Type analysis

Letterio Galletta

Introduction to type analysis

- Type analysis is one of the most famous and used static analysis
- Every static typed language performs type analysis when is compiled/interpret

The goal

The aim of type analysis is to ensure that no type errors occur at run-time. To do that a type system:

- classify program phrases according to values they compute
- ensure that each operation is applied to correct data

What is a type system?

What a type? (my answer)

A type of a term usually denotes some approximation of the term semantics:

- a set of values equipped with a set of operations
- something more complex describing the behavior of a term, e.g., effects

A type system is an instance of deduction systems where theorems to be proved are about program types

The recipe for defining a type system

Defining a new type system consists of 4 steps:

1. Define the syntax of types and the syntax of type environments
2. Define the format of typing judgments and the typing rules (the typing relation)
3. Prove the type system is sound with respect to the semantics of the language
4. Define the typing algorithm and prove it correct

Our running example: the fun language

$$
\begin{array}{rcll}
x, f & \in & \mathit{Id} & \\
c & \in & \mathit{Const} & \\
e \in \mathit{Exp} & ::= & c & \text{constant literal} \\
& | & x & \text{identifier} \\
& | & \textbf{if } e_1 \textbf{ then } e_1 \textbf{ else } e_2 & \text{conditionals} \\
& | & e_1 \diamond e_2 & \diamond \in \{+, -, *, \ldots\} \text{ primitive operators} \\
& | & \textbf{let } x = e_1 \textbf{ in } e_2 & \text{declarations} \\
& | & \textbf{fun } x \to e & \text{lambda abstraction} \\
& | & e_1\ e_2 & \text{function application}
\end{array}
$$

Defining types

- Usually, programming languages provide us values that are scalar, e.g., integers, strings, and values that are compound, e.g., lists, arrays, maps
- A type can be considered as a summary/description of a collection of values that share properties, e.g., 1 and 2 share the property to be integer values and that can be added
- The syntax of types describes which the type for scalar values are and how we can define new types by composing those already existent one
- The syntax of types is specified using a grammar, whose language corresponds to all possible types

Type syntax of fun

$$\tau ::= \tau_c \mid \tau_1 \to \tau_2 \qquad\qquad \tau_c \in \{int, bool, string, \ldots\}$$

- The type τ_c abstractly represents the type of a generic scalar value
- There is only one type constructor $\cdot \to \cdot$ that allows us to create functional types

Typing environments

A type environment records the association among variables and their types, formally it is a list of pairs

$$\Gamma ::= \emptyset \mid x : \tau, \Gamma$$

Usually, variables are assumed to be distinct and Γ is considered as a partial function, with the expected meaning for $\Gamma(x)$ and $dom(\Gamma)$

Type judgments

A judgment is a statement about the type of a program term typically it has the form

$$\Gamma \vdash M$$

where the free variables of M occur in Γ (we say Γ *closes* M)

A judgment usually express a relation between a term and a type (typing relation) or between two types (sub-typing relation)

Type judgments of fun

In our language we have only a kind of type judgment

$$\Gamma \vdash e : \tau$$

meaning that the expression e has type τ in Γ.

Typing rules

A typing rule specifies when a judgment is true:

$$\frac{\Gamma \vdash M_1 \cdots \Gamma \vdash M_n}{\Gamma \vdash M}$$

Rules without premises are axioms, thus, always true

A judgment is valid iff it can be derived from axioms, inductively applying judgments (derivation tree)

Saying that a term M has type τ in Γ means to find a derivation for the judgment $\Gamma \vdash M : \tau$

Typing rules of fun (1)

$$\textsc{TConst}\quad \frac{}{\Gamma \vdash c : \tau_c}$$

$$\textsc{TVar}\quad \frac{x \in dom(\Gamma) \qquad \Gamma(x) = \tau}{\Gamma \vdash x : \tau}$$

$$\textsc{TOp}\quad \frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma \vdash e_2 : \tau_2 \qquad \diamond : \tau_1 \times \tau_2 \rightarrow \tau}{\Gamma \vdash e_1 \diamond e_2 : \tau}$$

$$\textsc{TIf}\quad \frac{\Gamma \vdash e_1 : bool \qquad \Gamma \vdash e_2 : \tau \qquad \Gamma \vdash e_3 : \tau}{\Gamma \vdash \textbf{if}\ e_1\ \textbf{then}\ e_2\ \textbf{else}\ e_3 : \tau}$$

Typing rules of fun (2)

$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad x : \tau_1, \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \textbf{let}\, x = e_1 \,\textbf{in}\, e_2 : \tau_2}\ \textsc{TLet}$$

$$\frac{x : \tau_1, \Gamma \vdash e : \tau_2}{\Gamma \vdash \textbf{fun}\, x \to e : \tau_1 \to \tau_2}\ \textsc{TFun}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1\, e_2 : \tau_2}\ \textsc{TApp}$$

Note: in the rule TFun the type τ_1 of the formal parameter x is guessed

An example of derivation

$$\dfrac{\dfrac{\Gamma \vdash x : \mathit{int} \quad \Gamma \vdash 3 : \mathit{int} \quad <: \mathit{int} \times \mathit{int} \to \mathit{bool}}{\Gamma \vdash x < 3 : \mathit{bool}} \quad \dfrac{\Gamma \vdash x : \mathit{int} \quad \Gamma \vdash 1 : \mathit{int} \quad + : \mathit{int} \times \mathit{int} \to \mathit{int}}{\Gamma \vdash x + 1 : \mathit{int}} \quad \dfrac{\Gamma \vdash y : \mathit{int} \quad \Gamma \vdash 1 : \mathit{int} \quad + : \mathit{int} \times \mathit{int} \to \mathit{int}}{\Gamma \vdash y + 1 : \mathit{int}}}{\Gamma \vdash \textbf{ if } x < 3 \textbf{ then } x + 1 \textbf{ else } y + 1 : \mathit{int}}$$

where $\Gamma = x : \mathit{int}, y ; \mathit{int}, \emptyset$

An example of type error

$$\dfrac{\dfrac{\begin{array}{c}\Gamma \vdash x : \mathit{int} \\ \Gamma \vdash 3 : \mathit{int} \\ <: \mathit{int} \times \mathit{int} \to \mathit{bool}\end{array}}{\Gamma \vdash x < 3 : \mathit{bool}} \qquad \dfrac{\begin{array}{c}\Gamma \vdash x : \mathit{int} \\ \Gamma \vdash 1 : \mathit{int} \\ + : \mathit{int} \times \mathit{int} \to \mathit{int}\end{array}}{\Gamma \vdash x + 1 : \mathit{int}} \qquad \dfrac{\begin{array}{c}\Gamma \vdash y : \mathit{real} \\ \Gamma \vdash 1.0 : \mathit{real} \\ \boxplus : \mathit{real} \times \mathit{real} \to \mathit{real}\end{array}}{\Gamma \vdash y \boxplus 1.0 : \mathit{real}}}{\Gamma \vdash \textbf{ if } x < 3 \textbf{ then } x + 1 \textbf{ else } y \boxplus 1.0 : \textbf{error}}$$

where $\Gamma = x : \mathit{int}, y; \mathit{real}, \emptyset$

The $\textsc{Tif}$ rule requires that both branches have exactly the same type: *int* and *real* are different types because there is no implicit conversion nor sub-typing

Soundness

- The soundness ensures that the type system does what it was supposed to do, i.e., it prevents type errors to happen at run-time (well-typed programs don't go wrong)
- A sound type system discards statically all programs that may cause a type error, but it may discard more due to approximations
- The precision of a type system can be improved by enriching the type language, e.g., by adding polymorphism or sub-typing

An example of approximation

$$\dfrac{\emptyset \vdash \mathit{true} : \mathit{bool} \qquad \dfrac{\begin{array}{c}\emptyset \vdash 2 : \mathit{int} \\ \emptyset \vdash 1 : \mathit{int} \\ + : \mathit{int} \times \mathit{int} \to \mathit{int}\end{array}}{\emptyset \vdash 2 + 1 : \mathit{int}} \qquad \dfrac{\begin{array}{c}\emptyset \vdash \text{"}\mathit{hello}\text{"} : \mathit{string} \\ \emptyset \vdash \text{"}\mathit{world}\text{"} : \mathit{string} \\ \oplus : \mathit{string} \times \mathit{string} \to \mathit{string}\end{array}}{\emptyset \vdash \text{"}\mathit{hello}\text{"} \oplus \text{"}\mathit{world}\text{"} : \mathit{string}}}{\emptyset \vdash \mathbf{if}\ \mathit{true}\ \mathbf{then}\ 2 + 1\ \mathbf{else}\ \text{"}\mathit{hello}\text{"} \oplus \text{"}\mathit{world}\text{"} : \mathbf{error}}$$

The expression does not type check, but it never raises a run-time type error, yet:

$$\mathbf{if}\ \mathit{true}\ \mathbf{then}\ 2 + 1\ \mathbf{else}\ \text{"}\mathit{hello}\text{"} \oplus \text{"}\mathit{world}\text{"} \to 2 + 1 \to 3$$

Statically, we cannot decide which branch of **if** to take, thus, we consider both of them

Typing algorithm

- So far, we have considered the logical presentation of a type system but the actually to run the analysis we need a type-checking algorithm
- There are different techniques for defining these algorithms: sometimes translating rules in code suffices, but often to deal with advanced features, e.g., polymorphism, we need other approaches
- However, the algorithm must be proved to compute what the type system prescribes

  Soundness: if the algorithm returns that a term M has type t, then the judgment $\vdash M : t$ must be valid

  Completeness: if the term M has type t, the algorithm must return t as type of M

Typing algorithm for fun

In the rule TFUN we guess the type of the parameter x

$$\textsc{TFun}\quad \frac{x : \tau_1, \Gamma \vdash e : \tau_2}{\Gamma \vdash \textbf{fun}\, x \to e : \tau_1 \to \tau_2}$$

but guessing is not feasible algorithmically
Two possible approaches:

1. Introducing type annotations, the programmer specifies the types
2. Inferring the type of values from their usage, no annotations (see later)

Adding type annotations to fun

We annotate the formal parameter of a functional value with its type

$$
\begin{array}{rll}
e \in Exp ::= & \ldots & \\
 | & \mathbf{fun}\, x : \tau_1 \to e & \text{lambda abstraction} \\
 | & \ldots &
\end{array}
$$

The rule TFUN becomes

$$
\text{TFUN}\quad \frac{x : \tau_1, \Gamma \vdash e : \tau_2}{\Gamma \vdash \mathbf{fun}\, x : \tau_1 \to e : \tau_1 \to \tau_2}
$$

Our type system becomes algorithmic and its rules can be directly implemented. The resulting algorithm is sound and complete.

Demo

See the file fun.ml

A more interesting and complete implementation is MiniML from PLZoo project:
`https://github.com/andrejbauer/plzoo/`

Conclusion

- Type systems are one of the most used technique for the analysis of programs
- We describe the case of simple type checking for a functional language: type annotations in functions
- Many other topics: type inference, polymorphism á la ML, parametric polymorphism, type system for security, type and effect systems, the different family of dependent types, objects and classes, flow-sensitive types.