Summary of Language primitives and type disciplines for structured communication-based programming

Rumyana Neykova rumi.neykova@gmail.com

January 28, 2011

1 Paper Summary

In this paper the authors introduce the concept of binary session types as a way to express a series of reciprocal interactions between two parties. Basic language primitives and solid type discipline based on session types are established as a foundation for communication-based concurrent programming. The main theorems for the type system are formalized. Authors also show how proposed primitives can be used to express well-known communication patterns, applied in the programming languages, such as call-return and method invocation. At first the authors give a brief overview of the existing approaches for software based on communication,thus showing the need and importance of a new concept. It is pointed out that neither of the approaches provides a clear and concise way to express a series of reciprocal interactions between two parties. Such communications are expressed as series of distinct interactions, which leads to less code readability and concurrency bugs. The theory of session types aims to brings that gap.

2 Basic concepts and Operational Semantics

A central concept is the notion of session. A session a is series of interactions which serves as a unit of conversation. The communication between the parties is done via ports (channels).When a session is established a fresh channel is generated and it is used later on. In order to be able to describe complex communication structures via session types the authors present basic communication primitives(value passing, branching/selection, delegation). The standard synchronous message passing, known form CSP, is expressed through the data sending/receiving primitives. Branching/selection are a purified form of method invocation in Object-Oriented Programming. Channel sending/receiving is called delegation and involves passing a channel that is being used in a session to another process.The Standard structuring constructs for concurrent programming and pi-calculus(such as parallel composition, name hiding, conditional and recursion) are also given. Let's look at some of the interesting notations that are used.*request a(k) in P accept a(k) in P* denotes session initiation. Note that the channel k is bounded in both request and accept since the $(\cdot)$ is used as a

binder.The syntax is a little bit odd, since one might expects something similar to a <k> (request to a(k) correspond to the bound output in pi-calculus [1]) since we have output of the channel. The reason for the syntax and for channel k being bounded in request a(k) is explained in details in [2].Another a bit different syntax is the recursion(*Def X[$\tilde{e}\tilde{k}$] in P*) so let's give an example in order to illustrate it:
Def
foo($\tilde{x}$,$\tilde{k}$) = request a(k) ....foo2(...) and
foo2($\tilde{x}$,$\tilde{k}$) = ...foo(...) and ...
in foo(...)|foo2($\tilde{v}$,$\tilde{k}$)|...

The notations and semantics of the free names and free variables are preserved, two new notation are introduces: free channels (fc) and free process variables (fpv).Process of free variables and free channels is called program. After that the paper presents the operational semantics in terms of the reduction rules. I'll point out only the [Link] reduction rule since it illustrates one of the most important concepts of session types:

$$\text{[Link] (accept a(k) in } P_1)|(\text{request a(k) in } P_2) \rightarrow (\nu k)(P_1|P_2)$$

As can be seen from the rule [Link] and as we already emphasized a new name is generated whenever a session is establishes.

After introducing the operational semantics the paper shows how communication patterns such as call-return and method-invocation can be expressed in terms of session types. The semantics of the patterns are given by that of the structuring primitives via translation. From the examples can be seen the importance of the complementary types (definition is given in [1]) for the verification of the protocols. It can be concluded that although the translations are not complex at all, they succeed to declare the fixed communication patterns in concise way, even for the more complex scenarios.

3 Typing system

The basic notation for the typing system:
Typing Δ : finite partial map from channels to types
Basis Θ: finite partial map from variables to sequence of sorts and types
Sorting Γ: finite partial map from names to variables
Main sequence of the typing system: $\Theta;\Gamma \vdash P \triangleright \Delta$ should be read as ″under the environment $\Delta;\Gamma$ a process P has a typing Δ ″.Typing system syntax express the syntax for sorts (sorts are types different than session types) and session types.Some nontrivial sorts that are given are:<a,a> represents the type for shared names, $\perp$ indicates that no further connection is possible:$\mu t.\alpha$ denotes a recursive behavior, the process starts by doing α and, when t is encountered, recurs to α again.Example of recursive type: μx.s!<v>.x. Other important concepts in the paper are: compatible types and typing composition.Compatibility means that each channel k is associated with complementary behavior, thus ensuring the inaction on k to run without errors.Typing composition is essential for the parallel rule.The authors also formalize the basic theorems for the typing system: Invariance under $\equiv$, Subject reduction and Lack of run-time errors

References

[1] Honda, K., V. T. Vasconcelos and M. Kubo, Language primitives and type disciplines for structured communication-based programming, in: ESOP98, LNCS 1381 (1998), pp. 22138.

[2] Nobuko Yoshida and Vasco Thudichum Vasconcelos. Language primitives and type discipline for structured communication-based programming revisited: Two systems for higher-order session communication. Electr. Notes Theor. Comput. Sci., 171(4):7393, 2007.