A Confluent Relational Calculus for Higher-Order Programming with Constraints

Joachim Niehren, Gert Smolka

A Confluent Relational Calculus for Higher-Order Programming with Constraints

Joachim Niehren * Gert Smolka **

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

Abstract. We present the ρ-calculus, a relational calculus parametrized with a logical constraint system. The ρ-calculus provides for higher-order relational programming with first-order constraints, and subsumes higher-order functional programming as a special case. It captures important aspects of the concurrent constraint programming language Oz. We prove the uniform confluence of the ρ-calculus. Uniform confluence implies that all maximal derivations issuing from a given expression have equal length. But even confluence of a nonfunctional calculus modelling computation with partial information is interesting on its own right.

1 Introduction

We present the ρ-calculus, a relational calculus parametrized by a logical constraint system. The ρ-calculus provides for higher-order relational programming with first-order constraints.

The ρ-calculus captures interesting aspects of the concurrent constraint programming language Oz [3]. It is a minimalistic subcalculus of the Oz-Calculus [12] which, in contrast, integrates a variety of paradigms into a single formalism. The Oz-Calculus models functional, object-oriented, constraint-based and logic programming [11].

We prove the uniform confluence of the ρ-calculus. Uniform confluence implies that all maximal derivations issuing from a given expression have equal length. This, in particular, yields equivalence of normalization and strong normalization on the same expression. But even confluence of a nonfunctional calculus modelling computation with partial information is interesting on its own right.

* Joachim Niehren has been supported by a fellowship from the 'Graduiertenkolleg Informatik der Universität des Saarlandes'.

** Gert Smolka has been supported by the Bundesminister für Forschung und Technologie (FTZ-ITW-9105), the Esprit Project ACCLAIM (PE 7195), and the Esprit Working Group CCL (EP 6028).

In Jean-Pierre Jouannaud, ed. 1st International Conference on Constraints in Computational Logics, Munich, Germany, September 1994, LNCS 845.

Recently, two other minimalistic relational calculi have been proposed: the δ-calculus [8] and the γ-calculus [13] The δ-calculus models purely functional computation in a relational setting. It is uniformly confluent and embeds the eager λ-calculus. The γ-calculus is a extension of the δ-calculus being a minimalistic foundation for relational, concurrent, and object-oriented programming. In particular, it provides for lazy functional programming with sharing [5] integrating concurrent state [1, 6].

The γ-calculus provides for logical variables but not for first-order unification, which would amount to the integration of a tree constraint system. This choice makes the γ-calculus technically simpler than the ρ-calculus. However, in relational calculi providing for encapsulated search [11] or type inference [7], first-order unification or other forms of constraint solving are needed. For these purposes, extensions of the ρ-calculus are appropriate.

We continue with some technical remarks on uniform confluence. In [8], uniform confluence of the eager λ-calculus and the δ-calculus are proved. The proof for the eager λ-calculus is trivial, whereas the proof for the δ-calculus requires a deep analysis of parallel composition. We point out that proofs of uniform confluence are based on local considerations excluding termination. This is possible, since uniform confluence implies Huet's notion of strong confluence [4].

A central contribution of this paper is the proof of the uniform confluence of the ρ-calculus. This proof is hard. It is based on a method developed for the δ-calculus [8]. Additionally, it needs a novel method for the decomposition of equivalence relations. This new complexity comes with the freedom of constraint handling.

The confluence of the ρ-calculus depends essentially on the concept of names. Names are first-class citizens. This means that variables may denote names and that names can be passed as parameters. Names can be tested for equality and disequality in first-order logic. An equation $a = b$ between names is unsatisfiable if and only if a and b are distinct.

The ρ-calculus ensures that all abstractions are equipped with a unique name. On creation of a new abstraction a new name is created, relying on a mechanism independently proposed in [14, 10, 9]. Without the above invariant, confluence of the ρ-calculus would fail. For instance, consider the following expression E being a composition of two abstractions with the same name a:

$$a\!:\!x\,/\top \wedge a\!:\!y\,/\bot\,.$$

E reduces to non-joinable expressions when composed with an application $a\,z$:

$$E \wedge \bot \;\leftarrow\; E \wedge a\,z \;\rightarrow\; E \wedge \top\,.$$

The paper is organized as follows: First, we define constraint systems and present the ρ-calculus. Then, we give a typical example for programming in the ρ-calculus. After that, we investigate uniform confluence and its consequences for general calculi. Finally, we formulate our decomposition method and prove the uniform confluence of the ρ-calculus.

2 Constraint Systems

We introduce constraint systems based on first-order logic with equality as in [12, 13].

We assume an infinite set of *variables* ranged over by x, y, z and an infinite set of *names* denoted by a, b, c. The letters u, v and w stand for *references* being either variables or names. Throughout the paper, we will freely use the replacement operator $[u/v]$ (replace u for v) and apply it to logical formulae and other syntactical categories.

A *first-order signature* Σ declares constants, function symbols and predicate symbols. A *theory* Δ over Σ is a set of closed first-order formulae over Σ. A theory is *consistent* if it has a model. The formula $\tilde{\forall}\,\phi$ is an abbreviation for the universal closure of ϕ. A formula ϕ is *valid* with respect to a theory Δ if $\tilde{\forall}\,\phi$ is valid in all models of Δ. In this case, we write $\Delta \models \phi$.

A *constraint system* consists of a *constraint signature* Σ, a *constraint theory* Δ and a set of *constraints* ranged over by ϕ and ψ. Σ is a first-order signature containing all names as distinguished constants. Δ is a consistent theory over Σ. The set of constraints is a subset[3] of first-order formulae over Σ including bottom $\bot$, top $\top$, and equations $u = v$, and closed under conjunction $\phi \wedge \psi$, existential quantification $\exists x\,\phi$, and replacement $\phi[u/v]$. We require the following two conditions for all a, b and ϕ:

1. $\Delta \models \neg a = b$ if a and b are distinct,
2. $\Delta \models \phi \leftrightarrow \phi[a/b]$ if ϕ is closed and does not contain a.

The first condition allows to test names for disequality. The latter ensures consistency, when α-renaming for names comes into play. In particular, the above requirements imply that names are different from any value that can be described by a formula (see [12]).

3 The ρ-Calculus

We present the ρ-calculus with respect to a fixed constraint system. The ρ-calculus is the restriction of an auxiliary calculus that is specified by *expressions*, *structural congruence*, and *reduction*.

With respect to the structural congruence, an expression can be considered as *computation space* consisting of a *board* and a collection of *actors*. A board contains the information accumulated so far consisting of constraints and abstractions. Actors are conditionals and applications. They reduce driven by the information on the board and disappear. Reduction possibly adds new information and new actors.

[3] This slightly extends [12, 13] where all first-order formulae are constraints.

The abstract syntax of *expressions* ranged over by E and F is defined by the grammar in Figure 1. We write $\overline{u}$ ($\overline{x}$ and $\overline{a}$ resp.) as abbreviation for possibly empty, finite sequences of references (variables and names resp.).

$E, F ::= \phi$		*constraint*
	$a{:}\overline{y}\,/E$	*(named) abstraction*
	$u\,\overline{v}$	*application*
	$\mathsf{if}\ \phi\ \mathsf{then}\ E\ \mathsf{else}\ F\ \mathsf{fi}$	*conditional*
	$E \wedge F$	*(parallel) composition*
	$\exists u\ E$	*declaration*

Fig. 1. Expressions

An abstraction $a{:}\overline{y}\,/E$ is named by the name a, has *formal parameters* $\overline{y}$ and *body* E. An expression $u\,\overline{v}$ is an application of an abstraction named u with *actual parameters* $\overline{v}$. A conditional $\mathsf{if}\ \phi\ \mathsf{then}\ E\ \mathsf{else}\ F\ \mathsf{fi}$ has *guard* ϕ and *branches* E and F. Conjunction on constraints and composition on expressions coincide, likewise a existential quantification and declaration of variables.

Bound variables are introduced by constraints, as formal parameters of abstractions and by declaration. *Bound names* are introduced by declaration only. References that are not bound are called *free*. $\mathcal{F}(E)$ and $\mathcal{B}(E)$ denote the sets of free respectively bound references in E.

Abstractions may be named by variables. For this purpose, we introduce $x{:}\overline{y}\,/E$ as syntactic sugar for the expression $\exists a\ (x = a \wedge a{:}\overline{y}\,/E)$. This ensures that abstractions named by variables have a unique name.

The *structural congruence* $\equiv$ is the least congruence on expressions satisfying the axioms in Figure 2. It provides for the usual properties of composition and declaration, identifies logically equivalent constraints (*Equ*) and allows for the replacement of variables by equal references (*Repl*).

(α)	capture free renaming of bound references	
(ACI)	$\wedge$ is associative and commutative and satisfies $E \wedge \top \equiv E$	
(Ex)	$\exists u \exists v\ E \ \equiv\ \exists v \exists u\ E$	
(Mob)	$(\exists u\ E) \wedge F \ \equiv\ \exists u\ (E \wedge F)$	*if* $u \notin \mathcal{F}(F)$
(Equ)	$\phi \ \equiv\ \psi$	*if* $\Delta \models \phi \leftrightarrow \psi$
$(Repl)$	$x = u \wedge E \ \equiv\ x = u \wedge E[u/x]$	if $u \notin \mathcal{B}(E)$[4]

Fig. 2. Axioms of Structural Congruence

[4] For technical simplicity, we prefer to use $u \notin \mathcal{B}(E)$ instead of u *is free for* x *in* E.

With respect to declaration, the structural congruence treats names similar to variables. This simple machinery circumvents inconsistent equations between higher-order procedures using first-order constraints:

$$\begin{aligned} x{:}\overline{y}\,/E \wedge x{:}\overline{z}\,/F &= \exists a\,(x = a \wedge a{:}\overline{y}\,/E) \wedge \exists a\,(x = a \wedge a{:}\overline{z}\,/F) \\ &\equiv \exists a \exists b\,(x = a \wedge x = b \wedge a{:}\overline{y}\,/E \wedge b{:}\overline{z}\,/F) \\ &\equiv \bot \wedge \exists a \exists b\,(a{:}\overline{y}\,/E \wedge b{:}\overline{z}\,/F) \end{aligned}$$

Reduction $\rightarrow$ is the least binary relation on expressions satisfying the axioms in Figure 3 and the rules in Figure 4. We use an generalized replacement operator $[\overline{u}/\overline{x}]$ for simultaneous substitution. Its application implicitly requires that $\overline{u}$ and $\overline{x}$ have equal length and that $\overline{x}$ is linear (i.e. all elements of $\overline{x}$ are pairwise distinct).

$(Appl)$	$a{:}\overline{y}\,/E \wedge a\,\overline{u} \;\rightarrow\; a{:}\overline{y}\,/E \wedge E[\overline{u}/\overline{y}]$	*if* $\overline{u} \notin \mathcal{B}(E)$
$(Then)$	$\psi \wedge \mathsf{if}\ \phi\ \mathsf{then}\ E\ \mathsf{else}\ F\ \mathsf{fi} \;\rightarrow\; \psi \wedge E$	if $\Delta \models \psi \rightarrow \phi$
$(Else)$	$\psi \wedge \mathsf{if}\ \phi\ \mathsf{then}\ E\ \mathsf{else}\ F\ \mathsf{fi} \;\rightarrow\; \psi \wedge F$	if $\Delta \models \psi \rightarrow \neg\phi$

Fig. 3. Axioms of Reduction

Application (*Appl*) executes procedure calls by passing actual parameters for formal ones. Note that references are passed but not expressions. A conditional is reducible whenever sufficient information has been accumulated in form of constraints. Irreducible conditionals are called *suspended*. The axiom (*Then*) commits to the *then*-branch, if the guard is *entailed* ($\Delta \models \psi \rightarrow \phi$), whereas (*Else*) chooses the *else*-branch if the guard is *disentailed* ($\Delta \models \psi \rightarrow \neg\phi$).

An alternative formulation of reduction of conditionals is given in [13]. It is based on constraint propagation modelling relative simplification of guards. For the purpose of confluence, the presented version is simpler. However, relational calculi extended with deep guards require relative simplification [12].

The rules in Figure 4 formalize that reduction is closed under the structural congruence and allowed in every context but not in abstractions and conditionals.

$$\frac{E_1 \rightarrow E_2}{\exists u\, E_1 \rightarrow \exists u\, E_2} \qquad \frac{E_1 \rightarrow E_2}{E_1 \wedge F \rightarrow E_2 \wedge F} \qquad \frac{E_1 \equiv E_2 \quad E_2 \rightarrow E_3 \quad E_3 \equiv E_4}{E_1 \rightarrow E_4}$$

Fig. 4. Rules of Reduction

We continue restricting of the auxiliary calculus above. First, we exclude that several abstractions have the same name. Second, we take logical inconsistencies into account.

An expression E is *admissible* if it satisfies the following two conditions:

1. E does not contain two abstractions with the same name.
2. The name of an abstraction nested inside another abstraction of E is declared in the body of the enclosing abstraction.

Both restrictions do not diminish expressiveness. Real programs are usually written in suggared syntax and use abstractions $x{:}\,\overline{y}\,/E$ named by variables exclusively. Clearly, expressions obtained by expansion of those programs are admissible.

Proposition 1. *Admissibility is preserved by structural congruence and reduction.*

An expression is *failed* if it is congruent to $\bot \wedge E$ for some E and *unfailed* otherwise. Failure obviously destroys confluence. For instance, consider $\bot \wedge x\,y \wedge a{:}\,y\,/\top \wedge b{:}\,y\,/(\exists c\,c{:}\,z\,/\top \wedge x\,y)$ whichs leads to several finite and one infinite derivation that can not be joined.

One possible solution of the problem is to exclude expressions that will eventually fail. But we can be less restrictive exploiting the following property valid for all E: If there exists a derivation on E leading to failure then all finite derivations on E can be extended to failure. The solution, we finally choose, is to add the axiom (*Bot*) in the definition of reduction.

$$(Bot) \qquad \bot \wedge E \;\to\; \bot$$

(*Bot*) only applies to failed expressions forcing all maximal derivations to be infinite:

$$\bot \wedge E \;\to\; \bot \;\equiv\; \bot \wedge \top \;\to\; \bot \;\to\; \ldots$$

Conversely, uniform confluence requires that all maximal derivations on failed expressions are infinite, since failure may occur after an arbitrary number of reduction steps.

In the sequel we denote the relational composition of two binary relations $\to_1$ and $\to_2$ with $\to_1 \circ \to_2$. The restriction of a relation $\to$ to a set $\mathcal{E}$ is written as $\to_{|\mathcal{E}}$.

Definition 2. An *ρ-expression* is an admissible expression. The set of all ρ-expressions is denoted by $\mathcal{E}$. The *ρ-calculus* is the triple $(\mathcal{E}, \equiv_{|\mathcal{E}}, \to_{|\mathcal{E}})$.

The following property of the ρ-calculus is important. It will be generalized in Section 5 in order define a abstract notion of calculus.

Proposition 3. *The ρ-calculus satisfies* $\to_{|\mathcal{E}} \;=\; (\equiv_{|\mathcal{E}} \circ \to_{|\mathcal{E}} \circ \equiv_{|\mathcal{E}})$.

This is a consequence of Proposition 1 and the $\to \;=\; (\equiv \circ \to \circ \equiv)$.

Theorem 4 (Uniform Confluence). *The ρ-calculus is uniformly confluent; that is, for all admissible E and all F_1, F_2 such that $F_1 \leftarrow E \to F_2$ either $F_1 \equiv F_2$ or there exists G with $F_1 \to G \leftarrow F_2$.*

4 Examples

We illustrate the programmable control of data-flow of the ρ-calculus in connection with higher-order procedures.

First, we define two relational procedures *Add1* and *Add2* for the addition of integers. They are both correct with respect to the logical formula

$$\forall x \forall y \forall z \ (add(x, y, z) \leftrightarrow x + y = z) \ .$$

Add1 may proceed in computations with incomplete information, whereas *Add2* suspends until the values x and y are determined.

Second, we define two procedures *Sum1* and *Sum2* for the summation of lists of integers. They are created generically from *Add1* and *Add2* applying the higher-order procedure *Fold*. The data-flow of *Sum1* (resp. *Sum2*) generalizes the data-flow of *Add1* (resp. *Add1*).

We chose a constraint system providing for trees and integers with addition. Its signature Σ contains at least the binary function symbols *cons* and $+$, a unary relation symbol *int*, and constants *nil*, 0, $\bot 1$, $+1$, ... As constraints, we allow for all first-order formulae not containing universal quantification, negation and disjunction. An appropriate theory Δ can be defined combining the first-order theories of integers and trees. For instance, it provides for

$$\Delta \ \models \ x + 0 = z \ \leftrightarrow \ int(x) \wedge x = z \ .$$

We freely use syntactic sugar for lists and nesting. It $t, t_1, \ldots, t_n$ are terms over Σ then we define

$$\begin{array}{rcl} [t_1 \, t_2 \ldots t_n] & = & cons(t_1 \; cons(t_2 \; \ldots \; nil)) \\ u \, t_1 \ldots t_n & = & \exists x_1 \ldots \exists x_n \, (u \, x_1 \ldots x_n \wedge x_1 = t_1 \wedge \ldots \wedge x_n = t_n) \end{array}$$

As concrete variables (resp. names) we chose alpha-numeric expressions starting with a capital (resp. lower case) letter. The procedure *Add1* is defined by

$$Add1 : X \, Y \, Z \, / \, X + Y = Z$$

We remember that this is syntactic sugar that has to be expanded before reduction. The computation of $Add1 \, 2 \, Y \, 5$ is done by the following derivation:

$$\begin{array}{rcl} & & Add1 \, 2 \, Y \, 5 \ \wedge \ Add1 : X \, Y \, Z \, / \, X + Y = Z \\ & = & \exists X \exists Y \, (Add1 \, X \, Y \, Z \ \wedge \ X = 2 \ \wedge \ Z = 5) \ \wedge \ \exists a \, (Add1 = a \ \wedge \ a : \ldots) \\ & \equiv & \exists a \exists X \exists Z \, (X = 2 \ \wedge \ Z = 5 \ \wedge \ Add1 = a \ \wedge \ a \, X \, Y \, Z \ \wedge \ a : \ldots) \\ & \rightarrow & \exists a \exists X \exists Z \, (X = 2 \ \wedge \ Z = 5 \ \wedge \ Add1 = a \ \wedge \ X + Y = Z \ \wedge \ a : \ldots) \\ & \equiv & Y = 3 \ \wedge \ \exists a \, (Add1 = a \ \wedge \ a : \ldots) \\ & = & Y = 3 \ \wedge \ Add1 : \ldots \end{array}$$

We abbreviate the above derivation by $Add1 \, 2 \, Y \, 5 \ \rightarrow \ Y = 3$. Further possible derivations are $Add1 \, 2 \, 3 \, Z \ \rightarrow \ Z = 5$ and even $Add1 \, X \, 0 \, Z \ \rightarrow \ int(X) \ \wedge \ X = Z$. The second procedure for addition *Add2* is defined by:

$$Add2 : X \, Y \, Z \ / \ \exists a \, \mathsf{if} \ X = a \ \mathsf{then} \ \top \ \mathsf{else} \ \mathsf{if} \ Y = a \ \mathsf{then} \ \top \ \mathsf{else} \ X + Y = Z \ \mathsf{fi} \ \mathsf{fi}$$

It suspends until the parameters X and Y are determined. For example, the following application terminates with an suspending conditional:

$$Add2\,2\,Y\,5 \;\;\rightarrow^* \;\; \exists a\ (\mathsf{if}\ Y = a\ \mathsf{then}\ \top\ \mathsf{else}\ 2 + Y = 5\ \mathsf{fi})$$

Fold is well known from functional programming. As inputs, it takes a list $L = [X_1\ X_2\ \ldots\ X_n]$, a binary functional procedure P, and a start value S. Its output is the result of applying P recursively to the elements of L, from left to right, starting with S. Hence, $Fold(L, P, S) \;=\; P(P(\ldots P(P(S\,, X_1),\ X_2)\ldots),\ X_n)$. In the ρ-calculus, functional abstractions can be represented as relational ones by adding an explicit output parameter R:

$$\begin{array}{l} Fold : L\ P\ S\ R\ /\ \mathsf{if}\ L = nil\ \mathsf{then}\ R = S \\ \qquad\qquad \mathsf{else}\ \exists X_1 \exists L_1 \exists S_1\ (L = cons(X_1, L_1) \wedge P\ S\ X_1\ S_1 \wedge Fold\ L_1\ P\ S_1\ R)\ \mathsf{fi} \end{array}$$

The procedure *Create* abstracts over the second and third argument of *Fold*. Thus, *Create* inputs P and S and outputs a new abstraction named A using the remaining parameters L and R:

$$Create : P\ S\ A\ /\ (A : L\ R\ /\ Fold\ L\ P\ S\ R)$$

Now we can create the procedure *Sum1* (resp. *Sum2*) computing the sum of lists by application of *create* with $Add1$ and (resp. $Add2$):

$$Create\ Add1\ Sum1 \;\rightarrow\; Sum1 : L\ R\ /\ Fold\ L\ Add1\ S\ R$$

The data-flow of *Sum1* is as dynamic as the data-flow of *Add1*. For instance,

$$Sum1\ [2\ Y\ 3]\ 9 \quad \rightarrow^* \quad Y = 4 \;\; .$$

Choosing *Sum2* instead of *Sum1* ends up with a suspending conditional whenever one of the elements of the list is not determined.

5 General Calculi

We define an abstract notion of calculus appropriate for the investigation of uniform confluence in general. An (abstract) *calculus* consists of a set $\mathcal{E}$ an equivalence relation $\equiv$ on $\mathcal{E}$, and a binary relation $\rightarrow$ on $\mathcal{E}$ satisfying the property

$$\rightarrow \;=\; (\equiv \circ \rightarrow \circ \equiv)\ .$$

The elements of $\mathcal{E}$ are the *expressions* of the calculus, $\equiv$ is its *congruence*, and $\rightarrow$ its *reduction*. Note that every binary relation on a set $\mathcal{E}$ defines a calculus when taking the identity on $\mathcal{E}$ as congruence.

Given a calculus, we define the following relations:

$$\rightarrow_0 = \equiv\ , \qquad \rightarrow_{n+1} = \ (\rightarrow_n^{\circ} \rightarrow)\ , \qquad \rightarrow^* = \ \cup_{n \geq 0}\ \rightarrow_n\ .$$

A calculus is *confluent* iff $(^*\!\leftarrow \circ \rightarrow^*) \ \subseteq \ (\rightarrow^* \circ\ ^*\!\leftarrow)$ and *Church-Rosser* iff $(\leftarrow \cup \rightarrow)^* \subseteq (\rightarrow^* \circ\ ^*\!\leftarrow)$. It is *strongly confluent* iff $(\leftarrow \circ \rightarrow) \subseteq ((\equiv \cup \rightarrow) \circ\ ^*\!\leftarrow)$ and *uniformly confluent* iff $(\leftarrow \circ \rightarrow) \ \subseteq \ (\equiv \cup (\rightarrow \circ \leftarrow))$.

Proposition 5. *Uniform confluence implies strong confluence which implies confluence. Confluence and Church-Rosser property are equivalent.*

We define some further notions with respect to a given calculus. An expression E is *irreducible* iff there is no expression F with $E \to F$. A *derivation* is a finite sequence $(E_i)_{i=0}^{n}$ or a infinite sequence $(E_i)_{i=0}^{\infty}$ with $E_i \to E_{i+1}$ for all $i \geq 0$. A *derivation on* E is a derivation $(E_i)_{i\geq 0}$ with $E \equiv E_0$. A derivation is called *maximal* iff it is infinite or if its last element is irreducible. Reduction on E *normalizes* iff there is a maximal finite derivation on E. Reduction on E *strongly normalizes* if all maximal derivations on E are finite.

Theorem 6. *If E is an expression of a uniformly confluent calculus then all derivations on E have the same length.*

The proof is based on an inductive argument similar to proving that strong confluence implies confluence [4, 2].

Corollary 7. *If E is an expression of a uniformly confluent calculus then normalization on E and strong normalization on E are equivalent.*

6 Decomposition of Equivalence Relations

We present a method for the decomposition of an equivalence relation defined as least fixpoint. The method is independent of the underlying set.

Theorem 8 (Decomposition). *On a given set, we assume a confluent binary relation $\to$ and two equivalence relations $\approx$ and $\approx_1$ such that:*

1. $\approx \subseteq (\approx_1 \cup \to \cup \leftarrow)^*$,
2. $(\leftarrow \circ \approx_1) \subseteq (\approx_1 \circ \leftarrow)$.

Then $\approx \subseteq (\to^* \circ \approx_1 \circ {}^*\leftarrow)$ holds.

For instance, the decomposition $(\approx_1 \cup \to \cup \leftarrow)^* = (\to^* \circ \approx_1 \circ {}^*\leftarrow)$ holds under the assumption 2 of the theorem. For the proof we need the following statement that is not difficult to establish.

Lemma 9. *If $\to_1$ and $\to_2$ are transitive and reflexive relations, then we get:*

$$(\to_1 \cup \to_2)^* = \bigcup_{n\geq 3} \underbrace{\to_2 \circ \to_1 \circ \to_2 \circ \ldots \circ \to_2}_{n} .$$

The lemma states that the transitive reflexive closure is spawned by some of all possible compositions of $\to_1$ and $\to_2$ In particular, the length n of the composition is odd and greater or equal than 3.

Proof of the Decomposition Theorem. We define $\approx_2 = (\rightarrow \cup \leftarrow)^*$. Obviously, $(\approx_1 \cup \rightarrow \cup \leftarrow)^* = (\approx_1 \cup \approx_2)^*$ holds. Hence, Condition 1 is equivalent to $\approx \subseteq (\approx_1 \cup \approx_2)^*$. Applying Lemma 9 the theorem reduces to:

$$\underbrace{(\approx_2 \circ \approx_1 \circ \approx_2 \circ \ldots \circ \approx_2)}_{n} \subseteq (\rightarrow^* \circ \approx_1 \circ {}^*\leftarrow) \qquad \text{for all } n \geq 3 \tag{1}$$

This property can be shown by induction on n. For this purpose, we first formulate three simple properties. Confluence of $\rightarrow$ and Proposition 5 imply:

$$\approx_2 \subseteq (\rightarrow^* \circ {}^*\leftarrow) \tag{2}$$

A simple inductive argument applied to Condition 2 of the theorem yields:

$$({}^*\leftarrow \circ \approx_1) \subseteq (\approx_1 \circ {}^*\leftarrow) \tag{3}$$

Since $\approx_1$ is symmetric, Property (3) is equivalent to:

$$(\approx_1 \circ \rightarrow^*) \subseteq (\rightarrow^* \circ \approx_1) \tag{4}$$

Now, we are in position to prove Property (1). The case $n = 3$ works as follows:

$$\begin{array}{lll}
(\approx_2 \circ \approx_1 \circ \approx_2) & \subseteq (\approx_2 \circ \approx_1 \circ \rightarrow^* \circ {}^*\leftarrow) & \text{Property (2)} \\
& \subseteq (\approx_2 \circ \rightarrow^* \circ \approx_1 \circ {}^*\leftarrow) & \text{Property (4)} \\
& \subseteq (\approx_2 \circ \approx_1 \circ {}^*\leftarrow) & \text{definition of } \approx_2 \\
& \subseteq (\rightarrow^* \circ {}^*\leftarrow \circ \approx_1 \circ {}^*\leftarrow) & \text{Property (2)} \\
& \subseteq (\rightarrow^* \circ \approx_1 \circ {}^*\leftarrow \circ {}^*\leftarrow) & \text{Property (3)} \\
& \subseteq (\rightarrow^* \circ \approx_1 \circ {}^*\leftarrow) &
\end{array}$$

Next we consider the case $n \geq 4$ which implies $n \geq 5$ automatically:

$$\begin{array}{lll}
(\approx_2 \circ \approx_1 \circ \approx_2 \circ \ldots \circ \approx_2) & & \\
\quad \subseteq (\approx_2 \circ \approx_1 \circ \rightarrow^* \circ \approx_1 \circ {}^*\leftarrow) & & \text{induction hypothesis} \\
\quad \subseteq (\approx_2 \circ \rightarrow^* \circ \approx_1 \circ \approx_1 \circ {}^*\leftarrow \circ {}^*\leftarrow) & & \text{property (4)} \\
\quad \subseteq (\approx_2 \circ \approx_1 \circ \approx_2) & & \text{definition of } \approx_2 \\
\quad \subseteq (\rightarrow^* \circ \approx_1 \circ {}^*\leftarrow) & & \text{case } n = 3
\end{array}$$

7 Proving Uniform Confluence

Given a ρ-expression E we have to show how to join all F with $E \rightarrow F$ by means of reduction. But it is not obvious at all how to describe all those F in a finite manner. This problem comes with the syntactical flexibility provided by the structural congruence.

The idea of the proof is that all possible reductions may be performed on standardized expressions. These are unfailed, α-standardized expressions in prenex normal form and with separated constraints. We show how to reformulate congruence and reduction for standardized expressions. The reformulated versions are much simpler than the original ones with respect to the following aspects:

1. Quantifiers are not longer free to move into compositions.
2. Constraints are separated from abstractions, applications, and conditionals.
3. Reduction applies on the top-level of expressions and not below composition.

Standardization is performed by the following program: First, we circumvent failure. Next, we compute α-standardized prenex normal forms (PNF). In the following step, the structural congruence on PNFs is decomposed into the congruence over (ACI) and (Ex) and a directed relation corresponding to (α), (Mob), (Equ), and $(Repl)$. For decomposition we apply Theorem 8. Its assumptions require that the directed relation commutes with the remaining congruence. This statement would fail when choosing another decomposition treating some of the axioms (α), (Mob), (Equ), and $(Repl)$ independently. We can get rid of the directed relation, since it commutes with application of reduction axioms.

The remaining congruence is defined by (ACI) and (Ex). Reduction on PNFs with respect to the remaining congruence amounts to multiset rewriting. Hence, it is easy to describe and join all possible reductions on PNFs.

7.1 Congruences, Reductions and Other Relations

Let R be a binary relation on expressions. $\bar{\to}_R$ is the least relation containing R and closed under declaration and composition. $\Rightarrow_R$ is the least relation containing R and satisfying:

$$\frac{E_1 \;\Rightarrow_R\; E_2}{a{:}\overline{y}\,/E_1 \;\Rightarrow_R\; a{:}\overline{y}\,/E_2} \qquad \frac{E_1 \;\Rightarrow_R\; E_2}{\mathsf{if}\ \phi\ \mathsf{then}\ E_1\ \mathsf{else}\ F\ \mathsf{fi} \;\Rightarrow_R\; \mathsf{if}\ \phi\ \mathsf{then}\ E_2\ \mathsf{else}\ F\ \mathsf{fi}}$$

$$\frac{F_1 \;\Rightarrow_R\; F_2}{\mathsf{if}\ \phi\ \mathsf{then}\ E\ \mathsf{else}\ F_1\ \mathsf{fi} \;\Rightarrow_R\; \mathsf{if}\ \phi\ \mathsf{then}\ E\ \mathsf{else}\ F_2\ \mathsf{fi}}$$

$\equiv_R$ is the least congruence containing R.

All notations introduced above apply to axioms, since axioms may be identified with binary relations. For example, the axiom $(Refl)$ $E \equiv E$ corresponds to the binary relation $\{(E,E) \mid E \text{ is an expression}\}$. Hence, the following notations are defined:

$$\bar{\to}_{Mob}\,, \qquad \equiv_{\alpha}\,, \qquad \Rightarrow_{Equ}\ .$$

In analogy, sets of axioms can be considered as binary relations. This allows us to define the relation $\bar{\to}$ by $\bar{\to} = \bar{\to}_{\{Appl,Then,Else\}}$ where (Bot) is omitted.

7.2 Failure

Proposition 10. *Failure is preserved by structural congruence and reduction.*

Proof. The first statement can be proven using the following characterization of failed expressions: An expression E is unfailed iff the constraint is satisfiable which is obtained from E by replacing abstractions, applications, and conditionals with $\top$. We omit the details.

Proposition 11. *The ρ-calculus is uniformly confluent iff for all unfailed ρ-expressions E_1, E_2 and all F_1, F_2 such that $F_1 \overset{-}{\leftarrow} E_1 \equiv E_2 \overset{-}{\rightarrow} F_2$ either $F_1 \equiv F_2$ holds or there exists G with $F_1 \rightarrow G \leftarrow F_2$.*

Proof. Let E be a failed ρ-expression such that $F_1 \leftarrow E \rightarrow F_2$. Using Proposition 10, F_1 and F_2 are failed. Thus $F_1 \rightarrow \bot \leftarrow F_2$ holds.

7.3 Structure of the Proof

The structure of the proof is explained by Figure 5. This visualization is intended as a map of the proof. The proof is a travel starting in the north with $F_1 \overset{-}{\leftarrow} E_1 \equiv E_2 \overset{-}{\rightarrow} F_2$ and leading to G in the south. The expressions E_1 and E_2 are assumed to be unfailed and admissible. At a first glance, the reader should not be worried about undefined symbols. They will be explained on need.

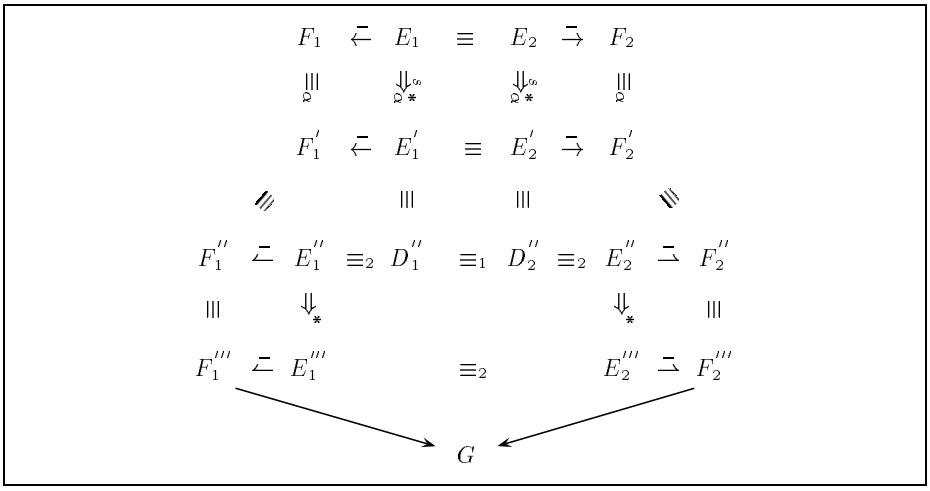

Fig. 5. Structure of the Proof

7.4 α-Standardization

An expression is *α-standardized* if none of its references is bound more than once and if there is no bound reference having a free occurrence. We define $E \overset{s}{\Rightarrow}_\alpha F$ iff F is obtained from E by replacing a bound reference in E with a reference not occurring in E.

Proposition 12. *For every E and every finite set of references Ref there exists an α-standardized F with $E \overset{s}{\Rightarrow}_\alpha^* F$ and $\mathcal{B}(F) \cap Ref = \emptyset$.*

This can be shown by induction on the structure of expressions. The proof of the next proposition uses the term rewriting techniques of [8].

Proposition 13. *If E is α-standardized and $E\,({}_{\alpha}\overset{s}{\Leftarrow} \circ \overset{-}{\rightarrow})\,F$ then $E\,(\overset{-}{\rightarrow} \circ \equiv_{\alpha})\,F$.*

7.5 Computing Prenex Normal Forms (PNFs)

PNFs are *extended expressions* using a noncommutative composition operator &. The definition of PNFs is mutual recursive with the definitions of two other forms of extended expressions, *molecules*, and *chemical solutions*:

$B ::= a{:}\,\overline{y}\,/D \mid u\,\overline{v} \mid \mathsf{if}\ \phi\ \mathsf{then}\ D_1\ \mathsf{else}\ D_2\ \mathsf{fi}$	molecules
$C ::= \top \mid B \mid C_1 \wedge C_2$	chemical solutions
$D ::= \phi\ \&\ C \mid \exists u\ D$	PNFs

Extended expressions can be considered as expressions by just replacing & by $\wedge$. Hence, relations defined on expressions carry over. Furthermore, it makes sense to overload the meta-variables E and F in order to denote extended expressions, whenever the distinction between & and $\wedge$ does not matter.

The rules in Figure 6 provide for computation of PNFs of α-standardized expressions. We use P and Q for quantifier prefixes $\exists u_1 \ldots \exists u_n$ with $n \geq 0$.

$$\mathcal{S}(\phi,\ \phi) \qquad \mathcal{S}(u\,\overline{v},\ \top\ \&\ u\,\overline{v}) \qquad \frac{\mathcal{S}(E,\ F)}{\mathcal{S}(a{:}\,\overline{y}\,/E,\ \top\ \&\ a{:}\,\overline{y}\,/F)}$$

$$\frac{\mathcal{S}(E,\ F)}{\mathcal{S}(\exists u\ E,\ \exists u\ F)} \qquad \frac{\mathcal{S}(E_1,\ P_1\,(\phi_1\ \&\ F_1)) \qquad \mathcal{S}(E_2,\ P_2\,(\phi_2\ \&\ F_2))}{\mathcal{S}(E_1 \wedge E_2,\ P_1P_2\,(\phi_2 \wedge \phi_2\ \&\ F_1 \wedge F_2))}$$

$$\frac{\mathcal{S}(E_1,\ F_1) \qquad \mathcal{S}(E_2,\ F_2)}{\mathcal{S}(\mathsf{if}\ \phi\ \mathsf{then}\ E_1\ \mathsf{else}\ E_2\ \mathsf{fi},\ \top\ \&\ \mathsf{if}\ \phi\ \mathsf{then}\ F_1\ \mathsf{else}\ F_2\ \mathsf{fi})}$$

Fig. 6. Computation of Prenex Normal Forms

Proposition 14. *If $\mathcal{S}(E,\ F)$ then F is a PNF. For all E there is (a not necessarily unique) F with $\mathcal{S}(E,\ F)$. If E is α-standardized then $E \equiv F$. Furthermore, F is α-standardized, $\mathcal{B}(E) = \mathcal{B}(F)$, and $\mathcal{F}(E) = \mathcal{F}(F)$.*

The proofs are straightforward by induction.

7.6 Congruence and Reduction for PNFs

The congruence $\equiv_1$ is the least congruence on PNFs satisfying the axioms in Figure 8. It is the appropriate counterpart of $\equiv$ when restricting to PNFs.

(α_1)	capture free renaming of bound references
(ACI_1)	$\wedge$ restricted to chemical solutions is associative and commutative, and satisfies $C \wedge \top \equiv_1 C$
(Ex_1)	$\exists u \exists v\ D \ \equiv_1\ \exists v \exists u\ D$
(Mob_1)	$\exists x\ (\phi\ \&\ C) \ \equiv_1\ (\exists x\ \phi)\ \&\ C$ if $x \notin \mathcal{F}(C)$
(Equ_1)	$\phi \equiv_1 \psi$ if $\Delta \models \phi \leftrightarrow \psi$
$(Repl_1)$	$\phi\ \&\ C \ \equiv_1\ \phi\ \&\ C[u/x]$ if $\Delta \models \phi \rightarrow x = u$

Fig. 7. Axioms of $\equiv_1$

Proposition 15. *If E_1 and E_2 are α-standardized, $E_1 \equiv E_2$, $\mathcal{S}(E_1, F_1)$, and $\mathcal{S}(E_2, F_2)$ then $F_1 \equiv_1 F_2$.*

The proof of this proposition is tricky and omitted due to lack of space.

The appropriate reduction on PNFs $\stackrel{-}{\rightharpoonup}$ is the least relation containing the axioms in Figure 8 (but not any rule).

$(Appl_t)$	$Q\ (\phi\ \&\ a{:}\overline{y}\ /E \wedge a\,\overline{u}\ \wedge F) \ \stackrel{-}{\rightharpoonup}\ Q\ (\phi \wedge a{:}\overline{y}\ /E \wedge E[\overline{u}/\overline{y}] \wedge F)$
$(Then_t)$	$Q\ (\phi\ \&$ if ψ then E_1 else E_2 fi $\wedge F) \ \stackrel{-}{\rightharpoonup}\ Q\ (\phi \wedge E_1 \wedge F)$ if $\Delta \models \phi \rightarrow \psi$
$(Else_t)$	$Q\ (\phi\ \&$ if ψ then E_1 else E_2 fi $\wedge F) \ \stackrel{-}{\rightharpoonup}\ Q\ (\phi \wedge E_2 \wedge F)$ if $\Delta \models \phi \rightarrow \neg\psi$

Fig. 8. Reduction on PNFs

We define $\equiv_2 = \equiv_{\{ACI_1, Ex_1\}}$. It has the nice property that it preserves α-standardization. Furthermore, the following proposition can be proved along the lines of [8].

Proposition 16. *The conditions $E_1 \stackrel{-}{\rightarrow} F$, $\mathcal{S}(E_1, E_2)$, and E_1 α-standardized imply $E_2(\equiv_2 \circ \stackrel{-}{\rightharpoonup} \circ \equiv)F$.*

7.7 Decomposition of $\equiv_1$

We want to decompose $\equiv_1$ into $\equiv_2$ and a directed relation corresponding to the set of axiom $\{\alpha_1, Mob_1, Equ_1, Repl_1\}$ by applying the Decomposition Theorem 8. The definition of the directed relation is based on the axioms in Figure 9.

For an arbitrary relation R on expressions we define $\stackrel{r}{\Rightarrow}_R$ to be the restriction of $\Rightarrow_R$ to α-standardized expressions. We define the directed relation $\Rightarrow$ by

$$\Rightarrow\ =\ (\stackrel{s}{\Rightarrow}_{\alpha_1} \cup \stackrel{r}{\Rightarrow}_{\{Mob_2, Equ_2, Repl_2\}})$$

(Mob_2) $P\exists xQ\ (\phi\ \&\ C)$	$\equiv_1$ $PQ\ (\exists x\ \phi\ \&\ C)$	if $x \notin \mathcal{F}(C)$
$(Repl_2)$ $Q\ (\phi\ \&\ C)$	$\equiv_1$ $Q\ (\phi\ \&\ C[u/x])$	if $\Delta \models \phi \rightarrow x = u$ and u is a name or $u \in \mathcal{F}(C)$ or $x \in \mathcal{B}(C)$
(Equ_2) $Q\ (\phi\ \&\ C)$	$\equiv_1$ $Q\ (\psi\ \&\ C)$	if $\Delta \models \psi \leftrightarrow \phi$, $\mathcal{F}(\psi) \subseteq \mathcal{F}(\phi)$, and $\mathcal{B}(\psi) \subseteq \mathcal{B}(\phi)$,

Fig. 9. Axioms needed for Directed Relation

Proposition 17. $(\Leftarrow \circ \equiv_2)\ \subseteq\ (\equiv_2 \circ \Leftarrow)$ *holds and* $\Rightarrow$ *is confluent.*

The proofs rely on the rewriting techniques of [8].

Corollary 18 (Decomposition). $\equiv_1\ \subseteq\ (\Rightarrow^* \circ \equiv_2 \circ\ ^*\Leftarrow)$.

Proof. We apply the Decomposition Theorem 8 instantiated with $\approx = \equiv_1$, $\approx_1 = \equiv_2$ and $\rightarrow = \Rightarrow$ The application conditions hold due to Lemma 17.

Proposition 19. $(^*\Leftarrow \circ \overset{-}{\rightharpoonup})\ \subseteq\ (\overset{-}{\rightharpoonup} \circ \equiv)$.

The proof again uses the rewriting techniques of [8].

7.8 The Final Case Distinction

Proposition 20. *Let* E_1 *and* E_2 *be* α*-standardized and admissible PNFs such that* $F_1 \overset{-}{\leftharpoonup} E_1 \equiv_2 E_2 \overset{-}{\rightharpoonup} F_2$. *Then* $F_1 \equiv F_2$ *or there is* G *with* $F_1 \rightarrow G \leftarrow F_2$.

Proof of Uniform Confluence. Applying Proposition 11, we have to join F_1 and F_2 assuming $F_1 \overset{-}{\leftarrow} E_1 \equiv E_2 \overset{-}{\rightarrow} F_2$ for unfailed ρ-expressions E_1 and E_2. We can assume $F_1 \not\equiv F_2$ without loss of generality. In the sequel, all statements hold for $i = 1$ and $i = 2$. Proposition 12 yields the existence of $E_i^{'}$ with $E_i \overset{s}{\Rightarrow}^*_{\alpha} E_i^{'}$. Applying Proposition 13 we get $E_i^{'} \overset{-}{\rightarrow} F_i^{'} \equiv_\alpha F_i$ for some $F_i^{'}$. By Proposition 14, there exists α-standardized PNFs $C_i^{''}$ with $\mathcal{S}(E_i^{'}, C_i^{''})$ and $E_i^{'} \equiv D_i^{''}$. Proposition 15 ensures $D_1^{''} \equiv_1 D_2^{''}$ and Proposition 16 implies $D_i^{''} \equiv_2 E_i^{''} \overset{-}{\rightharpoonup} F_i^{''} \equiv F_i^{'}$ for some $E_i^{''}$, $F_i^{''}$. $E_i^{''}$ is α-standardized and $E_1{''} \equiv_1 E_2^{''}$. By the Decomposition of $\equiv_1$ (Corollary 18) we get $E_1^{''} \Rightarrow^* E_1^{'''} \equiv_2 E_2^{'''}\ ^*\Leftarrow E_2^{'''}$. Proposition 19 yields $E_i^{'''} \overset{-}{\rightharpoonup} F_i^{'''} \equiv F_i^{''}$ for some $F_i^{'''}$. The final case distinction (Proposition 20) and $F_1^{'''} \not\equiv F_2^{'''}$ imply the existence of G with $F_1^{'''} \rightarrow G \leftarrow F_2^{'''}$. All together, this proves $F_1 \rightarrow G \leftarrow F_2$. □

8 Conclusion

Relational calculi provide for appropriate models of higher-order, concurrent, constraint programming. They cover important aspects of computation and have a rich mathematical theory. We have presented powerful methods solving some of the technical challenges when giving up syntactical position in favor of naming.

Acknowledgement The authors are grateful to Martin Müller, Tobias Müller and Christian Schulte for many suggestions and help.

References

1. Paul Barth, S. Nikhil Rishiyur, and Arvind. M-Structures: Extending a Parallel, Non-strict, Functional Language with State. In J. Hughes, editor, *Functional Programming Languages and Computer Architecture - 5th ACM Conference*, number 523 in LNCS, pages 538–568. Springer Verlag, August 1991.
2. N. Dershowitz and J.-P. Jouannaud. *Rewrite Systems*, volume B, chapter 6, pages 243–320. MIT Press, Cambridge, Massachusetts, 1990. Handbook of Theoretical Computer Science.
3. M. Henz, M. Mehl, M. Müller, T. Müller, J. Niehren, R. Scheidhauer, C. Schulte, G. Smolka, R. Treinen, and J. Würtz. The Oz Handbook. Research Report RR-94-09, DFKI, 1994.
4. Gérard Huet. Confluent Reductions: Abstract Properties and Applications to Term Rewriting Systems. *Journal of the ACM*, 27(4):797–821, October 1980.
5. John Launchbury. A Natural Semantics for Lazy Evaluation. In *Proceedings of 20th POPL*, pages 144–154. ACM, 1993.
6. Jean-Jaques Levy, Bent Thomsen, Lone Leth, and Alessandro Giacalone. Concurrency and Functions: Evaluation and Reduction. In *EATOS*, pages 88–106. ESPRIT, nov 1992. Esprit Basic Research Action 6454-CONFER.
7. Martin Müller and Joachim Niehren. Higher-Order Meta Programming in Oz. unpublished, May 1994.
8. Joachim Niehren and Gert Smolka. Functional Computation in a Calculus of Relational Abstraction and Application. Research Report RR-94-04, DFKI, March 1994.
9. Martin Odersky. A Functional Theory of Local Names. In *POPL*, pages 48–59, January 1994.
10. Andrew Pitts and Ian Stark. On the Observable Properties of Higher Order Functions that Dynamically Create Local Names. In *Proceedings of the ACM SIGPLAN Workshop on State in Programming Languages*, pages 31–45, June 1993.
11. Christian Schulte and Gert Smolka . Encapsulated Search in Higher-Order Concurrent Constraint Programming. Research report, DFKI, April 1994. to appear.
12. Gert Smolka. A Calculus for Higher-Order Concurrent Constraint Programming with Deep Guards. Research Report RR-94-03, DFKI, February 1994.
13. Gert Smolka. A Foundation for Concurrent Constraint Programming. In *CCL*, 1994. Invited Talk.
14. Gert Smolka, Martin Henz, and Jörg Würtz. Object-Oriented Concurrent Constraint Programming in Oz. Research Report RR-93-16, DFKI, April 1993.

The papers of the programming systems lab at DFKI are available via anonymous ftp from *tt*ps-ftp.dfki.uni-sb.de and via www from *tt*http://ps-www.dfki.uni-sb.de/.

This article was processed using the LaTeX macro package with LLNCS style