Modular Transformations of CLP Programs.

Sandro Etalle[1,2], Maurizio Gabbrielli[1]

[1] CWI
P.O. Box 94079, 1090 GB Amsterdam, The Netherlands.

[2] Dipartimento di Matematica Pura ed Applicata,
Università di Padova,
Via Belzoni 7, 35131 Padova, Italy.

email: {etalle, gabbri}@cwi.nl

Abstract

In this paper we propose an unfold/fold transformation system for Constraint Logic Programs. The framework is inspired by the one of Tamaki and Sato for pure logic programs [18]. The use of CLP permits a more concise definition for the folding operation. We provide conditions for applying the system in a modular way. Under these conditions the system is correct wrt a semantics (Ω-semantics) which is compositional wrt the union of programs. As corollaries we also prove the correctness of the non-modular system wrt the answer constraint semantics and the least model semantics. Finally, we show how these results can be applied to the logic programming case.

1 Introduction

As shown by a number of applications, program transformation is a powerful methodology for program development. In this field, the unfold/fold transformation rules were first introduced by Burstall and Darlington [5] for transforming clear, simple functional programs into equivalent, more efficient ones, and then adapted to logic programs both for program synthesis and for program specialization and optimization. Soon later, Tamaki and Sato [18] proposed an elegant framework for the transformation of logic programs based on unfold/fold rules. Their system was proved to be correct wrt the least Herbrand model semantics [18] and the computed answer substitution semantics [14]. The system was then extended to logic programs with negation and serious research effort has been devoted to proving its correctness wrt the various semantics available for normal programs.

The aim of this paper is to present an unfold/fold transformation system for CLP

based on [18]. The full use of CLP allows us to give new applicability conditions for the folding operation which are more concise and simple than the ones in the literature, in particular the use of substitutions is avoided.

We also introduce a definition of *modular* transformation sequence, which is obtained by adding to the standard definition some new simple applicability conditions. This allows us to transform separate parts of programs independently, and then to combine the results of transformation while preserving the original meaning of the program. This kind of modularity is particularly relevant from a practical point of view. Even if the program is not completely specified in all its components, we would often like to perform some transformations on it without affecting its original meaning. Such a possibility supports an incremental development of programs, according to a well established software-engineering technique. Each time a new part of program is added, instead of transforming the whole program from scratch, we can compose the transformed version of the new piece of program with the transformed version of the old one.

Our system is proved to be correct wrt a generalization of the Ω-semantics ([4]) for constraint logic programs. There are two main reasons that lead us to the choice of such a semantics.

First, it is a declarative semantics which generalizes both the least model semantics [12] and the answer constraint semantics [9]. As a corollary, we can then prove the correctness of the system wrt these semantics as well, and this applies also to the standard (non-modular) system.

Secondly, the Ω-semantics is *compositional* wrt the union of programs, that is, the semantics of a program can be obtained from the semantics its subprograms, or modules[1]. As such, the Ω-semantics provides a natural reference semantics for a modular transformation system: the correctness of the system wrt this semantics ensure us that we can safely recompose the results of separate transformations of modules.

Recently, an extension of the Tamaki-Sato method to CLP programs has also been proposed by Bensaou and Guessarian [2], in Section 6 we compare it to the one defined here. Another related work is the one of Maher [17], which considers transformations of deductive databases (with constraints, and allowing negation in the bodies of the clauses), and refers to the perfect model semantics. The main difference between this paper and [17] lies in the presence of negation (which is not allowed here) and the fact that the definition of folding used in [17] is rather restrictive, in particular it lacks the possibility of introducing recursion. This kind of folding has also been investigated in [11, 3].

The paper is organized as follows: Section 2 contains the preliminaries on CLP programs and the definition of the program composition we consider. Section 3 provides the definition of (modular) transformation sequences for CLP. In Section 4 we give the definition of Ω-semantics and we prove the correctness of the transformation. The system is then compared with Tamaki-Sato's for pure logic program in Section 5. Finally, Section 6 concludes by comparing our results to those contained in [2]. All

[1] This kind of compositionality is also called *OR*-compositionality or $\cup$-compositionality.

the proofs can be found in [7].

2 Preliminaries: CLP programs

The *Constraint Logic Programming* paradigm CLP(X) (CLP for short) has been proposed by Jaffar and Lassez [12] in order to integrate a generic computational mechanism based on constraints in the logic programming framework. Such an integration results in a framework which preserves the existence of equivalent operational, model-theoretic and fixpoint semantics. Indeed, as discussed in [17], most of the results which hold for pure logic programs can be lifted to CLP in a quite straightforward way.

The reader is assumed to be familiar with the terminology and the main results on the semantics of (constraint) logic programs. In this subsection we introduce some notations we will use in the following. Lloyd's book and the survey by Apt [16, 1] provide the necessary background material for logic programming theory. For constraint logic programs we refer to the original paper [12] by Jaffar and Lassez and to the recent survey [13] by Jaffar and Maher.

The CLP framework was originally defined using a many-sorted first order language. In this paper, to keep the notation simple, we consider a one sorted language (the extension of our results to the the many sorted case is immediate).

We assume programs defined on a signature with predicates where the set of predicate symbols, denoted by Π, is partitioned into two disjoint sets: Π_c (containing predicate symbols used for constraints) which contains also the equality symbol "=", and Π_b (containing symbols for user definable predicates).

We find convenient to use the notation $\exists_{-\bar{X}}\ \phi$ from [13] to denote the existential closure of the formula ϕ *except* for the variables $\bar{X}$ which remain unquantified. The notations $\bar{t}$ and $\bar{X}$ will denote a tuple of terms and of distinct variables respectively, while $\tilde{B}$ will denote a (finite, possibly empty) conjunction of atoms. The connectives "," and $\Box$ will often be used instead of "$\wedge$"to denote conjunction.

A *primitive constraint* is an atomic formula $p(t_1, \ldots, t_n)$ where $p \in \Pi_c$. A *constraint* is a first order formula built using primitive constraints[2]. A CLP rule is denoted by $H \leftarrow c \Box B_1, \ldots, B_n$. where c is a constraint, H (the head) and $B_1, \ldots, B_n$ (the body) are atomic formulas which use predicate symbols from Π_b only. Analogously a *goal* (or query) is denoted by $c \Box B_1, \ldots, B_n$.

The semantics of CLP programs is based on the notion of *structure* $\mathcal{D}$ which gives an interpretation for the constraints.

Given a structure $\mathcal{D}$ and a constraints c, $\mathcal{D} \models c$ denotes that ϕ is true under the interpretation provided by $\mathcal{D}$. Moreover if ϑ is a valuation (i.e. a mapping of variables on the domain D), and $\mathcal{D} \models c\vartheta$ holds, then ϑ is called a *$\mathcal{D}$-solution* of c ($c\vartheta$ denotes the application of ϑ to the variables in c).

[2] In the original formulation ([12]) a constraint was defined as a conjunction of atoms. We follow here the more general definition given in [13], since our results do not need such a restriction.

We refer to the mentioned papers for the basic notions and results concerning the semantics of CLP. Here we just recall that there exists ([12]) the least $\mathcal{D}$-model of a program P and this is considered the standard semantics of P. As for the operational model of CLP, it is obtained from SLD resolution by simply substituting $\mathcal{D}$-solvability for unifiability. More precisely, a derivation step for a goal $G : c_0 \Box B_1, \ldots, B_n$ in the program P results in a goal of the form $c_1 \Box B_1, \ldots, B_{i-1}, \tilde{B}, B_{i+1}, \ldots, B_n$ if B_i is the atom selected by the selection rule and there exists a clause in P renamed apart (i.e. with no variables in common with G) $H : -c \Box \tilde{B}$ such that $c_1 : (c_0 \wedge (B_i = H) \wedge c)$ is $\mathcal{D}$-satisfiable, that is, $\mathcal{D} \models \exists\ c_1$. Here and in the following, given the atoms A, H, we write $A = H$ as a shorthand for:

- $a_1 = t_1 \wedge \ldots \wedge a_n = t_n$, if, for some predicate symbol p and natural n, $A = p(a_1, \ldots, a_n)$ and $H = p(t_1, \ldots, t_n)$,
- *false*, otherwise.

This notation readily extends to conjunctions of literals.

The notion of derivation (in the program P) of a goal G_i from a goal G is the usual one and in the following is will be denoted by $G \overset{P}{\rightsquigarrow} G_i$. A derivation is *successful* if it is finite and its last element is a goal of the form ($c \Box$). In this case, $\exists_{-Var(G)}\ c$ is called the *answer constraint* [3]. Finally, we need a definition.

Definition 2.1 Let $cl_1 : A_1 \leftarrow c_1 \Box \tilde{B}_1$ and $cl_2 : A_2 \leftarrow c_2 \Box \tilde{B}_2$ be two clauses. We write

$$cl_1 \simeq cl_2$$

iff for any $i, j \in [1, 2]$ and for any $\mathcal{D}$-solution ϑ of c_i there exists an $\mathcal{D}$-solution γ of c_j such that $A_i\vartheta = A_j\gamma$ and $\tilde{B}_i\vartheta$ and $\tilde{B}_j\gamma$ are equal as multisets. □

For the sake of simplicity, we will denote the $\simeq$ equivalence class of a clause c by c itself.

2.1 Modular CLP Programs

Since we are interested in a modular transformation system, first of all we have to define precisely the notions of module and composition. A module, called Ω-open program for consistency with the notation used elsewhere, is a program P together with a set Ω containing the symbols of those predicate which are only partially specified in P.

Definition 2.2 (Ω-open program, [4]) An Ω-open program (Ω-program for short) is a program P together with a set Ω of predicate symbols. □

An Ω-program P is then a module which can be be composed with other modules which may further specify the predicates in Ω. A typical practical example can be

[3] We follow here the more recent terminology used in [13]. In the original paper ([12]) a derivation step was defined by rewriting in parallel all the atoms of the goal. As far as successful derivation are concerned the two formulations are equivalent. Moreover in [12] the answer constraint was considered c (without quantification).

a logic database whose intensional part (i.e. the rules) is completely known while the extensional one (i.e. the facts) is partially specified and could be incrementally added. The composition that we consider here is simply union, modified in order to take into account the "interface" described by the set Ω.

In the following, $Pred(E)$ denotes the set of predicate symbols which appear in the expression E and we say that a predicate p is defined in a program P, if P contains a clause whose head has predicate symbol p.

Definition 2.3 (Ω-union, [4]) Let P and Q be Ω-programs. If $(Pred(P) \cap Pred(Q)) \subseteq \Omega$ then $P \cup_\Omega Q$ is the Ω-program $P \cup Q$. Otherwise $P \cup_\Omega Q$ is not defined. □

Another notion that we have to establish is the kind of "observational" equivalences among programs that we want to maintain. We consider here the answer constraint notion of observable. This is a natural choice since, as previously mentioned, answers constraints are the standard results of CLP computations. Moreover we take into account also the contexts given by $\cup_\Omega$ composition, since these formalize our notion of module. Therefore the following.

Definition 2.4 (Observational equivalences) Let P_1, P_2 be Ω-open programs, we define

- $P_1 \approx_{ac} P_2$
  iff, for any goal G and for any $i, j \in [1,2]$, if there exists a derivation $G \overset{P_i}{\rightsquigarrow} c_i$ □. then there exists a derivation $G \overset{P_j}{\rightsquigarrow} c_j$ □. such that $\mathcal{D} \models \exists_{-Var(G)}\, c_i \leftrightarrow \exists_{-Var(G)}\, c_j$
- $P_1 \approx_\Omega P_2$
  iff for every Ω-program Q such that $P_i \cup_\Omega Q$, $i \in [1,2]$, is defined, we have that $P_1 \cup_\Omega Q \approx_{ac} P_2 \cup_\Omega Q$. □

So $P_1 \approx_{ac} P_2$ if P_1 and P_2 have the same answer constraints (up to logical equivlence in the structure $\mathcal{D}$), while $P_1 \approx_\Omega P_2$ iff their answer constraints are the same n any $\cup_\Omega$-context. By taking Q as the empty program we immediately see that $P_1 \approx_\Omega P_2$ then $P_1 \approx_{ac} P_2$. In the following, we will call $\approx_\Omega$ also *compositional* quivalence.

Unfold/fold transformations for CLP

this section we define an unfold/fold system for CLP programs. The system is inired by the one proposed by Tamaki and Sato [18] for pure logic programs. However, e extension to the context of constraint logic programs requires a generalization of e conditions for the folding operation in [18] which is not straightforward. We beve that the result is worth the effort, as now those applicability conditions can be pressed in a more concise way.

First, it is worth noticing that all the observable properties we refer to are invariant der $\simeq$; this implies that we can always replace any clause cl in a program P with

a clause cl', provided that $cl' \simeq cl$. This operation is often useful to *clean up* the constraints, and, in general, to present the clause in a more readable form.

We start from some requirements on the original (i.e. initial) program.

Definition 3.1 (Initial program) We call a CLP program P_0 an *initial program* if the following three conditions are satisfied:

(I1) P_0 is divided into two disjoint sets $P_0 = P_{new} \cup P_{old}$;

(I2) All the predicates which are defined in P_{new} occur neither in P_{old} nor in the bodies of the clauses in P_{new}. □

The following Example is kept simple for the sake of clarity. Programs are written in an unspecified $CLP(X)$ language.

Example 3.2 Let P_0 be the following Ω-program

```
member(Element, List) ←
    Element is an element of the list List.

member(El, List) ←List = [ El | _ ] □ .
member(El, List) ←List = [ _ | List2 ] □ member(El, List2).

meet(Path1, Path2) ←
    Path1 and Path2 intersect in a place which satisfy good.

meet(Path1, Path2) ← true □
    member(El, Path1), member(El, Path2), good(El).
```

where $\Omega = \{\texttt{good}\}$. So good is only partially specified in P_0 and its definition, which could involve some arithmetic constraints, can be added later. Moreover we assume that meet is the only *new* predicate. So P_{new} consists of the clause defining meet. □

The unfolding operation is basic to all the transformation systems, and it consists essentially in applying a resolution step to the unfolded atom in all possible ways. Here its definition is given modulo reordering of the bodies of the clauses, moreover, it is assumed that all the clauses are renamed apart.

Definition 3.3 (Unfolding) Let $cl : A \leftarrow c \,□\, H, \tilde{K}.$ be a clause of a program P, and $\{H_1 \leftarrow c_1 \,□\, \tilde{B}_1, \ldots, H_n \leftarrow c_n \,□\, \tilde{B}_n\}$ be the set of clauses of P for which $c \wedge c_i \wedge (H = H_i)$ is $\mathcal{D}$-satisfiable. For $i \in [1,n]$, let cl'_i be the clause

$$A \leftarrow c \wedge c_i \wedge (H = H_i) \,□\, \tilde{B}_i, \tilde{K}$$

Then *unfolding H in cl in P* consists of substituting cl by $\{cl'_1, \ldots, cl'_n\}$ in P. □

In this situation we say that $cl : A \leftarrow c \,□\, H, \tilde{K}.$ is the *unfolded* clause, while $\{H_1 \leftarrow c_1 \,□\, \tilde{B}_1, \ldots, H_n \leftarrow c_n \,□\, \tilde{B}_n\}$ are the *unfolding* ones.

Example 3.2 (part 2) By unfolding member(El, Path1) in the body of the clause defining meet, we obtain P_1, which, after cleaning up the constraints is:

```
meet(PathA, PathB) ←PathA = [ Place | _ ] □
   member(Place, PathB), good(Place).
meet(PathA, PathB) ←PathA = [ _ | PathA' ] □
   member(Place, PathA'), member(Place, PathB), good(Place).
```

together with the clauses defining predicate `member`. □

In the above example we have already renamed the variables in order to "prepare" the clauses for the next operation: the folding. This operation is often used in order to introduce recursion in the new definitions. Unlike unfolding, the folding operation requires some conditions which ensure its correctness also when modularity is not taken into account. Following [18], we define the transformation sequence and the folding operation in terms of each other.

Definition 3.4 (Transformation sequence) A *transformation sequence* is a sequence of programs $P_0, \ldots, P_n$, $n \geq 0$, such that P_0 is an initial program, and each P_{i+1}, $0 \leq i < n$, is obtained from P_i by unfolding or folding a clause of P_i. □

Here, we also assume the folding and the folded clause to be renamed apart, and, as a notational convenience, that the body of the folded clause had been reordered so that the atoms that are going to be folded are found on its left hand side.

Definition 3.5 (Folding) Let $P_0, \ldots, P_i$, $i \geq 0$, be a transformation sequence. Let also

$cl : A \leftarrow c_A \,\square\, \tilde{K}, \tilde{J}.$ be a clause in P_i,
$d : D \leftarrow c_D \,\square\, \tilde{H}.$ be a clause in P_{new}.

If $\tilde{K}$ is an instance of $\tilde{H}$ and $e = \{x_1 = t_1, \ldots, x_m = t_m\}$ is a constraint where $\{x_1, \ldots, x_m\} \subseteq Var(D)$ and $Var(t_1, \ldots, t_m) \subseteq Var(cl)$, then *folding D in cl via e* consists of substituting cl with

$$cl' : \; A \leftarrow c_A \wedge e \,\square\, D, \tilde{J}.$$

provided that the following three conditions hold:

(F1) *"If we unfold D in cl' using d as unfolding clause, then we obtain cl back":*

$$\mathcal{D} \models \; \exists_{-Var(A,\tilde{J},\tilde{H})} \; c_A \wedge e \wedge c_D \;\leftrightarrow\; \exists_{-Var(A,\tilde{J},\tilde{H})} \; c_A \wedge (\tilde{H} = \tilde{K})$$

(F2) *"d is the only clause of P_{new} that can be used to unfold D in cl'"*: there is no clause $b : B \leftarrow c_B \,\square\, \tilde{L}$ in P_{new} such that $b \neq d$ and $c_A \wedge e \wedge (D = B) \wedge c_B$ is $\mathcal{D}$-satisfiable"

(F3) *"No self-folding is allowed"*:

(a) either the predicate in A is an old predicate;

(b) or cl is the result of at least one unfolding in the sequence $P_0, \ldots, P_i$; □

Here, the constraint e acts as a bridge between the variables of d and cl.

Conditions **F1** and **F2** ensure that the folding operation behaves, to some extent, as the inverse of the unfolding one; the underlying idea is that if we unfolded the atom D in cl' using only clauses from P_{new} as unfolding clauses, then we would obtain cl back. In this context condition **F2** ensures that in P_{new} there exists no clause other than d that can be used as *unfolding clause*. Consequently, the result of such an operation would be the clause:

$$A \leftarrow c_A \wedge e \wedge (D = D') \wedge c'_D \,\square\, \tilde{H}', \tilde{J}$$

Where $d' : D' \leftarrow c'_D \,\square\, \tilde{H}'$ is an appropriate renaming of d. Now, by the standardization apart, the variables of $c_D, \tilde{H}, c'_D, \tilde{H}'$ which do not occur in D, D', do not occur anywhere else in this clause, so, by explicitating $(D = D')$, we can identify c'_D with c_D and $\tilde{H}'$ with $\tilde{H}$. The previous clause becomes then: $A \leftarrow c_A \wedge e \wedge c_D \,\square\, \tilde{H}, \tilde{J}$. Now, by **F1**, this can be rewritten as: $A \leftarrow c_A \wedge (\tilde{H} = \tilde{K}) \,\square\, \tilde{H}, \tilde{J}$, and, because of the constrain $(\tilde{H} = \tilde{K})$, this is equivalent (modulo $\simeq$) to $A \leftarrow c_A \wedge (\tilde{H} = \tilde{K}) \,\square\, \tilde{K}, \tilde{J}$. Now recall that the folding and the folded clause are assumed to be standardized apart, therefore $\tilde{H}$ has no variables in common with the rest of this clause (that is, with A, c_A, $\tilde{K}$ and $\tilde{J}$). Since we also assume that $\tilde{K}$ is an instance of $\tilde{H}$, it follows that the constraint $\tilde{H} = \tilde{K}$ can be eliminated, and we have the original clause cl back.

Of course, during a transformation sequence, such an "undo by unfolding" is often not possible; this is due to the fact that the folding clause is usually not found in the "current" program.

Finally, we should mention that the purpose of **F3** is to avoid the introduction of loops which can occur if a clause is folded by itself. This condition is the same one that is found in Tamaki-Sato's definition of folding for logic programs.

Example 3.2 (part 3) We can now fold `member(Place, PathA'), member(Place, PathB), good(Place)` in the second clause defining `meet` in program P_1. In this case, the constraint e has to be

$$\texttt{Path1} = \texttt{PathA'} \;\wedge\; \texttt{Path2} = \texttt{PathB}.$$

Checking **F1** is a trivial task: its left hand side is:

$\exists$`Path1, Path2, El  PathA=[_|PathA'] ∧ Path1=PathA' ∧ Path2=PathB'`

while its right hand side is

$\exists$`Path1, Path2, El  PathA=[_|PathA'] ∧ El=Place`

And it is immediate to see how one side reduces to the other one. The resulting program, P_3 after cleaning up the constraints, is:

```
meet(Path1, Path2) ←Path1 = [ El | _ ] □
   member(El, Path2), good(El).
meet(Path1, Path2) ←Path1 = [ _ | Path1' ] □meet(Path1', Path2').
```

together with the clauses defining predicate `member`

Notice that, because of this last operation, the definition of `meet` is now recursive. □

3.1 A Modular Transformation system

We are interested here in a *modular* transformation system. Previous conditions on the folding operation are sufficient to ensure the correctness of the system wrt the answer constraint and the least $\mathcal{D}$-model semantics (as we prove in the next section). However, as shown by the following example, if we want to obtain compositionally equivalent programs, we need to specify some further applicability conditions. Here and in the sequel, we call an atom Ω-*atom* if its predicate symbol is in Ω.

Example 3.6
(i) First, we cannot allow the unfolding of Ω-atoms. In fact let P_0 be the following program

```
p ←q.
q ←r.
```

where $\Omega = \{\mathtt{q}\}$. If we unfold q in the first clause, we obtain the program P_1:

```
p ←r.
q ←r.
```

Now the two programs are not compositionally equivalent. In fact, if we add the program $Q = \{\mathtt{q}\}$ then the query $\leftarrow \mathtt{p}$ succeeds in $P_0 \cup Q$ and fails in $P_1 \cup Q$ and hence $P_0 \not\approx_\Omega P_1$.

(ii) Secondly, we cannot let a *new* predicate to be also an Ω-predicate. Let P_0 be the following program

```
p ←q.
r ←q.
```

Where $\Omega = \{\mathtt{p}\}$ and $P_{new} = \{\mathtt{p} \leftarrow \mathtt{q}\}$. Since r is an old atom, we can fold q in the second clause of P_0; the resulting program P_1 is

```
p ←q.
r ←p.
```

Again $P_0 \not\approx_\Omega P_1$. In fact, if we add the program $Q = \{\mathtt{p}\}$ we have that the query $\leftarrow \mathtt{r}$ succeeds in $P_1 \cup Q$, but fails in $P_0 \cup Q$. □

Therefore the following.

Definition 3.7 (Modular transformation sequence) Let P_0 be an Ω-program and $P_0, \ldots, P_i$ be a transformation sequence. If

(O1) no Ω-atom is unfolded in $P_0, \ldots, P_i$ and

(O2) no *new* predicate belongs to Ω,

then we call $P_0, \ldots, P_i$ a *modular* transformation sequence. □

So, Ω-atoms can only be folded. Notice that this is what happens in the transformation sequence of Example 3.2, which is indeed a modular transformation sequence.

For $\Omega = \emptyset$, **O1** and **O2** are trivially satisfied and we have a standard (non modular) unfold/fold system. In fact when $\Omega = \emptyset$, composition is allowed only among programs which do not share predicate symbols, and, from a semantic point of view, this is equivalent to consider no composition at all.

4 Correctness of the unfold/fold system

The aim of this section is to show that a modular transformation sequence preserves the compositional equivalence. To this end, first we introduce a semantics ([10]) for CLP which models answer constraints and which is compositional wrt union of programs, then we show that a modular transformation sequence preserves this semantics. The desired result follows from the correctness of the semantics wrt the equivalence $\approx_\Omega$.

4.1 A compositional semantics for CLP

The semantics we introduce now is basically a straightforward lifting to the CLP case of that one defined in [4] for logic programs, where compositionality wrt union of programs is obtained by choosing a semantic domain based on clauses. This semantics can also be seen as a generalization of the answer constraint semantics of [9].

Using $\simeq$ to abstract from purely syntactic details, we define denotations as follows.

Definition 4.1 (Ω-Denotations) Let Ω be a set of predicate symbols and let $\mathcal{C}$ be the set of the $\simeq$-equivalence classes of the clauses in the given language. The *interpretation base* $\mathcal{C}_\Omega$ is the set $\{A \leftarrow c \,\square\, B_1, \ldots, B_n \in \mathcal{C} \mid Pred(B_1, \ldots, B_n) \subseteq \Omega\}$. An Ω-*denotation* is any subset of $\mathcal{C}_\Omega$. □

The following is an operational definition of the Ω-semantics for CLP. An equivalent fixpoint definition can be obtained by using a suitable operator ([10]).

Definition 4.2 ($\mathcal{O}_\Omega(P)$ semantics, [10]) Let P be an Ω-program. Then we define

$$\mathcal{O}_\Omega(P) = \{p(\bar{X}) \leftarrow c \,\square\, B_1, \ldots, B_n \in \mathcal{C}_\Omega \mid \text{there exists a derivation } \mathit{true} \,\square\, p(\bar{X}) \overset{P}{\rightsquigarrow} c \,\square\, B_1, \ldots, B_n.\ \}$$ □

The compositionality of previous semantics wrt $\cup_\Omega$ is proved in [10]. From such a result it follows the correctness of the Ω-semantics wrt the equivalence $\approx_\Omega$, as shown by the following Corollary of [10].

Corollary 4.3 Let P, Q be Ω-open programs. If $\mathcal{O}_\Omega(P) = \mathcal{O}_\Omega(Q)$ then $P \approx_\Omega Q$. □

In other words, $\mathcal{O}_\Omega(P)$ contains all the information necessary to model the behaviour of P, in terms of answer constraints, compositionally wrt union of programs. In the particular case $\Omega = \emptyset$, i.e. when all the predicates are completely defined, $\mathcal{O}_\Omega$

coincides with the answer constraint semantics defined in [9]. In this case we have that the semantics is not only correct, but also fully abstract wrt $\approx_{ac}$. In fact in [9] it is proven that $\mathcal{O}_\emptyset(P) = \mathcal{O}_\emptyset(Q)$ iff $P \approx_{ac} Q$.

4.2 Correctness result

We can now state the the main result of the paper: the unfold/fold modular transformation preserves the $\mathcal{O}_\Omega$ semantics.

Theorem 4.4 (Correctness) If P_0 is an Ω-program and $P_0, \ldots, P_n$ is a modular transformation sequence then for any $i, j \in [0, n]$,

- $\mathcal{O}_\Omega(P_i) = \mathcal{O}_\Omega(P_j)$ □

Notice also that, as shown in Example 3.6, conditions **O1** and **O2** are necessary to guarantee the correctness of a transformation sequence wrt the Ω-semantics.

From the correctness of the Ω-semantics wrt $\approx_\Omega$ we can derive the compositional equivalence of all the programs given by a modular transformation sequence.

Corollary 4.5 Let P_0, Q be Ω-open programs and $P_0, \ldots, P_n$ be a modular transformation sequence. If $P_0 \cup_\Omega Q$ is defined, then for any $i, j \in [0, n]$,

(i) $P_i \cup_\Omega Q$ is defined,

(ii) $P_i \approx_\Omega P_j$ □

In particular, we have that, for any program Q, the answer constraint semantics and the least $\mathcal{D}$-models [12] of $P_i \cup_\Omega Q$ and $P_j \cup_\Omega Q$ coincide.

Since both the least $\mathcal{D}$-model and the answer constraint semantics can be seen as abstractions of the Ω-semantics with $\Omega = \emptyset$, Theorem 4.4 implies that these semantics are preserved by any transformational sequence (conditions **O1, O2** are then trivially satisfied). Hence the following.

Corollary 4.6 Let $P_0, \ldots, P_n$ be a (non-modular) transformation sequence. Then, for any $i, j \in [0, n]$,

(i) $\mathcal{O}_\emptyset(P_i) = \mathcal{O}_\emptyset(P_j)$ (the answer constraint semantics of P_i and P_j coincide),

(ii) The least $\mathcal{D}$-models of P_i and P_j are equal. □

5 From CLP to LP

Pure logic programming (LP for short) can be seen as a particular instance of the constraint logic programming scheme. This is obtained by taking as structure the Herbrand universe where "=" is the only predicate symbol for constraints, and it is interpreted as identity.

In the following first we introduce the unfold/fold transformation system for logic programs proposed by Tamaki and Sato [18], then, by using a "canonical mapping" from logic to (pure) CLP programs[4], we show that it can be embedded in the one we presented in the previous section. This will allow us to show that, under the hypothesis expressed by conditions **O1** and **O2**, the unfold/fold system preserves the Ω-semantics for pure logic programs.

5.1 Unfold/fold transformations for LP

First, let us consider the unfold operation. Again, we assume that the clause are standardized apart.

Definition 5.1 (Unfolding, in LP) Let $cl : A \leftarrow H, \tilde{K}.$ be a clause of a logic program P, and let $\{H_1 \leftarrow \tilde{B}_1, \ldots, H_n \leftarrow \tilde{B}_n\}$ be the set of clauses of P whose heads unify with H, by mgu's $\{\theta_1, \ldots, \theta_n\}$. For $i \in [1, n]$ let cl'_i be the clause

$$(A \leftarrow \tilde{B}_i, \tilde{K})\theta_i$$

Then *unfolding H in cl in P* consists of substituting cl by $\{cl'_1, \ldots, cl'_n\}$ in P. □

We can now introduce the folding operation. In this context, we adopt the same definitions of *initial program* and of *transformation sequence* given for CLP. Again, we assume the folding and the folded clause to be renamed apart and (for notational convenience) that the body of the folded clause has been reordered (as in Definition 3.5).

Definition 5.2 (Folding, in LP, [18]) Let $P_0, \ldots, P_i$, $i \geq 0$, be a transformation sequence. Let also

$cl : A \leftarrow \tilde{K}, \tilde{J}.$ be a clause in P_i,

$d : D \leftarrow \tilde{H}.$ be a clause in P_{new}.

Let also $V = Var(\tilde{H}) \backslash Var(D)$ be the set of local variables of d. If there exists a substitution τ such that $Dom(\tau) = Var(d)$ and the following conditions hold:

(LP1) $\tilde{H}\tau = \tilde{K}$;

(LP2) For any $x, y \in V$

- $x\tau$ is a variable;
- $x\tau$ does not appear in A, $\tilde{J}$, $D\tau$;
- if $x \neq y$ then $x\tau \neq y\tau$;

(LP3) d is the only clause in P_{new} whose head is unifiable with $D\tau$;

(LP4) one of the following two conditions holds

1. the predicate in A is an old predicate;
2. cl is the result of at least one unfolding in the sequence $P_0, \ldots, P_i$;

then *folding D in cl (via τ)* consists of substituting cl with $cl' : A \leftarrow D\tau, \tilde{J}$. □

[4] *Pure* CLP programs are CLP programs in which the atoms in the clauses, apart from constraints, are always of the form $p(\tilde{X})$, where $\tilde{X}$ is a tuple of distinct variables.

5.2 LP vs CLP

While for the unfold operation it is clear that Definition 5.1 is the counterpart of Definition 3.3, for the fold operation the similarities are less obvious. In order to compare Definitions 3.5 and 5.2 we first need to define formally the "canonical" mapping μ from logic program to pure constraint logic programs.

Definition 5.3 Let $cl: p_0(\bar{t}_0) \leftarrow p_1(\bar{t}_1), \ldots, p_n(\bar{t}_n)$ be a clause in LP. Then $\mu(cl)$ is the CLP clause $p_0(\bar{x}_0) \leftarrow \bar{x}_0 = \bar{t}_0 \wedge \bar{x}_1 = \bar{t}_1 \wedge \ldots \wedge \bar{x}_n = \bar{t}_n \square p_1(\bar{x}_1), \ldots, p_n(\bar{x}_n)$, where $\bar{x}_0, \ldots, \bar{x}_n$ are tuple of new and distinct variables. □

By using the correspondence μ, we can compare Definition 3.5 with Definition 5.2. The following theorem shows that if cl' is the result of folding cl with d in the logic program P_i then we can obtain $\mu(cl')$ by folding $\mu(cl)$ with $\mu(d)$ in $\mu(P_i)$. In other words, if d and cl are such that the conditions **LP1, LP2, LP3** and **LP4** are satisfied, and cl' is the result of the folding operation, then by translating the whole program into CLP we end up in a situation in which conditions **F1, F2** and **F3** are satisfied. Moreover, by appropriately choosing the constraint e, we obtain $\mu(cl')$ by folding $\mu(cl)$ with $\mu(d)$.

Theorem 5.4 If $P_0, \ldots, P_n$ is a transformation sequence of logic programs, then $\mu(P_0), \ldots, \mu(P_n)$ is a transformation sequence in CLP. □

Clearly, even though a logic program is itself a CLP program, a transformation sequence of logic programs cannot be regarded as a transformation sequence of CLP programs. This is due to the fact that the unfold and fold operations for LP use substitutions whereas those for CLP use constraints. This is the reason why we use the mapping μ for proving the correspondence between the two kind of sequences.

5.3 Semantic Consequences for Logic Programs

Theorem 5.4 allows us to prove a counterpart of Theorem 4.4 for the LP case.

First we need the definition of the original Ω-semantics for pure logic programs [4]. It can be obtained from Definition 4.2 by simply replacing CLP derivations by LD derivations: $\mathcal{O}_\Omega(P)$ is now the set

$$\{(p(\bar{X})\vartheta \leftarrow B_1, \ldots, B_n)/\simeq \mid \text{there exists an SLD-derivation } p(\bar{X}) \overset{\vartheta}{\leadsto}_P B_1, \ldots, B_n. \text{ and } Pred(B_1, \ldots, B_n) \subseteq \Omega \}.$$

where $\simeq$ denotes variance and ϑ is the compositions of the mgu's used in the derivation $p(\bar{X}) \overset{\vartheta}{\leadsto}_P B_1, \ldots, B_n$. Unsurprisingly, this semantics enjoys the same correctness properties stated for the CLP case stated in and Corollary 4.3 (see [4]).

The semantic equivalence between the original pure logic program P and its CLP version $\mu(P)$ is due essentially to the isomorphism existing between the (lattice structures on) idempotent substitutions and equations [15]. Because of this isomorphism we can then use equivalently idempotent mgu's in SLD derivations or equations in

CLP derivations. In the first case, the result of the computation is the composition of the mgu's used, restricted to the variables in the goal. In the second the (equivalent) result is given by the answer constraint. This is formalized in [19] and (by extending the definition of μ to the $\simeq$-equivalence classes) brings to the following conclusion:

$$\mu(\mathcal{O}_\Omega(P)) = \mathcal{O}_\Omega(\mu(P))$$

We can now prove the correctness of the transformation sequence for logic programs wrt the Ω-semantics.

Corollary 5.5 Let $P_0, \ldots, P_n$ be a transformation sequence of pure logic programs and let Ω be a set of predicate symbols. If conditions **O1** and **O2** are satisfied then

- $\mathcal{O}_\Omega(P_0) = \mathcal{O}_\Omega(P_n)$. □

Analogously to the CLP case, the results summarized in Corollary 4.5 apply also to pure logic programs. Their proof follows directly from the previous Corollary. In particular we have that for any P_i, P_j in the transformation sequence and for any program Q, if $P_0 \cup_\Omega Q$ is defined, then a generic goal G has the same computed answer substitutions in $P_i \cup_\Omega Q$ and in $P_j \cup_\Omega Q$, and the least Herbrand models of $P_i \cup_\Omega Q$ and $P_j \cup_\Omega Q$ coincide.

6 Conclusions

A definition of unfold/fold transformation system for CLP based on [18] has also been given by Bensaou and Guessarian in [2], yet there are some substantial differences between [2] and our proposal:

First, the semantics they refer to is an extension to the CLP case of the C-semantics ([6, 8]). Such a semantics characterizes the logical consequences of the program on $\mathcal{D}$-models, but does not allow to model answer constraints. For example, the C-semantics identifies the programs $\{p(X) : -X = a \ \square., \ p(X) : -X = Y \ \square.\}$ and $\{p(X) : -X = Y \ \square.\}$ which have different answer constraint for the goal $p(X)$, and consequently are not identified by the answer constraint semantics in [9]. We believe that the answer constraints semantics provides a better reference semantics for unfold/fold systems, since answer constraints are the most natural properties that one would like to preserve while transforming programs. Moreover, the C-semantics can be obtained as an abstraction (upward closure) of the answer constraint semantics. As a consequence our correctness result are more general.

A second relevant difference is due to the fact that modularity is not take into account in [2].

Finally, a third point where our approaches depart from each other is that in [2] the folding conditions are obtained by a straightforward extension to the CLP case of **LP1**...**LP4**. In this respect, CLP programs are there treated as "extended" logic programs. Of course, [2] allows a more uniform treatment of LP and CLP, and, for those who know [18], it provides a more familiar definition of folding. On the other

hand, our choice of departing from the notation of [18] leads to a definition of folding which is more compact and does not need the use of substitutions.

To conclude, the contributions of this paper can be summarized as follows.

We have defined an unfold/fold transformation system for CLP based on the framework of Tamaki and Sato for logic programs [18]. The use of CLP allows us to express the applicability conditions for the folding operation in a more concise and elegant way.

A definition of a modular transformation sequence is given by adding to the usual definition some further simple applicability conditions. These conditions are shown to be necessary and sufficient to guarantee the correctness of the system wrt the Ω-semantics. Since this semantics is compositional wrt the union of programs, this provides a natural theoretical framework for the modular transformation of CLP programs. To the best of our knowledge, this is the first study of the issue of modularity in the context of transformation systems. Moreover, since the Ω-semantics is a generalization of the answer constraint semantics [9] and of the least model semantics [12], the correctness of the (normal, non-modular) system wrt those semantics follows as a corollary.

Finally, the relations between transformation sequences for CLP and LP are discussed. By "canonically" mapping into CLP the transformation system for LP proposed by Tamaki and Sato [18], we prove that, under the "modular" conditions **O1** and **O2**, the Ω-semantics is preserved by the transformation system also in the LP case.

Acknowledgements

The authors want to thank Annalisa Bossi and the referees for their useful suggestions.

References

[1] K. R. Apt. Introduction to Logic Programming. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B: Formal Models and Semantics, pages 495–574. Elsevier, Amsterdam and The MIT Press, Cambridge, 1990.

[2] N. Bensaou and I. Guessarian. Transforming Constraint Logic Programs. In F. Turini, editor, *Proc. Fourth Workshop on Logic Program Synthesis and Transformation*, 1994.

[3] A. Bossi and N. Cocco. Basic Transformation Operations which preserve Computed Answer Substitutions of Logic Programs. *Journal of Logic Programming*, 16:47–87, 1993.

[4] A. Bossi, M. Gabbrielli, G. Levi, and M. C. Meo. A Compositional Semantics for Logic Programs. *Theoretical Computer Science*, 122(1-2):3–47, 1994.

[5] R.M. Burstall and J. Darlington. A transformation system for developing recursive programs. *Journal of the ACM*, 24(1):44–67, January 1977.

[6] K. L. Clark. Predicate logic as a computational formalism. Res. Report DOC 79/59, Imperial College, Dept. of Computing, London, 1979.

[7] S. Etalle and M. Gabbrielli. Modular Transformations of CLP Programs. Technical report, CWI, Amsterdam, 1994. to appear.

[8] M. Falaschi, G. Levi, M. Martelli, and C. Palamidessi. A Model-Theoretic Reconstruction of the Operational Semantics of Logic Programs. *Information and Computation*, 102(1):86–113, 1993.

[9] M. Gabbrielli and G. Levi. Modeling Answer Constraints in Constraint Logic Programs. In K. Furukawa, editor, *Proc. Eighth Int'l Conf. on Logic Programming*, pages 238– 252. The MIT Press, Cambridge, Mass., 1991.

[10] M. Gabbrielli and G.M. Dore G. Levi. Observable Semantics for Constraint Logic Programs. *Journal of Logic and Computation*, 1994.

[11] P.A. Gardner and J.C. Shepherdson. Unfold/fold transformations of logic programs. In J-L Lassez and editor G. Plotkin, editors, *Computational Logic: Essays in Honor of Alan Robinson*. 1991.

[12] J. Jaffar and J.-L. Lassez. Constraint Logic Programming. In *Proc. Fourteenth Annual ACM Symp. on Principles of Programming Languages*, pages 111–119. ACM, 1987.

[13] J. Jaffar and M. Maher. Constraint Logic Programming: a Survey. *Journal of Logic Programming*, 1994. To appear.

[14] T. Kawamura and T. Kanamori. Preservation of Stronger Equivalence in Unfold/Fold Logic Programming Transformation. In *Proc. Int'l Conf. on Fifth Generation Computer Systems*, pages 413–422. Institute for New Generation Computer Technology, Tokyo, 1988.

[15] J.-L. Lassez, M. J. Maher, and K. Marriott. Unification Revisited. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 587–625. Morgan Kaufmann, Los Altos, Ca., 1988.

[16] J. W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, Berlin, 1987. Second edition.

[17] M.J. Maher. A transformation system for deductive databases with perfect model semantics. *Theoretical Computer Science*, 110:377–403, 1993.

[18] H. Tamaki and T. Sato. Unfold/Fold Transformations of Logic Programs. In Sten-Åke Tärnlund, editor, *Proc. Second Int'l Conf. on Logic Programming*, pages 127–139, 1984.

[19] D. A. Wolfram, M.J. Maher, and J-L. Lassez. A Unified Treatment of Resolution Strategies for Logic Programs. In Sten-Åke Tärnlund, editor, *Proc. Second Int'l Conf. on Logic Programming*, pages 263–276, 1984.

Towards A Functional Process Calculus*

Karsten Bohlmann
RWTH Aachen†

Rita Loogen
Philipps-Universität Marburg‡

Yolanda Ortega-Mallén
Universidad Complutense de Madrid§

Abstract

We extend a (lazy) Hindley-Milner-typed λ-calculus with process abstractions and applications in order to get a calculus for the functional specification of dynamically evolving process networks. The behaviour of processes is specified by process abstractions in a purely functional way as mapping input to output streams. Process applications lead to the dynamic creation of processes as "second class citizens". The redirection of streams by a special operator enables the definition of arbitrary topologies. Time-dependent behaviour is supported by predefined non-functional processes and by "semaphores", which are used within processes for the synchronization of outputs.

Our calculus preserves the functional paradigm and presents a natural model for a parallel declarative programming language. This is also reflected by its two-level operational semantics which embodies the reduction rules of the lazy λ-calculus in its kernel.

1 Introduction

During the last years, several attempts have been made to achieve an integration of the λ-calculus and concurrent process calculi. Milner [13] showed how to simulate the λ-calculus by a process calculus. Other approaches [1, 14] extend the λ-calculus so as to make processes 1^{st} class objects and communication and synchronization explicit in the style of process calculi like CCS and CSP.
A more practical work with the aim of integrating functional and concurrent programming is Facile [4]. It also employs synchronized explicit communication and CCS-like

*This research has been supported by the DAAD (Deutscher Akademischer Austauschdienst).
†Lehrstuhl für Informatik II, Ahornstraße 55, D-52056 Aachen, Germany,
email: karsten@zeus.informatik.rwth-aachen.de
‡Fachgebiet Informatik, Hans Meerwein Straße, Lahnberge, D-35032 Marburg, Germany,
email: loogen@informatik.uni-marburg.de
§Sec. Dept. de Informática y Automática, Facultad de C.C. Matemáticas, E-28040, Spain,
email: yolanda@dia.ucm.es

constructs (concurrent composition, nondeterministic choice) to spawn processes. The resulting programs exhibit a rather imperative style.

The main goal of our approach has been to develop a *functional process calculus* which is a conservative extension of the λ-calculus and may serve as a natural model for a parallel declarative programming language. This requires a clean semantic separation between processes and functional objects, which means that we do have processes, but as "2^{nd} class citizens". Unlike functions or process *abstractions*, processes are just operational entities without a functional denotation.
Communication is implicit — necessarily, because *commands* like "send" and "receive" are incompatible with the functional paradigm. Our process calculus is suitable for the specification of *deterministic process systems* (definition of entirely functional networks), as well as for modelling *concurrent systems*, which implies time-dependence and nondeterminism.
The *Kahn/MacQueen model* of functional process networks [8] has been our starting point. It adopts already a "functional notation" for specifying process behaviour, but is only first order, and thus far weaker than the full lazy λ-calculus.
In our approach the full expressive power of the λ-calculus becomes available not only for the description of internal process evolution, but also for the elegant definition of arbitrary process topologies.

In analogy to built-in constants, we use *predefined processes*, including `MERGE`, to introduce nondeterminism (which will thus be kept out of the purely functional processes). The necessity of `MERGE` for the Kahn/MacQueen model has been known for a long time as well. Its use obviously destroys the possibility to give the whole network a functional denotation, but operationally it presents no problem, and instead of struggling with the anomalies which result from a declarative treatment, we suggest an *observational* semantics as appropriate (as, for example, in CCS [12]). The basis for this approach is our operational semantics, which performs local lazy evaluation in a similar way as Launchbury [10] and uses *actions*, like "send value on stream" or "create process", to model inter-process behaviour.

2 Process Abstractions and Applications

An expression of our functional process calculus CFP (Concurrent Functional Processes) is either an expression of the typed λ-calculus with some straightforward extensions or a process abstraction (see Fig. 1). As indicated in the introduction, processes cannot be objects of the "functional world", but *process abstractions*, which specify process behaviour in a purely functional way, can.
A process abstraction

process **var**, ..., **var** *input* **var**, ..., **var** *output* **var**, ..., **var**
body **equation** ... **equation** *end*

contains declarations for the *input* and *output streams* of the process and one defining equation for every output. Moreover, auxiliary definitions are allowed, because different outputs often depend on a common sub-expression. This leads to a process *body* containing definitions at least for all the output streams.

```
exp ::=  const  |  (const exp ··· exp)          (constant | constant application)
       | var  |  pvar  |  svar                  (functional | process | semaphore variable)
       | if exp then exp else exp               (conditional)
       | λ var . exp                            (λ-abstraction)
       | (exp exp)                              (λ-application)
       | letrec equation ··· equation in exp    (local definition)
       | process var, ..., var  input var, ..., var  output var, ..., var
         body  equation ··· equation  end       (process abstraction)
       | bypass exp                             (stream redirection)
       | ⟨exp⟩ exp                              (eager process creation)
       | P exp, exp  |  V exp, exp              (semaphore operation)
equation ::=  var = exp                         (functional definition)
       | [pvar :](var, ..., var) = exp (exp, ..., exp)   (process application)
       | svar = semaphore                       (semaphore definition)
```

Figure 1: CFP Expressions

Furthermore, a process can have *parameters*, declared after the keyword *process*. This is basically λ-abstraction, but as we require process abstractions to be *combinators*, the expressions in the body may depend only on the parameters, the input streams, auxiliary definitions and, as recursively defined processes are allowed, the name p of the abstraction when it is defined in an equation with left hand side p. This ensures that in CFP a process is an independent unit which communicates exclusively via its stream connections.

The application of process abstractions to actual process parameters is expressed syntactically in the same way as for λ-abstractions (curried), but with quite a different interpretation, which is necessary to avoid *hidden communications* between processes, by means of shared variables. Let e_1 be a process abstraction with first parameter p_1, then the application $(e_1\ e_2)$ causes p_1 to be deleted from e_1's parameter list and a definition $p_1 = e_2$ to be added to the body, *plus definitions for all variables occurring free in* e_2 (if necessary, rename variables consistently to avoid conflicts with other local identifiers of e_1). The result is a new process abstraction containing the complete "graph" for e_2 in its body, bound to p_1. To be precise, this is only possible if e_2 does not depend on an input stream, a condition which can be checked statically. We count this application as a special case of λ-application, in contrast to *process application*, by which we mean the construct that causes creation of a new process from a process abstraction through binding its streams.

Streams, which are modelled by lists, will be *hyperstrict* [16], i.e. only finite, fully evaluated objects of basic and algebraic data types can be sent. This includes numbers, trees, lists (not streams), but not abstractions.

Process applications are only allowed as special right hand sides of equations (see Fig. 1). Exactly this restriction prevents processes from being first class citizens. The expressions intended for the input streams of the created process are written in a tuple after the (fully "λ-applied") process abstraction. The left hand side of such an equation is a tuple of variables which receive the output streams of the created process. The purpose of the optional label is explained in Section 5.

Example (Parallel operation). The following recursive process abstraction applies a given operation to a sequence of numbers in a binary tree of processes; e.g. when applied to the parameter λx. λy. (x*y) and the stream $[1, n]$, it computes $n!$; and when applied to the parameter λx. λy. (x+y) and the stream $[1, n]$, it computes $\sum_{i=1}^{n} i$.

```
par_op =  process op   input limits   output res
          body   res    = if low==high then [low] else [op (hd p1) (hd p2)]
                 low    = (hd limits)
                 high   = (hd (tl limits))
                 mid    = (low + high) / 2
                 (p1)   = (par_op op) ([low, mid])
                 (p2)   = (par_op op) ([mid+1, high])
          end
```

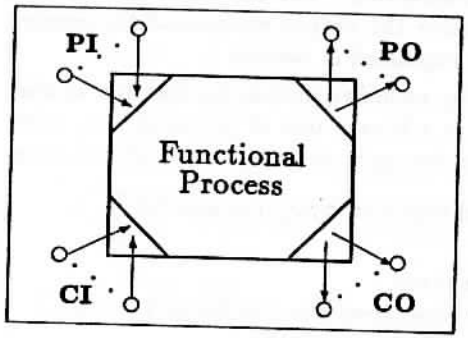
In order to calculate the result, a binary tree of processes is generated. However, due to lazy evaluation, this definition *fails* to achieve the desired parallelism. This will be explained in detail in Section 5, where we will introduce a vital mechanism to make it work. □

3 Communication and Termination

The connection transmitting a stream of values is called a *channel*, and the interface between a channel and a process an (input or output) *port*, or inport/outport. Thus, a port associates a local stream variable with a channel. Processes can communicate only via channels, which behave like unbounded buffers. A channel always connects exactly one *producer* to one *consumer* process, thus data cannot be implicitly broadcast. Fig. 2 depicts the icon used for processes in graphical network representations. When a child process is created, the father gets a new child outport (inport) for each parent inport (outport) of the child. So at the time of its creation, the child has parent ports only.

While in sequential functional programs there is always demand for the evaluation of one expression, which drives the computation, there is simultaneous demand for *all* output streams of a CFP process. Notice that every process must have at least one output, otherwise it could not do relevant work. Still, the demand for output streams does *not* depend on the consumer's request. A computation step consists of the (fair) choice of an outport and a reduction of the expression currently bound to the associated stream variable. Computation done for auxiliary definitions is shared by all outputs. Fully evaluated stream objects are transferred into the channel buffer.

When a process needs the contents of an input stream, this causes a transfer of data from the buffer. If it is empty, computation for the chosen output stream cannot proceed. If the inport does not even exist yet (thus it is a child port), the demand



The set of inports and outports of a process is divided into two subsets: *parent* and *child* input/output ports. The parent ports correspond to the ones declared in the process abstraction as *input* or *output* streams, whereas child ports are formed when a new process is created through the evaluation of a process application. We reserve one corner of the process icon for parent inports (PI), parent outports (PO), child inports (CI), and child outports (CO), respectively.

Figure 2: Input and Output Ports of a Process

for its data leads to the creation of the corresponding child process, i.e. the process application with the input variable on the left hand side is evaluated.

Streams are basically lists, and likewise, they are closed by [] ("nil"). When a process produces [] on an output stream, the corresponding outport is deleted from its set of outports. (The channel buffer remains.) An outport is also eliminated when its consumer terminates: further work on this stream would be wasted. We assume that terminating processes (implicitly) send a signal back on their incoming channels, which are then removed from the system.

A process terminates when it has no more outports (i.e. it has no more work to do). So termination is, like communication, implicit.

4 Stream Redirection

A crucial shortcoming of the calculus fragment introduced so far, resulting from "one at a time" process creation, is that it allows only tree-shaped networks. In order to overcome it, we introduce a new expression *bypass e*, where *e* must evaluate to a (parent or child) *input variable*. This expression is allowed solely as the definition of a (parent or child) *output variable*.

The meaning of the *bypass* construct is just to connect the inport directly with the outport, thus *short-circuiting* both channels, which fuse to a channel connecting the producer of the former inport and the consumer of the former outport. (At the moment of its creation, a process need not know its "true" communication partners. Connections are *always* established first between father and child, and redirected later.) Both channels must have been previously unused, i.e. no data may have been received or produced, resp.; this can be checked easily at run time for inports, and at compile time for outports. The short-circuited ports are removed from the inport/outport sets of the *bypass*ing process.

Figures 3 and 4 indicate how the *bypass* operator solves the problem of defining arbitrary networks: by *restructuring the creation topology* (a tree) to reflect the desired

communication topology. In principle, any process P_1 in the tree can communicate with any other, P_2, if the stream is sent up to the first common ancestor of P_1 and P_2, P_0, and then down to P_2. All the nodes in between have to serve as "relays" which do not really consume or manipulate the data. Using *bypass*, this pseudo-link between P_1 and P_2 is turned into a direct link.

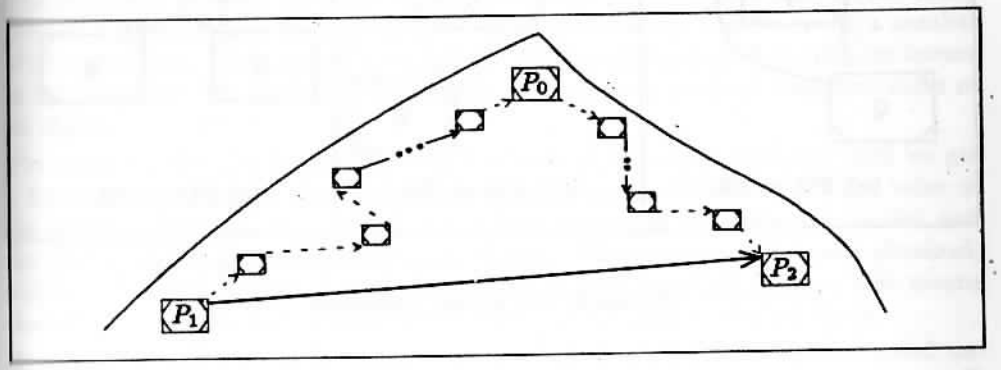


Figure 3: Communication in a Tree

Fig. 4 illustrates the three variants in which *bypass* is applied (dotted/solid arrows correspond to before/after *bypass*): Case (a) (redirecting a child input to a parent outport) is used at all nodes between P_1 and P_0; case (b) is used dually between P_0 and P_2; process P_0 applies case (c).

There is still a last alternative (short-circuiting a parent inport and a parent outport) which may seem useless at a first sight, but remember that the redirecting process may have other ports on which it works, and that redirection is dynamic, so the stream might have been processed in a different situation.

One of the big advantages of CFP is that higher-order process abstractions for parallel programming ("*skeletons*", [3]) can be defined which, when applied, build a certain topology of the desired size and with the given concrete functions. A simple pipeline skeleton is given in Figure 5. It describes the dynamic construction of a pipeline of processes for applying a given list of functions to an input stream. Each pipeline process maps a function of the function list to the input stream. The last process in the pipeline yields the final result. The other processes pass the modified input stream to the successor (son) process and bypass the result channel of the predecessor (father) process.

5 Lazy Evaluation vs. Eager Process Creation

A further problem arises from our wish for *lazy evaluation* within processes, i.e. arguments should be evaluated only if needed, only as far as needed, and they should not be evaluated more than once.

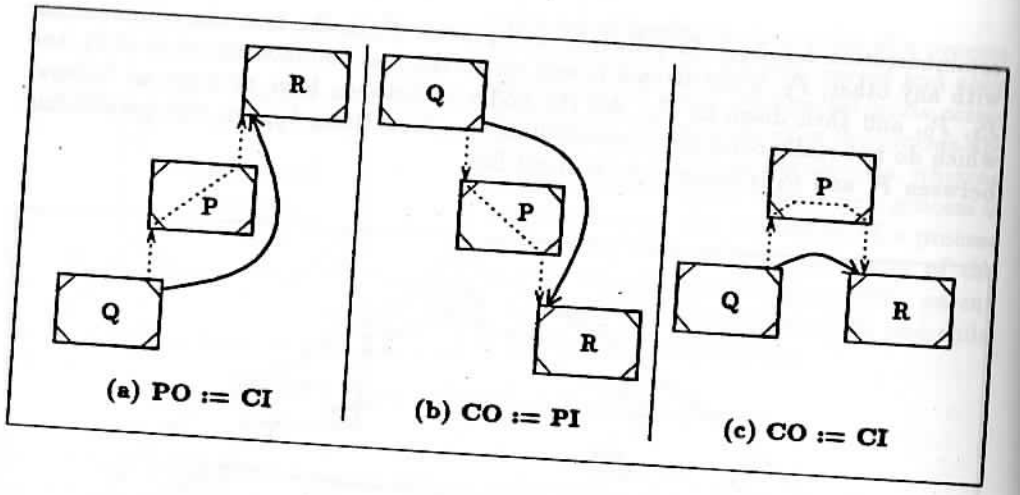


Figure 4: The *bypass* Construct

However, since everything is "demand driven", so is *process creation*! Up to now, child processes are only created when there is demand for their output. But if a process is not created until its results are actually needed, we will often fail to achieve the desired parallelism, which is based on the notion of *doing some work ahead*. Where we want parallelism in time, demand driven creation often causes only distribution in space, leading to *distributed sequentiality*. Let us reconsider the "parallel operation" example of Section 2: Since the arguments to 'op' are evaluated from left to right, there will be no demand for 'p2' until the result from the first child process has arrived (in 'p1'); therefore the second child process is created only after the termination of the first!

We need a way to override laziness, namely the ability to express *eager process creation*. Our solution comprises two elements:

```
pipe =  process function_list input nums output results
        body  results          =  if (isempty (tl function_list))
                                  then (map (hd function_list) nums)
                                  else (bypass son_results)
              (son_results)    =  (pipe (tl function_list))
                                  (map (hd function_list) nums)
              map              =  λf.λl.  if (isempty l) then []
                                          else [(f (hd l)) | (map f (tl l))]
        end
```

Figure 5: A Pipeline Skeleton

- A *process variable* (beginning with the special character ‡) can appear as a *label* on the left hand side of a process application (see Fig. 1).
- A new kind of expression $\langle e_1 \rangle e_2$ is introduced, where e_2 can be of any type and e_1 must evaluate to a *list of process names*. A process name arises from a label when the corresponding process application is evaluated.

The difference between a label and a process name is that a label can induce many process names, because the same process application can be used to create a number of processes (e.g., if it is part of a recursion). This is why we cannot do without *letrecs* in our language: We need the concept of a *local scope* to produce many instances of an object.

The meaning of $\langle e_1 \rangle e_2$ is the following: Evaluate e_1 to normal form, i.e. until we get a list of process names; then continue by evaluating e_2, whose value is the value of the overall expression. By evaluating e_1 all the associated processes are *created*, and the corresponding process names are produced. When process names are obtained, which are already contained in the list, this will have no effect, because this means that these processes were started before, and they will not be started again.

Example (Parallel operation, correct version). *Both* child processes are created eagerly:

```
par_op =  process op   input limits   output res
          body  res          =  if low==high then [low]
                                else (⟨[‡c1, ‡c2]⟩ [op (hd p1) (hd p2)]
                low          =  hd limits
                high         =  hd (tl limits)
                mid          =  (low + high) / 2
                ‡c1 : (p1)   =  (par_op op) ([low, mid])
                ‡c2 : (p2)   =  (par_op op) ([mid+1, high])
          end                                                    □
```

6 Modelling Time Dependent Systems

Section 5 completed the introduction of the CFP constructs which are necessary to describe purely functional (thus deterministic) process systems. However, a typical "concurrent system" shows a rather imperative behaviour, composed of temporally ordered actions. It is possible, (a) to *test* inputs for available data, and (b) to produce data on outputs *after* certain other events.

(a) leads to *value nondeterminism*: Since transmission times may vary, the value of an expression that depends on a tested input can vary as well. (b) in the absence of (a) just leads to *temporal ordering* in addition to what is induced by data dependences.

So far we are unable to express both (a) and (b) in CFP: Neither can the availability of data be tested, nor is it possible to postpone the sending of a value that does not depend on any more unevaluated expressions.

The best illustration of such behaviour is the task of modelling a *storage cell*. The cell should wait for new values from a writer and for requests from a reader; in

response, it should update its contents or send it to the reader. The cell cannot be modelled functionally, because it would have to commit itself to attend either to the value or to the request input, and the correct choice is unpredictable. — But the *reader* cannot be modelled either, because the intended discipline is that it sends a request if and only if it needs the contents of the cell. Since the request stream will be something like 'Req:Req:Req:···' (which can be evaluated immediately), and a process is eager on every output, requests will be output as soon as the reader is started. Synchronization cannot be achieved by introducing some additional data dependence, because the output will still *drive* the computation. What is needed is control in the other direction; otherwise we cannot describe such subordinate outputs.

Non-Functional Processes

The commonly used method for introducing the necessary nondeterminism, a non-functional (fair) stream merge operator merge, contradicts our wish to keep a clean separation between the functional and the concurrent domain. Therefore we do not allow such an operator, but only a complete *process* MERGE, which receives two streams and produces a single output stream.
Indeed, we extend the CFP syntax not only by MERGE, but by a whole family of *predefined process* symbols — which are just a special type of constants. In analogy to built-in functions, their semantics is given by an interpretation. It must properly interact with the system, e.g., when a stream is redirected or deactivated. An obvious restriction is that such processes can have no parameters, because these convey functional values.

Using MERGE, SPLIT (which divides a single stream nondeterministically into two streams) and SINK (which reads all values from its single input and forgets them), *unreliable communication media* can be defined in a simple manner. Fig. 6 gives definitions and pictures for two media which can change the order of data or duplicate data, resp. (we use special icons for the predefined processes). Note that 'medium1' terminates after it has created the MERGE and the SPLIT. Stream 'm' in 'medium2' is duplicated through occurring twice in the body, but only part of it comes back to be re-merged; the other output of SPLIT is lost in SINK.

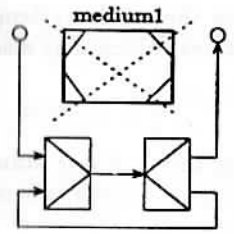
Synchronizing Outputs with Semaphores

Everything a process does results from the demand for an output stream. So synchronization of arbitrary events within one process boils down to synchronization of outputs (or vice versa). In contraposition to Section 5, where excessive laziness was conquered by creating artificial demand at such points of time, now excessive eagerness (of outputs) has to be conquered by restraining demand. We introduce therefore

- a new kind of equation svar = *semaphore*, which defines a semaphore (with the same scope as for normal identifiers);


```
medium1 = process
 input d_in    output d_out
 body d_out = bypass hd sl
      (m)   = MERGE (bypass d_in,
                     bypass hd (tl sl))
      (sl)  = SPLIT (bypass m)
 end
```

```
medium2 = process
 input d_in    output d_out
 body d_out = m
      (m)   = MERGE (bypass d_in,
                     bypass hd sl)
      (sl)  = SPLIT (m)
      ( )   = SINK (bypass hd (tl sl))
 end
```



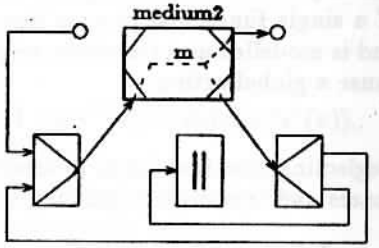


Figure 6: Example Processes "Unreliable Media"

- expressions P exp, exp and V exp, exp, where the first exp must evaluate to a semaphore and the second exp can be of any type.

The convention is that semaphore identifiers begin with the character '$'. A semaphore is always bound either to a positive integer or to a queue of outports. To evaluate $P\,e_1, e_2$ or $V\,e_1, e_2$, first e_1 is evaluated to a semaphore. Now assume that the value of e_s is a semaphore. When $P\,e_s, e$ is evaluated and the semaphore value is positive, it is decremented. Otherwise, the outport which lead to $P\,e_s, e$ is appended to the semaphore queue and *blocked*, i.e. it will not be served until it is dequeued by a V. The expression reduces to e. When $V\,e_s, e$ is evaluated and the semaphore's queue is empty, its value is incremented. Otherwise, the first outport in the queue is unblocked and removed from the queue. Then the expression reduces to e.

Example (Storage cell). The reader of a storage cell can now simply be modelled by blocking the request stream using a semaphore, until a new value is to be read from the storage cell. A merge process merges the request channel coming from the reader and the write channel coming from the writer and produces a single input channel for the storage cell.
The reader process is specified as follows:

```
reader =  process  ...  input  cell-in, ...  output  request, result ...
          body       $req     =  semaphore
                     request  =  P $req, ["Req" | request]
                     result   =  ··· V$req, (hd cell-in) ···
          end
```

Note that semaphores are fully integrated into the functional world. They provide for synchronization of "concurrent threads" within a single process, in contrast to synchronization between processes, which is accomplished by message passing.

7 Sketch of the Operational Semantics

The operational semantics of CFP consists of two descriptive levels: the *local* behaviour of processes, and the *global* behaviour of process networks. The local behaviour of a single functional process describes its own evolution within the whole system, and is modelled as a transition relation on the set Π of process states, which may also cause a global action:

$$\vdash \subset \Pi \times (\Pi \times Act).$$

Neglecting non-functional processes (which are integrated using their private state spaces and transition relations), Π is given by $\Pi := Inputs \times Outputs \times Defs$, where

- $Inputs := Powerset(\ Var \times Strms \times \{free, used\}\ )$ contains sets of inport descriptors.

  Var denotes the set of variables and $Strms$ is the set of channel identifiers. The tag *free* [*used*] indicates that the stream can[not] be bypassed.

- $Outputs := Powerset(\ Var \times Strms \times \{ready, blocked\}\ )$ contains sets of outport descriptors.

  The tag is used for semaphore handling.

- $Defs := \{\, defs : Var \cup PVar \cup SVar \dashrightarrow Comp \,\}$ contains environments of processes.

  $PVar$ and $Svar$ are the sets of process and semaphore variables, respectively. $Comp$ contains partially evaluated expressions.

Processes interact with the global system via *actions* out of the set Act, while the system may influence process states by subsequent *reactions* out of the set $ReAct$. The global behaviour of a program is described as a transition relation $\models \subset \Sigma \times \Sigma$ on the set of global configurations Σ. A *configuration* is a labelled process graph $\langle \mathcal{P}, \mathcal{S}, \gamma, state, buff \rangle$, where

- the set of vertices $\mathcal{P}$ contains process identifiers labelled with the corresponding process state via the mapping $state : \mathcal{P} \to \Pi$,
- the set of edges $\mathcal{S} \subseteq Strms$ contains stream (channel) identifiers which are labelled with the corresponding buffer contents by the mapping $buff : \mathcal{S} \to Val$. Where Val contains semantic values, including list structures and channel identifiers.
- the function $\gamma : \mathcal{S} \to \mathcal{P} \times \mathcal{P}$ defines the connections within the process graph.

7.1 The Local Behaviour of Processes

The behaviour of processes is determined by the equations in its process body, which are embedded in the environment component of its process state. In order to simplify this embedding, a preprocessor eliminates left hand side tuples in process applications by replacing

$$\S x : (x_1, \ldots, x_n) = RHS \quad \text{with } n+1 \text{ equations}$$
$$\S x = RHS, x_1 = \S x.1, \ldots, x_n = \S x.n.$$

For $defs \in Defs$ we define a dereferencing functions

$$defs^*(x) = \begin{cases} defs^*(y) & , defs(x) = y \in \mathbb{D}(defs) \\ x & , defs(x) \notin \mathbb{D}(defs) \end{cases}, \quad defs^+(x) := defs(defs^*(x)),$$

to simplify the access to the defining expressions for left hand side variables.
The *closure of an expression with respect to an environment* is defined by

$$\mathrm{cl}_{defs}(e) := e[x_1 \leftarrow \mathrm{cl}_{defs}(defs(x_1)), \ldots, x_n \leftarrow \mathrm{cl}_{defs}(defs(x_n))],$$

with $\{x_1, \ldots, x_n\} = \mathrm{free}(e) \cap \mathbb{D}(defs)$.
Now we are prepared to give the formal definition of the transition relation on process states:

$$\vdash \subset \Pi \times (\Pi \times Act)$$

We concentrate on the key definitions. Let $\pi = (ins, outs, defs)$.

Termination. The task of a process is the evaluation of its output channels. Therefore, a process without active outports terminates immediately, i.e.

- if $outs = \emptyset$, then $\pi \vdash (\pi, \mathbf{terminate})$.

The action "terminate" announces termination of a process to the system.

Closing outports. A process closes an outport, if the evaluation of this outport yields the empty list or if all outports are blocked by some semaphore variable. In the latter case the process closes all outports step by step and finally terminates. The closing of an outport is communicated to the system by the action $(\mathbf{close}, s_i)$, where s_i is the global channel identifier of the closed outport.

- If $(y, s, ready) \in outs$ and $defs^+(y) = []$
- or if $\{(y, s_i, \langle \text{r-tag} \rangle) \in outs \mid \langle \text{r-tag} \rangle = ready\} = \emptyset$ and $(y, s, blocked) \in outs$

then: $\pi \vdash ((ins, outs \setminus \{(y, s, ready)\}, defs[y/\bot]), (\mathbf{close}, s))$.
In the following, let $out = (y, s, ready) \in outs$, $x = defs^*(y)$ and $e = defs(x)$. The choice of the output channel, which is to be evaluated is nondeterministic.

Bypassing channels. An input channel of a process can only be bypassed, if it has not been used. This is indicated by the tag *free* or *used* in the channel descriptor. If the output channel y is defined by the expression $e = \mathit{bypass}\ z$ or $e = \mathit{bypass}\ x' \wedge \mathit{defs}^*(x') = z$, where $(z, s_z, (\mathrm{tag})) \in \mathit{ins}$, the input channel z has to be bypassed to the output channel y. This means that both channels disappear from the local process state and an action $(\mathrm{bypass}, s, s_z)$ is used to communicate the bypassing to the environment of the process. If the input channel has already been used, bypassing is no longer possible and the input channel is internally redirected to the output channel. The silent action τ is communicated to the environment.

- If $(z, s_z, \mathit{free}) \in \mathit{ins}$ then $\pi \vdash ((\mathit{ins} \setminus \{(z, s_z, \mathit{free})\}, \mathit{outs} \setminus \{\mathit{out}\}, \mathit{defs}[y/\bot]), (\mathrm{bypass}, s, s_z))$.
- If $(z, s_z, \mathit{used}) \in \mathit{ins}$ then $\pi \vdash ((\mathit{ins}, \mathit{outs}, \mathit{defs}[x/z]), \tau)$.

Passing a value via an outport. Whenever the evaluation of an output channel yields a value, this value is written into the corresponding outport in order to be passed via the channel to the inport of another process. Semantically this is modelled by send actions, which communicate the produced value to the environment.

- If $e = [e_1|e_2] \wedge \mathrm{cl}_{\mathit{defs}}(e_1) = v \in \mathit{Val}$, then:

$$\pi \vdash (\mathit{ins}, \mathit{outs}, \mathit{defs}[x/[v|x'],\ x'/e_2][y/x']), (\mathrm{send}, v, s)),$$

where $x' \in \mathit{Var}$ is a new variable, which is introduced to ensure sharing of the further evaluation of y.

Local evaluations are defined by a recursive function

$$\mathit{eval} : \mathit{Var} \times \mathit{Defs} \times \mathit{Inputs} \longrightarrow \mathit{Defs} \times \mathit{Inputs} \times \mathit{Act},$$

which incorporates the reduction rules of λ calculus for the modification of the environment. It models lazy evaluation. The overall evaluation is driven by the demand for the evaluation of the output streams to normal form.
Neglecting semaphore handling, *eval* doesn't modify the set of outputs.
Whenever $\mathit{eval}(x, \mathit{defs}, \mathit{ins}) = (\mathit{defs}', \mathit{ins}', a)$, this means: $\pi \vdash ((\mathit{ins}', \mathit{outs}, \mathit{defs}'), a)$.
The complete definition of the function *eval* is out of the scope of this paper. We discuss only the most interesting cases.

eta reduction. If the outport y is defined by an application of a λ-abstraction, a eta reduction is performed. Sharing of the parameter of the λ-abstraction is ensured by the introduction of a new equation.

- If $e = (e_1\ e_2) \wedge (e_1 = \lambda y.e' \vee e_1 = z \in \mathit{Var} \wedge \mathit{defs}^+(z) = \lambda y.e')$, then

$$\mathit{eval}(x, \mathit{ins}, \mathit{defs}) = (\mathit{defs}[x/e'[y \leftarrow x'], x'/e_2], \mathit{ins}, \tau),$$

where $x' \in \mathit{Var}$ be new.

Beta reduction of process applications. Process abstractions may have formal parameters, which have to be substituted by actual parameters, before the process abstraction can be used for the creation of a process. In principal, this substitution is done in the same way as the substitution of λ-abstracted variables. But it must be ensured that process abstractions remain closed expressions, i.e. the only variables which are allowed in the process body are the input and output variables, remaining process parameters and locally defined variables. Therefore an actual parameter expression must be closed with respect to the current environment, before it is substituted for the formal process parameter in the process body. This complicates the definition of process parameter substitution.

- If $e = (e_1\ e_2) \wedge (e_1 = \varphi \vee e_1 = z \in \mathit{Var} \wedge \mathit{defs}^+(z) = \varphi)$ with
  $\varphi = \mathit{process}\ \ p_1, \ldots, p_q\ \ \mathit{input}\ \ y_1, \ldots, y_r\ \ \mathit{output}\ \ z_1, \ldots, z_s\ ,$
  $\mathit{body}\ \ w_1 = b_1 \cdots w_t = b_t\ \ \mathit{end}$
  then $\mathit{eval}(x, \mathit{ins}, \mathit{defs}) = (\mathit{defs}[x/\varphi'], \mathit{ins}, \tau)$,

  where $\varphi' = \mathit{process}\ \ p_2, \ldots, p_q\ \ \mathit{input}\ \ y_1, \ldots, y_r\ \ \mathit{output}\ \ z_1, \ldots, z_s$
  $\mathit{body}\ \ w_1 = b_1 \cdots w_t = b_t\ w_1' = b_1' \cdots w_u' = b_u'\ \ \mathit{end},$

  and the new part of the environment $\mathcal{E} = \{w_1' = b_1', \ldots, w_u' = b_u'\}$ is defined as follows:

  Let X be the least subset of $\mathit{Var} \cup \mathit{PVar} \cup \mathit{SVar}$ with $\mathrm{free}(e_2) \subseteq X$ and $x \in X \Rightarrow \mathrm{free}(\mathrm{disablePS}(\mathit{defs}(x))) \subseteq X$, where 'disablePS' is a function on *Comp* which replaces all definitions (evaluated or not) of process and semaphore variables in an expression by $\bot$. Process and semaphore variables of the father process have no meaning for the child, so they are rendered ineffective by 'disablePS'.

  Let $X \cap \{p_1, \ldots, p_q, w_1, \ldots, w_t, y_1, \ldots, y_r, z_1, \ldots, z_s\} = \{x_1, \ldots, x_m\}$ and $x_1', \ldots, x_m'$ be new so that $\vartheta := [x_1 \leftarrow x_1', \ldots, x_m \leftarrow x_m']$ preserves the variable classes.

  Then $\mathcal{E} := \{p_1 = e_2\vartheta\} \dot\cup \{x = \mathrm{disablePS}(\mathit{defs}(x)) \mid x \in X\}\vartheta$.

Process creation is initiated, when a process application is evaluated. A create action is generated to ask the environment to generate the new process and to extend the inputs and outputs of the father process by the descriptors of the new inports and outports to the child process. New variables for the outports are included in the create message in order to provide the environment with the information necessary to perform a correct extension of the father's outputs.

- If $e = e'(e_1, \ldots, e_r) \wedge (e' = \varphi \vee e' = z \in \mathit{Var} \wedge \mathit{defs}^+(z) = \varphi)$ with $\varphi \in \mathit{CPExp}$, where *CPExp* is the set of non-functional process identifiers or parameterless process abstractions,

  then $\mathit{eval}(x, \mathit{ins}, \mathit{defs}) = (\mathit{defs}[x_1/e_1, \ldots, x_r/e_r], \mathit{ins}, (\mathrm{create}, \varphi, x, x_1 \cdots x_r))$,
  where $x_1, \ldots, x_r \in \mathit{Var}$ are new.

Requesting values from inports. The evaluation of an input channel causes a request action that informs the environment that the process wants to read a value from the buffer of the channel connected to the inport.

- If $(x, s_x, \langle \text{tag} \rangle) \in ins$ and $defs(x)$ is undefined (no input value available), then

$$eval(x, ins, defs) = (defs, ins, (\textbf{request}, s_x, x).$$

It is expected that the corresponding reaction of the environment transmits a value v from the buffer associated with s_x to the process body.

Actions. To sum up, the following actions are produced by the processes:

$$\begin{aligned} Act := \{\ & \textbf{terminate}, (\textbf{close}, s_i), (\textbf{send}, v, s_i), (\textbf{bypass}, s_i, s_j), \\ & (\textbf{request}, s_i, x), (\textbf{create}, \varphi, p, xs), \tau \mid \\ & v \in Val,\ s_i, s_j \in Strms,\ x \in Var,\ p \in PVar,\ xs \in Var^*,\ \varphi \in CPExp \} \end{aligned}$$

7.2 The Global Behaviour of Process Networks

The global behaviour of process networks is defined by the transition relation $\models \subseteq \Sigma \times \Sigma$ on system configurations. Actions caused by processes within the process graphs lead to global reconfigurations of the process system.

Let $\sigma = \langle \mathcal{P}, \mathcal{S}, \gamma, state, buff \rangle \in \Sigma$ and $P \in \mathcal{P}$ with $state(P) = \pi$ and $\pi \vdash (\pi', a)$. Depending on the process action a the following transition steps are possible. We discuss only three typical cases.

Moving a value into a channel buffer. If a process sends a value along an outport, a send-action $a = (\text{send}, v, s)$ communicates this event to the system, which behaves as follows:

- $\sigma \models \langle \mathcal{P},\ \mathcal{S},\ \gamma,\ state[P/\pi'],\ buff[s/buff(s) \cdot v)] \rangle$.

Bypassing. A process which bypasses an input channel to an output channel, produces an action $a = (\text{bypass}, s_{to}, s_{from})$, which causes the following global transition:

- Let $(P_{from}, P) = \gamma(s_{from})$ and $(P, P_{to}) = \gamma(s_{to})$. Then:

$$\begin{aligned} \sigma \models \langle & \mathcal{P}; \mathcal{S} \setminus \{s_{to}\}, \gamma[s_{to}/\bot,\ s_{from}/(P_{from}, P_{to})], \\ & state[P/\pi',\ P_{to}/update(state(P_t), (\text{bypass}, s_{to}, s_{from}))], buff[s_{to}/\bot] \rangle. \end{aligned}$$

The function $update : \Pi \times ReAct \to \Pi$ processes reactions of the system on the process states. In the case of the bypass reaction, it replaces the stream identifier s_{to}, which has been eliminated by the bypass, by the stream identifier s_{from}.

Process creation is the most interesting event, which causes the extension of the labelled process graph. When a process activates a child process, the action $a = (\text{create}, \varphi, x, x_1 \cdots x_r)$ with $\varphi = \begin{array}{lll} process & input & y_1, \ldots, y_r \quad output \quad z_1, \ldots, z_s \\ & body & w_1 = b_1 \cdots w_t = b_t \quad end \end{array}$ is generated. The system will now create a new process node, which is connected to the node P via the newly created input/output channels of the child process.

- Let $\tilde{P}$ be a new process identifier and $s_{i_1}, \ldots, s_{i_r}, s_{j_1}, \ldots, s_{j_s}$ be new stream identifiers. Then:

$$\begin{aligned} \sigma \models \langle & \mathcal{P} \dot{\cup} \{\tilde{P}\}, \mathcal{S} \dot{\cup} \{s_{i_1}, \ldots, s_{i_r}, s_{j_1}, \ldots, s_{j_s}\}, \gamma[s_{i_k}/(P, \tilde{P}),\ s_{j_l}/(\tilde{P}, P)], \\ & buff[s_{i_k}/\epsilon,\ s_{j_l}/\epsilon], state' \rangle, \end{aligned}$$

where $state' := state[P/\pi'',\ \tilde{P}/\tilde{\pi}]$ with

$$\begin{aligned} \pi'' &:= update(\pi', (\textbf{create}, x, x_1 \cdots x_r, s_{i_1} \cdots s_{i_r}, s_{j_1} \cdots s_{j_s})) \text{ and} \\ \tilde{\pi} &:= (ins, outs, defs) \text{ with } \begin{array}{l} ins = \{(y_1, s_{i_1}, free), \ldots, (y_r, s_{i_r}, free)\}, \\ outs = \{(z_1, s_{j_1}, ready), \ldots, (z_s, s_{j_s}, ready)\}, \\ defs = [w_1/b_1, \ldots, w_t/b_t]. \end{array} \end{aligned}$$

The effect of the *update*-function in case of the create-reaction is the back communication of the global stream identifiers for the child outports and inports of the father process.

This concludes the sketch of the operational semantics. Further details will be contained in the full version of this paper.

8 Summary

CFP, as an integration of the Hindley-Milner-typed λ-calculus with the Kahn / MacQueen model of parallel computation, is a **functional process calculus**, suitable for *Functional parallel programming:*

- **Closed process abstractions** describe functional units of parallel computation.
- **Hyperstrict, algebraic-type streams** provide efficient communication.
- **Explicit process applications** create dynamically evolving process networks.
- **Processes as second class citizens** secure clean differentiation of semantic objects.
- **Implicit sending and receiving** retains internally true functional programming.
- **Stream redirection** operator enables description of arbitrary topologies.
- **Eager process creation** offers functional scheduling during lazy evaluation.

and *Programming of time-dependent concurrent systems:*

- **Non-functional processes** constitute sources of nondeterminism.
- Especially, `MERGE` allows simultaneous waiting on several inputs.
- **Process-local semaphores** control relative computation order of outputs.

References

[1] G. Boudol: *Towards a Lambda-Calculus for Concurrent and Communicating Systems*, TAPSOFT '89, LNCS 351, Springer-Verlag 1989.

[2] F.W. Burton: *Annotations to Control Parallelism and Reduction Order in the Distributed Evaluation of Functional Programs*, ACM TOPLAS, Vol. 6, 1984.

[3] M. Cole: *Algorithmic Skeletons: A Structured Approach to the Management of Parallel Computation*, report CST-56-88, University of Edinburgh, 1988.

[4] A. Giacalone, P. Mishra, S. Prasad: *Facile: a Symmetric Integration of Concurrent and Functional Programming*, TAPSOFT '89, LNCS 352, Springer-Verlag 1989.

[5] P. Hudak, L. Smith: *Para-Functional Programming: A Paradigm for Programming Multiprocessor Systems*, 13th ACM Symp. on POPL, 1986.

[6] P. Hudak: *Para-Functional Programming in Haskell*, Chapter 5 in: B.K. Szymanski (ed.): Parallel Functional Languages and Compilers, ACM Press 1991.

[7] S.B. Jones, A.F. Sinclair: *Functional Programming and Operating Systems*, The Computer Journal, vol.32, no.2, 1989.

[8] G. Kahn, D.B. MacQueen: *Coroutines and Networks of Parallel Processes*, Information Processing 77, IFIP, North-Holland 1977.

[9] P. Kelly: *Functional Programming for Loosely-Coupled Multiprocessors*, Pitman 1989.

[10] J. Launchbury: *A Natural Semantics for Lazy Evaluation*, 20th ACM Symp. on POPL, 1993.

[11] F.T. Leighton: *Introduction to Parallel Algorithms and Architectures: Arrays – Trees – Hypercubes*, Morgan Kaufmann Publishers 1992.

[12] R. Milner: *Communication and Concurrency*, Prentice Hall 1989.

[13] R. Milner: *Functions as Processes*, Mathematical Structures in Computer Science 2, 1992.

[14] F. Nielson: *The Typed λ-Calculus with First-Class Processes*, PARLE '89, LNCS 366, Springer Verlag 1989.

[15] S.L. Peyton Jones: *Parallel Implementations of Functional Programming Languages*, The Computer Journal, vol.32, no.2, 1989.

[16] D.A. Turner: *An Approach to Functional Operating Systems*, Chapter 8 in: D.A. Turner (ed.): Research Topics in Functional Programming, Addison-Wesley 1990.

El Modelo RPS para la Gestión del Paralelismo AND Independiente en Programas Lógicos

Ramiro Varela Arias
Centro de Inteligencia Artificial.
Universidad de Oviedo.
Campus de Viesques. 33271. Gijón.
e-mail: ramiro@aic.uniovi.es

Resumen

En esta comunicación presentamos un modelo de gestión del paralelismo AND independiente conjuntamente con el paralelismo OR total en programas lógicos. Nos centramos en la descripción de la Red de Procesos y Soluciones (RPS) que es la estructura utilizada por los procesos AND para representar a las soluciones parciales que reciben de los procesos OR sucesores. Esta representación es uno de los problemas principales que se presentan cuando se explotan simultáneamente las dos fuentes de paralelismo. La solución que proponemos presenta algunas propiedades de interés, como por ejemplo una reducción significativa, con respecto a otros modelos, en el número total de procesos generados durante la evaluación de una pregunta.

1 Introducción

En los últimos años se han desarrollado un gran número de modelos de evaluación en paralelo de programas lógicos, debido por una parte a las expectativas que ofrecen las arquitecturas masivamente paralelas, y por otra a las numerosas fuentes de paralelismo que proporciona la programación lógica, de las que las principales son el paralelismo AND y el paralelismo OR. Algunos modelos explotan simultáneamente estos dos modos de paralelismo, por ejemplo los presentados en [4, 6, 9]. El principal inconveniente que presentan es el elevado número de procesos que se generan, sobre todo en programas altamente no deterministas, y la gestión del paralelismo AND. Así, en otros modelos en aras a la simplificación de los cálculos, se opta por restringir de alguna forma el paralelismo. Por ejemplo en muchos casos se utiliza el paralelismo AND independiente ([3, 4, 5, 7, 9]). En otros se utiliza únicamente el paralelismo OR ([2]). Y en algunos se sustituye el paralelismo OR por alguna estrategia de backtraking ([7]).

En esta comunicación presentamos un modelo que explota el paralelismo AND independiente, conjuntamente con el paralelismo OR, además de alguna de las fuentes secundarias. En la sección 2 describimos el modelo abstracto de interpretación que consiste en el recorrido del árbol AND/OR asociando un proceso independiente a cada uno de sus nodos. Definimos formalmente el modo de construir el árbol e incluimos un ejemplo sencillo. En la sección 3 describimos el Retículo de Flujo de Datos (RFD) que es la estructura utilizada para representar la ordenación parcial de los literales de una pregunta para la realización del paralelismo AND independiente. En la sección 4 describimos las redes de herencias múltiples que se utilizan para representar las soluciones parciales de cada predicado. Las secciones 5 y 6 constituyen la parte central de esta