Compiling Pattern Matching in Join-Patterns

Qin Ma and Luc Maranget

INRIA-Rocquencourt, BP 105, 78153 Le Chesnay Cedex, France
{Qin.Ma, Luc.Maranget}@inria.fr

Abstract. We propose an extension of the join-calculus with pattern matching on algebraic data types. Our initial motivation is twofold: to provide an intuitive semantics of the interaction between concurrency and pattern matching; to define a practical compilation scheme from extended join-definitions into ordinary ones plus ML pattern matching. To assess the correctness of our compilation scheme, we develop a theory of the applied join-calculus, a calculus with value-passing and value matching.

1 Introduction

The join-calculus [5] is a process calculus in the tradition of the π-calculus of Milner, Parrow and Walker [16]. One distinctive feature of join-calculus is the simultaneous definition of all receptors on several channels through *join-definitions*. A join-definition is structured as a list of *reaction rules*, with each reaction rule being a pair of one *join-pattern* and one *guarded process*. A join-pattern is in turn a list of channel names (with formal arguments), specifying the synchronization among those channels: namely, a join-pattern is matched only if there are messages present on all its channels. Finally, the reaction rules of one join-definition define competing behaviors with a non-deterministic choice of which guarded process to fire when several join-patterns are satisfied.

In this paper, we extend the matching mechanism of join-patterns, such that *message* contents are also taken into account. As an example, let us consider the following list-based implementation of a concurrent stack:[1]

$$\begin{array}{ll} \mathsf{def} & \mathit{pop}(r) \mathbin{\&} \mathit{State}(x::\mathit{xs}) \triangleright r(x) \mathbin{\&} \mathit{State}(\mathit{xs}) \\ \mathsf{or} & \mathit{push}(v) \mathbin{\&} \mathit{State}(\mathit{ls}) \triangleright \mathit{State}\ (v::\mathit{ls}) \\ \mathsf{in} & \mathit{State}(\texttt{[]}) \mathbin{\&} \ldots \end{array}$$

The second join-pattern $\mathit{push}(v)$ & $\mathit{State}(\mathit{ls})$ is an *ordinary* one: it is matched whenever there are messages on both *State* and *push*. By contrast, the first join-pattern is an *extended* one, where the formal argument of channel *State* is an *(algebraic) pattern*, matched only by messages that are cons cells. Thus, when the stack is empty (*i.e.*, when message [] is pending on channel *State*), pop requests are delayed.

[1] We use the Objective Caml syntax for lists, with *nil* being [] and *cons* being the infix ::.

P. Gardner and N. Yoshida (Eds.): CONCUR 2004, LNCS 3170, pp. 417–431, 2004.