Logical Operational Semantics of Parlog
Part II: Or-Parallelism *

Egon Börger
Dip. di Informatica
C.so Italia 40
I-56100 PISA
boerger@dipisa.di.unipi.it

Elvinia Riccobene
Dip. di Matematica
V.le Andrea Doria 6
I-95125 CATANIA
riccobene@mathct.cineca.it

Abstract

This paper refines the definition of a complete, mathematical semantics for the parallel logic programming language PARLOG provided in [3] by giving an explicit Evolving Algebras formalization of the OR-Parallelism in Parlog. In particular we extend the algebras of [3] by new rules which describe the dynamics of the crucial candidate clause search in Parlog which was left abstract in [3].

1 Signature extension of Parlog Algebras

Due to space limitations we skip the Introduction and refer the reader for motivation and basic definition to [1], [2], [3] of the latter of which this paper is a direct sequel. The abstract function candidate-clause(database, node, lit), which in [3] realizes the whole Parlog reduction process of a literal lit yielding as result the body of the selected clause and a unifying substitution of its head and lit, receives here an explicit description by Evolving Algebras rules. To this purpose we extend the previous Parlog Algebras, introducing new universes and functions and modifying the existing ones.

Due to space restrictions we can spell out here only a few new definitions and crucial rules, refering the reader for a full account to [4]. In the full paper [4] we prove the correctness of this candidate-clause specification w.r.t. its definition in [3] by showing that our extended Parlog algebras developed here are "conservative" over the Parlog algebras of [3].

When the Parlog computation system begins the reduction process of a given procedure call lit, it comes into the so-called test-commit-output-spawn phase. In the test phase

*The second author has been partially supported by "Progetto Finalizzato Sistemi Informatici e Calcolo Parallelo" of CNR, under Grant n.90.00671.69.
the system tries to find a clause which satisfies the **candidate clause** condition among those defining the procedure which comes in the form of

\[
\text{procdef}(\text{database}, \text{lit}) = C_{1_1}.C_{1_2}.\cdots.C_{1_m};\cdots;C_{n_1}.C_{n_2}.\cdots.C_{n_m}.
\]

The **parclauses** \( S_i \) are searched through sequentially (**seq-search**). Each **parclause** is a sequence \( C_1.C_2.\cdots.C_m \) of **guarded clauses** which are searched through in parallel (**or-search**). The **try-clause** computation, whether a clause is a **candidate clause** for \( \text{lit} \), checks whether the clause is an unguarded clause for which **input matching** \(^1\) of \( \text{lit} \) and **head** succeeds, or whether it is a guarded clause for which: a) the **input-match** computation (unification of \( \text{lit} \) and **head**) succeeds with a substitution \( s \); b) the **guard-eval** computation of the clause guard by the program succeeds with a substitution \( s' \); c) the substitutions \( s \) and \( s' \) are consistent. (**seq-search, or-search, try-clause, input-match, guard-eval** are new tags.)

When a **candidate clause** is found, the calling literal \( \text{lit} \) commits to it (**commit** phase) - interrupting the search for (other) candidate clauses -; output unification is performed between the output mode argument terms of \( \text{lit} \) and those of the selected clause (**output phase**); the \( \text{lit} \) computation is reduced to the evaluation of the **candidate clause**'s body under the computed output substitution (**spawn** phase).

For an explicit description of this **candidate clause** search, we imagine each **or-par** node as root of a computation subtree performing the **test-commit-output** phase and of another (later created) subtree to perform the **spawn** phase.

During the **input-match** computation a node may come into a new mode **suspended**, namely when there is a unifier for the calling literal and the clause head which however tries to bind a variable occurring (in a term) in an **input** argument position of the call. During the **spawn** phase we need a function \( \text{sub} : \text{Node} \rightarrow \text{Sub} \) which yields a substitution which is known at the given node, but which is not transparent to the main system. In the **or-par** node subtree description, the function \( \text{goal} \) will be used to pass from parent to child the calling literal \( \text{lit} \) which is responsible for the **candidate clause** search. The same function will be used to report from child to parent the **body** of the identified **candidate clause** for the calling literal.

Some other new mostly self explanatory functions will be introduced where needed in the rules.

### 2 The **or-par** node operation

The transition rules of Part I (see [3]) that perform the **or-par** node operation, are replaced by those that we are describing in this section.

When the **or-par** node has mode **starting**, it has just been created as child of a node

---

\(^1\) **Input matching** (of a literal \( \text{lit} \) and a term \( t \)) is defined as unification of \( \text{lit} \) and \( t \) in which no variable is substituted which occurs in an **input argument** of \( \text{lit} \) (with respect to that program).

In contrast to input matching one speaks of output unification to refer to a unification of two terms which appear in an **output mode argument** (of a literal w.r.t. a given program). This façon de parler stresses that for the unification of terms occurring in **output mode arguments** there is no restriction on the direction of the bindings (from goal to clause head (**output**) or viceversa (**input**)).