes, Effects and Criticality Analysis

This static analysis is the first step towards a tool enabling specification and verification of dependencies of a software component written in a functional language (like Spark-Ada does for Ada). This tool should allow the user to specify the real inputs and outputs, their dependencies and should be able to verify that the code is correct with respect to these specifications. In Section 5, we illustrate this on a small program. Then we conclude and give some possible extensions.

2 Related Work

Tools for model construction, based on information-flow analysis, exist for subsets of imperative languages. For example SPARK-Ada [6] gives the dependencies between inputs and outputs of software components written in an Ada subset. Functional languages begin to be used in development of critical software and tools related to safety. Unfortunately there are only very few tools dedicated to those languages and they are not as specifically designed for safety as are the tools for imperative languages.

Several motivations have led to information-flow analysis. The first one was compiler optimization with slicing [17], binding-time analysis [4,9], call-tracking [16] etc. Currently, the most studied matter is probably data security [7,13] (including secrecy and integrity).

Information-flow analysis for higher-order programming languages has been a long-term study.

In [2] the authors proposed a data-flow analysis for a lambda calculus. The goal of their analysis was run-time optimization using caching of previously computed values (do not compute an expression a second time if only independent values have changed). In their language, any sub-expression can be annotated with a label. Their run-time analysis is obtained by extending the usual operational semantics of the lambda calculus with a rule dealing with these labels. Hence, the analysis consists in evaluating the whole expression which leads to a value containing some of the labels present in the original expression. If a label is not present in the value, it means that the corresponding sub-expression is unused to compute the value.

In [1] the authors show that several kinds of information-flow analysis can be based on the same dependency calculus providing a general framework for secure information-flow analysis, binding-time analysis, slicing and call-tracking.

In [12] the analysis proposed by Abadi and al. in [2] is done statically using a simple translation and a standard type system. The authors claim that combined with the work of [1] their static analysis can also be used for all kinds of dependency analysis addressed in [1]. Later, in [13], the authors proposed a data-flow analysis technique for a lambda-calculus extended with references and exceptions. A new approach based on a specific type system was proposed to fill the lacks of the previous approach.

[6] see http://libre.adacore.com/libre/tools/spark-gpl-edition/

This latter approach, used as a basis to build the Flow Caml language, uses a lattice representing security levels in order to ensure secrecy properties. As previously shown in [1], this kind of framework can be used for other kinds of dependency analysis by using different lattices. From our experiments, Flow Caml can be used to compute dependencies by "cheating", "misusing" it and leads to pretty good results up to a certain point. The basic idea is to set one different security level per identifier to be traced, using a flat security lattice. Inference then mimics dependency analysis as long as no explicit type constraint with level annotation is present and as long as no references are used. This last point is the most blocking since invariance of references requires equality of levels. Hence, having different levels on the traced identifiers will make the type-checker rejecting such programs. In one sense, we need to be more conservative since we do not want to reject programs accepted by the "regular compiler". We could say that Flow Caml accurately works but at some point, not in the direction we need. Moreover, the type system can't "ignore" — i.e. really consider as opaque — dependencies for some identifiers the assessor doesn't matter about. This is an important point since considering identifiers "abstract" means that the safety analysis must not take them into account for some reasons and allows reducing "noise" (flooding) polluting valuable dependency information. Assigning a convenient level on functions to trace them is to be more difficult since explicit type annotations get quickly complex. We succeeded for first order functions, even polymorphic, but it is unclear how to handle higher order.

All of these approaches try to provide an automatic information-flow analysis. This automation is valuable for the different purposes they address. However, assessors need a parametric tool that can be used interactively in order to help them building their own models.

3 The Core Language

3.1 Syntax

The language we consider is an extended λ-calculus with the most common features of functional programming languages: constants, sums, `let`-binding, recursion and pattern-matching. It comprises the functional kernel of CamlLight.

Sums are built from constant and parametrized constructors. Parametrized constructors only contain one parameter without loss of expressiveness since constructors with several parameters can be encoded using one parameter being a pair. In the same way, tuples can be encoded as nested pairs.

The considered pattern-matching only contains two branches where the second pattern is a variable, but it does not affect the language expressiveness since a pattern-matching with more branches can be encoded by nested pattern-matching with two branches. Conditional expressions are encoded as a pattern-matching against two constants constructors `True` and `False.`

Expressions

e	$::= i$	Integer constant
	$\mid x$	Identifier
	$\mid C \mid D(e) \mid (e_1,\, e_2)$	Sum constructors and pair
	$\mid \texttt{let}\ x = e_1\ \texttt{in}\ e_2$	`let`-binding
	$\mid \texttt{fun}\ x \rightarrow e$	Abstraction (function)
	$\mid \texttt{rec}\ f\ x \rightarrow e$	Recursive abstraction
	$\mid e_1\ e_2$	Application
	$\mid \texttt{match}\ e_1\ \texttt{with}\ p \rightarrow e_2 \mid x \rightarrow e_3$	Pattern-matching

Patterns

p	$::= x$	Pattern variable
	$\mid i$	Integer constant pattern
	$\mid C \mid D(p)$	Sum constructor patterns
	$\mid (p_1, p_2)$	Pair pattern

Bindings

s	$::= \texttt{let}\ x = e$	*Toplevel* `let` binding

Programs (i.e. sequence of bindings)

prg	$::= \epsilon$
	$\mid s\ ;;\ prg$

3.2 Operational Semantics

This section shortly presents the operational semantics of the language. This is a standard call-by-value semantics evaluating an expression to a value:

Values

v	$::= i$	Integer
	$\mid C \mid D(v)$	Constant and parametrized constructors
	$\mid (v_1, v_2)$	Pair
	$\mid (\xi,\ \texttt{fun}\ x \rightarrow e)$	Simple closure (functional value)
	$\mid (\xi,\ \texttt{rec}\ f\ x \rightarrow e)$	Recursive closure

where ξ is an evaluation environment mapping identifiers onto values.

Evaluation rules will be given for the 3 kinds of constructs of the language, i.e. expressions, patterns and definitions (which are mostly expressions bound at top-level and must be processed specifically since they largely contribute to the representation of dependencies, the bound identifiers being those appearing in the "visible" result). These rules are similar to those commonly presented for other functional languages.

Evaluation of expressions

$$\xi \vdash C \rightsquigarrow C \qquad\qquad \frac{\xi \vdash e \rightsquigarrow v}{\xi \vdash D(e) \rightsquigarrow D(v)}$$

Sum expressions are evaluated to the value corresponding to the constructor as introduced in its hosting type definition, embedding the values coming from the evaluation of their arguments if there are some.

$$\xi \vdash \texttt{fun}\ id \rightarrow e \rightsquigarrow (\xi, \texttt{fun}\ id \rightarrow e) \qquad \xi \vdash \texttt{rec}\ f\ id \rightarrow e \rightsquigarrow (\xi, \texttt{rec}\ f\ id \rightarrow e)$$

Evaluation of a function expression leads to a "functional value" called a *closure*, embedding the current evaluation environment, the name of the formal parameter and the expression representing the body of this function.

$$\frac{\xi \vdash e_1 \rightsquigarrow (\xi_1,\ \texttt{fun}\ id \rightarrow e) \qquad \xi \vdash e_2 \rightsquigarrow v_2 \qquad (id, v_2) \oplus \xi_1 \vdash e \rightsquigarrow v_3}{\xi \vdash e_1\ \ e_2 \rightsquigarrow v_3}$$

$$\frac{\begin{array}{l}\xi \vdash e_1 \rightsquigarrow (\xi_1,\ \texttt{rec}\ f\ \ id \rightarrow e) \qquad \xi \vdash e_2 \rightsquigarrow v_2 \\ (id, v_2) \oplus (f, (\xi_1,\ \texttt{rec}\ f\ \ id \rightarrow e)) \oplus \xi_1 \vdash e \rightsquigarrow v_3\end{array}}{\xi \vdash e_1\ \ e_2 \rightsquigarrow v_3}$$

Evaluation of an application processes the argument expression, leading to a value v_2 and the functional expression leading to a closure. The environment of this closure is then extended (operator $\oplus$) by binding the formal parameter *id* to the value v_2 (and the function name to its closure in case of recursive function), then the body of the function gets evaluated in this local environment.

$$\frac{\xi \vdash e_1 \rightsquigarrow v_1 \qquad (x, v_1) \oplus \xi \vdash e_2 \rightsquigarrow v_2}{\xi \vdash \texttt{let}\ x\ \texttt{=}\ e_1\ \ \texttt{in}\ e_2 \rightsquigarrow v_2}$$

The `let in` construct locally binds the identifier x to a value: the definition expression e_1 is first evaluated, let v_1 be its value, then the expression e_2 is evaluated in the extended environment binding the identifier x to the value v_1.

$$\frac{\xi \vdash e_1 \rightsquigarrow v_1 \qquad p, v_1 \vdash_p \xi_1 \qquad \xi_1 \oplus \xi \vdash e_2 \rightsquigarrow v_2}{\xi \vdash \texttt{match}\ e_1\ \texttt{with}\ p\ \texttt{->}\ e_2 \mid x\ \texttt{->}\ e_3 \rightsquigarrow v_2}$$

$$\frac{\xi \vdash e_1 \rightsquigarrow v_1 \qquad \forall \xi_1 . \neg(p, v_1 \vdash_p \xi_1) \qquad (x, v_1) \oplus \xi \vdash e_3 \rightsquigarrow v_3}{\xi \vdash \texttt{match}\ e_1\ \texttt{with}\ p\ \texttt{->}\ e_2 \mid x\ \texttt{->}\ e_3 \rightsquigarrow v_3}$$

Evaluation of a pattern-matching first processes the matched expression, leading to a value v_1. We then need an extra operation, $\vdash_p$ verifying that a value is "compatible" with a pattern and if so returns the bindings of pattern variables to add to the environment before the evaluation of the right-side expression of a matching case. The rules describing this operation follow below.

If the first case of the pattern-matching has a pattern, p, compatible with e_1, then its induced pattern variables bindings are added to the current evaluation environment in which the right-side part expression of the case, e_2, is evaluated resulting in the whole expression value.

If the first case does not match, then the second one, x, will always match, extending the evaluation environment by binding x to v in which the right-side part expression is evaluated.

Matching values against patterns

$$x, v \vdash_p (x, v) \qquad C, C \vdash_p \emptyset \qquad \frac{p, v \vdash_p \xi}{D(p), D(v) \vdash_p \xi} \qquad \frac{p_1, v_1 \vdash_p \xi_1 \qquad p_2, v_2 \vdash_p \xi_2}{(p_1, p_2), (v_1, v_2) \vdash_p \xi_1 \oplus \xi_2}$$

Evaluation of top-level definitions

$$\frac{\xi \vdash e \rightsquigarrow v}{\xi \vdash \texttt{let}\ x = e \rightsquigarrow_s (x, v) \oplus \xi}$$

Evaluation of programs (successive top-level definitions)

$$\xi_0 \vdash \epsilon \rightsquigarrow_{prg} \xi_0 \qquad \frac{\xi_0 \vdash s \rightsquigarrow_s \xi_1 \quad \xi_1 \vdash prg \rightsquigarrow_{prg} \xi_2}{\xi_0 \vdash s\ ;;\ prg \rightsquigarrow_{prg} \xi_2}$$

4 Dependency Analysis

The dependency analysis is described as an alternative semantics (like an abstract interpretation [5]) called "dependency semantics". The dependency semantics performs a dependency inference by evaluating a program to a value (called dependency term) describing the dependencies of this program. This inference is the first step to verify dependencies stated by user specifications. The dependencies are expressed as structured values containing the "abstract" top-level identifiers having an impact on the evaluated program. External primitives or "trusted" components are given explicit dependencies since their definitions are either unavailable or unnecessary to analyze.

It is important to note that this analysis is more accurate than a "simple `grep`-like" command since the fact that an identifier does not appear in an expression does not mean that this expression does not depend on it. Moreover, the form of the dependency terms allows to keep trace of the structure of dependent data.

4.1 Dependency Semantics

The dependency on an identifier expression is simply described by the name of the identifier. A dependency term may be empty: a basic constant expression does not depend on anything. Constant and parametrized sum constructors as well as pairs allow representing the structure of the expression's value. Union of dependencies allows combining dependencies of several sub-expressions.

Values: Dependency terms

$\Delta ::= \bot$	Empty dependency
$\mid\ x$	Identifier (abstract or built-in)
$\mid\ C \mid D(\Delta)$	Constant and parametrized sum constructors
$\mid\ (\Delta, \Delta)$	Pair
$\mid\ < \lambda x.e, \Gamma >$	Closure
$\mid\ \Delta \otimes \Delta$	Union of dependencies

Dependency semantics: Evaluation of expressions

The following judgments use a dependency environment Γ binding free identifiers to their corresponding dependency term.

(*const*) $$\frac{}{\mathit{\Gamma} \vdash i \triangleright_e \bot}$$

A constant expression evaluates to an empty dependency: this means that the expression does not involve (depend on) any top-level identifier.

(*ident*) $$\frac{}{\mathit{\Gamma} \vdash x \triangleright_e \mathit{\Gamma}(x)}$$

Evaluation of an identifier returns the value bound to this identifier in the environment. This value can be either the identifier itself, if the top-level definition of this identifier has been “tagged” as abstract, or another dependency term corresponding to its definition otherwise.

(*pair*) $$\frac{\mathit{\Gamma} \vdash e_1 \triangleright_e \mathit{\Delta}_1 \quad \mathit{\Gamma} \vdash e_2 \triangleright_e \mathit{\Delta}_2}{\mathit{\Gamma} \vdash (e_1, e_2) \triangleright_e (\mathit{\Delta}_1, \mathit{\Delta}_2)}$$

A pair expression evaluates to a pair value in which each component is the evaluation of the corresponding component in the expression. This allows keeping trace of the dependency of each component separately. If only one of its components is used afterwards in the program, only the corresponding dependencies will be taken into account and not a significantly larger over-approximation.

(*constr-1*) $$\frac{}{\mathit{\Gamma} \vdash C \triangleright_e C}$$ (*constr-2*) $$\frac{\mathit{\Gamma} \vdash e \triangleright_e \mathit{\Delta}}{\mathit{\Gamma} \vdash D(e) \triangleright_e D(\mathit{\Delta})}$$

Sum constructor expressions are evaluated to their dependency term counterpart, embedding the dependencies of their argument if they have some.

(*lambda*) $$\frac{}{\mathit{\Gamma} \vdash \lambda x.e \triangleright_e < \lambda x.e, \mathit{\Gamma} >}$$

As in the operational semantics, functions evaluate to closures containing the **definition** of the function and the current environment. This rule has been chosen in order to provide a high level of precision, as opposed to the choice of analysing functions' bodies to synthesize a term containing only partial information.

(*recursion*) $$\frac{\mathit{\Delta} = \texttt{DepsOfFreeVars}(\lambda x.e, \mathit{\Gamma})}{\mathit{\Gamma} \vdash \texttt{rec}\ f\ x.e \triangleright_e < \lambda x.e, (f, \mathit{\Delta}) \oplus \mathit{\Gamma} >}$$

where $\texttt{DepsOfFreeVars}(\lambda x.e, \mathit{\Gamma})$ is the union ($\otimes$) of the dependency terms of all free variables present in the function definition. $\mathit{\Gamma}$ contains the binding of each free variable to its dependency term, which has already been computed.

Recursive functions also evaluate to closures. As opposed to the operational semantics, there is no need of recursive closure. However to avoid forgetting dependencies caused by effective recursive calls, one must admit that recursive calls possibly depend on all free variables present in the body of the function. This is mandatory as the following example shows:

```
let abstr_id = ... ;;   (* Assumed tagged ‘‘abstract’’. *)
let rec f x = if x then f false else abstr_id ;;
let v = f true ;;
```

where omitting `abstr_id` which is free in the body of `f` is wrong since calling `f` with `true` involves a dependency during the recursive call.

Note that this approach is correct because **recursive calls** do not introduce extra dependencies except those coming from the evaluation of parameters and the body.

(*apply-concr*)

$$\frac{\Gamma \vdash e_1 \rhd_e \Delta_1 \quad \Delta_1 =< \lambda x.e, \Gamma_x > \quad \Gamma \vdash e_2 \rhd_e \Delta_2 \quad (x, \Delta_2) \oplus \Gamma_x \vdash e \rhd_e \Delta}{\Gamma \vdash e_1\ e_2 \rhd_e \Delta}$$

(*apply-abstr*) $$\frac{\Gamma \vdash e_1 \rhd_e \Delta_1 \quad \Delta_1 \neq< \lambda x.e, \Gamma_x > \quad \Gamma \vdash e_2 \rhd_e \Delta_2}{\Gamma \vdash e_1\ e_2 \rhd_e \Delta_1 \otimes \Delta_2}$$

There are two cases for the evaluation of an application. Either the function evaluates to a closure (see Rule *apply-concr*), hence the evaluation takes the same form as in the operational semantics or the function evaluates to something else (Rule *apply-abstr*) and then the dependencies are the approximation of the dependencies of both left and right expressions. This happens when a top-level function is tagged as "abstract", (in this case the environment binds the identifier of the function to the dependency term reduced to this identifier, hence not revealing the dependency term of its body) or when the applied expression is not yet determined.

(*let-in*) $$\frac{\Gamma \vdash e_1 \rhd_e \Delta_1 \quad (x, \Delta_1) \oplus \Gamma \vdash e_2 \rhd_e \Delta_2}{\Gamma \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 \rhd_e \Delta_2}$$

The dependency semantics of a let binding mimics its operational semantics.

(*match-static-1*)

$$\frac{\Gamma \vdash e \rhd_e \Delta \quad \mathit{statically\_known}(\Delta) \quad p, \Delta \rhd_m \Gamma_1 \quad \Gamma_1 \oplus \Gamma \vdash e_1 \rhd_e \Delta_1}{\Gamma \vdash \texttt{match}\ e\ \texttt{with}\ p \to e_1 \mid x \to e_2 \rhd_e \Delta_1}$$

(*match-static-2*)

$$\frac{\Gamma \vdash e \rhd_e \Delta \quad \mathit{statically\_known}(\Delta) \quad \forall \Gamma_1 . \neg(p, \Delta \rhd_m \Gamma_1) \quad (x, \Delta) \oplus \Gamma \vdash e_2 \rhd_e \Delta_2}{\Gamma \vdash \texttt{match}\ e\ \texttt{with}\ p \to e_1 \mid x \to e_2 \rhd_e \Delta_2}$$

During pattern-matching analysis, the structure of the matched value maybe statically known as expressed by the following predicate:

$$\mathit{statically\_known}(\Delta) \triangleq \Delta = C \vee \Delta = D(\_) \vee \Delta = (\_\ ,\ \_)$$

In this case, we can then benefit from this information to deduce which branch to follow during the analysis as described by the two previous rules. The rule (*match-static-1*) applies when the dependency term of the matched expression has the same structure as the pattern of the first branch. If the first pattern does not match, the rule (*match-static-2*) applies. The $\rhd_m$ operation (whose rules are given below) computes the bindings induced by the pattern and the value and that must be added to the environment when analyzing each right-side part of the cases.

(*match*)

$$\frac{\begin{array}{c}\Gamma \vdash e \rhd_e \Delta \qquad \neg(\mathit{statically\_known}(\Delta)) \\ p, \Delta \rhd_p \Gamma_1 \qquad \Gamma_1 \oplus \Gamma \vdash e_1 \rhd_e \Delta_1 \qquad x, \Delta \rhd_p \Gamma_2 \qquad \Gamma_2 \oplus \Gamma \vdash e_2 \rhd_e \Delta_2\end{array}}{\Gamma \vdash \texttt{match}\ e\ \texttt{with}\ p \to e_1 \mid x \to e_2 \rhd_e (\Delta \otimes \Delta_1 \otimes \Delta_2)}$$

When the branch is not statically known, an over-approximation of the real dependencies is done. The dependencies of the matched value are computed. Then the expression of each branch is analyzed in the environment extended with the variables bound in the pattern (rules for $\rhd_p$ are given below). Finally the dependency of the whole expression is computed as the union of these three dependencies. Here, the structure of the value is lost because of the union of heterogeneous values.

Evaluation of definitions and programs

A top-level definition can be tagged as being either "abstract" or "concrete". This choice influences which dependency term will be bound to the identifier being defined, hence allows selecting the level of abstraction, granularity of the analysis. In this way, the abstraction is directly controlled by the user and not through modifications of the analysis algorithm itself.

(*let-concr*)
$$\frac{\Gamma \vdash e \rhd_e \Delta}{\Gamma \vdash \texttt{let}\ x = e \rhd_s (x, \Delta) \oplus \Gamma}$$

If the identifier is tagged as "concrete", the rule *(let-contr)* applies and the identifier is bound to the dependency term resulting from the evaluation of its definition through the dependency semantics.

(*let-abstr*)
$$\frac{}{\Gamma \vdash \texttt{let}\ x = e \rhd_s (x, x) \oplus \Gamma}$$

Conversely, if the identifier is tagged as "abstract", the rule *(let-abstr)* applies and the definition will bind the identifier to its name in the environment. This means that any use of this identifier in the program will be considered as depending only on this identifier and its own dependency term will be hidden.

$$\Gamma \vdash \epsilon \rhd_{prg} \Gamma \qquad\qquad (\mathit{prog})\ \frac{\Gamma_0 \vdash s \rhd_s \Gamma_1 \qquad \Gamma_1 \vdash \mathit{prg} \rhd_{prg} \Gamma_2}{\Gamma_0 \vdash s\ ;;\ \mathit{prg} \rhd_{prg} \Gamma_2}$$

The dependency semantics of a program mimics its operational semantics.

Evaluation of non-statically known pattern-matching

$$\frac{}{i, \Delta \rhd_p \emptyset} \qquad \frac{}{C, \Delta \rhd_p \emptyset} \qquad \frac{p, \Delta \rhd_p \Gamma}{D(p), \Delta \rhd_p \Gamma}$$

$$\frac{}{x, \Delta \rhd_p (x, \Delta)} \qquad \frac{p_1, \Delta \rhd_p \Gamma_1 \quad p_2, \Delta \rhd_p \Gamma_2}{(p_1, p_2), \Delta \rhd_p \Gamma_1 \oplus \Gamma_2}$$

A pattern variable matches any dependency term and binds the corresponding identifier to the dependency term in the returned environment. Pattern-matching of pairs is done using an over-approximation because the matched dependency term is not necessary a pair (values considered as "abstract" can be matched).

Evaluation of statically known pattern-matching

$$\frac{}{x, \Delta \rhd_m (x, \Delta)} \qquad\qquad \frac{}{C, C \rhd_m \emptyset}$$

$$\frac{p_1, \Delta_1 \rhd_p \Gamma_1 \quad p_2, \Delta_2 \rhd_p \Gamma_2}{(p_1, p_2), (\Delta_1, \Delta_2) \rhd_m \Gamma_1 \oplus \Gamma_2} \qquad\qquad \frac{p, \Delta \rhd_p \Gamma}{D(p), D(\Delta) \rhd_m \Gamma}$$

4.2 Proof of Correctness

The goal of the analysis is helping assessors to certify a given level of fault tolerance. In order to achieve this goal, the analysis must satisfy a correctness property based on the notion of fault tolerance. Hence, the notion of fault injection and its impact on the execution of a program are formalized and serve as a basis to express and prove a theorem of correctness.

Formalizing the notion of fault injection requires definitions of a reference environment (coming from the evaluation of the initial program) and an impact environment (from the evaluation after a fault injection). Similarity between those evaluation environments and the dependency environment (obtained from the dependency evaluation of the program) is defined below and serves as a basis to construct the proof of correctness of the analysis.

Definition 1 (Similar environments).
For a program $P = \texttt{let}\ x_1 = e_1, \ldots, \texttt{let}\ x_n = e_n$ *and an evaluation environment* $\xi_n = (x_n, v_n), \ldots, (x_1, v_1), \xi_\bullet$ *(where* $\xi_\bullet$ *is the environment corresponding to the free identifiers of the program), a dependency environment* Γ_n *is said similar to* ξ_n *if they share the same structure (i.e.* $\Gamma_n = (x_n, \Delta_n), \ldots, (x_1, \Delta_1), \Gamma_\bullet$*) and if for any identifier* x_i*, the value* v_i *(resp. the dependency term* Δ_i*) corresponds to the operational (resp. dependency) evaluation of* e_i *in the environment* ξ_{i-1} *(resp.* Γ_{i-1}*).*

Definition 2 (Fault injection on x_i).
Let $P = \texttt{let}\ x_1 = e_1, \ldots, \texttt{let}\ x_n = e_n$ *be a program. Injecting a fault in* (P, x_i) *at rank* $i < n$ *consists of building two environments* ξ_{ref} *(reference environment) and* ξ_{impact} *(impact environment) in the following way:*

1. ξ_{ref} *is the environment built during the evaluation of the program* P*.*
2. *If* v_i *is the value bound by the evaluation of the expression* e_i *(bound to* x_i *in the environment* ξ_{ref}*), then choose a value* v *different from* v_i*.*
3. *Build the environment* ξ_{impact} *by evaluating the rest of the program assuming that* x_i *is bound to the value* v *(instead of* v_i*):*
   - *for j = 1 ... i-1,* $\xi_{impact}(x_j) = \xi_{ref}(x_j)$*.*
   - $\xi_{impact}(x_i) = v$*, fault injection on the identifier* x_i *by bypassing evaluation of expression* e_i *and binding* x_i *to* v *in the environment*
   - *for j = i+1 ... n,* $\xi_{impact}(x_j) = v_j$ *with* v_j *being the result of the evaluation of the expression* e_j *: (*$\xi_{impact_{j-1}} \vdash e_j \rightsquigarrow v_j$*).*

Definition 3 (Impact of a fault injection).
Using the previous notations, given a program P *and a reference environment* ξ_{ref} *we say that an expression* e *is impacted by the fault injection if there exist two different values* v *and* v' *and an impact environment* ξ_{impact} *obtained by a fault injection in* (P, x_i)*, such that* $\xi_{ref} \vdash e \rightsquigarrow v$ *and* $\xi_{impact} \vdash e \rightsquigarrow v'$*.*

Theorem 1 (Correctness of the dependency analysis).
Let P *be a program,* e *an expression of this program and* x *an identifier defined at top-level in* P*. Assuming that* x *is the only identifier tagged as "abstract" in* P*, if* $\Gamma \vdash e \rhd_e \Delta$ *and* x *does not appear in* Δ *then* e *is not impacted by any fault injection in* (P, x)*.*

Sketch of the proof. A reference evaluation environment is built by evaluating the program P through the operational semantics. An impact environment is obtained by a fault injection on x. Then a dependency environment is built by evaluating the program P through the dependency semantics. We then prove that the dependency and the evaluation environments are similar (according to the definition above).

A proof by induction on the dependency evaluation of the program states that in each case, the value of e is the same in the reference and the impact environments. For more details, a complete proof has been published in [3].

5 An Example

We now present a simple but relevant program to show how our dependency analysis could be used in practice. In the following, we display both top-level definitions and the corresponding results of the dependency analysis. We consider that the initial dependency environment contains some primitives like `+, -, abs, assert, fst, snd` with their usual meanings. This sample code depicts a simple voter system where 3 inputs are compared together with a given tolerance. The output is the most represented value and a validity flag telling if a given minimal number of inputs agreeing together has been reached. Obviously we are interested in the dependencies on the inputs and the two fixed parameters of the system: the threshold and minimal number of agreeing inputs. Inputs may be coming from the external environment, but for the analysis, since they are tagged "abstract", they must be given each a dummy value.

The first five definitions introduce "abstract" identifiers. Their bodies being constants, their dependencies are $\perp$. However, since they are tagged "abstract", identifiers are bound to a dependency denoting their name. Subsequent definitions bind functions, leading to dependencies being closures embedding the current environment. The interesting part mostly arises in the last definition where all the defined functions are applied, leading to a non-functional result. This result shows two kinds of information. First, we remark that the structure of the result is kept, i.e. is a pair, since the toplevel structure is (Δ_1, Δ_2). Deeper structures (of Δ_1 and Δ_2) also exhibit the pair construct but are combined by $\otimes$, hence revealing that approximations of the analysis led to the loss of the knowledge of the real structure at this point. As long as a dependency term is not a union ($\otimes$), the analysis ensures that this term reflects the structure of the real value issued by effective computation. Conversely, apparition of $\otimes$ dependencies stops guarantying this property. This means that in some cases, dependencies will still show all "abstract" identifiers an expression depends on, but won't accurately show which part of the expression depends on which identifiers. The second interesting information is that, since the structure is kept in this case, the first component of the result does not depend on *_min_quorum* whereas the second does.

```
(* The tagged ``abstract'' values, i.e. those to be traced. *)
(* Tolerance and minimal number of agreeing inputs. *)
let _threshold = 2 ;;        $\rhd_e \perp$    $\Gamma_1 = (\text{"\_threshold"}, \_threshold) \oplus \Gamma$
let _min_quorum = 2 ;;       $\rhd_e \perp$    $\Gamma_2 = (\text{"\_min\_quorum"}, \_min\_quorum) \oplus \Gamma_1$

(* ``Dummy'' values set on inputs to perform the effective analysis. *)
let _sensor1 = 42 ;;         $\rhd_e \perp$    $\Gamma_3 = (\text{"\_sensor1"}, \_sensor1) \oplus \Gamma_2$
let _sensor2 = 45 ;;         $\rhd_e \perp$    $\Gamma_4 = \ldots$
let _sensor3 = 42 ;;         $\rhd_e \perp$    $\Gamma_5 = \ldots$

(* A comparison function modulo the threshold. *)
let eq v1 v2 =
  let delta = (v1 - v2) in
  (abs delta) <= _threshold ;;          $\rhd_e < \lambda\ v1.\lambda\ v2.\texttt{let}\ delta = \ldots, \Gamma_5 >$

(* A validity check function ensuring that the minimum
   number of agreeing inputs is reached. *)
let valid num = num >= _min_quorum ;;   $\rhd_e < \lambda\ num.(num\ \geq \_min\_quorum), \Gamma_6 >$

(* The vote function returning the most represented value
   and the number of inputs agreeing on this value. *)
let vote in1 in2 in3 =
 let cmp1 = eq in1 in2 in
 let cmp2 = eq in2 in3 in
 let cmp3 = eq in3 in1 in
 match (cmp1, cmp2, cmp3) with
 | (false, false, false) -> (0, 4)
 | (true, true, false) | (true, false, true) | (false, true, true) ->
     assert false
 | (true, true, true) -> (in1, 3)
 | (true, _, _) -> (in1, 2)
 | (_, true, _) -> (in2, 2)
 | (_, _, true) -> (in3, 2) ;;
$\rhd_e < \lambda in1.\lambda in2.\lambda in3.\texttt{let}\ cmp1 = eq\ in1\ in2\ \texttt{in}\ldots, \Gamma_7 >$

let main =
  let vot = vote _sensor1 _sensor2 _sensor3 in
  let response = fst vot in        (* Get first component of the pair. *)
  let accordance = snd vot in      (* Get the second one. *)
  let validity = valid accordance in
  (response, validity) ;;          $\rhd_e$
(
 (  $(\_threshold \otimes \_sensor2 \otimes \_sensor1, \_threshold \otimes \_sensor3 \otimes \_sensor2),$
    $\_threshold \otimes \_sensor1 \otimes \_sensor3$  )
 $\otimes(\_sensor1, \perp) \otimes (\_sensor3, \perp) \otimes (\_sensor2, \perp)$
,
 $\_min\_quorum \otimes (\_sensor2, \perp) \otimes (\_sensor3, \perp) \otimes (\_sensor1, \perp)$
 $\otimes$
 (  $(\_threshold \otimes \_sensor2 \otimes \_sensor1, \_threshold \otimes \_sensor3 \otimes \_sensor2),$
    $\_threshold \otimes \_sensor1 \otimes \_sensor3$  )
)
```

Future work addresses the problem of automatic verification of computed dependencies against those stated by the programmer. This implies being able to "understand" dependency terms. We explain here a few intuitions in this way. First of all, easy-to-read dependencies are those only containing identifiers or constructors. In this way, dependencies combined with a $\otimes$ must be seen as unions of simple sets where $\perp$ is the neutral element. Hence, $x_1 \otimes x_2 \otimes \perp$ means that dependency only implies identifiers x_1 and x_2. When dependencies show constructors, we must look at identifiers appearing inside them. For instance, $A(x_1) \otimes B(x_2)$ also represents $\{x_1; x_2\}$. Difficulties arise with closure dependencies since no application reduced the function. A possible approach is digging in the term, harvesting both occurrences of arguments and identifiers bound outside the function. This is a pretty crude approach but this still allows having information and in practice, we expect programs to be complete, i.e. with defined functions used, hence applied at some point. The loss of structure previously explained (see the code sample presentation) is also an issue since it makes more difficult checking the computed dependencies wihout structure against structured ones as the user could have stated them.

6 Conclusion and Further Work

Assessors perform huge tasks of manual software analysis. They may be assisted by automatic tools but they require tools specifically adapted for their needs and used interactively under their control. The dependency analysis presented in this paper has been implemented in a prototype (about 2000 lines of OCaml) and tested by an assessor on several examples such as a cruise control system. The adequacy of the analysis with higher-order functions has been demonstrated formally and experimentally. This tool answers a practical need of assessors by providing an information-flow analysis tool with a fine parametrization (using the tags abstract/concrete). This parametrization proved to be the backbone of the analysis.

Future work includes the development of the dependency verification tool as it was sketched on the example of Section 5. Next step is to produce a fine-grained analysis of pattern-matching based on the notion of execution paths. This more precise analysis would allow the assessor to recover or verify more accurately the dependencies between the real inputs/outputs, hence to get a finer model of the system under assessment.

Our analysis performs a kind of information-flow analysis sharing several aspects of the one done in Flow CAML. An extension of our analysis with notions present in Flow CAML (for instance security levels) may make our analysis able to check security properties along with the fault tolerance analysis.

Acknowledgement We would like to express our greatest thanks to Thérèse Hardin for her participation in the work presented in this paper.

References

1. M. Abadi, A. Banerjee, N. Heintze, and J. G. Riecke. A core calculus of dependency. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of programming languages*, POPL '99, pages 147–160, New York, USA, 1999. ACM.
2. M. Abadi, B. W. Lampson, and J.-J. Lévy. Analysis and caching of dependencies. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP'96)*, pages 83–91, 1996.
3. P. Ayrault. *Développement de logiciel critique en Focalize. Méthodologie et outils pour l'évaluation de conformité.* PhD thesis, Université Pierre et Marie Curie - LIP6, 2011.
4. C. Consel. Binding time analysis for high order untyped functional languages. In *Proceedings of the 1990 ACM conference on LISP and functional programming*, LFP '90, pages 264–272, New York, USA, 1990. ACM.

5. P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Conference Record of the Sixth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 269–282, San Antonio, Texas, 1979. ACM Press, New York, NY.
6. A. Fehnker, R. Huuck, P. Jayet, M. Lussenburg, and F. Rauch. Goanna: a static model checker. In *Proceedings of the 11th international workshop, FMICS 2006 and 5th international workshop, PDMC conference on Formal methods: Applications and technology*, FMICS'06/PDMC'06, pages 297–300, Berlin, Heidelberg, 2007. Springer-Verlag.
7. N. Heintze and J. G. Riecke. The slam calculus: Programming with secrecy and integrity. In *POPL*, pages 365–377, 1998.
8. Y. Minsky and S. Weeks. Caml trading &#8211; experiences with functional programming on wall street. *J. Funct. Program.*, 18(4):553–564, July 2008.
9. F. Nielson and R. H. Nielson. Automatic binding time analysis for a typed λ-calculus. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of programming languages*, POPL '88, pages 98–106, New York, USA, 1988. ACM.
10. B. Pagano, O. Andrieu, B. Canou, E. Chailloux, J.-L. Colaço, T. Moniot, and P. Wang. Certified development tools implementation in objective caml. In *Proceedings of the 10th international conference on Practical aspects of declarative languages*, PADL'08, pages 2–17, Berlin, Heidelberg, 2008. Springer-Verlag.
11. S. Peyton Jones, J.-M. Eber, and J. Seward. Composing contracts: an adventure in financial engineering (functional pearl). In *Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*, ICFP '00, pages 280–292, New York, NY, USA, 2000. ACM.
12. F. Pottier and S. Conchon. Information flow inference for free. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP'00)*, pages 46–57, Montréal, Canada, 2000.
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.
14. Standard Cenelec EN 50128. *Railway Applications - Communications, Signaling and Processing Systems - Software for Railway Control and Protection Systems*, 1999.
15. Standard IEC-61508, International Electrotechnical Commission. *Functional safety of electrical/electronic/programmable electronic safety-related systems*, 1998.
16. Y. M. Tang and P. Jouvelot. Effect systems with subtyping. In *Proceedings of the 1995 ACM SIGPLAN Symposium on Partial evaluation and semantics-based program manipulation*, PEPM '95, pages 45–53, New York, USA, 1995. ACM.
17. F. Tip. A survey of program slicing techniques. Technical report, Amsterdam, The Netherlands, 1994.