Type Inference via Symbolic Environment Transformations*

Ulrich Schöpp and Chuangjie Xu

fortiss GmbH, Munich, Germany

Region type systems are a powerful tool for e.g. pointer analysis and taint analysis [BGH13, GHL12] for object oriented programs. Extended with effect annotations, they can capture information about the possible event traces of programs and thus can be used to enforce programming guidelines [EHZ17, ESX21]. The idea of such a type-based analysis approach is to infer the type of a program which allows us to verify if the program satisfies certain properties. The type inference algorithms of [BGH13, GHL12, EHZ17, ESX21] infer a type for a method from the given types of its arguments. If the method is called with different arguments in different occasions of a program, then it is analyzed multiple times, one for each invocation.

To avoid analyzing the same piece of code multiple times, we introduce a *compositional* inference algorithm. Once a method has been analyzed, the result can be used directly in the analysis of its caller, assuming that it does not depend on the caller, i.e., it does not (indirectly) call the caller. The idea is to split the type inference into two steps: (1) Compute an environment transformation for each method. (2) Derive the type of the targeting method using its environment transformation. When analyzing a new method, we use the previously computed environment transformations of the callees rather than fully reanalyzing them as in previous work [BGH13, GHL12, EHZ17, ESX21]. We explain the idea in more detail.

Types and typing environments For simplicity, we work with a region type system without effects for Featherweight Java [IPW01] like the one in [BGH13]. A *type* is a region representing a property of a value such as its provenance information. For example, we may consider a region $\texttt{CreatedAt}(\ell)$ for references to objects that were created in the position with label ℓ. One can think of the label ℓ as a line number in the source code. This region allows us to track where in the program an object originates. Differing from previous work, we do not maintain a global table of field typing but instead add it into *typing environments*. For example, the environment

$$E = (x : \texttt{CreatedAt}(\ell_1),\ \texttt{CreatedAt}(\ell_1).f : \texttt{CreatedAt}(\ell_2))$$

means that x points to an object which is created at position ℓ_1 and the field f of any object created at ℓ_1 is an object created at ℓ_2.

Environment transformations The type system essentially performs flow analysis. The execution of a program may change the types of its variables and fields. Therefore, we can assign it an *environment transformation* that approximates how the types are updated in the program. For example, we assign the program

$$\begin{aligned} &y = x.f;\\ &x = \texttt{new}^{\ell_3}\ C(); \end{aligned}$$

the transformation

$$[y \mapsto x.f,\ x \mapsto \texttt{CreatedAt}(\ell_3)].$$

It updates the above environment E to $(x : \texttt{CreatedAt}(\ell_3),\ y : \texttt{CreatedAt}(\ell_2),\ \texttt{CreatedAt}(\ell_1).f : \texttt{CreatedAt}(\ell_2))$. Note that the substitutions are performed simultaneously.

*Supported by the German Research Foundation (DFG) under research grant 250888164 (GuideForce).

Field access graphs Directly using field access paths like $x.f$ as above is problematic, because the lengths of access paths may be unbounded. The computation of environment transformations involving such access paths may not terminate. For example, consider a class of linked lists with a field `next` : `Node` pointing the next node. The following method returns the last node of a list.

```
Node last() {
  if (next == null) {return this;}
  else {return next.last();}
}
```

It would have return type $\texttt{this} \vee \texttt{this.next} \vee \texttt{this.next.next} \vee \texttt{this.next.next.next} \vee \cdots$, expressing that the returned value has the same type of the variable `this` or the field `this.next` and so on. To solve this, we work with *access graphs* which provide a finite representation of access paths [KSK07]. For example, the `Node` class has three access graphs to represent all its access paths. Therefore, the return type of `last`() is the disjunction of these three graphs rather than the above infinite disjunction of access paths.

A theory of abstract transformations With the above ingredients, we now define a notion of abstract transformation. Like the usual ones, an *abstract transformation* consists of finitely many assignments $\kappa \mapsto u$. The value u is a disjunction of some atoms. An *atom* is a variable, a type or a field graph following a variable or a type. The key κ is a non-type atom, as we treat variables and fields similarly in order to involve field typing in environments. To model how types are updated in a program, we define the following operations on abstract transformations. We define the *composition* of abstract transformations to model type updates in a statement followed by another and the *join* of abstract transformations to tackle conditional branches. Moreover, we *instantiate* an abstract transformation to an endofunction on typing environments. Assignments are essentially subtyping constraints [PS91] and the instantiation solves the constraints.

Type inference Suppose we have a table T assigning an abstract transformation to each method of a program. Then we can compute an abstract transformation for any expression e of the program by induction on e. For example, when e is an invocation of a method, we lookup the table T. For any well-typed program, we can compute such a table T for it as follows. We start by initializing T for each method with the "bottom" transformation, that is, the transformation whose values are the empty disjunction. For each method, we compute the abstract transformation of its body, and then update the corresponding entry in T. We repeat this until no more update of T is possible. This procedure always terminates, because the update of T "weakens" its entries and there are only finitely many abstract transformations.

To infer the type of a method, we find its abstract transformation from T, feed it with a typing for its arguments, and then get the type of the return variable from the resulting typing environment. If we add a new method into the program, we can use T to analyze it as long as the old methods do not depend on it. In this way, we can reuse the existing analysis results to analyze new code rather than reanalyzing the whole program and library.

We note that our approach to type inference via abstract transformations is independent from region types. What essential is the finite lattice structure of types. For example, it also works for class types of the standard Java type system.

Lastly, we have a prototype implementation of the above type inference algorithm for the type system of [ESX21] based on the Soot framework [SG]. It takes a Java bytecode program and a programming guideline as inputs, infers the type and effect of the program, and then verifies if the program adheres to the guideline.

References

[BGH13] Lennart Beringer, Robert Grabowski, and Martin Hofmann. Verifying pointer and string analyses with region type systems. *Computer Languages, Systems & Structures*, 39(2):49–65, 2013.

[EHZ17] Serdar Erbatur, Martin Hofmann, and Eugen Zălinescu. Enforcing programming guidelines with region types and effects. In Bor-Yuh Evan Chang, editor, *Programming Languages and Systems (APLAS 2017)*, volume 10695 of *Lecture Notes in Computer Science*, pages 85–104. Springer, Cham, 2017.

[ESX21] Serdar Erbatur, Ulrich Schöpp, and Chuangjie Xu. Type-based enforcement of infinitary trace properties for Java. In *23rd International Symposium on Principles and Practice of Declarative Programming (PPDP 2021)*, pages 18:1–18:14. Association for Computing Machinery, 2021.

[GHL12] Robert Grabowski, Martin Hofmann, and Keqin Li. Type-based enforcement of secure programming guidelines — code injection prevention at SAP. In G. Barthe, A. Datta, and S. Etalle, editors, *Formal Aspects of Security and Trust (FAST 2011)*, volume 7140 of *Lecture Notes in Computer Science*, pages 182–197. Springer, Berlin, Heidelberg, 2012.

[IPW01] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight Java: a minimal core calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems*, 23(3):396–450, 2001.

[KSK07] Uday P. Khedker, Amitabha Sanyal, and Amey Karkare. Heap reference analysis using access graphs. *ACM Transactions on Programming Languages and Systems*, 30(1):1–41, 2007.

[PS91] Jens Palsberg and Michael I. Schwartzbach. Object-oriented type inference. *SIGPLAN Notices*, 26(11):146–161, 1991.

[SG] McGill University Sable Group. Soot - A framework for analyzing and transforming Java and Android applications.