1 Introduction

The typing system of a programming language determines some of the features available for the particular language. As an example, since Lisp is a dynamically typed language the type checking is done at run time. One can therefore use heterogeneous data types, for example the list [1, true, "string"]. This is not the case in programming languages that have strong static typing systems, as is the case of SML of New Jersey, or Pascal.

We call a typing system dynamic if the type checking of operands is done at run time, that is, if the typechecking of operands is done exactly before the operation is executed. If the typing system does the checking at compile time it is a static typing system. Examples of dynamically typed programming languages are Lisp and Smalltalk.

In practice, in dynamically typed programming languages, only primitive values and predefined functions have types. For example, the function car of Lisp has type list → S-expression, that is, it takes as input a list and returns an S-expression as a result, but the function first, defined as (defun first (x) (car x)) has no type. The expression (first 3) will eventually produce a runtime type error, since car expects a list, and this error will be raised when car is applied to 3, and not before, as would be done in a language like ML.

A dynamic typing discipline offers a high degree of flexibility that a static typing discipline cannot give. As an example, there are some program phrases that can be executed in a dynamic type checking discipline that cannot be written in a language with a static type checking discipline. An example of such a program is \((\lambda f . ((fK)(fI)))\) where \(K = \lambda x.\lambda y.x\) and \(I = \lambda x.x\). This term is rejected by the ML typing system because the types of \(K\) and \(I\) are not unifiable. If we "execute" this term, we get \(KI\) which reduces to \(\lambda x.I\), which has type \(\forall x.\forall y.\forall a. \beta\rightarrow \beta\). This program translated to Lisp runs without problems. This example shows another problem with statically typed languages. Sometimes there are programs that contain no typing error, but will be rejected by the typing system.

On the other hand, statically typed languages offer early error detection, and they give the opportunity for code optimization. Typically, a Lisp compiler has to "insert" code to do the necessary type-checking during run time, making the execution of programs more inefficient than it would be if the program had been written in a statically typed language, like ML.

One problem that sometime arises with statically typed languages is that programs that use some kind of external data cannot be typechecked at compile time, as are programs that read external files or distributed programs that make use of remote procedure calls. In these cases, some degree of dynamic typing is needed. As an example taken from [1] consider a program that reads a bitmap and displays it. If the bitmap is stored in an external file, the program has to read the contents of the file. If the contents is the representation of a bitmap, then there is no problem. If it is not, then there can be two ways of solving this, one is to treat the contents of the file as if it was a bitmap, and the other is to check that the contents of the file is a representation of a bitmap, and in case it is not raise an exception. So some dynamic typing is needed even in statically typed languages.

Another disadvantage of statically typed programming languages is the impossibility to have non homogeneous data objects. As an example, it is possible to handle the list \([1, true, "string"]\) in Lisp, but not in ML. This also motivates the addition of dynamisms to static languages [11].

Since it is undecidable whether a program is type correct, our aim is to find a typing system that allows greater flexibility by allowing programs to run even if there is no assurance that they contain no type error, but stops all programs it can guarantee will have type mismatches at run-time.

We present a type inference algorithm that infers types in the context of "Quasi Static Typing", which was introduced by Thatte in [12]. The idea is to have a system that combines static and dynamic types, not accepting some expressions that will lead to run time type error, while accepting others that may lead to error at run time.

In section 2 we present Thatte's Quasi-Static typing systems. In section 3 we present our type inference system based on QST, and finally. We conclude the paper in section 4.

2 Quasi-Static Typing (QST)

In this section we introduce the Quasi-static typing system, and present some basic results.

Quasi-Static Typing (QST) was introduced by Satish Thatte in [12]. It is a combination of Partial Types (see [11]) and automatic insertion of implicit positive (tagging) and negative (checking) coercions, see [12].

QST divides programs into three categories, well typed, ill typed and ambivalent, and it has two phases that can be integrated into one "pass". The first pass inserts implicit coercions where there is the possibility of a type mismatch, and the other does "plausibility checking". Ill typed programs are those that the system can prove will lead to run time type error, and therefore "rejected" by the typing system. Programs that pass the second phase are either well typed or ambivalent and are allowed to run. Well typed programs will never produce a run time type error, whereas ambivalent programs may or may not end in a run time type error.

The types assigned by QST are Partial Types, which were originally introduced in [11] with the intention of type checking heterogeneous objects. They include the type \(\Omega\) and a subtype relation. \(\Omega\) is assignable to all objects with the exception of a special object called \(\mathsf{wrong}\) which denotes run time type error. Partial types are defined by:

\[
\tau ::= \mathsf{int} \mid \mathsf{bool} \mid \Omega \mid \tau \rightarrow \tau
\]

The subtype relation, denoted \(\leq\) is defined as follows:

\[
\forall \tau, \tau' \leq \Omega
\]

\[
\tau_1 \rightarrow \tau_2 \leq \tau'_1 \rightarrow \tau'_2 \iff \tau'_1 \leq \tau_1 \text{ and } \tau_2 \leq \tau'_2
\]

The object language used here differs in a non relevant way from the one used in [12], and is defined by:

---

26

XXIV Conferencia Latinoamericana de Informática

Quito - Ecuador

27