Predicative Polymorphism in $\pi$-Calculus

Vasco T. Vasconcelos
Keio University
Department of Computer Science
3-14-1 Hiyoshi Kohoku-ku Yokohama 223 Japan
vasco@mt.cs.keio.ac.jp

Abstract

We present a formulation of the polyadic $\pi$-calculus featuring a syntactic category for agents, together with a typing system assigning polymorphic types to agents. The new presentation introduces an operator to express recursion, and an ML-style let-constructor allowing to associate an agent to an agent-variable, and use the latter several times in a program. The essence of the monomorphic type system is the assignment of types to names, and multiple name-type pairs to programs [14]. The polymorphic type system incorporates a form of abstraction over types, and inference rules allowing to introduce and eliminate the abstraction operator. The extended system preserves most of the syntactic properties of the monomorphic system, including subject-reduction and computability of principal typings. We present an algorithm to extract the principal typing of a process, and prove it correct with respect to the typing system. We also study, in the context of $\pi$-calculus, some well-known properties of the let-constructor.

1 Introduction

Predicative polymorphism allows to write a piece of code once, and use it in different contexts, with different types, instances of the type assigned to the original code.

The original proposal of the $\pi$-calculus [8] uses agent-constants and agent equations to provide for recursion. Agents are processes abstracted on a sequence of names, usually the free names of the process, while equations assign agent-constants to agents. Recursion is obtained by allowing an agent-constant to appear in the right-hand side of its defining equation. A typical program consists of a collection of equations defining agents, and a process instantiating these agents through their associated constants.

This mechanism not only provides for recursion, but also allows a form of polymorphism. In fact, an agent constant can be used to instantiate processes with different name-structures for the names in the instantiated process. This
name-structure, arising in name-passing calculi by the simple fact that names carry names, is captured by the notion of types for names [2, 7, 11, 13, 14]. Then we are in presence of a case of predicative polymorphism where names in the instantiated processes have different types, instances of a more general (polymorphic) type for the agent.

The particular formulation of the \( \pi \)-calculus we propose introduces a new syntactic category for agents (cf. [7, 12]), variables over agents (in contrast with constants), and two new constructors for processes. Agents are either processes abstracted on a sequence of names, \((x^n) P\), for \( x^n = x_1 \cdots x_n \) a sequence of names and \( P \) a process, or recursively defined agents, \( \text{rec } X.A \), binding agent-variable \( X \) in agent \( A \). The new process-constructors are agent application \( Av^n \), and an ML-like constructor \( \text{let } X = A \text{ in } P \), binding agent-variable \( X \) in process \( P \).

This new formulation leads to a natural notion of predicative polymorphism. Following Mitchell [9], the word predicative refers to the fact that polymorphism is introduced only after having defined all the base (monomorphic) types. To names we assign types meant to describe the kind of names a name is able to carry: when a name \( a \) carries a sequence of names of types \( \alpha_1, \ldots, \alpha_n \), we assign type \( (\alpha_1 \cdots \alpha_n)_a \) to \( a \). To agents we assign sequences of types: if \( x^n \) are the free names in a process \( P \), and \( \alpha^n \) their associated types, then to agent \((x^n) P\) we assign \( \alpha^n \), and call it a monomorphic type for the agent. In this way, we can abstract \( \alpha^n \) on some type-variable \( t \) to obtain a polymorphic type \( \forall t.\alpha^n \) for the agent.

We introduce a pair of rules allowing to abstract a given type on a particular type-variable, and to eliminate this from a polymorphic type. The polymorphic system preserves the important properties of the simple system, including the subject-reduction property and the existence and computability of principal typings. An algorithm to extract the principal typing of a process (cf. [1, 5, 16] for \( \lambda \)-calculi, and [14, 15] for name-passing calculi) is presented and proved correct with respect to the typing system. We also discuss some properties of the let- constructor and the relationship between the polymorphic and the monomorphic type systems, results that show remarkable similarities to what happens in ML.

The outline of the paper is as follows. The next section introduces a version of the polyadic \( \pi \)-calculus with variables over agents. Section 3 extends the basic typing assignment system [14] to handle the new constructors, and studies some properties of the system thus obtained. Section 4 introduces the polymorphic typing assignment system, and Section 5 presents an algorithm to extract the principal typing of a process. Section 6 discusses some further issues.

2 Polyadic \( \pi \)-calculus with variables over agents

This section presents a formulation of the polyadic \( \pi \)-calculus, where variables over agents and an explicit recursion operator provide for unbounded computation power. Unlike the original proposal of the calculus [8], we do not rely on auxiliary equations to provide for infinite behaviour, but else bring these equations directly into the syntax of processes, by introducing the notions of agents