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

Workshop on Automated Specification and Verification of Web Systems, 2011
Multi-tier architecture

Standard web applications have a multi-tier architecture

Each tier runs on a different computational environment characterized by its language and its data representation (impedance mismatch problem)
LINKS: a Multi-Tier Web Language

Multi-tier web languages allow one to blend server, client and database code and provide automatic mechanisms for the partition of the application over tiers.
LINKS: a Multi-Tier Web Language

Multi-tier web languages allow one to blend server, client and database code and provide automatic mechanisms for the partition of the application over tiers.

LINKS

LINKS is a functional multi-tier web language

- from a single source code the compiler generates code for each tier
- support an unified cross-tier programming model by exploiting web continuations
LINKS: a Multi-Tier Web Language

Multi-tier web languages allow one to blend server, client and database code and provide automatic mechanisms for the partition of the application over tiers.

LINKS

LINKS is a functional multi-tier web language:

- from a single source code the compiler generates code for each tier
- support an unified cross-tier programming model by exploiting web continuations

Web continuations in LINKS

Closures (expression to be executed plus bindings of free variables) stored in HTML pages
Baltopoulos and Gordon have shown that

- storing web continuation in HTML page is not secure
- an attacker can violate
  1. Secrecy
  2. Data Integrity
  3. Control Integrity
To overtake the security issues they have proposed a secure implementation that includes:

1. a compilation strategy based on authenticated encryption
2. a types-and-effects system to enable source level reasoning about the security of web applications
LINKS: security
Solution

To overtake the security issues they have proposed a secure implementation that includes

1. a compilation strategy based on authenticated encryption
2. a types-and-effects system to enable source level reasoning about security of web applications

The secure implementation has been formalized for TINYLINKS, a λ-calculus augmented with

1. XML values for representing web pages
2. event e assert annotation for expressing safety properties
**TinyLinks**

**Syntax**

\[
f, y, x
\]

\[
p
\]

\[
c ::= \text{Unit} \mid \text{Zero} \mid \text{Succ} \mid \text{String}
\]

\[
\mid \text{Nil} \mid \text{Cons} \mid \text{Tuple} \mid \text{Elem} \mid \text{Text}
\]

\[
g ::= + \mid - \mid * \mid /
\]

\[
L ::= p(V_1, \ldots, V_n)
\]

\[
V, U ::= x \mid c(V_1, \ldots, V_n) \mid \text{href}(E)
\]

\[
\mid \lambda x_1. \ldots, x_n. E \mid \text{form}([l_1, \ldots, l_n], E)
\]

\[
E ::= V \mid \text{var } x = E_1; E_2 \mid g(E_1, E_2)
\]

\[
\mid V(U_1, \ldots, U_n) \mid \text{post}([l_1 = V_1, \ldots, l_n = V_n], U)
\]

\[
\mid \text{get}(V) \mid \text{event } L \mid \text{assert } L
\]

\[
\text{switch}(V)\
\]

\[
\text{case } c(x_1, \ldots, x_n) \rightarrow E_1
\]

\[
\mid _\rightarrow E_2
\]
Types-and-effects system

A powerful extension of type systems which allows one to statically reason about program’s execution

\[ \Gamma \vdash M_1 : \tau_1 \& \phi_1 \quad \ldots \quad \Gamma \vdash M_n : \tau_n \& \phi_n \]

\[ \Gamma \vdash E(M_1, \ldots, M_n) : \tau \& \phi \]
Types-and-effects system

A powerful extension of type systems which allows one to statically reason about program’s execution

\[ \Gamma \vdash M_i : \tau_i \& \phi_i \]...

\[ \Gamma \vdash M_n : \tau_n \& \phi_n \]

\[ \Gamma \vdash E(M_1, \ldots, M_n) : \tau \& \phi \]
Types-and-effects system

A powerful extension of type systems which allows one to statically reason about program’s execution

Premise:

\[ \Gamma \vdash M_1 : \tau_1 & \phi_1 \quad \ldots \quad \Gamma \vdash M_n : \tau_n & \phi_n \]

Conclusion:

\[ \Gamma \vdash E(M_1, \ldots, M_n) : \tau & \phi \]
Types-and-effects system

A powerful extension of type systems which allows one to statically reason about program’s execution

\[ \Gamma \vdash M_1 : \tau_1 \& \phi_1 \quad \ldots \quad \Gamma \vdash M_n : \tau_n \& \phi_n \]

\[ \Gamma \vdash E(M_1, \ldots, M_n) : \tau \& \phi \]

It computes for each program phrase its type augmented with a semantic property

- if \( \Gamma \vdash M_i : \tau_i \& \phi_i \Rightarrow \tau_i \) is the type of the expression \( M_i \) and the semantic property \( \phi_i \) holds
- then \( \Gamma \vdash E(M_1, \ldots, M_n) : \tau \& \phi \Rightarrow \tau \) is the type of the expression \( E(M_1, \ldots, M_n) \) and the semantic property \( \phi \) holds
TINYLINKS’s types-and-effects system

Goal
Whenever an assertion `assert L` occurs in the execution there exists a previous occurrence of an event `event L`

\[
\Gamma; F \vdash E \overset{exp}{\Rightarrow} <_\_ : T > \{ F_1 \}
\]

\[
\Gamma; F \vdash E \overset{exp}{\Leftarrow} <_\_ : T > \{ F_1 \}
\]
**TINYLINKS’s types-and-effects system**

**Goal**
Whenever an assertion `assert L` occurs in the execution there exists a previous occurrence of an event `event L`

\[ \Gamma; F \vdash E \Rightarrow <_T> \{ F_1 \} \]

\[ \Gamma; F \vdash E \Leftarrow <_T> \{ F_1 \} \]

For each expression `E` compute
1. the type
TINYLINKS’s types-and-effects system

Goal
Whenever an assertion `assert L` occurs in the execution there exists a previous occurrence of an event `event L`

\[ \Gamma; F \vdash E \overset{exp}{\Rightarrow} \langle \_ : T \rangle \{ F_1 \} \]

\[ \Gamma; F \vdash E \overset{exp}{\Leftarrow} \langle \_ : T \rangle \{ F_1 \} \]

For each expression \( E \) compute
1. the type
2. the preconditions
TINYLINKS’s types-and-effects system

Goal
Whenever an assertion `assert L` occurs in the execution there exists a previous occurrence of an event `event L`

\[
\Gamma; F \vdash E \Rightarrow <_>: T > \{ F_1 \}
\]

\[
\Gamma; F \vdash E \Leftarrow <_>: T > \{ F_1 \}
\]

For each expression `E` compute
1. the type
2. the preconditions
3. the post-conditions
TINYLINKS’s types-and-effects system

Some example of rules

\[
\frac{
\Gamma; F \vdash V \stackrel{val}{\leftarrow} \text{xml} \quad 
\Gamma; F \vdash \text{get}(V) \stackrel{exp}{\Rightarrow} \langle \_ : \text{xml} \rangle \{ \} 
}{
\Gamma; F \vdash \text{get}(V) \Rightarrow \langle \_ : \text{xml} \rangle \{ \} 
}
\]
TINYLINKS’s types-and-effects system

Some example of rules

\[
\begin{align*}
\frac{\Gamma; F \vdash V_i \leftarrow \text{string} \quad \forall i \in \{1, \ldots, n\}}{
\Gamma; F \vdash \text{post}([l_1 = V_1, \ldots, l_n = V_n], U) \Rightarrow \langle \_ : \text{xml} \rangle \{\}}
\end{align*}
\]
**TINYLINKS’s types-and-effects system**

Some example of rules

(T-Assert)

\[ \Gamma \vdash \top \quad \text{fv}(F, L) \subseteq \text{dom}(\Gamma) \quad \text{L} \in F \]

\[ L = p(V_1, \ldots, V_n) \quad \Gamma; F \vdash V_i \Rightarrow T_i \quad \forall i \in \{1, \ldots, n\} \]

\[ \Gamma; F \vdash \text{assert} L \Rightarrow \langle \_ : \text{unit} \rangle \{ L \} \]
TINYLINKS’s types-and-effects system

Some example of rules

(T-App)

\[ \Gamma; F \vdash U \overset{val}{\Rightarrow} T \quad T = \langle x_1 : T_1, \ldots, x_n : T_n \rangle \{ F_1 \} \rightarrow W \quad f_v(T) = \emptyset \]

\[ \Gamma; F \vdash V_i \overset{val}{\Leftarrow} T_i \quad \forall i \in \{ 1, \ldots, n \} \quad F_1 [V_1/x_1] \ldots [V_n/x_n] \subseteq F \]

\[ \Gamma; F \vdash U(V_1, \ldots, V_n) \overset{exp}{\Rightarrow} W[V_1/x_1] \ldots [V_n/x_n] \]
TINYLINKS’s types-and-effects system

Safe Web Application

A web application $E$ is safe if and only if there exists a proof within the types-and-effects system of the judgment

$$\emptyset; \emptyset \vdash E \overset{exp}{\leftrightarrow} \langle _{-} : \text{xml} \rangle \{ \}$$
Types-and-effects system

Usually the definition of a types-and-effects analysis requires
1. Definition of rules
2. State and prove the soundness of analysis
3. Definition of inference algorithm
4. Prove that the algorithm is correct (soundness/completeness)
Types-and-effects system

Usually the definition of a types-and-effects analysis requires
1. Definition of rules
2. State and prove the soundness of analysis
3. Definition of inference algorithm
4. Prove that the algorithm is correct (soundness/completeness)

The approach used for the TINYLINKS’s types-and-effects system is different

- each expression is translated in an expression of F7
  - this translation hides the property of the analysis
Our goal

Reconstruct the TINYLINKS’s types-and-effects system by abstract interpretation
Our goal

Reconstruct the TINYLINKS’s types-and-effects system by abstract interpretation

Benefits

1. precise definition of relation between analysis and semantics
2. analysis is correct by construction
Our goal

Reconstruct the TINYLINKS’s types-and-effects system by abstract interpretation

Benefits
1. precise definition of relation between analysis and semantics
2. analysis is correct by construction

We follow Cousot’s methodology for type systems
1. we define a denotational semantics for TINYLINKS (concrete semantics)
2. we define a abstract semantics that computes types augmented by effects
Concrete semantics

A denotational semantics that considers
Concrete semantics

A denotational semantics that considers

- **TINYLINKS** as an untyped $\lambda$-calculus

\[
Eval = (\ldots + EEnv \rightarrow (Eval \times EEnv) + \ldots)_\perp
\]
Concrete semantics

A denotational semantics that considers

- **TINYLINKS** as an untyped λ-calculus

\[
Eval = (\ldots + EEnv \rightarrow (Eval \times EEnv) + \ldots)_{\perp}
\]

Domain of values

Injection (Text, Form, Fun, ...)
Concrete semantics

A denotational semantics that considers

- **TINYLINKS** as an untyped $\lambda$-calculus

\[
Eval = (\ldots + EEnv \rightarrow (Eval \times EEnv) + \ldots) \bot
\]

- the occurrence and assertion of the events

\[
EEnv = Pred \rightarrow (Dval \times Mark)
\]
Concrete semantics

A denotational semantics that considers

- **TINYLINKS** as an untyped \( \lambda \)-calculus

\[
Eval = (\ldots + EEnv \to (Eval \times EEnv) + \ldots)_{\bot}
\]

Domain of values

- the occurrence and assertion of the events

\[
EEnv = \text{Pred} \to (Dval \times \text{Mark})
\]

Events environment

Injections (Text, Form, Fun, \ldots)

\{ E, EA, A \}

values in the events are integers only
Concrete semantics

Two semantic functions

1. for values

\[ \mathcal{V}[-] : \text{VAL} \rightarrow \text{Env} \rightarrow \text{EEnv} \rightarrow \text{Eval} \]

2. for expressions

\[ \llbracket - \rrbracket : \text{EXP} \rightarrow \text{Env} \rightarrow \text{EEnv} \rightarrow (\text{Eval} \times \text{EEnv}) \]
Concrete semantics

Two semantic functions

1. for values

\[ \mathcal{V}[\_] : \text{VAL} \to \text{Env} \to \text{EEnv} \to \text{Eval} \]

2. for expressions

\[ \llbracket \_ \rrbracket : \text{EXP} \to \text{Env} \to \text{EEnv} \to (\text{Eval} \times \text{EEnv}) \]

Examples of semantic equation

\[
\llbracket \text{get}(V) \rrbracket \rho \phi = \text{let}^* v' = \mathcal{V}[V] \rho \phi \text{ in } \\
\text{case } v' \text{ of } \\
H\text{ref}(f) \to f \phi \\
_ \to (\llbracket \text{WrongValue}() \rrbracket, \iota)
\]
Concrete semantics

Two semantic functions
1. for values

\[ \mathcal{N}[-] : VAL \rightarrow Env \rightarrow EEnv \rightarrow Eval \]

2. for expressions

\[ [-] : EXP \rightarrow Env \rightarrow EEnv \rightarrow (Eval \times EEnv) \]

Examples of semantic equation

\[
\begin{align*}
\llbracket \text{assert } q(V) \rrbracket & \psi_\phi = \text{let}^* ev = \text{evalToDval}(\mathcal{N}[V]_\rho \phi) \text{ in} \\
& \text{let } (ev', m) = \phi q \\
& \text{if } ev = ev' \text{ then} \\
& \quad (\llbracket \text{Unit()} \rrbracket, \phi \llbracket (ev', EA)/q \rrbracket) \\
& \text{else} \\
& \quad (\llbracket \text{WrongValue()} \rrbracket, \iota)
\end{align*}
\]
Unsoundness in TINYLINKS analysis

Consider the expression

\[
\text{get(Text(“Hello World!”))}
\]
Unsoundness in TINYLINKS analysis

Consider the expression

\[
\text{get(Text("Hello World"))}
\]

Semantics

\[
[[\text{get(Text("Hello World"))}]] \rho \phi = (\lceil \text{WrongValue}(), \iota \rceil)
\]

Error: the denotation of \text{Text("Hello World!")} is not a link
Unsoundness in TINYLINKS analysis

Consider the expression

\[ \text{get(Text(”Hello World!”))} \]

Semantics

\[ \llbracket \text{get(Text(”Hello World!”))} \rrbracket_\rho \phi = (\llbracket \text{WrongValue()} \rrbracket, \iota) \]

Error: the denotation of Text("Hello World!") is not a link

TINYLINKS’s types-and-effects system

\[ \emptyset; \emptyset \vdash \text{get(Text(”Hello World!”))} \xrightarrow{\text{exp}} \llbracket \_ : \text{xml} \rrbracket \{ \} \]

the expression is type checked and the computed type is xml
Abstract domain

Values

Trouble

- types have annotations

\[
\text{integer}\{\}\rightarrow\text{integer}\{q:5, p:3\}
\]
Abstract domain

Values

Trouble

• types have annotations

\[
\text{integer}\{\}\rightarrow\text{integer}\{q:5,\ p:3\}\]

• annotated types are not a free algebra

\[
\text{integer}\{\}\rightarrow\text{integer}\{q:5,\ p:3\}\\
\text{integer}\{\}\rightarrow\text{integer}\{p:3,\ q:5\}
\]
Abstract domain

Values

Trouble

• types have annotations

\[
\text{integer } \{ \} \rightarrow \text{integer } \{ q : 5, p : 3 \}
\]

• annotated types are not a free algebra

\[
\begin{align*}
\text{integer } \{ \} & \rightarrow \text{integer } \{ q : 5, p : 3 \} \\
\text{integer } \{ \} & \rightarrow \text{integer } \{ p : 3, q : 5 \}
\end{align*}
\]

Solution: simple type and constraints
Abstract domain

Values

Trouble

- types have annotations

\[
\text{integer} \{ \} \rightarrow \text{integer} \{ q : 5, p : 3 \}
\]

- annotated types are not a free algebra

\[
\begin{align*}
\text{integer} \{ \} & \rightarrow \text{integer} \{ q : 5, p : 3 \} \\
\text{integer} \{ \} & \rightarrow \text{integer} \{ p : 3, q : 5 \}
\end{align*}
\]

Solution: simple type and constraints

- we substitute the annotations in the types with annotation variables
Abstract domain

Values

Trouble

• types have annotations

\[ \text{integer} \{ \} \rightarrow \text{integer} \{ q: 5, p: 3 \} \]

• annotated types are not a free algebra

\[ \text{integer} \{ \} \rightarrow \text{integer} \{ q: 5, p: 3 \} \]
\[ \text{integer} \{ \} \rightarrow \text{integer} \{ p: 3, q: 5 \} \]

Solution: simple type and constraints

• we substitute the annotations in the types with annotation variables

\[ \text{integer}(\gamma_1) \rightarrow \text{integer}(\gamma_2) \quad \gamma_1 \supseteq \emptyset \quad \gamma_2 \supseteq \{ p: 3, q: 5 \} \]
Abstract domain

Values

- the events depend on the concrete values \( Dval \)
Abstract domain

Values

- the events depend on the concrete values \((Dval)\)
- the abstract domain need to include the concrete values
Abstract domain

Values

- the events depend on the concrete values ($Dval$)
- the abstract domain need to include the concrete values

\[ TipoS \times Dval \times Constr \times TPred \]

- (type, substitution) $\cup$ NoType
- concrete value
- constraints
- $Pred \rightarrow Dval$
Abstract semantics

Example of semantic equation

\[
\begin{align*}
[&\text{get}(V)]^a & \rho \phi = \\
& \gamma \in V_a & \text{fresh} \\
\text{let} & (ts, d, C, f) = V[V]^a & \rho \phi \text{ in} \\
\text{if} & ts \neq \text{NoType} \text{ then} \\
& \text{case} \ \text{mgu}(\{ ts.t = \text{link}(\gamma) \} \cup ts.\theta) \ of \\
& \text{S}(\theta) \rightarrow \text{let} C' = \{ (\theta(\gamma), q) \in \theta(C) \} \ \text{in} \\
& \text{if} \ \text{check}(\theta(f \leftarrow C'), \phi) \ \text{then} \\
& (((\theta(\text{xml}(\gamma)), \theta), \\
& \nodval, \theta(C) \setminus C', \theta(f \downarrow C)), \phi) \\
& \text{else} \\
& (\text{Error}, \iota) \\
& \rightarrow (\text{Error}, \iota) \\
\text{else} \\
& (\text{Error}, \iota)
\end{align*}
\]
Analyzer

Both the concrete and abstract semantics have been implemented as OCaml programs
Analyzer

Both the concrete and abstract semantics have been implemented as OCaml programs

- TINYLINKS programs are represented in abstract syntax
Both the concrete and abstract semantics have been implemented as OCaml programs

- **TINYLINKS** programs are represented in abstract syntax
- the implementation have an unique semantic function parametrized with respect to
Both the concrete and abstract semantics have been implemented as OCaml programs

- **TINYLINKS** programs are represented in abstract syntax
- the implementation have an unique semantic function parametrized with respect to
  - the primitive operations
Analyzer

Both the concrete and abstract semantics have been implemented as OCaml programs

- TINYLINKS programs are represented in abstract syntax
- the implementation have an unique semantic function parametrized with respect to
  - the primitive operations
  - the semantic domain
Example 1

Expression

```javascript
fun buy(value, dbpass) {
    var _ = assert PriceIs(value);
    Text("a")
}
```
Example 1

Expression

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

Abstract semantics

```
(type - :
Function(_, Integer(), _,
    Function(_, _, Xml(_, _,
        No_dval [(_, PriceIs)]
    {PriceIs -> _, {}}
```
Example 2

Expression

buy 5
Example 2

Expression

buy 5

Abstract semantics

(type - : 
  Function(_#dbpass#var3_, _typevar3_, _annvar7_, 
           Xml(_annvar9_), _annvar8_)
  Unknown [(_annvar7_,PriceIs)] {PriceIs -> 5}, {}
Example 3

Expression

buy 5 "a"
Example 3

Expression

buy 5 "a"

Abstract semantics

Exception: No_type "apply_fun: no preconditions"
Conclusions

We have reconstructed the TINYLINKS’s types-and-effects system by abstract interpretation

- we have precisely defined relationship between semantics and analysis
- we have shown unsoundness of TINYLINKS’s types-and-effects system
Conclusions

We have reconstructed the TINYLINKS’s types-and-effects system by abstract interpretation

- we have precisely defined relationship between semantics and analysis
- we have shown unsoundness of TINYLINKS’s types-and-effects system

Future work

- Consider a type system with sub-types (link <: xml, form <: xml)
- Extend the class of value that can be used in the events
- Generalize the methodology and apply it to further analysis