A Reconstruction of a Type-and-Effect Analysis by Abstract Interpretation

Letterio Galletta

Dipartimento di Informatica - Università di Pisa
galletta@di.unipi.it

Abstract. Type-and-effect systems (TESs) have been widely exploited to specify static analyses of programs, for example to track computational side effects, exceptions and communications in concurrent programs. We adopt abstract interpretation techniques to reconstruct a TES developed to handle security problems of a multi-tier web language. In order to reconstruct this type-and-effect analysis we extended the Cousot's methodology used for type systems and the corresponding type inference algorithms by defining an abstract domain able to express types augmented by semantic annotations. This abstract domain is an extension of the Hindley's monotypes with a new kind of variables and constraints. We show that this abstract domain allows us to reconstruct different type-and-effect analyses by properly changing the set of monotypes and the shape of the constraints. In particular, we apply this approach to reconstruct the *Call-Tracking Analysis*. As usual with abstract interpretation, the analysis is correct by construction. The analyser has been implemented in OCaml.

1 Introduction

Type-and-effect systems (TESs) are a powerful extension of type systems which allows one to express general semantic properties and to statically reason about program execution. The underlying idea is to refine the type information so to express further intentional or extensional properties of the semantics of the program. In practice, TES compute the type of each program sentence and an approximate (but sound) description of its run-time behaviour.

Type systems (and the corresponding type inference algorithms) have been reconstructed as a hierarchy of abstract interpretations by Cousot [3]. In [5,6] we have extended the Cousot's methodology to reconstruct the TES used in [1] to handle security issues in the multi-tier web language LINKS [2]. We have shown that the original analysis was unsound. We have fixed the flaw and derived a correct analyser as an abstract semantics. In order to reconstruct this TES we have defined an abstract domain able to express types augmented by semantic annotations and concrete values. This abstract domain is an extension of Hindley's monotypes [7,4,3,9] and it could be easily implemented. Its definition is based on an approach described in [11] and exploits specific annotated types (*simple types*), where the annotations are replaced by a special kind of variables (*annotation variables*) whose values satisfy suitable constraints. An abstract value is indeed a pair (t_s, C) where t_s is a Hindley's monotype with annotation variables and C is a constraint whose solution represents the annotation.

We argue that this abstract domain is general enough to reconstruct different TESs by properly changing the set of monotypes and the shape of the constraints. To show the flexibility and the feasibility of our abstract domain we have reconstructed a quite simple analysis, *Call-Tracking Analysis (CTA)*, for which a TES was provided in [11]. This analysis allows one to determine, for each expression, the type of the computed value and the function applications which may occur during the evaluation. We defined this analysis, for a minimal ML core (TINYML). For example, the expression

```
let rec fact = fun[fact_point] n -> if (iszero n) then 1
                                    else  n * (fact (n - 1))
```

defines a recursive function which computes the factorial of a number and which is uniquely identified by the label fact_point. The CTA computes an abstract value expressing the annotated type $\mathtt{int} \xrightarrow{\{\mathtt{fact\_point}\}} \mathtt{int}$, i.e. a function from integers to integers which could apply the function denoted by the label fact_point during its evaluation.

To the best of our knowledge the only paper relating TESs and abstract interpretation is [12]. In this paper Vouillon and Jouvelot introduce a simple program time complexity estimator for a λ-calculus with recursion. They also define an abstract interpretation and a TES for this analysis. Their main result is that these two a priori distinct approaches are equivalent. Their abstract semantics computes for each expression the set of ground types compatible with the value given by the concrete evaluation. Hence, this abstract domain is more adapt to relate and prove equivalent two different approaches than to directly implement an analyser. Our abstract domain, instead, is designed to be easily implemented.

In this extended abstract we survey the ideas and the methodology underlying our abstract domain (Section 2) and we show some results obtained by the analyser, implemented in OCaml (Section 3). Page limitation prevent us from giving more details on the formal development and the implementation issues. A full presentation, including all the proof and the code can be found in [5].

2 Reconstruction of the Call-Tracking Analysis

In this section we describe the ideas of our reconstruction of the TES for CTA. TINYML is a minimal core of ML so its syntax is standard and we assume each λ-abstraction to have a unique label ($\mathtt{l} \in \mathtt{Point}$).

We define a denotational semantics as concrete semantics[3], that computes a pair for each expression: its value and the set of the function labels applied during the evaluation (the effects). To store the effects we use the *effects environment*. Since TINYML is an untyped λ-calculus, we define the semantic domain of values *Eval* as a recursive sum of cpos, where each element of the sum represents a suitable class of values.

As usual then we take the powerset of the concrete semantics as the collecting one. In a TES types can be annotated (e.g. to record latent effects), the definition of a suitable abstract domain requires particular care. In [6] we extend the Hindley's monotypes by introducing a new kind of variables (*annotation variable*) and constraints. In practice, an abstract value is a pair (t, C) where t is an Hindley's monotype with annotation variables and C is a set of constraints whose solution represent the annotation that the type can have. For the CTA the domain of abstract values is $TypeA = TypeS \times Constr$ where $TypeS$ is the set of Hindle's monotypes with annotation variables[1] (V_a) and $Constr = \mathcal{P}(V_a \times Point)$ is the set of constraints. A constraint is as a set of pairs $(annotation\,variable, label)$: $(\delta, \mathtt{l})$ means that the label `l` is a member of the type annotation represented by the variable δ.

A Galois connection relates the abstract and the concrete domain. The connection is defined in [5] by using standard results, e.g. *representation function* [10].

The definition of the abstract semantics equations for CTA follows the same schema of [5,6], but in this case the computed effects concern the function applications encountered during the evaluation.

3 Examples

The abstract semantics of TINYML has been implemented in OCaml [8]. To illustrate the analyser, consider the expression

```
let a = fun[a_point] x -> true in
let b = fun[b_point] x -> false in
  (a 1) or (b 1)
```

defines two constant functions, `a` and `b`. The first function returns `true`, the second one `false`. We take the disjunction of applying both function to 1. The analyser computes

```
(type - : Boolean [(_annvar2_,a_point), (_annvar3_,b_point)] &
  {a_point, b_point})
```

that is the computed value is a boolean and during the evaluation we might apply both functions. Notice that this happens because during the abstraction process we loose precision. Since we do not know the values of the disjuncts, we have to evaluate them both. As a consequence, the resulting effect is not precise, yet safe and valid.

As second example consider the expression

```
(fun[x_point] x -> x) (fun[y_point] y -> y)
```

representing the application of the identity function with label `x_point` to the identity function with label `y_point`. The analyser computes

```
(type - : Function(_typevar1_, _annvar1_, _typevar1_)
    [(_annvar2_,x_point), (_annvar1_,y_point)]  & {x_point})
```

that is the computed type is a function and during the evaluation we might apply the function identified by the label `x_point`.

[1] Actually, for technical reasons $TypeS$ is the lifting of Hindle's monotypes with idempotent substitutions and a new bottom, see [5,6]

4 Conclusions

We have shown that abstract interpretation can deal with TESs. We defined an abstract domain able to express types augmented by semantic annotations and at the same time simple enough to allow the implementation of an analyser in OCaml. Our abstract domain extends Hindley's monotypes with a new kind of variables and constraints. We have prove the expressive power of this domain by showing that it allows us to reconstruct different TESs by only changing the set of monotypes and the shape of constraints. As an example we have reconstructed the CTA.

References

1. Baltopoulos, I.G., Gordon, A.D.: Secure Compilation of a Multi-Tier Web Language. In: TLDI '09: Proceedings of the 4th international workshop on Types in language design and implementation. pp. 27–38. ACM, New York, NY, USA (2009)
2. Cooper, E., Lindley, S., Wadler, P., Yallop, J.: Links: Web programming without tiers. In: In 5th International Symposium on Formal Methods for Components and Objects (FMCO). Springer-Verlag (2006)
3. Cousot, P.: Types as abstract interpretations, invited paper. In: Conference Record of the Twentyfourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 316–331. ACM Press, New York, NY, Paris, France (Jan 1997)
4. Damas, L., Milner, R.: Principal type-schemes for functional programs. In: POPL '82: Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages. pp. 207–212. ACM, New York, NY, USA (1982)
5. Galletta, L.: Una semantica astratta per l'inferenza dei tipi ed effetti in un linguaggio multi-tier. Master's thesis, Università di Pisa (2010), available at `http://www.cli.di.unipi.it/~galletta/tesi.html`
6. Galletta, L., Levi, G.: An abstract semantics for inference of types and effects in a multi-tier web language. In: Proceedings of the 7th International Workshop on Automated Specification and Verification of Web Systems (2011)
7. Hindley, R.: The principal type-scheme of an object in combinatory logic. Transactions of the American Mathematical Society 146, 29–60 (1969)
8. INRIA: The Caml Language, `http://caml.inria.fr`, wWW publication
9. Monsuez, B.: Polymorphic typing by abstract interpretation. In: Proceedings of the 12th Conference on Foundations of Software Technology and Theoretical Computer Science. pp. 217–228. Springer-Verlag, London, UK (1992)
10. Nielson, F., Nielson, H.R.: Type and Effect Systems. In: Olderog, E.R., Steffen, B. (eds.) Correct System Design, pp. 114–136. No. 1710 in Lecture Notes in Computer Science, Springer (1999), `http://www2.imm.dtu.dk/~nielson/Papers/NiNi99tes.pdf`
11. Nielson, F., Nielson, H.R., Hankin, C.: Principles of Program Analysis. Springer, 1st ed. 1999. corr. 2nd printing, 1999 edn. (2005)
12. Vouillon, J., Jouvelot, P.: Type and effect systems via abstract interpretation (1995), `http://www.cri.ensmp.fr/classement/doc/A-273.pdf`