An Abstract Semantics for Inference of Types and Effects in a Multi-Tier Web Language

Letterio Galletta Giorgio Levi
Dipartimento di Informatica, Università di Pisa
{galletta, levi}di.unipi.it

Types-and-effects are type systems, which allow one to express general semantic properties and to statically reason about program's execution. They have been widely exploited to specify static analyses, for example to track computational side effects, exceptions and communications in concurrent programs. In this paper we adopt abstract interpretation techniques to reconstruct (following the Cousot's methodology) a types-and-effects system developed to handle security problems of a multi-tier web language. Our reconstruction allows us to show that this types-and-effects system is not sound with respect to the semantics of the language. In addition, we correct the soundness issues in the analysis and systematically construct a correct analyser.

1 Introduction

Types-and-effects systems are a powerful extension of type systems which allows one to express general semantic properties and to statically reason about program's execution. The underlying idea is to refine the type information so as to express further intensional or extensional properties of the semantics of the program: in practice, they compute the type of each program's sentence and an approximate (but sound) description of its run-time behavior. Since they are defined over the well understood theory of type systems, they are an intuitive framework for specifying and for developing static analyses. Such systems were originally introduced in [12] to statically track side effects in languages that mix functional and imperative feature. However, they have been employed to control many other kinds of computational effects and analyses, e.g. exceptions [18], region inference [22] and communications in concurrent programs [21]. Recently, they have been used in [2] to handle security issues in LINKS [4].

LINKS is a strict, typed, functional language for web applications. Its main feature is to be multi-tier, that is, it enables the developer to mix client, server and database source code by delegating the charge of code and data partitioning to the compiler: from a single source file the compiler generates code for the database back-end, for the web server and the client front-end, ensuring that all data is stored either in client or in database. In [2] Baltopoulos and Gordon have shown that storing unencrypted application data on the client opens LINKS to attacks that may expose secrets and modify control-flow and application data. In order to overcome these problems they have proposed a compilation strategy based on authenticated encryption[1] and a types-and-effects system to enforce programs to satisfy a particular class of integrity constraints (event-based assertions). This types-and-effects system formalizes source level reasoning about LINKS programs and allows them to prove security properties by inspection of the source code. For the definition of this system they have followed a methodology characterized by translating each LINKS expression to an expression of a concurrent λ-calculus with refinement types [3]. This translation hides the properties of the analysis, and does not guarantee the soundness with respect to the

[1] a combination of secrecy and integrity protection obtained by encrypting together data and its hash.

L. Kovacs, R.Pugliese, and F. Tiezzi (Eds.): Workshop on Automated Specification and Verification of Web Systems (WWV 2011)
EPTCS 61, 2011, pp. 81–95, doi:10.4204/EPTCS.61.6

semantics of the language. Hence, we decided to study the properties of this analysis by reconstructing it by abstract interpretation [11].

Abstract interpretation [6, 7, 8, 9] is a general theory for approximating the semantics of dynamic systems. The key idea behind abstract interpretation is that the description of the behavior of a system (at various levels of abstraction) is an approximation of its formal semantics. In static analysis this means that every property of a program can be observed in its semantics and computed as an approximation: the intuition is that the analysis can be systematically derived by throwing away superfluous information from the semantics. In practice, the approximated semantics (abstract semantics) is obtained from the standard one (called concrete) by substituting the actual (concrete) domain of computation and its basic semantic operations with abstract domain and abstract semantic operations, respectively. The basic idea is that the abstract domain is a representation of some properties of interest about concrete domain's values, while abstract operations simulate, over the abstract domain, the behavior of their concrete counterparts. Hence the abstract semantics computes the properties of interest and the analysis algorithm corresponds to evaluating programs over the abstract domain. Since the abstract domain is a sound approximation of the concrete one, the analysis algorithm is correct with respect to the semantics by construction.

Type systems (and corresponding type inference algorithms) have been reconstructed as a hierarchy of abstract interpretations by Cousot [5]. In order to reconstruct the types-and-effects analysis of LINKS we extend Cousot's methodology by defining an abstract domain able to express types augmented by effects. In this paper we give the following contributions:

- we demonstrate that the analysis defined by Baltopoulos and Gordon is not sound: in fact, the expression `get(Text( "Hello!" ))` is type-checked but it results in a run-time type error (Section 3)
- we show how to fix this unsoundness issue (Section 3)
- we systematically derive an abstract semantics which represents a correct analyser (we have implemented it in OCaml [16]) (Sections 4 and 5)

In the next sections we first will sketch the type-and-effect system proposed for LINKS (Section 2), then we describe the ideas and the methodology underlying our reconstruction.

2 Secure Compilation of LINKS

Standard web applications have a multi-tier architecture: user interface, application logic and data management are implemented over three different tiers. Each tier runs on a different computational environment (web browser, web server and database respectively) characterized by its own language and its data representation. This heterogeneity gives rise to the problem of *impedance mismatch* [19]: because each language has its own data type, data exchanged between tiers of same application have a different representation. This problem complicates the development of web applications because programmers need to define routines to interchange and convert data. To solve this problem a new class of web languages (multi-tiers languages) have has been developed. These languages allow programmers to blend server, client and database source code and provide automatic mechanisms for the partition of the application over tiers.

LINKS is a functional programming language for web applications that belongs to the class of multi-tiers languages. LINKS enables developers to mix client, server and database source code by delegating

the charge of code and data partitioning to the compiler: from a single source code the compiler generates code for the database back-end, for the server and for the client front-end.

In this way LINKS overcomes the problem of impedance mismatch by abstracting details of a single tier and by supporting an unified programming model similar to the one used for GUI applications. To realize this cross-tier programming model LINKS exploits the mechanism of the web continuation [23]. These continuations are implemented as closures (expression to be executed plus values of free variables) and are stored in HTML pages either as hidden fields of forms or as URL parameters. This approach gives rise to security risks since a malicious client may modify those closures to enforce unexpected computations on the server.

In particular, Baltopoulos and Gordon in [2] have demonstrated that the approach adopted by LINKS of storing unencrypted data on the client is not secure because an attacker may violate the data secrecy, the data integrity and the control-flow integrity of the application. To overtake these problems they have proposed a secure implementation of LINKS that includes a compilation strategy based on authenticated encryption to protect the closures held in the browser and a types-and-effects system to enable source level reasoning about security of web applications. This secure implementation has been formalized for TINYLINKS, a simple subset of LINKS.

TINYLINKS is a λ-calculus augmented with XML values for representing web pages and annotation expressions for expressing safety properties. Its syntax is shown in Figure 1. HTML pages are values created by applying the data constructors `Text` and `Elem`: the first one represents simple text in HTML document, the second one a generic tag element. To express links and forms exists two ad-hoc data constructors that contains suspended expressions [2]. $\texttt{href}(\texttt{E})$ is a link that, when clicked, evaluates the expression `E`. $\texttt{form}([\texttt{l}_1, \ldots, \texttt{l}_n], \texttt{E})$ is a HTML form with a suspended computation (the expression `E`) which requires user input. The input is represented by labels $[\texttt{l}_1, \ldots, \texttt{l}_n]$ that will contain the values inserted in the input fields of the form. The evaluation of `href` and `form` can be accomplished by using the operators `get` and `post`, respectively [3]. The annotations `eventL` and `assertL` have no computational meaning. They allow us to annotate TINYLINKS programs with event-based assertions expressing suitable safety properties. An expression is safe if whenever an assertion `assertL` occurs in the execution, there exists a previous occurrence of an event `eventL`.

Baltopoulus and Gordon have defined a dependent types-and-effects system to verify that each expression of a program is safe. This system is specified by a set of inductively defined typing judgments. These judgments are of the form $\Gamma; \texttt{F} \vdash \texttt{E} \stackrel{exp}{\Rightarrow} \langle \_ : \texttt{T} \rangle \{ \texttt{F}' \}$, where Γ is the typing environment, `F` is the set of events which have occurred and are needed to safe evaluation of the expression `E` (precondition); `T` and $\texttt{F}'$ are, respectively, the type of value and the set of events (post-condition) yielded by the execution of `E`.

The typing rules for the operations `get` and `post`, for the annotations `event` and `assert` and for the function application are shown in Figure 2. Rule (T-Get) establishes that the type assigned to `get` is `xml` (that represent the type of a generic HTML tag) with empty effect, provided that `V` is another HTML tag. By (T-Post), the type of `post` expression is `xml` with empty effect, provided that the values associated with submission labels are strings and that `U` is a HTML tag. By (T-Event) `eventL` has type `unit` and effect `L`, provided that the values in the event `L` have a type. Rule (T-Assert) is similar to (T-Event) except that requires $\texttt{L} \in \texttt{F}$, that is the precondition of the judgment includes `L`. Rule (T-App) is typical for application and shows how the mechanism of the annotations works: the expression is type checked if only if the events in the precondition $\texttt{F}_1$ of the function have occurred in `F` with same values. The

[2] we can look at these values as special kinds of functional abstractions.

[3] we can look at these operations as special kinds of function application.

$\mathtt{f},\mathtt{y},\mathtt{x}$	Variables
$\mathtt{p}$	Predicates
$\mathtt{c} ::= \mathtt{Unit} \mid \mathtt{Zero} \mid \mathtt{Succ} \mid \mathtt{String}$ $\mid \mathtt{Nil} \mid \mathtt{Cons} \mid \mathtt{Tuple} \mid \mathtt{Elem} \mid \mathtt{Text}$	Data constructors
$\mathtt{g} ::= + \mid - \mid * \mid /$	Primitive operators
$\mathtt{L} ::= \mathtt{p}(\mathtt{V}_1, \ldots, \mathtt{V}_n)$	Events: a predicate and a list of values
$\mathtt{V},\mathtt{U} ::= \mathtt{x} \mid \mathtt{c}(\mathtt{V}_1, \ldots, \mathtt{V}_n) \mid \mathtt{href}(\mathtt{E})$ $\mid \lambda \mathtt{x}_1. \ldots, \mathtt{x}_n.\mathtt{E} \mid \mathtt{form}([\mathtt{l}_1, \ldots, \mathtt{l}_n], \mathtt{E})$	Values
$\mathtt{E} ::= \mathtt{V} \mid \mathtt{var}\,\mathtt{x} = \mathtt{E}_1; \mathtt{E}_2 \mid \mathtt{g}(\mathtt{E}_1, \mathtt{E}_2)$ $\mid \mathtt{V}(\mathtt{U}_1, \ldots, \mathtt{U}_n) \mid \mathtt{post}([\mathtt{l}_1 = \mathtt{V}_1, \ldots, \mathtt{l}_n = \mathtt{V}_n], \mathtt{U})$ $\mid \mathtt{get}(\mathtt{V}) \mid \mathtt{event}\,\mathtt{L} \mid \mathtt{assert}\,\mathtt{L}$ $\mathtt{switch}(\mathtt{V})\{$ $\mid \quad \mathtt{case}\,\mathtt{c}(\mathtt{x}_1, \ldots, \mathtt{x}_n) \to \mathtt{E}_1$ $\quad \_ \to \mathtt{E}_2$ $\}$	Expressions

Figure 1: Syntax of TINYLINKS

events generated after application include the ones of the post-condition of U.

We say that a web application E is safe if and only if there is a derivation within the types-and-effects system of the judgment $\emptyset;\emptyset \vdash \mathtt{E} \overset{exp}{\Leftarrow} \langle \_ : \mathtt{xml} \rangle \{\}$, meaning that E is a closed expression which requires no precondition and which yields a web page without generating further events.

After the definition of typing rules, the standard methodology requires to state and prove the soundness theorem which guarantees the validity of the analysis with respect to the semantics of the language. Baltopoulus and Gordon adopt a different approach by translating each TINYLINKS expression to an expression of a concurrent λ-calculus with *refinement types*. This translation hides the details and the properties of the defined types-and-effects system, in particular the soundness. For instance, the expression $\mathtt{get}(\mathtt{Text}(\text{"Hello!"}))$ is safe because a derivation exists for the judgment $\emptyset;\emptyset \vdash \mathtt{get}(\mathtt{Text}(\text{"Hello!"})) \overset{exp}{\Leftarrow} \langle \_ : \mathtt{xml} \rangle \{\ \}$. However, we will show in the next section that the proposed types-and-effects system is not sound because, even if this expression is type checked, its evaluation results in a run-time type error.

3 A Denotational Semantics for TINYLINKS

In this paper we adopt the approach described by Cousot in [5]. We define a denotational semantics for TINYLINKS, by considering it as an untyped λ-calculus. Furthermore, since we deal with effects, we explicitly consider assertions of events. To this purpose we introduce a special environment (*events environment*) which will store occurred events. The semantics of $\mathtt{assert}\,\mathtt{q}(\mathtt{V}_1, \ldots, \mathtt{V}_n)$ will require checking that q is bound in this environment to values $\mathtt{V}_1, \ldots, \mathtt{V}_n$. If this check succeeds, the evaluation yields a Unit value, otherwise a "sentinel" value indicating an error.

For the sake of simplicity, we restrict the values in an event to integers only. We will also assume that functions have a single argument and predicates in events are bound to a single value. Since we regard TINYLINKS an untyped λ-calculus, we define the semantics domain of values (*Eval*) as a recursive sum of cpos, by using the inverse limit construction described in [24]. Each element of this sum represents a specific class of values. For instance, $\mathscr{Z}$ is the set of integers; *U* and *S* are singletons of the unit value

$$\text{(T-Get)}\ \frac{\Gamma;\mathtt{F} \vdash \mathtt{V} \overset{val}{\Leftarrow} \mathtt{xml}}{\Gamma;\mathtt{F} \vdash \mathtt{get}(\mathtt{V}) \overset{exp}{\Rightarrow} \langle \_ : \mathtt{xml} \rangle \{\ \}}$$

$$\text{(T-Post)}\ \frac{\Gamma;\mathtt{F} \vdash \mathtt{V_i} \overset{val}{\Leftarrow} \mathtt{string} \quad \forall i \in \{1,\ldots,n\} \quad \Gamma;\mathtt{F} \vdash \mathtt{U} \overset{val}{\Leftarrow} \mathtt{xml}}{\Gamma;\mathtt{F} \vdash \mathtt{post}([(\mathtt{l_1} = \mathtt{V_1},\ldots,\mathtt{l_n} = \mathtt{V_n})],\mathtt{U}) \overset{exp}{\Rightarrow} \langle \_ : \mathtt{xml} \rangle \{\ \}}$$

$$\text{(T-Event)}\ \frac{\Gamma \vdash \diamond \qquad fv(\mathtt{F},\mathtt{L}) \subseteq dom(\Gamma) \qquad \mathtt{L} = \mathtt{p}(\mathtt{V_1},\ldots,\mathtt{V_n}) \quad \Gamma;\mathtt{F} \vdash \mathtt{V_i} \overset{val}{\Rightarrow} \mathtt{T_i} \quad \forall i \in \{1,\ldots,n\}}{\Gamma;\mathtt{F} \vdash \mathtt{event}\,\mathtt{L} \overset{exp}{\Rightarrow} \langle \_ : \mathtt{unit} \rangle \{\mathtt{L}\}}$$

$$\text{(T-Assert)}\ \frac{\Gamma \vdash \diamond \qquad fv(\mathtt{F},\mathtt{L}) \subseteq dom(\Gamma) \qquad \mathtt{L} \in \mathtt{F} \qquad \mathtt{L} = \mathtt{p}(\mathtt{V_1},\ldots,\mathtt{V_n}) \quad \Gamma;\mathtt{F} \vdash \mathtt{V_i} \overset{val}{\Rightarrow} \mathtt{T_i} \quad \forall i \in \{1,\ldots,n\}}{\Gamma;\mathtt{F} \vdash \mathtt{assert}\,\mathtt{L} \overset{exp}{\Rightarrow} \langle \_ : \mathtt{unit} \rangle \{\mathtt{L}\}}$$

$$\text{(T-App)}\ \frac{\Gamma;\mathtt{F} \vdash \mathtt{U} \overset{val}{\Rightarrow} \mathtt{T} \quad \mathtt{T} = \langle \mathtt{x_1} : \mathtt{T_1},\ldots,\mathtt{x_n} : \mathtt{T_n} \rangle \{\mathtt{F_1}\} \to \mathtt{T_2}\{\mathtt{F_2}\} \quad fv(\mathtt{T}) = \emptyset \quad \Gamma;\mathtt{F} \vdash \mathtt{V_i} \overset{val}{\Leftarrow} \mathtt{T_i} \quad \forall i \in \{1,\ldots,n\} \quad \mathtt{F_1}[\mathtt{V_1}/\mathtt{x_1}]\ldots[\mathtt{V_n}/\mathtt{x_n}] \subseteq \mathtt{F}}{\Gamma;\mathtt{F} \vdash \mathtt{U}(\mathtt{V_1},\ldots,\mathtt{V_n}) \overset{exp}{\Rightarrow} \mathtt{T_2}\{\mathtt{F_2}[\mathtt{V_1}/\mathtt{x_1}]\ldots[\mathtt{V_n}/\mathtt{x_n}]\}}$$

Figure 2: Some examples of rules specifying the type-and-effect system for the correspondences analysis.

and the error value [4]; $EEnv \to Eval \to (Eval \times EEnv)$, $EEnv \to (Eval \times EEnv)$ and $EEnv \to [Eval] \to (Eval \times EEnv)$ are the sets of the denotations of functions, links and forms, respectively.

The environment (Env) is a function from identifier (Ide) to values ($Eval$). The events environment ($EEnv$) maps predicates ($Pred$) to pairs formed by an element of $Dval$ and an element of $Mark$. $Dval$ denotes values which can occur in an event [5]. $Mark$ is the state of an event: E indicates that the event has occurred, EA that has occurred and has been asserted, A that has only been asserted.

We define two semantic functions $\mathscr{V}[\![-]\!] : \mathtt{VAL} \to Env \to EEnv \to Eval$ for values and $[\![-]\!] : \mathtt{EXP} \to Env \to EEnv \to (Eval \times EEnv)$ for expressions. The semantics of values is straightforward, because we only need to construct the corresponding denotation. Some examples of semantic equation are shown in Figure 3. In the definition, we use injections into $Eval$ (like $Unit$, $Href$, Fun), continuous semantic operators (like *bindList*) and a meta-language which includes:

- $if\, e_1\, then\, e_2\, else\, e_3$ (conditional);
- $let\, x = e_1\, in\, e_2$ as a cleaner notation for $((\lambda x.e_2)\, e_1)$;
- $let^\star\, x = e_1\, in\, e_2$ for $((\lambda x.e_2)^\star\, e_1)$;
- $case\, e_1\, of\, in_1(x_1) \to e_2\ \_ \to e_3$ for $[\lambda x_1.e_1, \lambda x_2.e_3, \ldots, \lambda x_2.e_3]$;
- $let\, (x_1\, x_2) = e_1\, in\, e_2$ for $let\, y = e_1\, in\, let\, x_1 = \pi_1(y)\, let\, x_2 = \pi_2(y)\, in\, e_2$;

where π_i, $\lfloor - \rfloor$, $\star$ and $[-,\ldots,-]$ are the standard operators for product, lifting and sum of cpos [26].

The semantics of expressions is similar to the one of the untyped λ-calculus. The most interesting cases of semantic equations are shown in Figure 4 and below we give some comments about them.

[4] this value is used to show a run-time type error

[5] in the following we will call them denotable values

$$\begin{aligned}
\mathscr{V}[\![\lambda \mathtt{x}.\mathtt{E}]\!]\rho\,\phi &= \lfloor Fun(\lambda\phi'.\lambda v.\,[\![\mathtt{E}]\!]\rho\,[v/x]\;\phi')\,\rfloor \\
\mathscr{V}[\![\mathtt{Href}(\mathtt{E})]\!]\rho\,\phi &= \lfloor Href(\lambda\phi'.[\![\mathtt{E}]\!]\rho\,\phi')\,\rfloor \\
\mathscr{V}[\![\mathtt{Form}(\mathtt{ll},\mathtt{E})]\!]\rho\,\phi &= \lfloor Form(\lambda\phi'.\lambda vl.let^{\star}\;\rho' = bindList(\rho, ll, vl)\;in\;[\![\mathtt{E}]\!]\rho'\,\phi')\,\rfloor
\end{aligned}$$

Figure 3: Examples of semantic equations for values.

The semantics of $\mathtt{get}(\mathtt{V})$ asks to evaluate $\mathtt{V}$; if the evaluation results into the denotation of a link ($Href(f)$), we evaluate the corresponding suspended expression (the closure f), otherwise we return an error value.

The semantics of $\mathtt{post}(\mathtt{VL},\mathtt{V})$ is similar: if the evaluation of $\mathtt{V}$ is a form ($Form(f)$) and the evaluation of $\mathtt{VL}$ is a list of strings, we return the result of the application of the functional value f to the denotation of $\mathtt{VL}$ and to the current events environment ϕ.

The semantics of $\mathtt{event\,q}(\mathtt{V})$ requires the evaluation of $\mathtt{V}$; if the produced value is an integer, we create a new binding for the predicate $\mathtt{q}$ in ϕ and return a unit value otherwise we raise an error.

The semantics of $\mathtt{assert\,q}(\mathtt{V})$ is similar, but requires the evaluation of $\mathtt{V}$ to be equal to the value bound to the predicate $\mathtt{q}$ in ϕ. In this case we update the state of the event in ϕ and return a unit value.

By using the semantic equation of `get` we prove that the evaluation of the expression $\mathtt{get}(\mathtt{Text}($ "$\mathtt{Hello!}$" $))$ results in a run-time type error (the value $\lfloor WrongValue()\,\rfloor$) because the denotation of $Text($"$Hello$!"$)$ is not a link. Although a link is an XML value, it is different from other XML values because it is a special kind of functional abstraction. Notice that the type-and-effect system proposed for TINYLINKS does not handle this special nature of links correctly, because it assigns the same type to the all XML values. Note that the same remark can be made for forms. Our above arguments demonstrate that the types-and-effects system of [2] is unsound because exists an expression which is type checked but its evaluation yields yet a run-time type error. We argue that the solution to this problem is to use a type system with subtypes. For the sake of simplicity, in our reconstruction we will not use subtypes, but we will instead define two ad-hoc types for forms and links which will handled so as have a sound analysis.

4 An Abstract Semantics for Inference of Types and Effects

Following the classical methodology of abstract interpretation, once we have defined a concrete semantics, we need to define a collecting semantics by extending $\mathscr{V}[\![-]\!]$ and $[\![-]\!]$ to the powerset.

The concrete semantics properties, which we are interested in, are the types and the event-based annotations. We need to define a suitable domain for both. One possibility is to define the abstract domain as the set of Hindley's monotypes (terms) with variables [15, 10, 5, 20, 13]. However, this is not possible, since types are annotated by effects. For example, a function type will have the form $\mathtt{T}_1\,\{\,\mathtt{F}_1\,\}\rightarrow\mathtt{T}_2\,\{\,\mathtt{F}_2\,\}$, where $\mathtt{F}_1$ are the events which have to be occurred before the function application, whereas $\mathtt{F}_2$ are the events which we can consider occurred afterwards. Hence, we need to define a domain of annotated types. The main problem is that the algebra of annotated terms is not free. In fact, two types can be identified even if their syntax is different. For example, the types $\mathtt{xml}\{\,\mathtt{q}(10),\mathtt{p}(1)\,\}\rightarrow\mathtt{xml}\{\,\}$ and $\mathtt{xml}\{\,\mathtt{p}(1),\mathtt{q}(10)\,\}\rightarrow\mathtt{xml}\{\,\}$ have a different representation, but they are equal because the effects $\{\,q(10),p(1)\,\}$ and $\{\,p(1),q(10)\,\}$ denote the same set. Therefore, we cannot use a syntactic unification algorithm [17] to solve equations between terms.

$$
\begin{aligned}
[\![\texttt{get(V)}]\!]\rho\,\phi &= let^\star\, v' = \mathscr{V}[\![\texttt{V}]\!]\rho\,\phi\ in\\
&\quad case\ v'\ of\\
&\quad\quad Href(f) \to f\,\phi\\
&\quad\quad \_ \to (\lfloor Wrong Value() \rfloor, \iota)\\
[\![\texttt{post(VL, V)}]\!]\rho\,\phi &= let^\star\, v' = \mathscr{V}[\![\texttt{V}]\!]\rho\,\phi\ in\\
&\quad let^\star\, v_2 = checkStringList(map(\lambda x.\, \mathscr{V}[\![\texttt{x}]\!]\rho\,\phi)\,\texttt{VL})\ in\\
&\quad case\ v'\ of\\
&\quad\quad Form(f) \to case\ v_2\ of\\
&\quad\quad\quad\quad\quad V(vl) \to f\,vl\,\phi\\
&\quad\quad\quad\quad\quad \_ \to (\lfloor Wrong Value() \rfloor, \iota)\\
&\quad\quad \_ \to (\lfloor Wrong Value() \rfloor, \iota)\\
[\![\texttt{event q(V)}]\!]\rho\,\phi &= let^\star\, d = evalToDval(\mathscr{V}[\![\texttt{V}]\!]\rho\,\phi)\ in\\
&\quad if\ d = dint(n)\ then\\
&\quad\quad (\lfloor Unit() \rfloor, \phi\,[(d, E)/\texttt{q}])\\
&\quad else\\
&\quad\quad (\lfloor Wrong Value() \rfloor, \iota)\\
[\![\texttt{assert q(V)}]\!]\rho\,\phi &= let^\star\, ev = evalToDval(\mathscr{V}[\![\texttt{V}]\!]\rho\,\phi)\ in\\
&\quad let\ (ev', m) = \phi\,\texttt{q}\\
&\quad if\ ev = ev'\ then\\
&\quad\quad (\lfloor Unit() \rfloor, \phi\,[(ev', EA)/\texttt{q}])\\
&\quad else\\
&\quad\quad (\lfloor Wrong Value() \rfloor, \iota)
\end{aligned}
$$

Figure 4: Examples of semantic equations for expressions.

One solution would be to use an algorithm for unifying terms in non-free algebras (semantic unification). Such algorithms do exist [1], but they are not usable in practice.

Our reconstruction does not rely on semantic unification but on another approach described in [22]. This approach exploits special annotated types (simple types), where annotations are replaced by variables (annotation variables), whose values have to satisfy some constraint. For example, the annotated type $\texttt{xml}\{\,\texttt{q}(10), \texttt{p}(1)\,\} \to \texttt{xml}\{\,\}$ becomes $\texttt{xml}(\alpha) \to \texttt{xml}(\beta)$, where α and β are the minimal annotations A and B which satisfy the constraints $A \supseteq \{\, q(10),\, p(1)\,\}$ and $B \supseteq \{\,\}$, respectively. The algebra of simple types is free. Hence, the introduction of a new kind of variable in terms requires a simple variation of the unification algorithm: an annotation variable unifies with another annotation variable only.

However, this solution is not completely adequate to define an abstract domain for the properties which we are concerned with, because the effects depend on the values. Hence we need to include them in the abstract domain. Since events in the precondition and post-condition of a function type may depend on the value bound to a formal parameter we need to remember it. We then introduce in the set of terms another kind of variables, called identifier variables. Identifier variables are handled by simple

modification of the unification algorithm: an identifier variable unifies with another identifier variable only.

The domain of abstract values will contain also substitutions as in [13]. The role of substitutions can be explained as follows. At some point in the evaluation of the abstract semantics (for example, in the semantics of function abstraction), we will introduce new type variables, with the meaning "any possible type". During the evaluation (for example, of the function body), this information will be subject to instantiations, computed by unifications and represented as an idempotent substitution. Since the abstract semantic evaluation functions are defined by structural recursion, the easiest way to provide the instantiation information to the caller is to include it in the returned value.

Although we have now all necessary information for defining an adequate abstract domain, there is a problem concerning the representation of effects in the constraints. Intuitively we can simply represent them by using a set of pairs, where the first component is the predicate and the second one is the denotable value. The problem is in partial order, since we should consider both set inclusion and the relative precision of denotable values. We can achieve this by using power domains [14, 24]. We use a different approach: we define an effect as a function from predicates to denotable values (we will name it correspondence function). We can then represent constraints by splitting them in two parts: the first part is a set of pairs (annotation variable, predicate) and the second one is a correspondence function.

Let V_t be a countable set of type variables, V_a be a countable set of annotation variables, Ide be a countable set of identifier variable ($V_t \cap V_a \cap Ide = \emptyset$) and $\Sigma = \{unit : 0, int : 0, string : 0, xml : 1, link : 1, form : 1, list : 1, fun : 5\} \cup \{tuple_n : n \mid n \geq 2\}$ be a numerable set of function symbol, T_s is the set of terms with variables $V_t \cup V_a \cup Ide$ modulo renaming, ordered by the inverse instance relation. It is worth noting that we have introduced two new types $form$ and $link$ in order to solve the problem relating forms and links which we described in Section 3. Furthermore we will use annotation variables in *xml*, $link$, $form$ and fun only; in fun there are two annotation variables representing the precondition and the post-condition respectively. We further assume that the first argument of fun is an identifier variable. We obtain $TypeS$ by lifting T_s with idempotent substitutions [13] and by adding a new bottom element $Notype$.

As we described above, the first part of a constraint is a pair (annotation variable, predicate): (δ, q) means that the predicate q is in the effect represented by the variable δ. We use inverse inclusion as partial order: if C_1 is included in C_2, then C_1 has less information than C_2, hence, its value is less precise. Let V_a be the set of annotation variables and $Pred$ be the set of predicates. We define $Constr = \wp(V_a \times Pred)$. The second part of a constraint is a correspondence function whose domain is $TPred = Pred \to Dval$ ordered by using the dual of usual partial order. We assume that cb: $\wp(TPred) \to TPred$ is the glb operator and ζ is the bottom element.

The domain of abstract values is $TypeA = TypeS \times Dval \times Constr \times TPred$. In the following, we will denote by $Error$ the bottom element of this domain.

The domain of abstract environment (type environment) is $AEnv = Ide \to TypeA$. We are now in the position to define our abstract domains $AV = AEnv \to EEnv \to TypeA$ for values and $AE = AEnv \to EEnv \to (TypeA \times EEnv)$ for expressions.

To relate the abstract domain to the concrete one we need to define a Galois connection. In [11] we formally built this connection in in various steps, by using properly defined representation functions [22] and propositions.

Some examples of abstract semantic equations are shown in Figures 5, 6 and 7. In these definitions, we assume to have a function *mgu*, which, given a set of term equations, computes a solution by using the unification algorithm. If there exists a solution, it returns the unifier $S(\theta)$; otherwise, it returns F to denote failure. The set of equations is denoted by $\{t_1 = t'_1, \ldots, t_n = t'_n\}$. Since idempotent substitutions

$$
\begin{array}{l}
\mathscr{V}[\![\texttt{href(E)}]\!]^a\rho\,\phi\,= \\
\quad \gamma \in V_a \quad fresh \\
\quad let\,((ts,\_,C,\,f),\phi') = [\![\texttt{E}]\!]^a\rho\,\phi\,in \\
\quad let\,A = assert(\phi',\phi)\,in \\
\quad let\,E\, = event(\phi',\phi)\,in \\
\quad if\,E = \emptyset \wedge ts \neq NoType\,then \\
\quad\quad case\,mgu(\{\,ts.t = xml(\gamma)\,\} \cup ts.\theta)\,of \\
\quad\quad\quad S(\theta) \rightarrow let\,C' = C \cup \{\,(\gamma,q) \mid q \in A\,\} \\
\quad\quad\quad\quad let\;f' = cb\{\;f, eenvToT\,pred(diff(\phi',\phi))\;\}\;in \\
\quad\quad\quad\quad ((\theta(link(\gamma)),\theta), nodval, \theta(C'), \theta(f')) \\
\quad\quad\quad \_ \rightarrow Error \\
\quad else \\
\quad\quad Error \\
\mathscr{V}[\![\lambda\texttt{x.E}]\!]^a\rho\,\phi\,= \\
\quad \alpha \in V_t \quad \gamma_1,\gamma_2 \in V_a \quad fresh \quad \varepsilon \quad identity\;substituition \\
\quad let\,((ts,\_,C_1,\,f_1),\phi') = [\![\texttt{E}]\!]^a\rho\,[((\alpha,\varepsilon),var(x),\emptyset,\zeta)/x]\,\phi\,in \\
\quad if\,ts \neq NoType\,then \\
\quad\quad let\,\phi_d = \theta(\phi) \\
\quad\quad let\,C' = \{\;(\gamma_1,q) \mid q \in assert(\phi',\phi_d)\;\}\;in \\
\quad\quad let\,C'' = \{\;(\gamma_2,q) \mid q \in event(\phi',\phi_d)\;\}\;in \\
\quad\quad let\;f_2 = eenvToT\,pred(diff(\phi',\phi_d))\,in \\
\quad\quad\quad ((\theta(fun(x,\alpha,\gamma_1,ts.t,\gamma_2)),\theta), \\
\quad\quad\quad\quad nodval, \theta(C_1 \cup C' \cup C''), \theta(cb\{\,f_1,\,f_2\,\})) \\
\quad else \\
\quad\quad Error
\end{array}
$$

Figure 5: The abstract semantics of links and functional abstractions.

are isomorphic to solved form equations, we will use $\{t_1 = t'_1, \ldots, t_n = t'_n\} \cup \theta$ to refer the union of equations in $\{t_1 = t'_1, \ldots, t_n = t'_n\}$ and equations defined by θ. For the sake of simplicity, the components of the elements of the domain *TypeS*, will be identified by a notation similar to the one used to access the fields of a structure in an imperative language. Given $ts = (t', \theta') \in TypeS$, then $ts.t = t'$ and $ts.\theta = \theta'$.

Given an element C of *Constr* and a substitution θ, we will denote by $\theta(C) = \{(\theta(\delta), l) \mid (\delta, l) \in C\}$ the pair obtained by applying θ to all the annotation variables in C.

Given a correspondence function $f \in TPred$ and a substitution θ, we define $\theta(f) = \lambda q.\theta(f\,q)$, where if $d \neq var(x)$ for some x then $\theta(d) = d$.

Furthermore we assume that for $f \in TPred$ and $C \in Constr$ $f \downarrow C$ and $f \leftarrow C$ are the correspondence functions achieved by removing from f the predicates occurring and not occurring in C respectively; that

$$
\begin{array}{l}
[\![\texttt{get(V)}]\!]^a \rho\, \phi = \\
\quad \gamma \in V_a \quad fresh \\
\quad let\ (ts, d, C, f) = \mathscr{V}[\![\texttt{V}]\!]^a \rho\, \phi\ in \\
\quad if\ ts \neq NoType\ then \\
\quad\quad case\ mgu(\{\, ts.t = link(\gamma) \,\} \cup ts.\theta)\ of \\
\quad\quad\quad S(\theta) \rightarrow let\ C' = \{\, (\theta(\gamma), q) \in \theta(C) \,\}\ in \\
\quad\quad\quad\quad if\ check(\theta(f \leftarrow C'), \phi)\ then \\
\quad\quad\quad\quad\quad (((\theta(xml(\gamma)), \theta), \\
\quad\quad\quad\quad\quad\quad\quad nodval, \theta(C) \setminus C', \theta(f \downarrow C)), \phi) \\
\quad\quad\quad\quad else \\
\quad\quad\quad\quad\quad (Error, \iota) \\
\quad\quad\quad \_ \rightarrow (Error, \iota) \\
\quad\quad else \\
\quad\quad\quad (Error, \iota) \\
[\![\texttt{E}_1\, \texttt{E}_2]\!]^a \rho\, \phi = \\
\quad x \in Ide\ \alpha_1 \in V_t\ \gamma_1, \gamma_2 \in V_a \quad fresh \\
\quad let\ ((ts_1, \_, C_1, f_1), \phi_1) = [\![\texttt{E}_1]\!]^a \rho\, \phi \\
\quad let\ ((ts_2, d_2, C_2, f_2), \phi_2) = [\![\texttt{E}_2]\!]^a \rho\, \phi_1 \\
\quad if\ ts_1 \neq NoType \wedge ts_2 \neq NoType\ then \\
\quad\quad case\ mgu(\{\, ts_1.t = fun(x, \alpha, \gamma_1, ts_2.t, \gamma_2) \,\} \cup \\
\quad\quad\quad\quad\quad \cup ts_1.\theta \cup ts_2.\theta)\ of \\
\quad\quad\quad S(\theta) \rightarrow let\ C' = \{\, (\delta, q) \in \theta(C_1) \mid \delta \in prvar(\theta(ts_1.t)) \,\}\ in \\
\quad\quad\quad\quad\quad let\ C'' = \{\, (\delta, q) \in \theta(C_1) \mid \delta \in psvar(\theta(ts_1.t)) \,\}\ in \\
\quad\quad\quad\quad\quad let\ f_1' = \theta(f_1)[\theta(x), d_2]\ in \\
\quad\quad\quad\quad\quad if\ check(\theta(f_1' \leftarrow C'), \phi_2)\ then \\
\quad\quad\quad\quad\quad\quad (((\theta(ts_2.t), \theta), \top, \theta(C_1 \cup C_2) \setminus (C' \cup C'), \\
\quad\quad\quad\quad\quad\quad\quad\quad cb\{\, \theta(f_1) \downarrow (C' \cup C''), \theta(f_2) \,\}), incl(\phi_2, (\theta(f_1') \leftarrow C'')))) \\
\quad\quad\quad\quad\quad else \\
\quad\quad\quad\quad\quad\quad (Error, \iota) \\
\quad\quad\quad \_ \rightarrow (Error, \iota) \\
\quad\quad else \\
\quad\quad\quad (Error, \iota)
\end{array}
$$

Figure 6: The abstract semantics of `get` expression and function application.

$f[x, d]$ for $\texttt{x} \in \texttt{Ide}$, $d \in DVal$ and $f \in TPred$ is the correspondence function achieved by binding d to all predicates which are bound to $var(x)$ in f; that given a $f \in TPred$ and $\phi \in EEnv$ the function $check(f, \phi)$

returns *true* if the events represented by f have been occurred in ϕ, $false$ otherwise; that for $\phi \in EEnv$ $eenvToTPred(\phi)$ is the correspondence function achieved from ϕ; that given ϕ_1,ϕ_2 $assert(\phi_2, \phi_1)$ is the set of predicates of events asserted in ϕ_2 but not in ϕ_1, that $event(\phi_2, \phi_1)$ is the set of predicates of events generated in ϕ_2 but not in ϕ_1 and that $diff(\phi_2, \phi_1)$ is the events environment which contains the events of ϕ_2 which are not in ϕ_1 and the events of ϕ_1 which changed their value or state in ϕ_2. Furthermore we assume that, given $t \in T_s$, $prvar(t)$ and $psvar(t)$ denote the set of annotation variables of t for preconditions and post-conditions, respectively.

$$
\begin{array}{l}
[\![\texttt{assert q(V)}]\!]^a \rho\, \phi\ = \\
\quad let\ (ts, d, C, f) = \mathscr{V}[\![\texttt{V}]\!]^a \rho\, \phi\ in \\
\quad if\ ts \neq NoType \wedge (d = nint(n) \vee d = var(x))\ then \\
\quad\quad case\ mgu(\{\, ts.t = int \,\} \cup ts.\theta)\ of \\
\quad\quad\quad S(\theta) \rightarrow if\ q \notin dom(\phi) \vee \pi_1(\phi(q)) = d\ then \\
\quad\quad\quad\quad\quad\quad (((unit, \theta), nodval, \theta(C), \theta(f)), \phi\,[(d, A)/q]) \\
\quad\quad\quad\quad\quad else \\
\quad\quad\quad\quad\quad\quad (Error, \iota) \\
\quad\quad\quad \_ \rightarrow (Error, \iota) \\
\quad else \\
\quad\quad (Error, \iota) \\
[\![\texttt{event q(V)}]\!]^a \rho\, \phi\ = \\
\quad let\ (ts, d, C, f) = \mathscr{V}[\![\texttt{V}]\!]^a \rho\, \phi\ in \\
\quad if\ ts \neq NoType \wedge (d = nint(n) \vee d = var(x))\ then \\
\quad\quad case\ mgu(\{\, ts.t = int \,\} \cup ts.\theta)\ of \\
\quad\quad\quad S(\theta) \rightarrow if\ q \notin dom(\phi) \vee \phi(q) = (d, T)\ then \\
\quad\quad\quad\quad\quad\quad (((unit, \theta), nodval, \theta(C), \theta(f)), \phi\,[(d, E)/q]) \\
\quad\quad\quad\quad\quad else \\
\quad\quad\quad\quad\quad\quad (Error, \iota) \\
\quad\quad\quad \_ \rightarrow (Error, \iota) \\
\quad else \\
\quad\quad (Error, \iota)
\end{array}
$$

for some $n \in \mathscr{Z}$, $x \in Ide$ and where $T \in \{\, E, EA \,\}$

Figure 7: The abstract semantics of `event` and `assert` annotations.

The semantics of links consists in the evaluation of the expression `E`. If in this evaluation no errors ($ts \neq NoType$) and no new events (this is required by the rule described in Section 2) occur, then we check that the computed value has type *xml*. Since in our reconstruction *xml*, *link* and $form$ are different

types without any relation, this check rejects all the expressions which return a value of type *link* or $form$. Although this behavior may seem too restrictive, because it rejects some legal expressions like `href(href(Text("Hello")))`, it guarantees us safety and simplicity in the management of these different and unrelated types. If this check has success, we return an abstract value where the simple type is *link* and the constraint is risen by properly extending the result of the evaluation of `E`.

The semantics of forms is similar. We evaluate `E` in a type environment where the labels `ll` are bound to the abstract value with simple type *string* and constraint empty and we return an abstract value where the simple type is $form$.

The semantics of functional abstraction consists in the evaluation of the body `E` in a type environment, where the formal parameter `x` is bound to a generic type. If in this evaluation no errors occur, we compute the events which are included in the precondition (represented by C' and f_2) and in the post-condition (represented by C'' and f_2). We return an abstract value where the simple type is obtained by applying the substitution $ts.\theta$ to the functional type $(fun(x, \alpha, \gamma_1, ts.t, \gamma_2))$ and the constraint is obtained by combining C with C' and C'' and f with f_2.

The semantics of `get` requires the evaluation of `V` to be successful and yields a value of type *link*. If the preconditions are satisfied, that is if they are in ϕ and have occurred before, we construct an abstract value where the simple type is *xml* and the constraint is obtained from the one returned by the evaluation of `V` by removing the information about preconditions. The pair which is returned has in the first component this abstract value and in the second one the events environment ϕ. This is correct because the semantics of `href` guarantees that no new events have occurred during the evaluation of the suspended expression.

The semantics of `post` is similar except that we ask that the elements of list `VL` are strings and that the value yielded by the evaluation of `V` has type $form$.

In the semantics of function application we evaluate the sub-expressions $\texttt{E}_1$ and $\texttt{E}_2$: if both evaluations do not produce errors, we check that the simple type of $\texttt{E}_1$ is a function type where the argument has the simple type of $\texttt{E}_2$ and that the precondition of function is satisfied in the events environment ϕ_2, obtained from evaluating both the sub-expressions. In order to perform this last check, we substitute the denotable value bound to `x` in f_1 by the one returned by the evaluation of $\texttt{E}_2$. Then, by using the function *check*, we ask that the events required by the function body are in ϕ_2. If we succeed, we construct an abstract value where the simple type is $\theta(\alpha)$ and the constraint is obtained by composing those returned by the evaluation of the sub-expressions, where the events of preconditions and post-conditions are removed. We return a pair composed by this abstract value and by the events environment ϕ_2 extended with the events of the post-condition of the function.

The semantics of `assert` consists in the evaluation of `V`. If it yields an abstract value whose simple type is *int* and whose denotable value is a specific integer or a specific identifier, we check that there is in ϕ at most the same event which we are generating. In this way we are sure that it is impossible to change the value bound to a predicate. If this check has success, we build an abstract value where the simple type is *unit* and the constraint is the one returned by the evaluation of `V`. This abstract value is the first component of returned pair; the second component consists of the events environment ϕ extended with the new event.

The semantics of `event` is similar except that we ask that, if the event is in ϕ, then its state has to be either E or EA.

5 Implementation and Examples

Both the concrete and the abstract semantics have been implemented as OCaml [16] programs. The language provides a feature, the mechanism of functors, which allows us to have a unique semantic function (realised by the functor `Semantics`), parametrized with respect to the primitive operations and the semantic domain. We can thus construct the concrete semantics interpreter, which executes programs, and the abstract interpreter, which analyzes programs in terms of types and effects, by instantiating the same functor `Semantics`.

Programs are represented in abstract syntax, although, for the sake of simplicity, we will use in the following LINKS-like syntax. For example, the expression

```
fun buy(value, dbpass) {
  var _ = assert PriceIs(value);
    Text("Hello")
}
```

defines a function which requires that the event `PriceIs(value)` has occurred and which returns an XML value. The result of its evaluation by the abstract semantics interpreter is

```
(type - :
 Function(_#value#var0_, Integer(), _annvar0_,
   Function(_#dbpass#var1_, _typevar1_, _annvar2_,
            Xml(_annvar4_), _annvar3_),
   _annvar1_)
 No_dval [(_annvar2_,PriceIs)] {PriceIs -> _#value#var0_}, {})
```

meaning that the computed type is a function type whose first argument has a type integer and the second one has type variable [6] where the precondition (represented by the annotation variable `_annvar2_`) includes the event composed by the predicate `PriceIs` and the value bound to the first formal parameter. If we give a value (for example 5) to the first parameter, the abstract semantics is

```
(type - :
 Function(_#dbpass#var3_, _typevar3_, _annvar7_,
          Xml(_annvar9_), _annvar8_)
 Unknown [(_annvar7_,PriceIs)] {PriceIs -> 5}, {})
```

that is the computed type is a specialization of that one computed for `buy` where the predicate `PriceIs` is bound to the value `5` in the precondition. The abstract semantics of the application of the function `buy` to `5` and `"a"` is an error

```
Exception: No_type "apply_fun: no preconditions"
```

because we are applying a function whose precondition is not satisfied.

6 Conclusions

We have described how to reconstruct a types-and-effects system, proposed to handle some security issues in LINKS, as an abstract interpretation of a denotational semantics which explicitly models the types and the effects. By our reconstruction we have precisely defined the relation between the semantics and the analysis, we have systematically constructed a correct analyser and we have shown that the proposed types-and-effects system was not sound. We have stressed that the unsoundness derived from the fact of

[6] since the `dbpass` parameter is not used in the body, the analyzer cannot compute a more precise type

considering forms and links as simple XML values forgetting their own differentiating features. In our reconstruction we have solved this problem by using two new specific types and we have managed them in ad-hoc manner. We plan to extend our reconstruction to consider a type system with sub-types so as to be able to manage links and forms in a more uniform and elegant way and to use additional values in the effects.

One advantage of abstract interpretation approach on the type system approach is that the analysis is directly derived from the semantics and is sound by construction. This forces one to tackle from the very beginning subtle problems such as the ones described in Section 3 that might only be revealed while trying to prove the soundness theorem following the type system approach. On the other hand we have shown that abstract interpretation can easily handle extensions of types, such as types and effects. There is only one example in the literature of an abstract interpretation reconstruction of a type and effect static analysis [25].

References

[1] F. Baader & J. H. Siekmann (1994): *Unification theory*. In Dov M. Gabbay, Christopher J. Hogger, J. A. Robinson & Jörg H. Siekmann, editors: *Handbook of Logic in Artificial Intelligence and Logic Programming (2)*. Oxford University Press, pp. 41–126.

[2] I. G. Baltopoulos & A. D. Gordon (2009): *Secure Compilation of a Multi-Tier Web Language*. In: *TLDI '09: Proceedings of the 4th international workshop on Types in language design and implementation*. ACM, New York, NY, USA, pp. 27–38, doi:10.1145/1481861.1481866.

[3] J. Bengtson, K. Bhargavan, C. Fournet, A. D. Gordon & S. Maffeis (2008): *Refinement Types for Secure Implementations*. In: *Proceedings of the 2008 21st IEEE Computer Security Foundations Symposium*. IEEE Computer Society, Washington, DC, USA, pp. 17–32, doi:10.1109/CSF.2008.27. Available at http://portal.acm.org/citation.cfm?id=1380848.1381243.

[4] Ezra Cooper, Sam Lindley, Philip Wadler & Jeremy Yallop (2007): *Links: Web Programming Without Tiers*. In Frank de Boer, Marcello Bonsangue, Susanne Graf & Willem-Paul de Roever, editors: *Formal Methods for Components and Objects. Lecture Notes in Computer Science* 4709, Springer Berlin / Heidelberg, pp. 266–296, doi:10.1007/978-3-540-74792-5_12.

[5] P. Cousot (1997): *Types as Abstract Interpretations, invited paper*. In: *Conference Record of the Twenty-fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM Press, New York, NY, Paris, France, pp. 316–331, doi:10.1145/263699.263744.

[6] P. Cousot & R. Cousot (1977): *Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints*. In: *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM Press, New York, NY, Los Angeles, California, pp. 238–252, doi:10.1145/512950.512973.

[7] P. Cousot & R. Cousot (1979): *Systematic design of program analysis frameworks*. In: *Conference Record of the Sixth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM Press, New York, NY, San Antonio, Texas, pp. 269–282, doi:10.1145/567752.567778.

[8] P. Cousot & R. Cousot (1992): *Abstract Interpretation and Application to Logic Programs*. *Journal of Logic Programming* 13(2–3), pp. 103–179, doi:10.1016/0743-1066(92)90030-7.

[9] P. Cousot & R. Cousot (1992): *Abstract Interpretation Frameworks*. *Journal of Logic and Computation* 2(4), pp. 511–547, doi:10.1093/logcom/2.4.511.

[10] L. Damas & R. Milner (1982): *Principal type-schemes for functional programs*. In: *POPL '82: Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM, New York, NY, USA, pp. 207–212, doi:10.1145/582153.582176.

[11] L. Galletta (2010): *Una semantica astratta per l'inferenza dei tipi ed effetti in un linguaggio multi-tier*. Master's thesis, Università di Pisa.

[12] D. K. Gifford & J. M. Lucassen (1986): *Integrating functional and imperative programming*. In: *Proceedings of the 1986 ACM conference on LISP and functional programming*. LFP '86, ACM, New York, NY, USA, pp. 28–38, doi:10.1145/319838.319848.

[13] R. Gori & G. Levi (2002): *An Experiment in Type Inference and Verification by Abstract Interpretation*. In: *VMCAI '02: Revised Papers from the Third International Workshop on Verification, Model Checking, and Abstract Interpretation*. Springer-Verlag, London, UK, pp. 225–239, doi:10.1007/3-540-47813-2_16.

[14] C. A. Gunter & D. S. Scott (1990): *Semantic Domains*, chapter 12, pp. 634–674. Handbook of Theoretical Computer Science, Elsevier Science.

[15] R. Hindley (1969): *The Principal Type-Scheme of an Object in Combinatory Logic*. *Transactions of the American Mathematical Society* 146, pp. 29–60.

[16] INRIA: *The Caml Language*. Available at http://caml.inria.fr. WWW publication.

[17] J.-L. Lassez, M. J. Maher & K. Marriott (1988): *Unification revisited*. In: *Foundations of deductive databases and logic programming*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, pp. 587–625.

[18] X. Leroy & F. Pessaux (2000): *Type-based analysis of uncaught exceptions*. *ACM Trans. Program. Lang. Syst.* 22, pp. 340–377, doi:10.1145/349214.349230.

[19] E. Meijer, W. Schulte & G. Bierman (2003): *Programming with circles, triangles and rectangles*. In: *In XML Conference and Exposition*.

[20] B. Monsuez (1992): *Polymorphic Typing by Abstract Interpretation*. In: *Proceedings of the 12th Conference on Foundations of Software Technology and Theoretical Computer Science*. Springer-Verlag, London, UK, pp. 217–228, doi:10.1007/3-540-56287-7_107.

[21] F. Nielson & H. Nielson (1994): *Constraints for polymorphic behaviours of concurrent ML*. In Jean-Pierre Jouannaud, editor: *Constraints in Computational Logics. Lecture Notes in Computer Science* 845, Springer Berlin / Heidelberg, pp. 73–88, doi:10.1007/BFb0016845.

[22] F. Nielson, H. Riis Nielson & C. Hankin (2005): *Principles of Program Analysis*, 1st ed. 1999. corr. 2nd printing, 1999 edition. Springer.

[23] C. Queinnec (2000): *The influence of browsers on evaluators or, continuations to program web servers*. *SIGPLAN Not.* 35, pp. 23–33, doi:10.1145/357766.351243.

[24] D. Schmidt (1986): *Denotational Semantics: A Methodology for Language Development*. William C Brown Pub.

[25] J. Vouillon & P. Jouvelot (1995): *Type and Effect Systems via Abstract Interpretation*. Available at http://www.cri.ensmp.fr/classement/doc/A-273.pdf.

[26] G. Winskel (1993): *The Formal Semantics of Programming Languages*. MIT Press.