A Static Type Inference for Python

Eva Maia
Faculdade de Ciências da Universidade do Porto
Portugal
emaia@dcc.fc.up.pt

Nelma Moreira
CMUP, Faculdade de Ciências da Universidade do Porto
Portugal
nam@dcc.fc.up.pt

Rogério Reis
CMUP, Faculdade de Ciências da Universidade do Porto
Portugal
rvr@dcc.fc.up.pt

ABSTRACT

Dynamic languages, like Python, are attractive because they guarantee that no correct program is rejected prematurely. However, this comes at a price of losing early error detection, and making both code optimization and certification harder tasks when compared with static typed languages. Having the static certification of Python programs as a goal, we developed a static type inference system for a subset of Python, known as RPython. Some dynamic features are absent from RPython, nevertheless it is powerful enough as a Python dialect and exhibits most of its main features. Our type inference system tackles with almost all language constructions, such as object inheritance and subtyping, polymorphic and recursive functions, exceptions, generators, modules, etc., and is itself written in Python.

Categories and Subject Descriptors

F.3.2 [**Logics and Meaning of Programs**]: Semantic of Programming Languages—*Program Analysis*; F.3.3 [**Logics and Meaning of Programs**]: Studies of Program Constructors—*Type structure*

General Terms

Languages, Theory, Verification

Keywords

Dynamic Languages, Python, Static Type Systems, Type Inference

1. INTRODUCTION

Formal software verification is becoming very important due to the increased need to certify software as reliable. Special attention is devoted to critical and embedded systems in order to ensure its integrity, safety, and correction. Due to performance constraints, these systems are usually implemented in C or Java. However, when time performance is not absolutely critical, the speed and simplicity of the development cycle and the safety and clarity of the code justify the use of high-level languages, such as Python [15].

Python is a very high-level, object-oriented programming language with a clear syntax which facilitates the readability of the code and program development. Python is dynamically typed and allows variables to take values of different types in different program places. When an assignment is evaluated the type of the assignment right side is given to the variable on left side. This variable could already have a type, but at the moment of the assignment it acquires the new type. Python supports inheritance, in other words, it allows to derive new classes from existing classes. A derived class has access to the attributes and methods of the base classes, and it can redefine them. Inheritance can be single or multiple depending on whether the class inherits from one or more existing classes. Python has also introspective and reflective features which allow to obtain, at runtime, information about the object types.

The enforcement of static type systems for several dynamic languages have been studied in the literature (see [7] and references therein). However, for Python only some partial or non formal approaches have been attempted [2, 3].

Our goal is to add a type discipline that ensures static type safety for Python programs. We begin by choosing a Python subset, called RPython, and that was introduced within the PyPy project [12]. The PyPy project aims to produce a flexible and fast Python implementation. The main idea is to write a high-level specification of the interpreter to be translated into lower-level efficient executables such as C/POSIX environment, JVM, and CLI [14]. Other goal of PyPy is to develop a Just-In-Time compiler for Python in order to improve the performance of the language. The PyPy interpreter is written in RPython [1], which is a subset of Python where some dynamic features have been removed. The main features of this language are:

a) the variables can not change type;

b) complex types have to be homogeneous, *i.e.* all elements of a list must have the same type; the dictionary keys must have all the same type, as well as the values, but the keys do not have to be of the same type of the values;

c) not having some introspective or reflective characteristics;

d) not allowing the use of special methods (__*__);

e) not allowing the definition of functions inside functions;

f) not allowing the definition and the use of global variables;

g) not allowing the use of multiple inheritance.

The first three features are the most relevant, as they are the ones that make possible the static type inference.

In the next sections we describe the development of a static type inference system for a superset of RPython, corresponding to features a) - c). We begin by describing formally the (abstract) syntax of the language and the type system. Then we present some type inference rules, and an illustrative example. Finally we comment on some experimental tests and address future work.

2. TYPE SYSTEM

A *type system* [10, 4] is composed by a set of types and a set of rules that define the assignment of these types to program constructors, in a given context. The association of a type τ to a constructor e is called a *type assignment*, and it is denoted by $e :: \tau$. A *context* Γ is a set of type assignments to variables. Given a context Γ, a constructor e and a type τ, $\Gamma \vdash e :: \tau$ means that considering the context Γ it is possible to infer that the constructor e has type τ.

The following grammar describes the abstract syntax of an extended version of RPython.

$$
\begin{array}{rcl}
p & ::= & [s_1, \ldots, s_n] \\
s & ::= & \mid x = e \mid x \; op \; = e \mid \texttt{del}\; e \mid \texttt{print}\; e \mid \texttt{pass} \\
 & & \mid \texttt{assert}\; e \mid \texttt{break} \mid \texttt{continue} \\
 & & \mid \texttt{return}\; [e]^? \mid \texttt{raise}\; [e]^? \\
 & & \mid \texttt{if}\; e :\; p\; [\texttt{else}\; p]^? \mid \texttt{while}\; e :\; p\; [\texttt{else}\; p]^? \\
 & & \mid \texttt{def}\; f\; (x_1, \ldots, x_n) :\; p \;\mid \texttt{for}\; x\; \texttt{in}\; e :\; p\; [\texttt{else}\; p]^? \\
 & & \mid \texttt{class}\; c() :\; p \mid \texttt{class}\; c(l) :\; p \\
 & & \mid \texttt{with}\; e\; [\texttt{as}\; x]^? : p \mid \texttt{try} : p_1\; \texttt{finally} :\; p_2 \\
 & & \mid \texttt{try} :\; p_1\; [\texttt{except}\; \bar{e}\; \texttt{as}\; x : p_2]^?\; [\texttt{else} :\; p_3]^? \\
 & & \mid \texttt{try} : p_1\; [\texttt{except}\; (\bar{e}_1, \ldots, \bar{e}_n) : p_2]^?\; [\texttt{else} : p_3]^? \\
 & & \mid \texttt{exec}\; e_1\; \texttt{in}\; e_2 \\
 & & \mid \texttt{exec}\; e \mid \texttt{exec}\; e_1\; \texttt{in}\; e_2, e_3 \mid \texttt{import}\; l\; [\texttt{as}\; l_1]^? \\
 & & \mid \texttt{from}\; l\; \texttt{import}\; * \mid \texttt{from}\; l\; \texttt{import}\; l_1, \ldots, l_m \\
e, \bar{e} & ::= & n \mid l \mid x \mid (e_1, \ldots, e_n) \mid [e_1, \ldots, e_n] \mid e\; op\; e \mid e\; opc\; e \\
 & & \mid e\; opb\; e \mid opu\; e \mid \{\bar{e}_1 : e_1, \ldots, \bar{e}_n : e_n\} \\
 & & \mid e[\bar{e}] \;\mid e[\bar{e}_1 : \bar{e}_2] \;\mid e[\bar{e}_1 :] \;\mid e[: \bar{e}_2] \\
 & & \mid e\; \texttt{if}\; e\; \texttt{else}\; e \mid [e_1\; \texttt{for}\; x\; \texttt{in}\; e_2\; (\texttt{if}\; e_3)^*] \\
 & & \mid f\; (e_1, \ldots, e_n) \mid c\; (e_1, \ldots, e_n) \\
 & & \mid e.m(e_1, \ldots, e_n) \mid e(\bar{e}_1, \ldots, \bar{e}_n).m\; (e_1, \ldots, e_n) \\
 & & \mid \texttt{lambda}\; e_1, \ldots, e_n : \bar{e} \;\mid \texttt{yield}\; [e]^?
\end{array}
$$

The optional elements are enclosed in $[\;]^?$. A program p is a statement list. A statement can be an assignment in which the right side must be an expression, a conditional, a function definition, a class definition, etc. An expression e is, for example, a number n, a constant l, a variable x, a tuple, a list, an operation, or a dictionary. In operations, we can use binary operators (*op*), comparison operators (*opc*), boolean operators (*opb*), and unary operators (*opu*).

2.1 Types

In Python, everything is an object, and every object is a class instance. However, we can assign types to them, according to the context in which they are used. Figure 1 presents a grammar for the set of types τ and type schemes η of our system.

$$
\begin{array}{rcl}
\tau, \alpha & ::= & \sigma \in TVar \mid eInt \mid eFloat \mid eLong \mid eString \\
 & & \mid eBool \mid eNone \mid eList(\tau) \mid eTuple(\tau_1, \ldots, \tau_n) \\
 & & \mid eAcc(\{i_1 : \tau_{i_1}, \ldots, i_n : \tau_{i_n}\}) \mid eDict(\alpha, \tau) \\
 & & \mid eIter(\tau) \mid eArrow([\tau_1, \ldots, \tau_n], \alpha) \\
 & & \mid eGen(\tau) \mid eGer(f, \Omega) \mid eClass(c, \Omega) \\
 & & \mid eAbs([(x_1, \eta_1), \ldots, (x_n, \eta_n)], \eta) \\
 & & \mid eCt(x, [\tau_1, \ldots, \tau_n]) \mid eMod(x, \Omega) \\
\eta & ::= & \tau \\
 & & \mid \; eAll([\sigma_1, \ldots, \sigma_n], \tau))
\end{array}
$$

Figure 1: Types

$TVar$ is the set of the type variables σ. Numeric values have type $eInt$, $eFloat$ or $eLong$. Constants have type $eString$. `True` and `False` are special constants which have type $eBool$, as well as the comparison operations. The statements which do not return a value have type $eNone$. Lists are homogeneous objects, *i.e.*, all their elements have the same type. A list has type $eList(\tau)$, where τ is the common type of all its elements. On the other hand, tuple elements do not need to be of the same type. Because of this, a tuple has type $eTuple([\tau_1, \ldots, \tau_n])$, where τ_i is the type of the element in the ith position.

If it is possible to infer a type only for some tuple positions and we don't know the tuple arity, type $eAcc(\{i_1 : \tau_{i_1}, \ldots, i_n : \tau_{i_n}\})$ is used to indicate that the i_jth tuple position has an object of type τ_{i_j}, for $1 \leq j \leq n$.

In a dictionary, all keys must have the same type α and all values must have the same type τ. Thus, $eDict(\alpha, \tau)$ is the dictionaries' type.

Python supports the concept of an iteration over a container. All the elements of a given container have the same type τ. So, $eIter(\tau)$ is the iterator type.

A non polymorphic function has type $eArrow([\tau_1, \ldots, \tau_n], \alpha)$, where τ_i is its ith argument type and α is the return type. The type of a polymorphic function is $eAll([\sigma_1, \ldots \sigma_n], eArrow([\tau_1, \ldots \tau_n], \alpha))$, where σ_i are type variables, τ_i is the ith argument type, and α is the return type.

The use a `yield` statement in a function definition is sufficient to turn that definition into a generator function instead of a normal one. The type of an `yield` statement is $eGen(\tau)$, where τ is the type of the associated expression. When a generator function is called it returns an iterator, known as *generator*. So, $eGer(f, \Omega)$ is the type of this kind of call, where f is the function name, and Ω is a local environment with the two methods that characterize the *generator*: *__iter__* and *next*. These generators are a simple tool for creating iterators, which permit the access to elements in a container, one at a time. The iterators type is $eIter(\tau)$.

The local context Ω of a class c assigns types to its methods, *i.e.* $\Omega \;\; ::= \;\; \{m_0 :: \eta_0, \ldots, m_n :: \eta_n\}$, where m_i are method names. A class has type $eClass(c, \Omega)$, where c is its name.

To deal with inheritance, we must consider the potential existence of the abstract methods. Often it is useful to define a class that serves as a *model* for a particular purpose, and the classes that inherit from it can complete that model with a more specific behaviour. In this case, methods of the *model* classes can use methods that do not belong to the current class but to a more specific one. This methods are called *abstract methods*. The type of an abstract method is $eAbs([(m_1, \eta_1), \ldots, (m_n, \eta_n)], \eta)$, where m_i is the name of a method that do not belong to the current class and η_i is its type, and η is the type of the abstract method if those methods exist in the class.

Union types are types describing values which type can be one of a set of types. The type $eCt(x, [\tau_1, \ldots, \tau_n])$ describes a collection of types.

A module is a file containing Python definitions and statements. Code reuse is one of the primary reasons for having modules. The $eMod(x, \Omega)$ is the type of a module which name is x and Ω is its local context.

2.2 Subtypes

The notion of subtype expresses the intuitive concept of inclusion between types, where types are treated as collections of values [10, 9]. Given a subtyping relation $<:$, τ is a subtype of α ($\tau <: \alpha$) if any term of type τ can be used in the context where a term of type α is expected.

The subtyping relation must be reflexive and transitive. Some of the subtyping rules are presented in the Figure 2. Notice that if a type is a type variable, unification must be used. As usual, the function rule SF is contravariant.

$$\tau <: \tau \tag{Ref}$$
$$eNone <: \tau \tag{Sen}$$
$$eInt <: eLong \tag{Sil}$$
$$eInt <: eFloat \tag{Sif}$$
$$eLong <: eFloat \tag{Slf}$$
$$\frac{\tau <: \alpha}{eList(\tau) <: eList(\alpha)} \tag{Sl}$$
$$\frac{\alpha_2 <: \alpha_1 \quad \tau_1 <: \tau_2}{eDict(\alpha_1, \tau_1) <: eDict(\alpha_2, \tau_2)} \tag{Sd}$$
$$\frac{\tau_i <: \alpha_i \quad 1 \le i \le n}{eTuple(\tau_1, \ldots, \tau_n) <: eTuple(\alpha_1, \ldots, \alpha_n)} \tag{St}$$
$$\frac{\alpha_i <: \tau_i \quad 1 \le i \le n \quad \tau <: \alpha}{eArrow([\tau_0, \ldots, \tau_i], \tau) <: eArrow([\alpha_1, \ldots, \alpha_i], \alpha)} \tag{Sf}$$
$$\frac{l_1 \subseteq l_2}{eCt(id_1, l_1) <: eCt(id_2, l_2)} \tag{SCt}$$

Figure 2: Subtyping Rules

The transitivity and the termination of the subtyping algorithm was proved.

3. TYPE INFERENCE RULES

Let $\Gamma ::= \{t_0 :: \tau_0, \ldots, t_n :: \tau_n\}$ be a context where $t_i \in \{x, f, c\}$. We denote by Γ_f the restriction of Γ to function and class names:

$$\Gamma_f ::= \{t_i :: \eta_i \mid \ \eta_i \texttt{ is } eArrow([\tau], \alpha) \ \text{ or } \eta_i \texttt{ is } eAll([\sigma], \alpha) \ \text{ or } \eta_i \texttt{ is } eClass(c, \Omega)\}$$

Figure 3 shows some of the our type inference rules. In the following paragraphs we describe briefly these rules.

(Var) A variable x has type τ, if the assignment of type τ to the variable x exists in Γ.

(Atr) An assignment of an expression e to a variable x has type *eNone*, if it is possible to infer $e :: \tau_1$ and $x :: \tau_2$, and $\tau_1 <: \tau_2$ or $\tau_2 <: \tau_1$.

$$\Gamma \vdash n :: \tau \tag{Num}$$
where $\tau \in \{eInt, eFloat, eLong\}$

$$\Gamma \ \vdash \ x :: \tau, \ if \ (x :: \tau) \ \in \ \Gamma \tag{Var}$$
$$\frac{\Gamma \vdash \ e :: \tau_1 \quad \Gamma \vdash \ x :: \tau_2 \quad \tau_1 <: \tau_2 \ or \ \tau_2 <: \tau_1}{\Gamma \vdash x = e \ :: \ eNone} \tag{Atr}$$
$$\frac{\Gamma \vdash \ e_1 :: \tau_1 \quad \Gamma \vdash e_2 :: \tau_2 \qquad \tau_1 <: \tau_2 \ or \ \tau_2 <: \tau_1}{\Gamma \vdash e_1 \ opc \ e_2 \ :: eBool} \tag{Opc}$$
$$\Gamma \vdash return \ :: eNone \tag{Return1}$$
$$\frac{\Gamma \ \vdash \ e :: \tau}{\Gamma \vdash return \ e \ :: \ \tau} \tag{Return2}$$
$$\frac{\begin{array}{c}\Gamma \vdash \ e_0 :: eBool \quad \Gamma \vdash e_1 :: \tau \\ \Gamma \vdash e_2 :: \alpha \qquad \tau <: \alpha\end{array}}{\Gamma \vdash if \ e_0 : \ e_1 \ else \ e_2 \ :: \alpha} \tag{Cond1}$$
$$\frac{\begin{array}{c}\Gamma \vdash \ e_0 :: eBool \quad \Gamma \vdash e_1 :: \tau \\ \Gamma \vdash e_2 :: \alpha \qquad \alpha <: \tau\end{array}}{\Gamma \vdash if \ e_0 : \ e_1 \ else \ e_2 \ :: \tau} \tag{Cond2}$$
$$\frac{\begin{array}{c}\bar{\Gamma} = \{x_i :: \tau_i\} \quad 1 \le i \le n \\ \bar{\Gamma} \cup \Gamma_f \cup \{e :: \tau\} \vdash e :: \alpha \quad \tau <: \alpha\end{array}}{\Gamma'' \vdash def \ f \ (x_1, \ldots, x_n) : e \ :: eNone} \tag{DefFunc}$$
where $\Gamma'' = \Gamma \cup f :: eArrow([\tau_1, \ldots, \tau_n], \alpha)$

$$\frac{\begin{array}{c}\Gamma \vdash f :: eArrow([\tau_1, \ldots, \tau_n], \alpha) \\ \Gamma \vdash \bar{e_i} :: \alpha_i \qquad \alpha_i \ <: \ \tau_i \quad 1 \le i \le n\end{array}}{\Gamma \vdash f(\bar{e_1}, \ldots, \bar{e_n}) \ :: \ \alpha} \tag{Application1}$$
$$\frac{\begin{array}{c}\Gamma \vdash f :: eAll([\sigma_1, \ldots, \sigma_n], eArrow([\tau_1, \ldots, \tau_n], \alpha)) \\ \Gamma \vdash \bar{e_i} :: \alpha_i \qquad \alpha_i \ <: \ \tau_i \quad 1 \le i \le n\end{array}}{\Gamma \vdash f(\bar{e_1}, \ldots, \bar{e_n}) \ :: \ \alpha} \tag{Application2}$$
$$\frac{\Gamma_f \subseteq \Gamma \quad \Gamma_f \vdash e_1 :: \eta_1, \ \ldots, \ \Gamma_f \vdash e_n :: \eta_n}{\Gamma'' \vdash class \ c() : [e_1, \ldots, e_n] \ :: \ eNone} \tag{DefCla1}$$
where $\Gamma'' = \Gamma \cup eClass(c, \{m_1 :: \eta_1, \ldots, m_n :: \eta_n\})$

$$\frac{\begin{array}{c}\Gamma_f \subseteq \Gamma \quad \Gamma \vdash l :: eClass(id, \Omega) \\ \Gamma' = \Gamma_f \cup \Omega \quad \Gamma' \vdash e_1 :: \eta_1, \ \ldots, \ \Gamma' \vdash e_n :: \eta_n\end{array}}{\Gamma'' \vdash class \ c(l) : [e_1, \ldots, e_n] \ :: \ eNone} \tag{DefCla2}$$
where $\Gamma'' = \Gamma \cup eClass(c, \Omega \cup \{m_1 :: \eta_1, \ldots, m_n :: \eta_n\})$

$$\frac{\begin{array}{c}\Gamma \vdash c :: eClass(c, \Omega) \\ \Gamma, \ \Omega \vdash \_\_init\_\_(e_1, \ldots, e_n) :: eNone\end{array}}{\Gamma \vdash c(e_1, \ldots, e_n) :: eClass(c, \Omega)} \tag{Inst1}$$
$$\frac{\begin{array}{c}\Gamma \vdash c :: eClass(c, \Omega) \\ \Gamma, \ \Omega \vdash \_\_init\_\_(e_1, \ldots, e_n) :: eClass(c, \Omega)\end{array}}{\Gamma \vdash c(e_1, \ldots, e_n) :: eClass(c, \Omega)} \tag{Inst2}$$
$$\frac{\begin{array}{c}\Gamma \vdash c(e_1, \ldots, e_n) :: eClass(c, \Omega) \\ \Omega \vdash m(\tau_1, \ldots, \tau_n) :: \eta \quad \Gamma \vdash \bar{e_i} :: \alpha_i \\ \alpha_i <: \tau_i \quad 2 \le i \le n\end{array}}{\Gamma \vdash c(e_1, \ldots, e_n).m(\bar{e_1}, \ldots, \bar{e_n}) :: \eta} \tag{Acm1}$$

Figure 3: Type Inference Rules

(Opc) One operand type must be a subtype of the other. The comparison operation *opc* has always type *eBool*. For example, $1 < 2 :: eBool$.

(Return[1,2]) When no value is returned, the type of the statement is *eNone*. Otherwise, the statement has the type of object that is returned. For example, $return\ 1 :: eInt$.

(Cond[1,2]) Conditional tests have type *eBool*. The type of one branch must be a subtype of the type of the other branch. For example, $if\ 2 < 3 :\ return\ 1.0\quad else :\ return\ 2 :: eFloat$, because $2 < 3 :: eBool$, $return\ 1.0 :: eFloat$, $return\ 2 :: eInt$, and $eInt <: eFloat$.

(DefFunc) A function has type $eArrow([\tau_1, \ldots, \tau_2], \alpha)$. The argument types are inferred in a new context $\bar{\Gamma}$. Initially we assign to the body of the function, a type variable σ which later is unified with α. This type assignment is added to context $\bar{\Gamma}$. The type of the function body is inferred in the context $\bar{\Gamma} \cup \Gamma_f$.

(Application[1,2]) The types of the function call arguments must agree with the correspondent parameter types of its definition. In that case, the type of the call is the return type of the function.

(DefCla[1,2]) In a class definition, we infer the type (η_i) of each of its methods (m_i). These types are stored in the class local context which is part of the class type. When a class B inherits from another class A, the context of the class A, removing all type assignments that occur in B, is added to the context of the class B.

(Inst[1,2]) When an instance of a class A is created, the function *__init__* is invoked in the class local context. Even if the *__init__* function would have *eNone* as return type, the instance type is always the class type.

(Acm1) Class methods may or may not have formal arguments. Either way a first argument, the context, is always present.

Besides the rules here presented our system has type inference rules for all constructors presented in the abstract syntax [8].

3.1 Example

Consider the following Python code:

```
class Num():                              1
  def __init__(self):                     2
    return self                           3
                                          4
  def fac(self, num):                     5
    if num <= 1:                          6
       return 1                           7
    else:                                 8
       return num*self.fac(num-1)         9
                                          10
f = Num().fac(3)                          11
```

The first statement is a class definition. According to the DefCla1 rule, we need to infer the type of the methods, using the DefFunc rule. In method fac, we infer the type of the body which is a conditional statement. For that the Cond1 rule is used: the type of num ≤ 1 has to be *eBool*. In the comparison, one argument must be subtype of the other, so num $:: \tau$ and $\tau <: eInt$. The type of the statement in line 7 must be a subtype of the statement in line 9. The type of the statement **return** 1 is *eInt*.To infer the type of the statement **return** num∗self.fac(num−1), the type of the expression is inferred. As we already know that num type is a subtype of *eInt*, the return type of function (self.fac(num−1)) is also a subtype of *eInt*, because they are the operands of a multiplication operation. The argument self is an special argument, which refers the class itself. In the following, when representing the class itself, we omit its local context. Recall that instead of *eInt* any supertype can be used, so the fac method has type

$$eArrow([eClass(Num), eCt(eInt, eFloat, eLong)],\\ eCt(eInt, eFloat, eLong)),$$

where *Int*, *Float*, and *Long* are the class names of the corresponding type. After inferring the type of __init__ and fac methods, we conclude that the type of the class Num is

$$\begin{aligned}&eClass(Num,\\&\{fac :: eArrow([eClass(Num), eCt(eInt, eFloat, eLong)],\\&\qquad eCt(eInt, eFloat, eLong),\\&\_\_init\_\_ :: eArrow([eClass(Num)], eClass(Num)\}).\end{aligned}$$

To infer the type of the assignment in line 11 we use the Atr* rule. The type of the assignment right side is *eInt*. As f does not have any type assigned, we conclude that f:: *eInt*.

4. CONCLUSIONS

Our type inference system is written in Python, using as input the program syntactic trees produced by the Python AST module, which defines the Python abstract syntax [13]. Built-in functions and some Python modules must be prototyped by hand, as they are not written in Python. We are currently testing the system with different applications, like FAdo [5] and Yappy [16]. We are also using some of the PyPy benchmarks [11] in the tests.

As future work, we aim to use this type inference system within a certification system based in Hoare logics, like the Why tool [6]. Why is multi-language and multi-prover in the sense that it can be used for annotated programs in various programming languages and the generated proof obligations verified by several automatic or interactive-assistant provers. The proof obligations, when checked, enforce the correctness and security of the program towards the annotated properties.

5. REFERENCES

[1] D. Ancona, M. Ancona, A. Cuni, and N. D. Matsakis. RPython: a step towards reconciling dynamically and statically typed oo languages. In *Proc. of DLS '07*, pages 53–64, New York, NY, USA, 2007. ACM.

[2] J. Aycock. Aggressive type inference, 2004.

[3] B. Cannon. Localized type inference of atomic types in Python. Master's thesis, California Polytechnic State University, 2005.

[4] L. Cardelli and P. Wegner. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys*, 17:471–522, 1985.

[5] FAdo Project Team. Fado: Tools for formal languages manipulation. http://fado.dcc.fc.up.pt/. Access date: 1.04.2011.

[6] J.-C. Filliâtre. Why: a multi-language multi-prover verification tool. Technical Report 1366, LRI, Université Paris Sud, March 2003.

[7] M. Furr, J.-h. D. An, J. S. Foster, and M. Hicks. Static type inference for Ruby. In *Proceedings of the 2009 ACM symposium on Applied Computing*, SAC '09, pages 1859–1866, New York, NY, USA, 2009. ACM.

[8] E. Maia, N. Moreira, and R. Reis. A static type inference system for rpython (with extensions). Technical Report DCC-2011-04, FCUP (Faculdade de Ciências da Universidade do Porto), Porto, Portugal, 2011.

[9] J. C. Mitchell. Type inference with simple subtypes. *J. Funct. Program.*, 1(3):245–285, 1991.

[10] B. C. Pierce. *Types and Programming Languages.* MIT Press, 2002.

[11] PyPy Project Team. Pypy benchmarks. http://codespeak.net/svn/pypy/benchmarks/. Access date: 1.04.2011.

[12] PyPy Project Team. PyPy: flexible and fast Python implementation. http://codespeak.net/pypy. Access date: 1.04.2011.

[13] Python Software Foundation. Abstract syntax trees. http://docs.python.org/library/ast.html. Access date: 1.04.2011.

[14] A. Rigo and S. Pedroni. Pypy's approach to virtual machine construction. In *OOPSLA '06*, pages 944–953, New York, NY, USA, 2006. ACM.

[15] G. Rossum. Python reference manual. Technical report, CWI (Centre for Mathematics and Computer Science), Amsterdam, The Netherlands, 1995.

[16] Yappy Project Team. Yappy: Lr parser generator for Python. http://www.dcc.fc.up.pt/ rvr/naulas/Yappy/. Access date:1.04.2011.