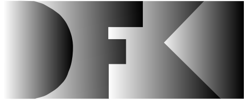


Deutsches
Forschungszentrum
für Künstliche
Intelligenz GmbH

Research Report
RR-94-16

A Foundation for Higher-order Concurrent Constraint Programming

Gert Smolka

June 1994

Deutsches Forschungszentrum für Künstliche Intelligenz GmbH

Postfach 20 80
67608 Kaiserslautern, FRG
Tel.: + 49 (631) 205-3211
Fax: + 49 (631) 205-3210

Stuhlsatzenhausweg 3
66123 Saarbrücken, FRG
Tel.: + 49 (681) 302-5252
Fax: + 49 (681) 302-5341

This work appears in the Proceedings of the *1st International Conference on Constraints in Computational Logics*, edited by Jean-Pierre Jouannaud, Springer LNCS, September 7–9, 1994, München, Germany.

This work has been supported by the Bundesminister für Forschung und Technologie (contract ITW 9105), the Esprit Basic Research Project ACCLAIM (contract EP 7195), and the Esprit Working Group CCL (contract EP 6028).

© Deutsches Forschungszentrum für Künstliche Intelligenz 1994

This work may not be copied or reproduced in whole of part for any commercial purpose. Permission to copy in whole or part without payment of fee is granted for nonprofit educational and research purposes provided that all such whole or partial copies include the following: a notice that such copying is by permission of the Deutsche Forschungszentrum für Künstliche Intelligenz, Kaiserslautern, Federal Republic of Germany; an acknowledgement of the authors and individual contributors to the work; all applicable portions of this copyright notice. Copying, reproducing, or republishing for any other purpose shall require a licence with payment of fee to Deutsches Forschungszentrum für Künstliche Intelligenz.

A Foundation for Higher-order Concurrent Constraint Programming

Gert Smolka
Programming Systems Lab
German Research Center for Artificial Intelligence (DFKI)
Stuhlsatzenhausweg 3, 66123 Saarbrücken, Germany
email: `smolka@dfki.uni-sb.de`

June 9, 1994

Abstract

We present the γ-calculus, a computational calculus for higher-order concurrent programming. The calculus can elegantly express higher-order functions (both eager and lazy) and concurrent objects with encapsulated state and multiple inheritance. The primitives of the γ-calculus are logic variables, names, procedural abstraction, and cells. Cells provide a notion of state that is fully compatible with concurrency and constraints. Although it does not have a dedicated communication primitive, the γ-calculus can elegantly express one-to-many and many-to-one communication.

There is an interesting relationship between the γ-calculus and the π-calculus: The γ-calculus is subsumed by a calculus obtained by extending the asynchronous and polyadic π-calculus with logic variables.

The γ-calculus can be extended with primitives providing for constraint-based problem solving in the style of logic programming. A such extended γ-calculus has the remarkable property that it combines first-order constraints with higher-order programming.

Contents

1 Introduction **3**

2 The Gamma Calculus **4**

3 The Chemical Metaphor **6**

4 Creating Fresh Names **8**

5 Possible Indeterminisms **9**

6 Embedding of the Eager Lambda Calculus **10**

7 Embedding of the Lazy Lambda Calculus **11**

8 Records **12**

9 Procedures with Encapsulated State **13**

10 Objects **14**

11 Communication **17**

12 An Execution Strategy **18**

13 First-order Constraints and Search **19**

13.1 Constraints 20
13.2 Conditionals 21
13.3 Disjunctions 21
13.4 Failure 21
13.5 Search 22
13.6 Higher-order Programming and First-order Constraints 22

14 Relationship with the π-calculus **22**

15 Future Research **24**

1 Introduction

Concurrent constraint programming [22] is a research direction aiming at a unified framework for high-level concurrent programming and constraint-based problem solving. Its roots are concurrent logic programming [24] and constraint logic programming [3, 10]. Although concurrent programming and constraint-based problem solving have different structure and applications, they do have significant commonalities:

- both come in a relational and concurrent setting
- constraint propagation is a concurrent activity
- logic variables are the canonical form of reference for constraints and concurrent computation.

This paper presents the γ-calculus, a computational calculus for higher-order concurrent programming. As is, the calculus can elegantly express higher-order functions (both eager and lazy) and concurrent objects with encapsulated state and multiple inheritance. Constraint-based problem solving in the style of logic programming requires additional primitives, which can be chosen such that one obtains a combination of higher-order programming with first-order constraints. This is in sharp contrast to approaches based on higher-order logic [18], where higher-order programming comes with the operational burden of higher-order constraints.

An extension [25, 23] of the γ-calculus providing for constraint-based problem solving serves as the foundation of Oz [8], a full-fledged programming language and system under development at the Programming Systems Lab of DFKI.[1]

The primitives of the γ-calculus are logic variables, names, procedural abstraction, and cells. Cells provide a notion of state that is fully compatible with concurrency and constraints. Although it does not have a dedicated communication primitive, the γ-calculus can elegantly express one-to-many and many-to-one communication.

It is illuminating to compare the γ-calculus with the π-calculus [17, 16, 15]. Both are concurrent systems with first-class names. While the γ-calculus has logic variables, the π-calculus has formal input arguments only (as in functional programming). As is well-known from logic programming, logic variables do not necessitate a static distinction between input and output, thus providing for a free data flow combining smoothly with concurrent control. While the π-calculus has communication as its principal primitive, the γ-calculus has logic variables, procedural abstraction, and cells as its principal primitives. The primitives of the γ-calculus were chosen with the consideration that programming abstractions such as higher-order functions and concurrent objects be easily expressible. If we extend the π-calculus with logic variables, it can express procedural abstraction and cells. Logic

[1] The Oz programming system and its documentation are available through anonymous ftp from `ps-ftp.dfki.uni-sb.de` or through WWW from `http://ps-www.dfki.uni-sb.de/`.

variables increase the expressivity of the π-calculus in two crucial aspects: They allow to equate communication links, and they provide the possibility to express procedures with input *and* output arguments (recall that a function is a procedure with input and output).

The paper is organized as follows. Section 2 gives the formal definition of the γ-calculus. Sections 3–5 provide important intuitions and examples for the expressivity of the γ-calculus. Sections 6 and 7 show how the eager and the lazy λ-calculus can be embedded into the γ-calculus. Section 8 shows how the γ-calculus can express records. Sections 9 and 10 show how the γ-calculus can express concurrent objects with encapsulated state and multiple inheritance. Section 11 discusses communication issues. Section 12 presents a possible execution strategy for the γ-calculus. Section 13 shows how the γ-calculus can be extended with general first-order constraints. Section 14 clarifies the relationship between the γ-calculus and the π-calculus.

2 The Gamma Calculus

Figure 1 shows the syntax of the γ-calculus. It assumes that an infinite alphabet of *variables* and a disjoint and infinite alphabet of *names* are given. Variables and names are jointly referred to as *references*. Variables are placeholders for names. There are no other values but names.

The expressions of the γ-calculus are relational, as in logic programming or the π-calculus. Seen from the perspective of predicate logic, expressions play the role of formulas and references play the role of terms. Composition is like conjunction in logic programming and parallel composition in the π-calculus. A declaration $\exists uE$ introduces a new reference u with scope E. Declaration of variables is like existential quantification in logic programming; declaration of names is like restriction in the π-calculus. Equations are like equations in logic. Names stand for themselves and thus are different if they are syntactically different (so-called unique name assumption). A (named) abstraction $a{:}\,\overline{x}/E$ consists of a name a, formal arguments $\overline{x}$ ($\overline{x}$ stands for a possibly empty sequence of variables), and a body E (the expression being abstracted from). There is the side condition that the sequence of formal arguments $\overline{x}$ be linear (i.e., consist of pairwise distinct variables). Abstractions can be seen as procedure or predicate definitions. An application $u\overline{v}$ consists of a reference u designating the abstraction to be applied and the actual arguments $\overline{v}$. Applications can be seen as procedure or predicate calls. A conditional **if** $u = v$ **then** E **else** F reduces to either E or F, depending on whether u and v turn out to be equal or different. A cell $a{:}u$ has the name a and holds the reference u; reduction with an application avw will impose the equation $u = v$ and update the cell to hold w.

From the above it is clear that the γ-calculus has one binder for names ($\exists aE$) and two binders for variables ($\exists xE$ and $a{:}\,\overline{x}/E$). *Free and bound references* of expressions are defined accordingly.

The γ-calculus is an expressive computational system. We will show that it can elegantly

Symbols

x, y, z	:		variables	
a, b, c	:		names	
u, v, w	$::=$	$x \mid a$	references	

Expressions

E, F, G	$::=$	$\top$	null	
	$\mid$	$E \wedge F$	composition	
	$\mid$	$\exists u E$	declaration	
	$\mid$	$u = v$	equation	
	$\mid$	$a{:}\,\overline{x}/E$	abstraction	($\overline{x}$ linear)
	$\mid$	$u\overline{v}$	application	
	$\mid$	**if** $u = v$ **then** E **else** F	conditional	
	$\mid$	$a{:}u$	cell	

Figure 1: Syntax of the γ-calculus

express higher-order functional programming, data structures, and concurrent objects with encapsulated state and multiple inheritance.

A distinctive feature the γ-calculus shares with logic programming is that variables can be used without explicitly saying how their values are obtained (so-called logic variables). Information about the values of variables can be stated through equations, which can be seen as constraints. Equations can express partial (e.g, $x = y$) and total (e.g., $x = a$) information. Recall that names are the only values variables can take in the γ-calculus.

The computational intuitions expressed above are formalized by rules rewriting the expressions of the calculus. This is a common setup also found in the λ-calculus (functional computation) and SLD-resolution (relational computation). For the γ-calculus, this setup is refined in that the rules are applied modulo a structural congruence, and in that the rules can only be applied to specific positions.[2]

Applying rewrite rules modulo a structural congruence is actually quite common, although it is often not made explicit. In the λ-calculus, it is common practise to "identify" expressions that are equal up to α-conversion (consistent renaming of bound variables). In logic programming and unification, one typically rewrites multisets of atomic formulas, where

[2] A similar setup is used in a recent presentation of the π-calculus devised by Milner [16, 15].

the multisets are obtained by making conjunction associative and commutative. First-order rewriting modulo equations is an established topic [5], serving as a foundation for the specification language OBJ [6] and Meseguer's rewriting logic [14].

The *structural congruence* of the γ-calculus is the least congruence "$E \equiv F$" on the set of expressions satisfying the following laws:

- composition $E \wedge F$ of expressions is associative, commutative, and satisfies $E \wedge \top \equiv E$ (thus we can see composition as multiset union and $\top$ as the empty multiset)
- declaration $\exists u E$ of references allows for consistent renaming of the declared reference u and satisfies

$$\begin{aligned} \exists u E \wedge F &\equiv \exists u(E \wedge F) \qquad \text{if } u \text{ not free in } F \\ \exists u \exists v E &\equiv \exists v \exists u E \\ \exists u \top &\equiv \top \end{aligned}$$

  (thus declarations can always be moved above compositions, and declarations of references not being used can be deleted)
- abstractions $a{:}\,\overline{x}/E$ allow for consistent renaming of the formal arguments $\overline{x}$
- equations $u = v$ are symmetric.

Reduction in the γ-calculus is defined in Figure 2 by a system of inference rules. Only the structure rules have premises, all other rules are axioms. The structure rules say that reduction is modulo structural congruence, and that reductions of subexpressions not appearing beneath abstractions and conditionals can be taken as reductions of the entire expression. A reduction $E \to F$ is possible if and only if it can be derived with the structure rules from exactly one instance of an axiom. The Application Rule comes with the side condition that the number $|\overline{u}|$ of actual arguments in the application equals the number $|\overline{x}|$ of formal arguments in the abstraction.

Proposition 1 *Let* contexts *be defined as* $C ::= \bullet \mid C \wedge E \mid E \wedge C \mid \exists u C$. *Then* $E \to E'$ *is a reduction in the* γ*-calculus if and only if there exists a context* C *and an instance* $G \to G'$ *of an axiom in Figure 2 such that* $E \equiv C[G]$, *and* $C[G'] \equiv E'$.

3 The Chemical Metaphor

Reduction in the γ-calculus can be seen as evolution of a *computation space* containing a multiset of freely floating *molecules*.[3] The molecules are equations, abstractions, applications, conditionals, and cells. The structural congruence of the γ-calculus is defined such

[3] The metaphor of seeing concurrent computation as chemical reaction appeared with Berry and Boudol's chemical abstract machine [4].

Structure

$$\frac{E \equiv E' \quad E' \to F' \quad F' \equiv F}{E \to F} \qquad \frac{E \to E'}{E \wedge F \to E' \wedge F} \qquad \frac{E \to E'}{\exists u E \to \exists u E'}$$

Elimination

$$\exists x(x = u \ \wedge \ E) \ \to \ E[u/x] \qquad \text{if } x \neq u \text{ and } u \text{ free for } x \text{ in } E$$

Application

$$a\overline{u} \ \wedge \ a{:}\,\overline{x}/E \ \to \ E[\overline{u}/\overline{x}] \ \wedge \ a{:}\,\overline{x}/E \qquad \text{if } \overline{u} \text{ free for } \overline{x} \text{ in } E \text{ and } |\overline{u}| = |\overline{x}|$$

Conditional

$$\textbf{if } u = u \textbf{ then } E \textbf{ else } F \ \to \ E \qquad \textbf{if } a = b \textbf{ then } E \textbf{ else } F \ \to \ F \qquad \text{if } a \neq b$$

Exchange

$$a{:}u \ \wedge \ avw \ \to \ a{:}w \ \wedge \ v = u$$

Figure 2: Reduction in the γ-calculus

that every expression can be seen as a computation space: After pushing all declarations to the top (possibly involving α-conversion), we are left with a conjunction of molecules. The expression $\top$ describes the empty computation space. Expressions appearing as the constituents of abstractions and conditionals do not yet contribute to the computation space.

A computation space evolves by reduction with the rules given in Figure 2. The Application and Exchange Rules describe reactions between two molecules sharing a name. The rules for the conditional describe transformations of a single molecule. The Elimination Rule deletes an equational molecule and eliminates a variable by replacing all its occurrences with another reference.

When a conditional reduces, it injects one of its constituent expressions into the computation space, thus possibly contributing new molecules and new references (the operational reading of a declaration $\exists uE$ is: Create a new reference u). Similarly, when an application reacts with an abstraction, a copy of the body of the abstraction is injected into the computation space, where the actual arguments of the application replace the formal arguments of the abstraction. The Application Rule is the only rule that copies expressions. As the space evolves, the number of molecules and the number of connecting references can increase and decrease. Every infinite reduction chain $E_1 \to E_2 \to \ldots$ must involve the Application Rule.

The Elimination Rule provides all the constraint handling needed in the γ-calculus. If a computation space contains a molecule $x = u$, then x can be eliminated by replacing it

with u, provided u is different from x. We assume that a computation space does not have free variables. Injecting an equation $x = a$ into a computation space amounts to an attempt to fix the value of the variable x to a. There might be competing such attempts, as in

$$\exists x(x = a \ \wedge \ x = b \ \wedge \ E).$$

Which value is taken for x is an indeterministic choice: The space can either reduce to $a = b \wedge E[a/x]$ or to $a = b \wedge E[b/x]$, where the choice being made cannot be retracted. Note that all occurrences of x will be replaced with only one of the two names. The fact that there were conflicting attempts to fix the value of x remains partly visible since the "inconsistent" equation $a = b$ remains in the space.[4] There are three possibilities to handle such a conflict: consider it a regular event (the choice taken in the γ-calculus), consider it a run-time error, or consider it a failure in the sense of logic programming (we will say more about failure in Section 13).

The expression (a is not free in E)

$$\exists a(\textbf{if } x = a \textbf{ then } \top \textbf{ else } E)$$

has an interesting operational reading: inject the expression E in the computation space once the variable x has been assigned a value (i.e., has been replaced by a name). Put more informally, the above expression synchronizes E upon the event that the value of x becomes known.

The Exchange Rule describes a reaction of a cell $a{:}u$ with an application avw. The reaction updates the reference hold by the cell to w and equates the references u and v (exploiting logic variables). Thus reading and writing of a cell are merged into one atomic operation. Cells yield a notion of state that is fully compatible with concurrency and constraints. Cells are essential for expressing objects.

The Application and the Exchange Rule have in common that they describe reactions between two molecules that agree on the same name (i.e., a). As computation proceeds, new abstractions and cells may be created. This necessitates the creation of fresh names, an operation elegantly expressible in the γ-calculus.

4 Creating Fresh Names

The operational reading of $\exists a(x = a)$ is: Create a fresh name and make it the value of the variable x. To see why this is so, consider the expression

$$\exists x \exists y(\exists a(x = a) \ \wedge \ \exists a(y = a) \ \wedge \ \textbf{if } x = y \textbf{ then } E \textbf{ else } F)$$

[4] Equations of the form $u = u$ and $a = b$ do not have a computational effect. Hence they can be deleted in an implementation of the γ-calculus.

and suppose that x and y are distinct variables that do not occur free in E and F. Moreover, assume that a is a name not occurring free in E and F. We will show that the expression reduces to F.

First, we move the left declaration of a to the outside of the expression using the laws for declarations and compositions and exploiting the assumption that a does not occur free in E and F.

$$\equiv\ \exists a \exists x \exists y(x = a\ \wedge\ \exists a(y = a)\ \wedge\ \textbf{if}\ x = y\ \textbf{then}\ E\ \textbf{else}\ F)$$

Next we exchange the declarations of x and y and eliminate x with the Elimination Rule.

$$\rightarrow\ \exists a \exists y(\exists a(y = a)\ \wedge\ \textbf{if}\ a = y\ \textbf{then}\ E\ \textbf{else}\ F)$$

Next we rename the inner name a to b, where b is assumed to be different and to not occur free in E and F.

$$\equiv\ \exists a \exists y(\exists b(y = b)\ \wedge\ \textbf{if}\ a = y\ \textbf{then}\ E\ \textbf{else}\ F)$$

This brings us in a position where we can eliminate y in the same way we did it for x before.

$$\rightarrow\ \exists a \exists b(\textbf{if}\ a = b\ \textbf{then}\ E\ \textbf{else}\ F)$$

Now, since a and b are different, we obtain

$$\rightarrow\ \exists a \exists b F$$

using the appropriate rule for the conditional. It remains to get rid of the declarations of the names a and b. This can be done using the congruence laws:

$$\equiv\ \exists a \exists b(\top \wedge F)\ \equiv\ (\exists a \exists b \top)\ \wedge\ F\ \equiv\ \top\ \wedge\ F\ \equiv\ F.$$

5 Possible Indeterminisms

The γ-calculus involves several indeterminisms:

1. if there are two applications for the same cell, the order of their reduction is indeterministic
2. if there are two equations $x = a$ and $x = b$ for the same variable, the choice of the name replacing x is indeterministic
3. if an application matches more than one abstraction or cell, the choice of the abstraction or cell it reacts with is indeterministic.

The first indeterminism is essential for concurrent computation (see the section on objects). The other indeterminisms should not occur with well-written programs.

The third form of indeterminism can be excluded with a straightforward syntactic condition: extend the γ-calculus with the syntactic variants

$$\begin{array}{rcl} x\colon \overline{y}/E & :\equiv & \exists a(x = a \wedge a\colon \overline{y}/E) \\ x\colon u & :\equiv & \exists a(x = a \wedge a\colon u) \end{array}$$

and admit only initial expressions not containing the primitive forms $a\colon \overline{y}/E$ and $a\colon u$. One can show that reduction sequences issuing from such expressions cannot involve the third form of indeterminism.

Provided one excludes cells, there is a syntactic condition excluding all remaining indeterminisms; a thus restricted version of the γ-calculus is the δ-calculus studied and proven confluent in [21]. The δ-calculus seems to be a promising alternative to the λ-calculus for the foundation of functional programming.

Remark. The syntactic extensions $x\colon \overline{y}/E$ and $x\colon u$ defined above are *static*; that is, they must be expanded *before* a reduction rule is applied. This is since $x\colon \overline{y}/E$ changes its meaning when the elimination rule replaces x with a name a.

6 Embedding of the Eager Lambda Calculus

To embed the eager λ-calculus (see [27]) into the γ-calculus, we extend the expressions of the γ-calculus such that one can write λ-terms in equations:

$$\begin{array}{rcl} E, F, G & ::= & \ldots \mid x = M \\ M, N & ::= & x \mid \lambda x M \mid MN. \end{array}$$

The semantics of the new equations is given by the congruences

$$\begin{array}{rcl} x = \lambda y M & :\equiv & x\colon yz/z = M \\ x = MN & :\equiv & \exists y \exists z(y = M \ \wedge \ z = N \ \wedge \ yzx) \end{array}$$

providing a translation from the extended syntax to the base syntax (the syntactic extension $x\colon yz/E$ was defined in the previous section). As one would expect, functional abstractions translate into relational abstractions with an input and an output argument. It is instructive to consider the translation of the identity function:

$$x = \lambda y y \quad \equiv \quad \exists a(x = a \ \wedge \ a\colon yz/z = y).$$

The translation of functional applications exploits that functional nesting can be expressed by composition and declaration of auxiliary variables.

The soundness of the embedding is established by the following theorem [19].

Theorem 2 *Let M be a closed λ-term. Then M converges in the eager λ-calculus if and only if $\exists x(x = M)$ converges in the γ-calculus.*

In contrast to the λ-calculus, the γ-calculus can express (mutual) recursion directly. For instance, $\exists x \exists y\,(x = \lambda u M \;\wedge\; y = \lambda v N \;\wedge\; E)$ defines two possibly mutually recursive functions x and y that can be used in E.

Eager functional programming with mutual recursion can in fact be expressed in a confluent subcalculus of the γ-calculus, called the δ-calculus [21].

7 Embedding of the Lazy Lambda Calculus

The embedding of the lazy λ-calculus (see [27]) into the γ-calculus is more subtle than the embedding of the eager λ-calculus. The basic idea is to represent a lazy function by an abstraction with three arguments: one argument for the input of the function, one argument for the output of the function, and one argument for requesting that the input of the function be computed.

In the following we will use r and s to denote variables used to request subcomputations. We extend the syntax of the γ-calculus as follows:

$$\begin{array}{rcl} E, F, G & ::= & \ldots \mid x{\cdot}r = K \\ K, L & ::= & x \mid \lambda x K \mid KL \mid x{\cdot}r. \end{array}$$

An equation $x{\cdot}r = K$ equates x to the result of the λ-term K, where evaluation of K must be requested explicitly through the variable r.

The semantics of the new expressions is given by the congruences

$$\begin{array}{rcl} x{\cdot}r = y & :\equiv & x = y \\ x{\cdot}r = y{\cdot}s & :\equiv & x = y \wedge r = s \\ x{\cdot}r = \lambda y K & :\equiv & x{:}\, ysz/z{\cdot}r = K[y{\cdot}s/y] \\ x{\cdot}r = KL & :\equiv & \exists y \exists y' \exists z \exists s\,(y{\cdot}r = K \;\wedge\; ryy' \;\wedge\; y'zsx \;\wedge\; z{\cdot}s = L) \end{array}$$

providing a translation from the extended syntax to the base syntax of the γ-calculus. The translation of an equation $x \cdot r = K$ will admit no other rule but the Elimination Rule, eliminating unnecessary auxiliary variables (e.g., the translation of $x \cdot r = y(z \cdot s)$ will reduce to $\exists y'(ryy' \wedge y'zsx)$). Evaluation of $x \cdot r = K$ must be requested explicitly by composing it with $r = \lambda xx$ (the "eager" equation $r = \lambda xx$ was defined in the previous section). Evaluation is made lazy by switching the connection between abstractions and applications only when the result of the application is needed. The switch is realized by an application ryy', which is fired by equating r to the identity function.

Concerning the correctness of the embedding of the lazy λ-calculus, we conjecture the following theorem to hold.

Theorem 3 *Let M be a closed λ-term. Then M converges in the lazy λ-calculus if and only if $\exists x \exists r (r = \lambda yy \wedge x \cdot r = M)$ converges in the γ-calculus.*

Reduction in the lazy λ-calculus is not a fully satisfactory model of reduction in lazy functional programming languages [13]. The problem is that β-reduction possibly copies the arguments of applications, which will duplicate reductions to be done if the arguments are reducible terms. For instance, $(\lambda x(xx))M$ will reduce to MM containing two copies of the possibly reducible term M. The γ-calculus avoids this problem completely since it copies the bodies of abstractions rather than the actual arguments of functional applications. Launchbury [13] carefully analyses sharing in lazy functional programming and provides an operational semantics providing an accurate model for sharing.

The following facts provide evidence that the γ-calculus is superior to the λ-calculus as an operational model of functional programming languages:

- The γ-calculus can directly express (mutual) recursion;
- the γ-calculus can express sharing;
- the γ-calculus can mix lazy with eager functions;
- the γ-calculus provides a unified framework for functional and concurrent programming.

8 Records

Records can be expressed in the γ-calculus as functions mapping field names to their associated values. For instance, the record

```
[A:U B:V C:W]
```

can be expressed as the function

```
fun {F}
   if F=C then W
   elseif F=B then V
   elseif F=A then U
   else undefined fi
end
```

returning the name `undefined` in case the argument is not equal to one of the field names `A`, `B`, `C`. We have now switched to a concrete syntax for the γ-calculus. Variables are written as identifiers starting with capital letters, and names are written as identifiers starting with lower case letters (e.g., `undefined`). Functional notation translates as in the section on the embedding of the eager λ-calculus.

Note that the field names of the above record are given as variables. In case two or more field names turn out to be equal, the rightmost value specification wins.

Record adjunction takes the union of two records, where conflicts are resolved by giving priority to the right record; for instance,

```
[a:1 b:2 c:3] * [b:8 d:6] = [a:1 b:8 c:3 d:6]
```

In the γ-calculus, record adjunction can be expressed as the higher-order function

```
Adjoin = fun {R S}
            fun {F}
               local V = {S F} in
                  if V=undefined then {R F} else V fi
               end
            end
         end
```

9 Procedures with Encapsulated State

The following defines a procedure `{Num X}` maintaining an internal counter initialized with `0`.

```
local C = {NewCell 0} in
   proc {Num X}
      local Y in {C X Y} {Plus X 1 Y} end
   end
end
```

An application `{Num X}` will equate `X` with the current value of the counter and then increment the counter. It is straightforward to represent numbers in the γ-calculus. The procedure `NewCell` is defined as

$$\exists a(\texttt{NewCell} = a \wedge a{:}\, xy/\exists c(x = c \wedge c{:}y)).$$

Now suppose the computation space contains the applications

```
{Num X}  {Num Y}  {Num Z}
```

Then the variables `X`, `Y`, and `Z` will be equated to different numbers and the internal counter of `Num` will be incremented three times. One possible outcome is `X=0`, `Y=1`, `Z=2`. Another possible outcome is `X=1`, `Y=0`, `Z=2`. However, `X=3`, `Y=0`, `Z=2` is impossible, provided there are no other applications of `Num` but the ones above.

The procedure `Num` builds a state sequence

$$u_1,\ u_2,\ u_3,\ \ldots,\ u_k$$

whose members are linked by constraints {Plus u_i 1 u_{i+1}}, and whose respective last member is hold in the cell C. Concurrent applications of Num create concurrent exchange requests for the cell C, which are serialized indeterministically. Reduction of an application {C X Y} will equate X to the current end of the sequence and make Y the new end of the sequence. Note that this construction makes crucial use of logic variables, and that mutual exclusion of the competing state accesses is obtained for free.

The procedure Num is unsafe in so far that an application {Num 567}, say, may set the counter to 568, due to the indeterministic choice of the equation to be used with the Elimination Rule. A safe version of Num is

```
local C = {NewCell 0} in
   proc {Num A}
      local X Y in {C X Y} {Plus X 1 Y} {Wait X A} end
   end
end
```

where {Wait X A} is defined as $\exists a($**if** X $= a$ **then** $\top$ **else** $X = A)$.

10 Objects

Objects are procedures with encapsulated state. They are specified by a collection of methods, possibly obtained by inheritance from other objects. Objects are applied to messages. A message is a record [methodName:M ...] specifying the name M of the method to be applied, possibly together with input and output arguments. A method is a possibly indeterministic function

$$\textit{method}\colon \textit{state} \times \textit{message} \times \textit{object} \;\rightarrow\; \textit{state}$$

evolving the state of the object according to the message and the object itself (the so-called self reference).

When an object is applied to a message, the method requested by the message

```
Method = {MethodTable {Message methodName}}
```

is obtained from the method table of the object (represented as a record). Next a request

```
{C State NewState}
```

to extend the state sequence of the object is issued (C is the encapsulated cell holding the end of the state sequence) and the selected method is applied

```
{Method State Message O NewState}
```

```
proc {Create MethodTable O}
   local C = {NewCell EmptyRecord} in
      proc {O Message}
         local Method State NewState in
            Method = {MethodTable {Message methodName}}
            {C State NewState}
            {Method State Message O NewState}
         end
      end
   end
end
```

Figure 3: Object creation

to link the new state with the old state.

Figure 3 shows a procedure `{Create MethodTable O}` creating a new object `O` from a method table given as argument. States are represented as records, and the initial state is the empty record represented as follows:

```
EmptyRecord = fun {F} undefined end
```

The procedure `Create` is oversimplified in that it does not

- handle the case where the requested method is undefined
- provide a possibility to initialize the state of the newly created object (which is a must in a concurrent setting)
- provide more sophisticated synchronization, for instance, state access only after the method to be applied is known
- provide a possibility to close an object.

All these features can be incorporated easily [8]. Initialization can be taken care of by giving `Create` an initial message as extra argument.

Using the syntax of Oz [8], a simple counter object `C` can be created as follows:

```
create C
   meth init(X)  val <- X end
   meth inc(X)  val <- @val+X end
   meth read(X)  X=@val end
end
```

```
[init: proc {InState Message Self OutState}
          OutState = {Adjoin InState [val: {Message arg}]}
       end
 inc:  proc {InState Message Self OutState}
          OutState = {Adjoin InState
                           [val: {Plus {InState val} {Message arg}}]}
       end
 read: proc {InState Message Self OutState}
          OutState = InState
          {Message arg} = {InState val}
       end
]
```

Figure 4: Method table of a simple counter object

This translates in an application of the procedure `Create` in Figure 3 to the method table shown in Figure 4. The state of the counter is represented as a one field record `[val:_]`. The methods `init` and `inc` "update" the attribute `val` by means of record adjunction.[5] A message requesting that the counter be incremented by 67, say, takes the form `[methodName:inc arg:67]`. The generality obtained by representing states as records and attribute updates as adjunctions is needed when the methods of the counter are inherited to objects with additional attributes.

Creating an object O by inheritance from objects $O_1, \ldots, O_n$ means to obtain the method table of O by combining the method tables of $O_1, \ldots, O_n$, possibly by record adjunction. To enable inheritance, the method table of an object must be made accessible. One straightforward way to do this is to equip an object with a pseudo-method returning its method table.

From our discussion it should be clear that there is more than one style of object-orientation the γ-calculus can express. A fully developed style of object-orientation based on the ideas outlined here is realized in Oz [8].

The observation that objects are procedures with encapsulated state is well-known in the Lisp community [1]. Our contribution here is to show that this idea carries over smoothly to the concurrent setting of the γ-calculus.

Our object model can express private methods and private attributes by restricting the visibility of method and attribute names exploiting the statically scoped setting of the γ-calculus. Although attributes are not directly accessible, they may be visible to methods added by inheritance.

[5] Attributes are the field names of states and represent what is called an instance variable in Smalltalk.

```
proc {NewPort Port Stream}
   local C = {NewCell Stream} in
      proc {Port Message}
         local S in {C [token:Message next:S] S} end
      end
   end
end
```

Figure 5: Creating ports

11 Communication

We have seen that we can express communicating concurrent objects as procedures with encapsulated state. This model is different from the established model, where a concurrent object is an agent reading messages from a communication medium (e.g., streams in concurrent logic programming [24], mail boxes in the actor model [9], and ports [12] in AKL). Moreover, the principal notion of process algebras and the π-calculus is communication through channels. So, how is it that the γ-calculus can express communicating concurrent objects without a dedicated communication prmitive?

The answer is simple: Explicit communication is unnecessary if procedures can be applied concurrently and can have encapsulated state. State is obtained from cells, which can be seen as a primitive and standardized form of procedures with state. Thus, communication and state turn out to be different sides of the same coin. This observation is fundamental, but certainly not new.

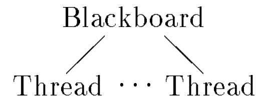
Our object model provides for straightforward many-to-one communication. In contrast, streams in concurrent logic programming [24] provide for easy one-to-many communication, but have severe problems with many-to-one communication (see [12] for a discussion of this issue).

Ports [12] are a communication structure well-suited for both many-to-one and one-to-many communication. Ports can be easily expressed in the γ-calculus. The procedure `{NewPort Port Stream}` in Figure 5 creates a new port (a procedure) and connects it to a stream (a logic variable to be constrained incrementally to a list). An application `{Port Message}` extends the stream associated with the port with the reference `Message`. One easily obtains many-to-many communication since the port can be shared by many message senders and the stream can be shared by many message receivers.

12 An Execution Strategy

A programming language based on the γ-calculus must make some assumptions about the order in which possible reduction steps are to be carried out. Such assumptions are needed so that the programmer can write fair[6] and efficient programs. We will outline one possible execution strategy below.

Our execution strategy organizes a computation space into a blackboard and a collection of threads.



The *blackboard* is a composition of abstractions, cells, and redundant equations of the form $u = u$ or $a = b$.[7] A *thread* is a nonempty stack of expressions. The execution strategy considers the threads of a computation space in a round-robin fashion making sure that every reducible thread will make progress. As computation proceeds, existing threads may terminate and new threads may be created.

A thread is reduced by considering its topmost expression. The reduction rules for threads are derived from the rules of the γ-calculus. A thread is not reducible if it consists of a single expression E and E is either a conditional whose guard does not have the form $u = u$ or $a = b$, or an application that does not match an abstraction or a cell on the blackboard. In all other cases, a thread is reduced by popping its topmost expression and if it is

1. $E \wedge F$: push first F and then E
2. $\exists x E$: create a fresh variable y and push $E[y/x]$
3. $\exists a E$: create a fresh name b and push $E[b/a]$
4. $x = u$ or $u = x$, where $x \neq u$: replace all occurrences of x with u
5. $u = u$, $a = b$, $a{:}\overline{x}/E$, or $a{:}x$: write it on the blackboard
6. $a\overline{u}$ and the blackboard contains a matching abstraction $a{:}\overline{x}/E$: push $E[\overline{u}/\overline{x}]$
7. **if** $u = u$ **then** E **else** F: push E
8. **if** $a = b$ **then** E **else** F, where $a \neq b$: push F
9. avw and the blackboard contains a matching cell $a{:}u$: push $v = u$ and replace $a{:}u$ with $a{:}w$ on the blackboard

[6] Fairness roughly means that reduction steps that could be done will be done eventually.

[7] Equations of the form $u = u$ or $a = b$ have no computational significance and can be dropped in an implementation.

10. an application or conditional that cannot reduce yet with one of the above rules: make it the single expression of a new thread (**Suspension Rule**).

The congruence laws must not be applied. We assume that computation starts with a computation space where no variable is free and no free name is declared. These assumptions ensure that capturing of references cannot occur. The rules have the remarkable property that a reducible thread stays reducible if other threads are reduced before it.

The idea is to start with a computation space with an empty blackboard and a single thread containing a single expression. If the top of a nonsingleton thread is not yet reducible, it is *suspended* by moving it to a newly created thread. This way the thread is not blocked and the next expression can be reduced. One can force the creation of a new thread executing E by writing

$$\exists x\,(\mathbf{if}\ x = a\ \mathbf{then}\ E\ \mathbf{else}\ \top\ \wedge\ x = a).$$

An expression is called *sequential* if it will execute with a single thread; that is, if we start with a computation space consisting just of one singleton thread containing the expression, it cannot evolve into a space with more than one thread. An expression is called *quasi-sequential* if it is congruent to a sequential expression. If E_1 and E_2 are sequential, then

$$\exists x\,(\mathbf{if}\ x = a\ \mathbf{then}\ E_1\ \mathbf{else}\ \top\ \wedge\ \mathbf{if}\ x = a\ \mathbf{then}\ E_2\ \mathbf{else}\ \top\ \wedge\ x = a)$$

is quasi-sequential but not sequential.

A implementation may execute several threads in parallel. Our execution strategy has the interesting property that a sequential expression may be easily rewritten such that it executes with several possibly parallel threads.

Let M be a closed λ-term. Then the expression $\exists x(x = M)$ obtained with the translation embedding the eager λ-calculus into the γ-calculus is sequential. Expressions obtained with the translation embedding the lazy λ-calculus are in general not even quasi-sequential.

13 First-order Constraints and Search

We will now extend the γ-calculus with general first-order constraints. The extension to general constraints will confront us with the problem of failure, which we could circumvent nicely for the simple constraints of the γ-calculus.

In the following we can only present some basic ideas concerning the extension of the γ-calculus to general constraints and search. For a deeper investigation of these issues we refer the reader to [25, 23, 20].

We base our notion of constraint system on first-order predicate logic with equality. A *constraint system* consists of

1. a signature Σ (a set of constant, function and predicate symbols)

2. a consistent theory Δ (a set of sentences over Σ having a model)
3. an infinite set of constants in Σ called *names* satisfying two conditions:
   (a) $\Delta \models \neg(a \doteq b)$ for every two distinct names a, b
   (b) $\Delta \models \phi \leftrightarrow \psi$ for every two sentences ϕ, ψ over Σ such that ψ can be obtained from ϕ by permutation of names.

Given a constraint system, we will call every first-order formula over its signature a *constraint.* We use $\bot$ for the constraint that is always false, and $\top$ for the constraint that is always true.

The minimal constraint system has no other symbols but names in its signature. The usual tree constraint systems (finite or rational constructor trees) can be made into constraint systems in our sense by simply distinguishing infinitely many constants as names.

We now extend the γ-calculus with three new forms

$$E, F, G \quad ::= \quad \ldots \mid \phi \mid \textbf{if } \phi \textbf{ then } E \textbf{ else } F \mid E \nabla F$$
$$\phi, \psi \quad : \quad \text{constraints}$$

called *constraints*, *conditionals*, and *disjunctions*, respectively. We assume that all constraints are taken from some fixed constraint system. Recall that a constraint is simply a first-order formula over the constraint signature. A real programming language will of course carefully restrict the constraints a programmer can actually write (see, for instance, Oz [8]). The new expressions subsume the expressions $\top$, $u = v$, and **if** $u = v$ **then** E **else** F of the γ-calculus.

13.1 Constraints

The semantics of constraints in the extended γ-calculus is given by four congruence laws:

1. conjunction of constraints is congruent to composition of constraints
2. existential quantification $\exists x \phi$ of constraints is congruent to variable declaration $\exists x \phi$ over constraints
3. $\phi \equiv \psi \quad$ if $\Delta \models \phi \leftrightarrow \psi$
4. $x = u \wedge E \;\equiv\; x = u \wedge E[u/x] \quad$ if u free for x in E.

The first three laws provide for constraint simplification. Law (4) extends the equality imposed by constraints to all expressions. The Elimination Rule of the γ-calculus is subsumed by the new congruence laws and is thus not present in the extended calculus.

Proposition 4 *If $\Delta \models \phi \wedge \psi \leftrightarrow \psi'$, then $\phi \wedge \psi \wedge E \;\equiv\; \psi' \wedge E$. If $\Delta \models \phi \rightarrow \psi$, then $\phi \wedge E \;\equiv\; \psi \wedge \phi \wedge E$.*

13.2 Conditionals

The semantics of the conditional is given by the congruence law

$$\phi \wedge \textbf{if } \psi \textbf{ then } E \textbf{ else } F \quad \equiv \quad \phi \wedge \textbf{if } \phi \wedge \psi \textbf{ then } E \textbf{ else } F$$

providing for relative simplification of conditional guards (see [2, 26]) and two reduction rules

$$\textbf{if } \top \textbf{ then } E \textbf{ else } F \ \rightarrow \ E \qquad\qquad \textbf{if } \bot \textbf{ then } E \textbf{ else } F \ \rightarrow \ F$$

subsuming the corresponding rules of the γ-calculus.

Proposition 5 *If* $\Delta \models \phi \rightarrow \psi$, *then* $\phi \wedge \textbf{if } \psi \textbf{ then } E \textbf{ else } F \ \rightarrow \ \phi \wedge E$. *If* $\Delta \models \phi \rightarrow \neg\psi$, *then* $\phi \wedge \textbf{if } \psi \textbf{ then } E \textbf{ else } F \ \rightarrow \ \phi \wedge F$.

A useful generalization of the conditional is obtained by allowing for multiple clauses

$$\textbf{if } \phi_1 \textbf{ then } E_1 \ [] \ \ldots \ [] \ \phi_n \textbf{ then } E_n \textbf{ else } F$$

where the conditional can reduce with any clause whose guard is entailed. This introduces a new form of indeterminism known as committed choice. If the guards of all clauses are disentailed, then the generalized conditional can reduce to the else constituent.

13.3 Disjunctions

The semantics of disjunctions is given by the congruence laws

$$E \nabla F \ \equiv \ F \nabla E \qquad\qquad \phi \ \wedge \ (E \nabla F) \ \equiv \ (\phi \wedge E) \nabla (\phi \wedge F)$$

and the reduction rules

$$(\bot \wedge E) \nabla F \ \rightarrow \ F \qquad\qquad \top \nabla F \ \rightarrow \ \top.$$

Note that disjunctions do not introduce any form of backtracking. Read from right to left, the second congruence law allows to lift shared constraints (an idea also realized in the constructive disjunction of [7]). For instance,

$$(x = 1 \wedge y = 1) \ \nabla \ (x = 1 \wedge y = 2) \ \ \equiv \ \ x = 1 \ \wedge \ (y = 1 \ \nabla \ y = 2).$$

13.4 Failure

A expression E is called *failed* if $E \equiv E \wedge \bot$. In a failed expression, all conditionals and disjunctions become trivially reducible. Thus computation must be stopped as soon as failure occurs. Note that this is in contrast to the situation in the pure γ-calculus, where computation can proceed orderly in the presence of inconsistent equations $a = b$.

13.5 Search

The extension of the γ-calculus to first-order constraints is of practical use only in conjunction with a facility for search.

Search in the style of Prolog can be provided as follows: Computation proceeds as long as reduction rules are applicable and failure does not occur. If computation arrives at an unfailed and irreducible expression, a disjunctive molecule $E \nabla F$ is selected (if there is any) and two don't know alternatives are created by replacing $E \nabla F$ with E and F, respectively. The alternatives are reduced as before and may be explored following a backtracking strategy. Unfailed and irreducible expressions not containing disjunctive molecules are taken as solutions.

Prolog-style search suffers from many problems. For one thing, it is not obtained within the computational calculus but formulated at the meta-level. Moreover, the idea of backtracking is incompatible with the idea of concurrent and reactive computation.

Combining reactive computation with search has been one of the (unsolved) challenges of the Japanese Fifth Generation Project. A computational calculus solving the problem through encapsulation of search into deep guard combinators has been devised with the concurrent constraint language AKL [11]. Oz realizes a more flexible scheme based on the γ-calculus and a higher-order search combinator spawning a local computation space [23].

13.6 Higher-order Programming and First-order Constraints

The extended γ-calculus has the remarkable property that it combines first-order constraints with higher-order programming. The only requirement on constraints imposed by higher-order programming is the accommodation of names. This is in sharp contrast to approaches based on higher-order logic [18], where higher-order programming comes with the operational burden of higher-order constraints. Although we do not doubt the usefulness of higher-order constraints for some applications (e.g., reasoning about programs), we feel that higher-order programming and higher-order constraints are two separate issues that should be decoupled as much as possible.

The ρ-calculus [20] is a confluent subcalculus of the γ-calculus with constraints, which provides for deterministic higher-order programming with first-order constraints.

14 Relationship with the π-calculus

It is illuminating to compare the γ-calculus with the π-calculus [17, 16, 15], a calculus of concurrent computation that evolved from research on algebraic process calculi. Although the γ-calculus and the π-calculus were conceived with very different goals and intuitions—a unified model of computation in the case of the γ-calculus and a model of communicating processes in the case of the π-calculus—they are strikingly close technically. In fact, both

calculi can be obtained as specializations of a slightly more general calculus, which is obtained from the polyadic π-calculus [15] by distinguishing between names and variables and making variables logical. Logic variables increase the expressivity of the π-calculus in two crucial aspects: They allow to equate communication links, and they provide the possibility to express procedures with input *and* output arguments (recall that a function is a procedure with input and output).

While the γ-calculus has logic variables, the π-calculus has formal arguments only (as in functional programming). While the π-calculus has communication as its principal primitive, the γ-calculus has logic variables, abstraction, and cells as principal primitives. We shall show below that the π-calculus can be extended with logic variables, and that the thus extended asynchronous π-calculus can express abstractions and cells.

To put the comparison of the two calculi on solid ground, we introduce yet another calculus, called the *κ-calculus*. The κ-calculus is an asynchronous and polyadic version of the π-calculus in [16] extended with equations. Its abstract syntax is given by

$$A, B \ ::= \ \top \mid A \wedge B \mid \exists x A \mid x :: \overline{y}/A \mid x\overline{y} \mid x : \overline{y}/A \mid x = y$$

where $\top$ is null, $A \wedge B$ is composition, $\exists x A$ is restriction, $x :: \overline{y}/A$ is an input agent, $x\overline{y}$ is an asynchronous output agent, and $x : \overline{y}/A$ is a replicating input agent (i.e, $!\, x :: \overline{y}/A$). The only form not present in the π-calculus are equations $x = y$. In contrast to the π-calculus, where x and y would be called names, they are called variables in the κ-calculus.

Seen from the perspective of the γ-calculus, we have dropped conditionals and the distinction between names and variables, and we have added the form $x :: \overline{y}/A$, which will turn out to be a once-only abstraction.

The structural congruence of the κ-calculus is given by the usual laws for composition and restriction, α-conversion for both input agents, symmetry for equations, and replication for replicating input agents:

$$x : \overline{y}/A \ \equiv \ x :: \overline{y}/A \ \wedge \ x : \overline{y}/A.$$

The reduction axioms are the Communication Rule

$$x\overline{y} \ \wedge \ x :: \overline{z}/A \ \rightarrow \ A[\overline{y}/\overline{z}] \qquad \text{if } \overline{y} \text{ free for } \overline{z} \text{ in } A$$

and the Elimination Rule

$$\exists x(x = y \ \wedge \ A) \ \rightarrow \ A[y/x] \qquad \text{if } x \neq y \text{ and } y \text{ free for } x \text{ in } E.$$

The structural reduction rules are the usual ones.

Seen from the perspective of the γ-calculus, an output agent is an application and a replicating input agent is an abstraction. Ordinary input agents are once-only abstractions providing extra expressivity. In fact, cells can be expressed using once-only abstractions:

$$x : y \quad :\equiv \quad \exists z\, (x :: uv/(u = y \wedge zv) \ \wedge \ z : w/\, x :: uv/(u = w \wedge zv)).$$

The κ-calculus does not make a distinction between variables and names. Without this distinction, there is nothing that can make two variables different. Hence the symmetric conditional of the γ-calculus does not carry over to the κ-calculus. However, we could still have an asymmetric conditional just testing for equality.

One easily verifies that our embeddings of the eager and lazy λ-calculus into the γ-calculus carry over to the κ-calculus. Due to the presence of logic variables, they are simpler than the ones for the π-calculus given by Milner [16]. In contrast to Milner's encoding, our embedding of the lazy λ-calculus shares reductions of arguments (as in implementations of lazy functional programming).

It seems that the κ-calculus cannot express record adjunction and, consequently, inheritance with method overwriting. The problem is that two variables cannot be established as different. Thus names and a corresponding symmetric conditional seem to be crucial for modeling inheritance.

15 Future Research

Our investigations of the γ-calculus are at an early stage. So far, they have mainly been driven by considerations concerning the design and implementation of the programming language Oz, of which it formalizes important aspects. Directions for future research are type disciplines and reasoning about programs. In particular, a declarative characterization of program equivalence is desirable, the investigation of which may start from the techniques developed for the π-calculus. Another interesting topic are extensions of the γ-calculus so that it can model distributed computation and mobility.

Acknowledgements

I'm thankful to Martin Müller and Joachim Niehren for continued discussions accompanying the development of the γ-calculus. Martin Henz, Andreas Podelski, Ralf Treinen and Jörg Würtz helped by commenting on a draft version of this paper.

References

[1] H. Abelson, G. J. Sussman, and J. Sussman. *Structure and Interpretation of Computer Programs.* The MIT Press, Cambridge, Mass., 1985.

[2] H. Aït-Kaci, A. Podelski, and G. Smolka. A feature-based constraint system for logic programming with entailment. *Theoretical Computer Science*, 122(1–2):263–283, January 1994.

[3] F. Benhamou and A. Colmerauer, editors. *Constraint Logic Programming: Selected Research.* The MIT Press, Cambridge, Mass., 1993.

[4] G. Berry and G. Boudol. The chemical abstract machine. In *Proceedings of the 17th ACM Conference on Principles of Programming Languages*, pages 81–94, 1990.

[5] N. Dershowitz and J.-P. Jouannaud. Rewrite systems. In *Handbook of Theoretical Computer Science*, volume B, chapter 15. North–Holland, Amsterdam, Holland, 1990.

[6] K. Futatsugi, J. Goguen, J.-P. Jouannaud, and J. Meseguer. Principles of OBJ2. In *Proceedings of the 12th ACM Conference on Principles of Programming Languages*, pages 52–66, 1985.

[7] P. v. Hentenryck, V. Saraswat, and Y. Deville. Design, implementations, and evaluation of the constraint language cc(FD). Technical Report CS-93-02, Brown University, Box 1910, Providence, RI 02912, 1993.

[8] 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, Deutsches Forschungszentrum für Künstliche Intelligenz, Stuhlsatzenhausweg 3, D-66123 Saarbrücken, Germany, 1994.

[9] C. Hewitt, P. Bishop, and R.Steiger. A universal modular ACTOR formalism for artificial intelligence. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 235–245, 1973.

[10] J. Jaffar and M. J. Maher. Constraint Logic Programming: A Survey. *The Journal of Logic Programming*, to appear, 1994.

[11] S. Janson and S. Haridi. Programming paradigms of the Andorra kernel language. In V. Saraswat and K. Ueda, editors, *Logic Programming, Proceedings of the 1991 International Symposium*, pages 167–186, San Diego, USA, 1991. The MIT Press.

[12] S. Janson, J. Montelius, and S. Haridi. Ports for objects. In *Research Directions in Concurrent Object-Oriented Programming.* The MIT Press, Cambridge, Mass., 1993.

[13] J. Launchbury. A natural semantics for lazy evaluation. In *Proceedings of the 20th ACM Conference on Principles of Programming Languages*, pages 144–154, 1993.

[14] J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96:73–155, 1992.

[15] R. Milner. The polyadic π-calculus: A tutorial. ECS-LFCS Report Series 91-180, Laboratory for Foundations of Computer Science, University of Edinburgh, Edinburgh EH9 3JZ, October 1991.

[16] R. Milner. Functions as processes. *Journal of Mathematical Structures in Computer Science*, 2(2):119–141, 1992.

[17] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, I. *Information and Computation*, 100(1):1–40, September 1992.

[18] G. Nadathur and D. Miller. An overview of λProlog. In R. A. Kowalski and K. A. Bowen, editors, *Proceedings of the Fifth International Conference and Symposium on Logic Programming*, pages 810–827, Seattle, Wash., 1988. The MIT Press.

[19] J. Niehren. Embedding the eager lambda calculus into the delta calculus. Research report, DFKI, Stuhlsatzenhausweg 3, D-66123 Saarbrücken, Germany, 1994. Forthcoming.

[20] J. Niehren and G. Smolka. A confluent calculus for higher-order relational programming. In *Proceedings of the 1st International Conference on Constraints in Computational Logics*, Munich, Germany, September 1994. LNCS, Springer-Verlag.

[21] J. Niehren and G. Smolka. Functional computation in a calculus of relational abstraction and application. Research Report RR-94-04, DFKI, Stuhlsatzenhausweg 3, D-66123 Saarbrücken, Germany, March 1994.

[22] V. A. Saraswat. *Concurrent Constraint Programming.* The MIT Press, Cambridge, Mass., 1993.

[23] C. Schulte and G. Smolka. Encapsulated search in higher-order concurrent constraint programming. Technical report, DFKI, Stuhlsatzenhausweg 3, D-66123 Saarbrücken, Germany, April 1994.

[24] E. Shapiro. The family of concurrent logic programming languages. *ACM Computing Surveys*, 21(3):413–511, September 1989.

[25] G. Smolka. A calculus for higher-order concurrent constraint programming with deep guards. Research Report RR-94-03, Deutsches Forschungszentrum für Künstliche Intelligenz, Stuhlsatzenhausweg 3, D-66123 Saarbrücken, Germany, February 1994.

[26] G. Smolka and R. Treinen. Records for logic programming. *The Journal of Logic Programming*, 18(3):229–258, April 1994.

[27] G. Winskel. *The Formal Semantics of Programming Languages.* The MIT Press, Cambridge, Mass., 1993.

Remark. Papers of authors from the Programming Systems Lab of DFKI are available through anonymous ftp from `ps-ftp.dfki.uni-sb.de` or through WWW from `http://ps-www.dfki.uni-sb.de/`.