Constraints for Free in Concurrent Computation

Joachim Niehren Martin Müller
Programming Systems Lab
German Research Center for Artificial Intelligence (DFKI)
Stuhlsatzenhausweg 3, 66123 Saarbrücken, Germany
{niehren,mmueller}@dfki.uni-sb.de

Abstract. We investigate concurrency as unifying computational paradigm which integrates functional, constraint, and object-oriented programming. We propose the \( \rho \)-calculus as a uniform foundation of concurrent computation and formally relate it to other models: The \( \rho \)-calculus with equational constraints provides for logic variables and is bisimilar to the \( \gamma \)-calculus. The \( \rho \)-calculus without constraints is a proper subset of the \( \pi \)-calculus. We prove its Turing completeness by embedding the eager \( \lambda \)-calculus in continuation passing style. The \( \rho \)-calculus over an arbitrary constraint system is an extension of the standard \( cc \)-model with procedural abstraction.

1 Introduction

Concurrent computation allows the unification of many programming paradigms. This observation underlies Milner's \( \pi \)-calculus [14, 13], Saraswat's concurrent constraint (\( cc \)) model [21], and Smolka's \( \gamma \)-calculus [23]. It is also central to the actor model by Hewitt and Agha [1]. Concurrency is the key to the programming language Oz [24] which integrates functional [16], object-oriented [7] and constraint programming [9, 15].

In this paper we start to relate several computational calculi. An overview is given in the picture below. We formulate the relations by comparison with the \( \rho \)-calculus [19], a concurrent calculus with first-order constraints, higher-order procedural abstraction, and indeterminism via cells. Any constraint system determines an instance of the \( \rho \)-calculus. The \( \rho \)-calculus serves as a foundation of the concurrent constraint language Oz [24], is part of its language definition [25] and a basis for its implementation [11].

We prove bisimilarity of the \( \gamma \)-calculus [23] and the calculus \( \rho(x=y) \): The \( \gamma \)-calculus has been designed to model concurrent objects, while \( \rho(x=y) \) instantiates the \( \rho \)-calculus with equational constraints to provide for logic variables. Our bisimulation allows to consider the \( \rho \)-calculus as an extension of the \( \gamma \)-calculus.
with constraints. To obtain this result, we simplified the original ρ-calculus [19]:

Now, constraints actually “come for free” in ρ, in contrast to previous extensions of γ with constraints [22, 23, 19, 25].

The ρ-calculus over the trivial constraint system ρ(θ) is a proper subset of the asynchronous polyadic π-calculus [12, 3, 8]. This result is immediate from the identification of procedural abstractions with replicated input agents. Once-only input agents are not available in ρ(θ). Surprisingly, ρ(θ) is still Turing complete: Higher-orderness allows us to embed the eager λ-calculus. A continuation passing style [20] avoids logic variables which have been employed in an earlier embedding of the eager λ-calculus into γ [23, 16].¹ We prove the adequacy of our embedding based on a simulation and uniform confluence [16].²

The ρ-calculus is syntactically compositional: Constraints, applications, conditionals, and cells can be freely combined by composition, declaration, and abstraction. The reduction relation of ρ is defined up to a structural congruence, as familiar from recent presentations of π [13, 3, 8] and γ [23]. The central novelty in the version presented here is the distinction of logical conjunction (∧) on constraints from composition (λ). In the standard cc-model [21, 5, 6], these distinctions hold implicitly due to a monolithic constraint store. In a compositional syntax, the separation of conjunction and composition is central since it yields simple normal forms.

On reduction, applications, cells, or conditionals interact with an arbitrary constraint in their environment, but only one of them. For instance, the conditional if \( x=y \) then \( E \) else \( F \) fi is irreducible in the context of \( x=1 \land y=1 \), since none of \( x=1 \) or \( y=1 \) entails or disentails \( x=y \), but reducible in the context of \( x=1 \land y=1 \models x=y \). Constraints must be combined explicitly by reduction:

\[
\phi_1 \land \phi_2 \rightarrow \phi_1 \land \phi_2
\]

This combination rule is the essential difference of the ρ-calculus in this paper to its predecessor [19]. It plays the role of elimination in γ where no conjunction is apparent:

\[
\exists x \exists y(x=y \land E) \rightarrow \exists y E[y/x]
\]

The separation of conjunction and composition leads to a transparently distributed constraint store. From this point of view, combination can be interpreted as unification which may or may not involve a network transfer.

Related Work: Most surprisingly, the lazy λ-calculus can be embedded into ρ(θ) with call-by-need complexity (see [17]). Alternatively, the call-by-need λ-calculus [2] could be directly embedded into ρ(θ). Both results are stronger than the analogous results for π [4], since both embeddings map into a uniformly confluent subset of ρ(θ) and π. Furthermore, Milner’s embedding of the lazy λ-calculus into π [13] does not capture call-by-need complexity, and Smolka’s

¹ We owe the idea to personal communication with Gert Smolka and Martin Odersky.
² Indeterminism via cells cannot arise in ρ(θ) and is not needed for functional computation.