**Deutsches
Forschungszentrum
für Künstliche
Intelligenz GmbH**

**Research
Report**

RR-94-03

A Calculus for Higher-order Concurrent Constraint Programming with Deep Guards

Gert Smolka

February 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

Deutsches Forschungszentrum für Künstliche Intelligenz

The German Research Center for Artificial Intelligence (Deutsches Forschungszentrum für Künstliche Intelligenz, DFKI) with sites in Kaiserslautern and Saarbrücken is a non-profit organization which was founded in 1988. The shareholder companies are Atlas Elektronik, Daimler-Benz, Fraunhofer Gesellschaft, GMD, IBM, Insiders, Mannesmann-Kienzle, Sema Group, and Siemens. Research projects conducted at the DFKI are funded by the German Ministry for Research and Technology, by the shareholder companies, or by other industrial contracts.

The DFKI conducts application-oriented basic research in the field of artificial intelligence and other related subfields of computer science. The overall goal is to construct systems with technical knowledge and common sense which - by using AI methods - implement a problem solution for a selected application area. Currently, there are the following research areas at the DFKI:

Intelligent Engineering Systems
Intelligent User Interfaces
Computer Linguistics
Programming Systems
Deduction and Multiagent Systems
Document Analysis and Office Automation.

The DFKI strives at making its research results available to the scientific community. There exist many contacts to domestic and foreign research institutions, both in academy and industry. The DFKI hosts technology transfer workshops for shareholders and other interested groups in order to inform about the current state of research.

From its beginning, the DFKI has provided an attractive working environment for AI researchers from Germany and from all over the world. The goal is to have a staff of about 100 researchers at the end of the building-up phase.

Dr. Dr. D. Ruland
Director

A Calculus for Higher-order Concurrent Constraint Programming with Deep Guards

Gert Smolka

DFKI-RR-94-03

This work has been supported ba a grant from The Federal Ministry for Research and Technology (FKZ ITWM-9105), the Esprit Project ACCLAIM (PE 7195), and the Esprit Working Group CCL (EP 6028).

A Calculus for Higher-order Concurrent Constraint Programming with Deep Guards

Gert Smolka

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

February 1994

Abstract

We present a calculus providing an abstract operational semantics for higher-order concurrent constraint programming. The calculus is parameterized with a first-order constraint system and provides first-class abstraction, guarded disjunction, committed-choice, deep guards, dynamic creation of unique names, and constraint communication. The calculus comes with a declarative sublanguage for which computation amounts to equivalence transformation of formulas. The declarative sublanguage can express negation.

Abstractions are referred to by names, which are first-class values. This way we obtain a smooth and straightforward combination of first-order constraints with higher-order programming.

Constraint communication is asynchronous and exploits the presence of logic variables. It provides a notion of state that is fully compatible with constraints and concurrency.

The calculus serves as the semantic basis of Oz, a programming language and system under development at DFKI.

Contents

1 Introduction **3**

2 Constraints 4

3 Calculus A **5**

3.1 Syntax 6
3.2 Structural Congruence 7
3.3 The Blackboard Metaphor 11
3.4 Reduction 11
3.5 Termination 15
3.6 Entailment and Negation 16
3.7 Relative Simplification 16
3.8 Confluence 17
3.9 Distribution Rule 17
3.10 Relation to SLD- and SLDNF-resolution 18
3.11 Freeze 19

4 Extensions **19**

4.1 Guarded Disjunction 19
4.2 Committed-Choice 20
4.3 Names 21
4.4 First-class Abstraction 22

5 Calculus B **23**

5.1 Constraint Systems 23
5.2 Syntax 24
5.3 Structural Congruence 25
5.4 Reduction 25

6 Constraint Communication **30**

1 Introduction

Concurrent constraint programming [28] brings together ideas from constraint and concurrent logic programming. Constraint logic programming [13, 2], on the one hand, originated with Prolog II [6] and was prompted by the need to integrate numbers and data structures in an operationally efficient, yet logically sound manner. Concurrent logic programming [30], on the other hand, originated with the Relational Language [5] and was promoted by the Japanese Fifth Generation Project, where logic programming was conceived as the basic system programming language and thus had to account for concurrency, synchronization and indeterminism. For this purpose, the conventional SLD-resolution scheme had to be replaced with a new computation model based on the notion of committed-choice. At first, the new model developed as an ad hoc construction, but finally Maher [18] realized that commitment of agents can be captured logically as constraint entailment. The first practical language design combining committed-choice with encapsulated search is AKL [14]. AKL's primary mechanism for encapsulation of nondeterminism are deep guards.

In 1991 the DFKI started the research project Hydra with the goal to design, investigate and implement a high-level concurrent programming language bringing together the merits of logic and object-oriented programming. Our starting point was the existing work on concurrent and constraint logic programming, and some ideas for concurrent control of generalized constraint logic programming [31]. It soon became clear that some of our ideas were related to the ideas realized in AKL. However, to arrive at a smooth and practical integration of constraint and object-oriented programming, we felt that it is absolutely necessary that the underlying language is higher-order, that is, that procedures and agents are first-class citizens. We also came to the conclusion, that the established model of communication in concurrent logic programming based on streams was not compatible with our goals, both because it induces a tedious and low-level programming style, and because it poses serious implementation problems due to the need for fair stream merging (for a similar argumentation see also [15]).

Our investigations resulted 1992 in the design and implementation of a first version of Oz [11], a higher-order object-oriented concurrent constraint programming language. Some aspects of Oz have been reported in [12, 32]. The major difficulty encountered in the design of Oz was the lack of a sufficiently powerful framework for designing such a language (i.e., specifying its operational semantics). Saraswat's framework [28], for instance, accommodates neither deep guards nor first-class procedures. In fact, it not even accounts for the incremental aspects of the operational semantics of flat guards. The tree rewriting semantics specifying the Extended Andorra Model [9] and the structural operational semantics for AKL [10] turned out to be more helpful. Finally, we learned from the setup of the π-calculus [21] how a tree rewriting semantics can be made sufficiently abstract: rather than employing real trees, one can use abstract trees obtained by taking the quotient with respect to an abstract equality called structural congruence. This idea provided a sufficient base for coming up with a flexible calculus specifying

the abstract operational semantics of Oz. This setup nicely combines declarative aspects with operational aspects: Aspects that are accounted for by laws for the structural congruence are put in declaratively, while aspects that are accounted for by reduction rules are put in operationally. In our calculi constraint propagation and simplification are accommodated purely declaratively.

This paper attempts to convey the calculus underlying Oz to readers having a background in logic programming. For this reason we start with a Calculus A still maintaining a close connection to first-order Predicate Logic. Calculus A constitutes an operational semantics for (constraint) logic programming with negation that is profoundly different from the conventional SLDNF-resolution [17]. Its distinctive primitive is a deep guard conditional. Calculus A will convey a number of important and general ideas: the setup of structural congruence and reduction, nonclausal syntax, deep guards and propagation laws, relative simplification, and internal representation of don't know choices. The next step adds guarded disjunction and a committed-choice combinator (generalizing the conditional). Another, orthogonal generalization gives first-class status to abstractions. This in turn necessitates the introduction of a facility for the dynamic creation of new and unique names. Taken together, these extensions lead to Calculus B. Finally, a new form of asynchronous communication, called constraint communication, is introduced. Constraint communication also introduces a notion of state that is fully compatible with constraints and concurrency.

The way our calculus provides for higher-order programming is unique in that denotation and equality of variables are captured by first-order logic only. In fact, denotation of variables and the facility for higher-order programming are completely orthogonal concepts. This is in contrast to existing approaches to higher-order logic programming [22, 4]. The paper [24] investigates the relationship between higher-order functional computation and higher-order relational computation as realized in Calculus B.

Chapters 2–4 provide the connection to Logic Programming and motivate and explain the setup of Calculus B. Chapter 5 presents Calculus B in a technically self-contained manner. Chapter 6 extends Calculus B with constraint communication.

Practical examples illustrating the expressivity of our calculus can be found in [32], where we show how concurrent objects and multiple inheritance can be expressed with Calculus B and constraint communication.

Calculus B can be conservatively extended with a facility for encapsulated search. This will be the subject of a future paper.

2 Constraints

The calculi presented in this paper are parameterized with respect to a constraint system. One can see them as constructions extending constraint systems with programming facilities.

For our purposes it will suffice to found the notion of constraint system on

first-order Predicate Logic, similar to how it is done in Jaffar and Lassez' CLP-framework [13]. We are aware that there exist more general and foundationally less heavy alternatives for setting up the notion of a constraint system (e.g., [29, 7]); however, by taking Predicate Logic as the starting point, we can build on well-established intuitions, notions and notations, and proceed quickly to the issues we want to bring across.

A **constraint system** consists of a signature Σ (a set of first-order function and predicate symbols) and a consistent theory Δ (a set of first-order sentences over Σ having at least one model). Often the constraint theory Δ will be given as the set of all sentences valid in a certain structure (e.g., the structures of finite trees, rational trees, integers, or rational numbers). A **constraint** is any formula over the signature of the constraint system (here we deviate from [13]). The **basic constraints** are the atomic formulas over Σ closed under conjunction:

$$\phi, \psi \quad ::= \quad \bot \mid \top \mid s \doteq t \mid r(s_1, \ldots, s_n) \mid \phi \wedge \psi.$$

The symbol $\bot$ is the truth constant false, $\top$ is the truth constant true, s and t are terms, and r is a predicate symbol. The letters x, y, z will always denote variables (of which we assume countably infinitely many), and the overlined letters $\overline{x}$, $\overline{y}$, ... are used to denote finite, possibly empty sequences of variables. For a formula $\exists x_1 \ldots \exists x_n \phi$, where $n \geq 0$, we will often write $\exists \overline{x} \phi$. Moreover, $\exists \phi$ abbreviates $\exists x_1 \ldots \exists x_n \phi$, where $x_1, \ldots, x_n$ are the free variables of ϕ. The notations $\forall \overline{x} \phi$ and $\forall \phi$ are defined analogously.

Our calculi make use of the following relationships for constraints:

$$\begin{aligned} \phi \models\!\!\!\models_\Delta \psi \quad &:\Longleftrightarrow \quad \forall\,(\phi \leftrightarrow \psi) \text{ is true in every model of } \Delta \\ \phi \models_\Delta \psi \quad &:\Longleftrightarrow \quad \phi \models\!\!\!\models_\Delta \phi \wedge \psi \end{aligned}$$

It is understood that ϕ and ψ may have free variables. Given a constraint system, a constraint ϕ is called **satisfiable** if $\phi \not\models_\Delta \bot$ (i.e., there is at least one model of Δ in which ϕ is satisfiable).

For examples we will use the finite tree constraint system $\mathcal{H}$ (often called Herbrand) [16, 19] underlying conventional logic programming. The signature of $\mathcal{H}$ consists of infinitely many function symbols for every arity, and the theory of $\mathcal{H}$ (known as Clark's Equality Theory) is given by the schemes:

$$\begin{aligned} f(\overline{x}) \doteq f(\overline{y}) \quad &\rightarrow \quad \overline{x} \doteq \overline{y} \\ f(\overline{x}) \doteq g(\overline{y}) \quad &\rightarrow \quad \bot \qquad\qquad (f \neq g) \\ x \doteq f(\cdots x \cdots) \quad &\rightarrow \quad \bot. \end{aligned}$$

3 Calculus A

Calculus A is a didactic vehicle for conveying our model of concurrent deep guard computation. Its distinctive primitive is a deep guard conditional that can express

negation.[1] Calculus A is formulated in the familiar setting of first-order Predicate Logic, which we will leave for the full calculus. Although deep guards already appeared with Concurrent Prolog, they resisted formalization for a long time. The only other formalization of deep guard computation we know of is the structural operational semantics of AKL [10].

Calculus A models simplification and propagation of constraints purely declaratively by means of its structural congruence.[2] Given this setup, deep guards can be accommodated straightforwardly without any extra machinery. Moreover, our model can account for the incremental aspects of constraint propagation and simplification. This is in contrast to the structural operational semantics of AKL, which does not separate constraint propagation and simplification from the reduction rules.

Calculus A employs a nonclausal syntax that alleviates the distinction between program and query. This prepares the ground for the switch to higher-order abstraction in Calculus B.

3.1 Syntax

The syntax of Calculus A is shown in Figure 1. It is parameterized with respect to the signature of the underlying constraint system, and an additional alphabet of distinct symbols, called **defined predicate symbols**. We identify conjunction and quantification of constraints in the calculus with conjunction and existential quantification of constraints in Predicate Logic.

Every expression of Calculus A corresponds to a first-order formula, where conjunction translates to conjunction, quantification to existential quantification, disjunction to disjunction, and abstraction, application and conditional translate as follows:

$$\begin{aligned} p{:}\overline{x}/E &\implies \forall \overline{x}\,(p(\overline{x}) \leftrightarrow E) \\ p\overline{x} &\implies p(\overline{x}) \\ \mathit{if}\ E\ \mathit{then}\ F\ \mathit{else}\ G\ \mathit{fi} &\implies (E \wedge F)\ \vee\ (\neg E \wedge G). \end{aligned}$$

Abstractions serve as procedure definitions, and applications as procedure calls. We require that the formal arguments of an abstraction be pairwise distinct. The conventional separation between program and query is alleviated by the nonclausal syntax of the calculus. Given a conditional $\mathit{if}\ E\ \mathit{then}\ F\ \mathit{else}\ G\ \mathit{fi}$, we call the constituent E the **guard** of the conditional. A guard is called **flat** if E is a

[1] A deep guard conditional with the same declarative semantics as ours has been proposed and implemented in Nu-Prolog by Lee Naish [23]. The operational semantics of Naish's conditional is however different from ours: it delays until its guard is ground. A deep guard conditional with unsound operational semantics based on cut existed already in Edinburgh's Dec-10 Prolog [25].

[2] Note that SLD-resolution (i.e., the operational semantics of Horn clauses) accommodates constraints operationally rather than declaratively (e.g., the notion of unification is an operational notion).

Symbols

x, y, z, u, v, w	:	*variable*
p, q	:	*defined predicate*

Constraints

ϕ, ψ

Expressions

E, F, G, H	::=	ϕ	*constraint*
		$E \wedge F$	*conjunction*
		$\exists x E$	*quantification*
		$p{:}\overline{x}/E$	*abstraction*
		$p\overline{x}$	*application*
		$E \vee F$	*disjunction*
		$if\ E\ then\ F\ else\ G\ fi$	*conditional*

Figure 1: Syntax of Calculus A.

constraint, and **deep** otherwise. Since we have the logical equivalence

$$\neg E \quad \models\!\!\!\models \quad if\ E\ then\ \bot\ else\ \top\ fi,$$

Calculus A can express negation.

The **variable binders** of Calculus A are quantification and abstraction. A quantification $\exists x E$ binds x with scope E, and an abstraction $p{:}\overline{x}/E$ binds its formal arguments $\overline{x}$ with scope E. The **free variables** of an expression are defined accordingly. We use $\mathcal{V}E$ to denote the set of variables that occur free in E.

An expression is called an **actor** if it is either an application, a disjunction, or a conditional.

Logical equivalence for the expressions of Calculus A is defined as

$$E \models\!\!\!\models_\Delta F \quad :\Longleftrightarrow \quad \Delta \models \forall(E \leftrightarrow F),$$

where Δ is the theory of the underlying constraint system. The signature underlying logical equivalence is the signature of the constraint system together with the alphabet of defined predicate symbols.

3.2 Structural Congruence

The operational semantics of Calculus A will be defined as a reduction relation "$E \to F$" on expressions. It will respect logical equivalence in that $E \to F$ always

$$\frac{}{E \equiv E} \qquad \frac{E \equiv F}{F \equiv E} \qquad \frac{E \equiv F \quad F \equiv G}{E \equiv G}$$

$$\frac{E \equiv E' \quad F \equiv F'}{E \wedge F \equiv E' \wedge F'} \qquad \frac{E \equiv E'}{\exists x E \equiv \exists x E'} \qquad \frac{E \equiv E'}{p{:}\overline{x}/E \equiv p{:}\overline{x}/E'} \qquad \frac{E \equiv E' \quad F \equiv F'}{E \vee F \equiv E' \vee F'}$$

$$\frac{E \equiv E' \quad F \equiv F' \quad G \equiv G'}{if\ E\ then\ F\ else\ G\ fi \equiv if\ E'\ then\ F'\ else\ G'\ fi}$$

Figure 2: Structural congruence laws of Calculus A.

implies $E \models\!\!\models_\Delta F$. The reduction relation will be defined on a quotient of the expressions with respect to an equivalence relation called structural congruence. This setup is familiar in the theory of term rewriting [8], and has been applied to the semantics of concurrent programming in the Chemical Abstract Machine [3] and the π-Calculus [21].

A binary relation $\equiv$ on the expressions of Calculus A is called a **congruence** if it satisfies the structural congruence laws in Figure 2.

Proposition 3.1 *The relation "$E \models\!\!\models_\Delta F$" is a congruence.*

We define the **structural congruence** "$E \equiv F$" of Calculus A to be the least congruence satisfying the proper congruence laws appearing in Figure 3. Except for the first and second propagation law, which are essential for deep guard computation, the laws are familiar from Predicate Logic.

Proposition 3.2 *For all expressions E, F and all constraints ϕ, ψ:*

1. $E \equiv F \;\Rightarrow\; E \models\!\!\models_\Delta F$
2. $\phi \equiv \psi \;\Leftrightarrow\; \phi \models\!\!\models_\Delta \psi$.

Proof. The second claim follows from the first claim and Law SS. To show the first claim, it suffices to show that "$E \models\!\!\models_\Delta F$" satisfies every congruence law in Figure 3, since "$E \models\!\!\models_\Delta F$" is already established as a congruence. All laws but SPC are obvious. That $\models\!\!\models_\Delta$ satisfies SPC follows easily with:

$$\begin{aligned} \pi \wedge \neg E \models\!\!\models_\Delta \bot \vee (\pi \wedge \neg E) \quad &\models\!\!\models_\Delta \quad (\pi \wedge \neg\pi) \vee (\pi \wedge \neg E) \\ &\models\!\!\models_\Delta \quad \pi \wedge (\neg\pi \vee \neg E) \\ &\models\!\!\models_\Delta \quad \pi \wedge \neg(\pi \wedge E). \end{aligned}$$

Renaming

SR: $E \equiv F$ if E and F are equal up to consistent renaming of bound variables

Conjunction

SC: $\wedge$ is associative, commutative, and satisfies $E \equiv E \wedge \top$

Quantification

SQE: $\exists x \exists y E \equiv \exists y \exists x E$

SQM: $\exists x E \wedge F \equiv \exists x (E \wedge F)$ if x does not occur free in F

Disjunction

SD: $E \vee F \equiv F \vee E$

Simplification

SS: $\phi \equiv \psi$ if $\phi \models\!\models_{\Delta} \psi$

Equality

SE: $x \doteq y \wedge E \equiv x \doteq y \wedge E[y/x]$ if y free for x in E

where $E[y/x]$ is obtained from E by replacing every free occurrence of x with y.

Propagation

SPD: $\pi \wedge (E \vee F) \equiv \pi \wedge ((\pi \wedge E) \vee F)$

SPC: $\pi \wedge if\ E\ then\ F\ else\ G\ fi \equiv \pi \wedge if\ \pi \wedge E\ then\ F\ else\ G\ fi$

where π must be a constraint or an abstraction

Replication

SPR: $p{:}\overline{x}/E \equiv p{:}\overline{x}/E \wedge p{:}\overline{x}/E$

Figure 3: Proper congruence laws of Calculus A.

Proposition 3.3 *If x does not occur free in E, then $\exists x E \equiv E$.*

Proof. If x does not occur free in E, then

$$\exists x E \equiv \exists x (\top \wedge E) \equiv \exists x \top \wedge E \equiv \top \wedge E \equiv E$$

by the congruence laws SC, SQM, and SS.

It is important to have a good intuitive understanding of the quotient that structural congruence imposes on the set of expressions. The laws for conjunction make conjunction into an operator building multisets of nonconjunctive expressions, where $\top$ plays the role of the empty multiset. To use the metaphor of the Chemical Abstract Machine [3], conjunction creates the chemical solution in which concurrent computation can take place. The Quantifier Mobility Law SQM and the Renaming Law ensure that quantification does not hinder the flow of the chemical solution. With the congruence laws mentioned so far we can rewrite every expression into the form

$$\exists \overline{x} \, (\pi \wedge E),$$

where π is a conjunction of constraints and abstractions, and where E is a conjunction of actors. Law SQE ("quantifier exchange") turns the variable sequence $\overline{x}$ into a multiset. The commutativity law for disjunctions, SD, takes away the order between the two branches. The Simplification Law makes constraints denotational, that is, their syntax does not matter. The Equality Law extends equalities entailed by constraints to conjoined expressions.

The two propagation laws SPD and SPC make conjoined constraints and abstractions visible in the branches of disjunctions and the guards of conditionals. Read from right to left, they provide for the deletion of abstractions and constraints that are present higher up. For example, taking $\mathcal{H}$ as the underlying constraint system and "." as a binary function symbol, we have

$$\begin{pmatrix} x \doteq 1.u.v.w \\ y \doteq v.w \\ x = 1.u.y \vee x \doteq 1.z.y \end{pmatrix} \equiv \begin{pmatrix} x \doteq 1.u.y \\ y \doteq v.w \\ \top \vee z \doteq u \end{pmatrix}$$

using the laws for conjunction, simplification and propagation. (The rows of a matrix are conjoined by conjunction.) Taken together, the laws for simplification and propagation provide for something we call relative constraint simplification. The intuition behind this name is made explicit in the next proposition, where the constraint in the guard of the conditional is simplified with respect to the constraint above. Similar statements hold for the branches of a disjunction.

Proposition 3.4 (Relative Simplification) *If $\phi \wedge \psi \models\!\!\models_\Delta \phi \wedge \psi'$ and no variable in $\overline{x}$ occurs free in ϕ, then*

$$\phi \wedge \mathit{if}\ \exists \overline{x} \, (\psi \wedge E)\ \mathit{then}\ F\ \mathit{else}\ G\ \mathit{fi} \quad \equiv \quad \phi \wedge \mathit{if}\ \exists \overline{x} \, (\psi' \wedge E)\ \mathit{then}\ F\ \mathit{else}\ G\ \mathit{fi}.$$

Figure 4: The blackboard metaphor.

The following variant of the above proposition will be useful in examples.

Proposition 3.5 (Relative Simplification) *If* $\phi \wedge \psi \models\!\!\!\models_\Delta \phi \wedge \psi'$, *then*

$$\phi \wedge if\ \psi\ then\ E\ else\ F\ fi \quad \equiv \quad \phi \wedge if\ \psi'\ then\ E\ else\ F\ fi.$$

The Replication Law SPR allows for copying and merging of abstractions. It is needed to render the reduction rules confluent.

3.3 The Blackboard Metaphor

We can visualize an expression modulo structural congruence as a computation space consisting of a number of actors connected to a blackboard (see Fig. 4). The actors are either applications, disjunctions or conditionals. The blackboard consists of constraints and abstractions. Conjunction and quantification provide the glue keeping actors and blackboard together. Since conditional and disjunctive actors spawn local computation spaces (i.e., the guards of conditionals and the branches of disjunctions), the computation system is actually a tree-like structure of computation spaces (see Fig. 4).

The reduction rules we will give in the next section can be seen as animation rules for computation spaces. The actors read the blackboard and reduce once the blackboard contains sufficient information. The information on the blackboard increases monotonically. When an actor reduces, it may put new information on the blackboard and create new actors. The actors of a computation space are short-lived: once they reduce they disappear.

3.4 Reduction

We define the **reduction relation** $\rightarrow$ of Calculus A to be the least relation satisfying the structural laws in Figure 5 and the proper laws (called **reduction rules**) in Figure 6. Put more intuitively, we have $E \rightarrow F$ if and only if there are expressions E' and F' such that $E \equiv E'$, $F \equiv F'$, and F' is obtained from E' by applying a reduction rule to a subexpression of E' not appearing in a protected position. A position in an expression is called **protected** if it is within

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

$$\frac{E \to E'}{E \vee F \to E' \vee F} \qquad \frac{E \to E'}{\mathit{if}\ E\ \mathit{then}\ F\ \mathit{else}\ G\ \mathit{fi} \to \mathit{if}\ E'\ \mathit{then}\ F\ \mathit{else}\ G\ \mathit{fi}}$$

Figure 5: Structural reduction laws of Calculus A.

Unfolding

RU: $p\overline{x} \wedge p{:}\overline{y}/E \to \exists \overline{y}\,(\overline{y} \doteq \overline{x} \wedge E) \wedge p{:}\overline{y}/E$
if $\overline{x}$ and $\overline{y}$ are disjoint and of equal length

Disjunction

RDF: $(\bot \wedge E) \vee F \to F$

RDT: $\top \vee E \to \top$

Conditional

RCF: $\mathit{if}\ \bot \wedge E\ \mathit{then}\ F\ \mathit{else}\ G\ \mathit{fi} \to G$

RCT: $\mathit{if}\ \top\ \mathit{then}\ F\ \mathit{else}\ G\ \mathit{fi} \to F$

Figure 6: Reduction rules of Calculus A.

an abstraction, or within the second or third constituent of a conditional. Due to the fact that the reduction relation is defined as the *least* relation satisfying the reduction laws, the protected positions are in fact just those where the reduction relation is not forced by a structural law to satisfy the compatibility property. Disallowing reduction at protected positions makes it possible to write terminating recursive programs. Note that the conditional is the only construct of the calculus that can express sequentializing and synchronizing control.

Proposition 3.6 *For all expressions E, F: if $E \to F$ then $E \models\!\!\models_\Delta F$.*

Proof. It suffices to show that $\models\!\!\models_\Delta$ satisfies all reduction laws. Since $E \models\!\!\models_\Delta F$ is a congruence and $E \equiv F \Rightarrow E \models\!\!\models_\Delta F$ by Proposition 3.2, it suffices to show that $E \models\!\!\models_\Delta F$ satisfies every reduction rule. This is easily verified.

The proposition states that reduction is sound with respect to logical equivalence. Of course, reduction is not complete with respect to logical equivalence. For instance, $p \wedge \neg p \models\!\!\models_\Delta \bot$ although $p \wedge \neg p$ is irreducible.

Proposition 3.7 *If x is free for y in E, then:*

$$px \ \wedge \ p{:}y/E \quad \rightarrow \quad E[x/y] \ \wedge \ p{:}y/E.$$

Proof. Without loss of generality we can assume that x and y are distinct (otherwise we rename the formal argument of the abstraction to a fresh variable). Now we have:

$$\begin{array}{lcll} px \ \wedge \ p{:}y/E & \rightarrow & \exists y\,(y \doteq x \wedge E) \ \wedge \ p{:}y/E & \text{by } RU \\ & \equiv & \exists y\,(y \doteq x) \ \wedge \ E[x/y] \ \wedge \ p{:}y/E & \text{by } SE, SQM \\ & \equiv & E[x/y] \ \wedge \ p{:}y/E & \text{by } SS. \end{array}$$

Example 3.8 *We will now see how reduction in Calculus A works. After going through a flat guard computation, we will see a deep guard computation.*

Let us assume $\mathcal{H}$ as the underlying constraint system. Then the recursively defined predicate nat

$$NAT \ := \ nat{:}x \,/\, if \ x \doteq 0 \ then \ \top \ else \ \exists y\,(x \doteq s(y) \wedge nat\, y) \ fi$$

holds exactly for the trees 0, $s(0)$, $s(s(0))$, ... Now suppose we want to reduce the expression

$$nat\, x \ \wedge \ x \doteq s(0) \ \wedge \ NAT.$$

By unfolding according to Proposition 3.7 we obtain

$$if \ x \doteq 0 \ then \ \top \ else \ \exists y\,(x \doteq s(y) \wedge nat\, y) \ fi \ \wedge \ x \doteq s(0) \ \wedge \ NAT.$$

As usual, we tacitly exploit the associativity and commutativity of conjunction. By relative simplification (Proposition 3.5) we obtain

$$if \ \bot \wedge \top \ then \ \top \ else \ \exists y\,(x \doteq s(y) \wedge nat\, y) \ fi \ \wedge \ x \doteq s(0) \ \wedge \ NAT.$$

Application of the reduction rule RCF yields

$$\exists y\,(x \doteq s(y) \wedge nat\, y) \ \wedge \ x \doteq s(0) \ \wedge \ NAT,$$

from where we proceed to

$$\exists y\,(y \doteq 0 \ \wedge \ if \ y \doteq 0 \ then \ \top \ else \ \exists z\,(y \doteq s(z) \wedge nat\, z) \ fi) \ \wedge \ x \doteq s(0) \ \wedge \ NAT$$

using constraint simplification ($x \doteq s(y) \wedge x \doteq s(0) \ \models\!\!\!|_{\mathcal{H}} \ y \doteq 0 \wedge x \doteq s(0)$) and unfolding according to Proposition 3.7. By relative simplification we obtain

$$\exists y\,(y \doteq 0 \ \wedge \ if \ \top \ then \ \top \ else \ \exists z\,(y \doteq s(z) \wedge nat\, z) \ fi) \ \wedge \ x \doteq s(0) \ \wedge \ NAT$$

from where we proceed to $\exists y\,(y \doteq 0 \ \wedge \ \top) \ \wedge \ x \doteq s(0) \ \wedge \ NAT$ using the reduction rule RCT. Now application of the Simplification Law yields the irreducible expression $x \doteq s(0) \ \wedge \ NAT$.

We are now ready to consider a deep guard computation (the capital letter B is a variable):

$$\begin{array}{ll} & if\ nat\, x\ then\ B \doteq 1\ else\ B \doteq 0\ fi\ \wedge\ x \doteq s(0)\ \wedge\ NAT \\ \equiv & if\ nat\, x \wedge x \doteq s(0) \wedge NAT\ then\ B \doteq 1\ else\ B \doteq 0\ fi\ \wedge\ x \doteq s(0)\ \wedge\ NAT \\ \rightarrow^* & if\ x \doteq s(0) \wedge NAT\ then\ B \doteq 1\ else\ B \doteq 0\ fi\ \wedge\ x \doteq s(0)\ \wedge\ NAT \\ \equiv & if\ \top\ then\ B \doteq 1\ else\ B \doteq 0\ fi\ \wedge\ x \doteq s(0)\ \wedge\ NAT \\ \rightarrow & B \doteq 1\ \wedge\ x \doteq s(0)\ \wedge\ NAT. \end{array}$$

The first congruence follows by the propagation law for conditionals. The following reduction chain on the guard was established above. (We exploit that reduction can be applied to subexpressions if they are not in protected positions.) Using the Propagation Law in the opposite direction, we can rewrite the guard to $\top$. Now the reduction rule RCT for conditionals applies and produces the final expression.

Example 3.9 *Assume the constraint system $\mathcal{H}$ and consider the following definition of a membership predicate for lists:*

$$\begin{array}{lll} MEM & := & mem : X\, L\ /\ if\ L \doteq nil\ then\ \bot \\ & & \qquad else\ \exists H\, R\, (L \doteq H.R\ \wedge\ (X \doteq H\ \vee\ mem\, X\, R))\ fi. \end{array}$$

One can verify the following two derivations:

$$\begin{array}{rcl} \exists L\, (L \doteq 1.2.X.Y\ \wedge\ mem\, X\, L)\ \wedge\ MEM & \rightarrow^* & MEM \\ \exists X\, L\, (X \doteq 7\ \wedge\ L \doteq 1.Y.nil\ \wedge\ mem\, X\, L)\ \wedge\ MEM & \rightarrow^* & Y \doteq 7\ \wedge\ MEM. \end{array}$$

Note that the first derivation employs Rule RDT.

Example 3.10 *Assume the constraint system $\mathcal{H}$ and consider the following definition of a length predicate for lists:*

$$\begin{array}{lll} LEN & := & len : L\, N\ /\ (L \doteq nil\ \wedge\ N \doteq 0)\ \vee \\ & & \qquad \exists H\, R\, M\ (L \doteq H.R\ \wedge\ N \doteq s(M)\ \wedge\ len\, R\, M). \end{array}$$

Due to the symmetry of the operational semantics of disjunctions, the predicate computes numbers for lists and lists for numbers:

$$\begin{array}{rcl} \exists L\, (L \doteq X.Y.nil\ \wedge\ len\, L\, N)\ \wedge\ LEN & \rightarrow^* & N \doteq s(s(0))\ \wedge\ LEN \\ \exists N\, (N \doteq s(0)\ \wedge\ len\, L\, N)\ \wedge\ LEN & \rightarrow^* & \exists X\, (L \doteq X.nil)\ \wedge\ LEN. \end{array}$$

If we define the length predicate with a conditional

$$\begin{array}{lll} LEN & := & len : L\, N\ /\ if\ L \doteq nil\ then\ N \doteq 0 \\ & & \qquad else\ \exists H\, R\, M\ (L \doteq H.R\ \wedge\ N \doteq s(M)\ \wedge\ len\, R\, M)\ fi, \end{array}$$

the symmetry is lost and only the first derivation remains possible.

3.5 Termination

An expression E is called **failed** if $E \equiv \bot \wedge F$ for some F. The reduction rules RDF and RCF are called **failure rules**. An expression is called **nervous** if it is not failed and a failure rule applies to it (e.g., $\bot \vee E$). An infinite derivation $E_1 \rightarrow E_2 \rightarrow E_3 \rightarrow \cdots$ is called **admissible** if no E_i is failed, and E_{i+1} is obtained from E_i by a failure rule whenever E_i is nervous. An expression is called **terminating** if there exists no admissible infinite derivation issuing from it.

Example 3.11 *Assume the constraint system $\mathcal{H}$ and consider the recursively defined predicate nat from Example 3.8. It is easy to see that the expression $\phi \wedge nat\, x \wedge NAT$ is terminating for every constraint ϕ.*

Example 3.12 *Assume the constraint system $\mathcal{H}$ and consider*

$$DNAT \;:=\; nat\colon x \;/\; x \doteq 0 \vee \exists y\, (x \doteq s(y) \wedge nat\, y).$$

Logically, NAT and $DNAT$ are equivalent, that is, $NAT \models\!\models_{\mathcal{H}} DNAT$. Operationally, they behave differently as it comes to termination. For instance, we obtain an admissible infinite derivation issuing from $nat\, x \wedge DNAT$ by applying the unfolding rule repeatedly. However, if we constrain the argument of $nat\, x$ sufficiently, we obtain termination. For instance, $x = s(s(0)) \wedge nat\, x \wedge DNAT$ is terminating.

The examples show that one needs the conditional to write recursive predicates that terminate for underconstrained arguments. Conditionals have however the disadvantage that they destroy the symmetry of relational definitions (see for instance the length predicate in Example 3.10). Calculus B will fix this problem by providing so-called guarded disjunctions, which can express the control needed for termination.

Example 3.13 *Assume the constraint system $\mathcal{H}$ and consider the addition predicate*

$$\begin{aligned} ADD \;:=\; & add\colon x\, y\, z \;/\; if\; x \doteq 0\; then\; y \doteq z \\ & \qquad else\; \exists u\, v\, (x \doteq s(u) \wedge z \doteq s(v) \wedge add\, u\, y\, v)\; fi. \end{aligned}$$

It is not difficult to see that the expression

$$x \doteq s(z) \wedge add\, x\, y\, z \wedge ADD$$

is not terminating.[3] *Note that $x \doteq s(z) \wedge add\, x\, y\, z \wedge ADD \models\!\models_{\mathcal{H}} \bot$. We can enforce termination by sequentializing with a deep guard:*

$$x \doteq s(z) \wedge if\; nat\, x\; then\; add\, x\, y\, z\; else\; \top\; fi \wedge ADD.$$

[3] This example was brought to my attention by Thom Frühwirth.

3.6 Entailment and Negation

Knowing Saraswat's ask and tell calculus [27, 28], one would expect a reduction rule for conditionals that fires upon entailment of the guard by the context:

$$\textit{RCE:} \quad \phi \wedge \mathit{if}\ \psi\ \mathit{then}\ F\ \mathit{else}\ G\ \mathit{fi} \ \rightarrow\ F \qquad \text{if } \phi \models_\Delta \psi.$$

It is clear that Rule RCE can simulate Rule RCT. However, RCT can also simulate RCE. To see this, assume $\phi \models_\Delta \psi$. Then

$$\phi \wedge \mathit{if}\ \psi\ \mathit{then}\ F\ \mathit{else}\ G\ \mathit{fi} \ \equiv\ \phi \wedge \mathit{if}\ \top\ \mathit{then}\ F\ \mathit{else}\ G\ \mathit{fi}$$

by constraint propagation and simplification.

A constraint system is called **independent**[4] if it satisfies

$$\phi \models_\Delta \bigvee_{i=1}^{n} \exists \overline{x}_i \psi_i \ \Rightarrow\ \exists i\colon\ \phi \models_\Delta \exists \overline{x}_i \psi_i$$

for all basic constraints $\phi, \psi_1, \ldots, \psi_n$. The usual tree constraint systems are independent [33]. In particular, this is the case for the finite tree constraint system $\mathcal{H}$ (provided an infinite signature is taken, as required in this paper; see [33] for a counter example).

Proposition 3.14 *Assume the constraint system is independent. Then*

$$\phi \ \wedge\ \bigwedge_{i=1}^{n} \mathit{if}\ \exists \overline{y}_i \psi_i\ \mathit{then}\ \bot\ \mathit{else}\ \top\ \mathit{fi} \ \rightarrow^*\ \bot \quad \Longleftrightarrow \quad \phi \ \wedge\ \bigwedge_{i=1}^{n} \neg \exists \overline{y}_i \psi_i \ \models\!\!\!\models_\Delta\ \bot,$$

provided $\phi, \psi_1, \ldots, \psi_n$ are basic constraints.

Proof. The direction from left to right is obvious since reduction is an equivalence transformation (Proposition 3.6). To show the other direction, we can assume $\phi \models_\Delta \exists \overline{x}_i \psi_i$ for some i since the constraint system is independent. Thus we can use Rule RCE, which yields $\phi \wedge \mathit{if}\ \exists \overline{y}_i \psi_i\ \mathit{then}\ \bot\ \mathit{else}\ \top\ \mathit{fi} \rightarrow^* \phi \wedge \bot$. Now we obtain the left hand side of the equivalence by using Rule RCE with the context $\bot$.

3.7 Relative Simplification

A **relative simplification procedure** for a constraint system is a procedure that, given two basic constraints ϕ and ψ, where ϕ must be satisfiable, produces a constraint ψ' such that $\phi \wedge \psi \ \models\!\!\!\models_\Delta\ \phi \wedge \psi'$. A relative simplification procedure is **complete** if its output ψ' satisfies

1. $\phi \models \exists \overline{x} \psi \ \Rightarrow\ \Delta \models \exists \overline{x} \psi'$ (provided no variable in $\overline{x}$ occurs in ϕ)

[4] Note that our definition of independence involves existential quantification, which is not the case for the conventional definition [16]. Our notion of independence agrees however with the definitions in [33, 20].

2. $\phi \wedge \psi \models\!\dashv_\Delta \bot \;\Rightarrow\; \psi' = \bot$.

A complete relative simplification procedure together with a test "$\Delta \models \exists\overline{x}\phi$" is the basic operational machinery one has to provide for the underlying constraint system in order to decide whether a reduction rule is applicable. Relative simplification procedures for feature tree constraint systems have been developed in [1, 33], and the full version of [33] provides an abstract machine for relative simplification.

A complete relative simplification procedure for the finite tree constraint system $\mathcal{H}$ can easily be obtained from a unification procedure. Given two basic constraints ϕ and ψ, where ϕ is satisfiable, one first computes an idempotent most general unifier θ of ϕ. Next one computes an idempotent most general unifier ω for $\theta\psi$. If ω does not exist, take $\psi' = \bot$. Otherwise, take for ψ' the equational representation of ω. That this in fact specifies a complete relative simplification procedure for $\mathcal{H}$ follows from the results in [16].

3.8 Confluence

An expression is called **admissible** if it is congruent to an expression that contains at most one abstraction per predicate, and that does not nest abstractions into abstractions. It is easy to see that reduction preserves admissibility of expressions.

Conjecture 3.15 *Reduction in Calculus A is confluent on admissible expressions. That is, if E is admissible, $E \to^* F$, and $E \to^* G$, then there exists an expression H such that $F \to^* H$ and $G \to^* H$.*

Example 3.16 *Consider the nonadmissible expression (a and b are distinct constants):*

$$N \quad := \quad (p\colon x/x \doteq a) \;\wedge\; (p\colon x/x \doteq b).$$

It is easy to see that $N \wedge py$ reduces to two noncongruent normal forms $N \wedge y \doteq a$ and $N \wedge y \doteq b$.

3.9 Distribution Rule

A rule obviously missing from Calculus A is the **Distribution Rule**:

$$\textit{RDD:} \quad (E \vee F) \wedge G \;\to\; (E \wedge G) \vee (F \wedge G).$$

With the Distribution Rule the otherwise irreducible expression

$$(x \doteq 1 \vee x \doteq 2) \wedge (x \doteq 3 \vee x \doteq 4)$$

(we assume $\mathcal{H}$ as the underlying constraint system) reduces to $\bot$. The Distribution Rule may quickly lead to combinatorial explosion since it introduces a new copy

of the conjoined G. Moreover, adding the Distribution Rule destroys confluence on admissible expressions:

$$\begin{array}{lll} (\top \vee p) \wedge q \;\rightarrow\; \top \wedge q \;\equiv\; q & & by\ RDT \\ (\top \vee p) \wedge q \;\rightarrow\; (\top \wedge q) \vee (p \wedge q) \;\equiv\; q \vee (p \wedge q) & & by\ RDD. \end{array}$$

We conjecture that confluence can be preserved if Rule RDT is given preference over Rule RDD. Disallowing Rule RDT completely should also recover confluence.

3.10 Relation to SLD- and SLDNF-resolution

It is interesting to relate Calculus A to SLD-resolution [17]. A **goal** is a conjunction of constraints and applications. A **definite abstraction** is a closed (i.e., no free variables) expression of the form

$$p{:}\overline{x} \;/\; \bigvee_{i=1}^{n} \exists \overline{y}_i G_i$$

where the G_i's are goals. A **definite program** is an admissible conjunction of definite abstractions. Clark's completion [17] translates Horn clause programs into definite programs.

Now consider a definite program Π and a goal G. An SLD-derivation with respect to Π issuing from E can be simulated in Calculus A if the Distribution Rule is added. The simulation does not employ the rules RDT, RCT and RCF. Furthermore, because there are no conditionals, and disjunctions are moved above conjunctions with the Distribution Rule, the propagation laws are not needed. However, it is necessary to add two new congruence laws:

$$\begin{array}{rcl} E \vee (F \vee G) & \equiv & (E \vee F) \vee G \\ \exists x\,(E \vee F) & \equiv & \exists x E \vee \exists x F. \end{array}$$

The unfolding rule RU is applied only after an expression has been rewritten to disjunctive normal form $\bigvee_{i=1}^{n} (\Pi \wedge \exists x_i G_i)$. The disjunctive normal form corresponds in fact to the frontier of an SLD-tree, and answers show up as goals G_i that are satisfiable constraints. Finite failure of a goal G amounts to a derivation $\Pi \wedge G \rightarrow^* \Pi \wedge \bot$.

Note that this simulation reveals backtracking as a space efficient implementation of the distribution rule, which constructs only one clause of the disjunctive normal form at a time.

We can also simulate SLDNF-resolution [17], where negation $\neg E$ is expressed as $if\ E\ then\ \bot\ else\ \top\ fi$. Now we need the reduction rules for the conditional and also Rule RDT for disjunction. We also need the propagation law for the conditional, but the propagation law for disjunction is still not needed.

3.11 Freeze

Prolog II's freeze can be expressed as

$$if\ (if\ x \doteq a\ then\ \top\ else\ \top\ fi)\ then\ E\ else\ \top\ fi$$

where we assume $\mathcal{H}$ as the underlying constraint system, and a to be a constant of $\mathcal{H}$. Note that this expression is logically equivalent to E, and that E is released for reduction if and only if the context is strong enough to either entail or disentail $X \doteq a$. This will be the case if and only if x is "bound" to a nonvariable term by the context.

4 Extensions

This section discusses informally the extensions leading from Calculus A to Calculus B.

4.1 Guarded Disjunction

The problem with disjunction in Calculus A is that it lacks the control needed to obtain termination. This is illustrated by the recursive abstraction

$$DNAT \quad := \quad nat\colon x \,/\, x \doteq 0 \vee \exists y\,(x \doteq s(y) \wedge nat\, y)$$

and the nonterminating expression $nat\, x \wedge DNAT$.

It is not difficult to provide the missing control. To this purpose, we extend the syntax of Calculus A with a clause combinator

$$E\ then\ F$$

whose declarative reading is $E \wedge F$. In contrast to conjunction, however, the clause combinator is not commutative and prevents its second argument from reduction. The first argument is called the guard of the clause, and the second argument is called the body of the clause. If we rewrite $DNAT$ to

$$DNAT \quad := \quad nat\colon x \,/\, (x \doteq 0\ then\ \top) \vee \exists y\,(x \doteq s(y)\ then\ nat\, y),$$

then the expression $nat\, x \wedge DNAT$ is obviously terminating. Of course, we need an additional propagation law

$$\pi \wedge (E\ then\ F) \quad \equiv \quad \pi \wedge (\pi \wedge E\ then\ F)$$

for clauses and must arrange things such that if a disjunction reduces to a clause, the body of the clause is released. To this purpose we replace the old reduction rules for disjunction with the following new ones:

$$\begin{array}{rcll} (\bot \wedge E)\ then\ F & \to & \bot & \\ \exists\overline{x}\,(E\ then\ F) \vee \bot & \to & \exists\overline{x}\,(E \wedge F) & \\ \exists\overline{x}\,(\phi\ then\ \top) \vee G & \to & \top & \text{if } \exists\overline{x}\,\phi \equiv \top. \end{array}$$

We can now rewrite the definition of the length predicate from Example 3.10 to

$$\begin{aligned} LEN \quad := \quad & len : L\, N \,/\, (L \doteq nil \;\wedge\; N \doteq 0 \; then \; \top) \;\vee \\ & \exists\, H\, R\, M \; (L \doteq H.R \;\wedge\; N \doteq s(M) \; then \; len\, R\, M) \end{aligned}$$

and obtain a terminating and symmetric solution satisfying

$$\begin{aligned} \exists L \, (L \doteq X.Y.nil \;\wedge\; len\, L\, N) \;\wedge\; LEN \quad & \rightarrow^* \quad N \doteq s(s(0)) \;\wedge\; LEN \\ \exists N \, (N \doteq s(0) \;\wedge\; len\, L\, N) \;\wedge\; LEN \quad & \rightarrow^* \quad \exists X \, (L \doteq X.nil) \;\wedge\; LEN. \end{aligned}$$

Our solution will work fine with binary disjunctions, but not with disjunctions taking more alternatives, for instance,

$$(x \doteq 1 \; then \; E_1) \;\vee\; (y \doteq 2 \; then \; E_2) \;\vee\; (z \doteq 3 \; then \; E_3).$$

This problem can be resolved by having a disjunction combinator

$$or \; (C_1 \cdots C_n)$$

taking a multiset $C_1 \cdots C_n$ of possibly quantified clauses

$$C \quad ::= \quad \exists \overline{x} \, (E \; then \; F)$$

as argument.

Let us summarize. The guarded disjunction combinator spawns any number of possibly quantified clauses. The clauses can be thought of as competing computations. Reduction takes place in the guards of the clauses, but not in their bodies. If a clause has failed (i.e., its guard has reduced to $\bot \wedge E$), it is discarded. If only one clause is left, the disjunction combinator commits to this clause and the body of the clause is released. Moreover, the disjunction can reduce to $\top$ if the guard of a clause whose body is $\top$ is satisfied.

4.2 Committed-Choice

Calculus B will also have a committed-choice combinator

$$if \; C_1 \cdots C_n \; else \; G$$

taking a multiset $C_1 \cdots C_n$ of possibly quantified clauses and an expression G as arguments. The clauses can be thought of as competing computations. Reduction takes place in the guards of the clauses, but not in their bodies. If the guard of a clause is satisfied, the committed-choice combinator can commit to this clause and the body of the clause is released:

$$if \; \exists \overline{x} \, (\phi \; then \; F) \;\; C_2 \cdots C_n \; else \; G \quad \rightarrow \quad \exists \overline{x} \, (\phi \wedge F) \qquad \text{if } \exists \overline{x} \, \phi \equiv \top.$$

If a clause has failed (i.e., its guard has reduced to $\bot \wedge E$), it is discarded. If no clause is left, the committed choice combinator reduces to the else constituent G.

The conditional of Calculus A can be obtained from the committed-choice combinator by having only one unquantified clause: $if\ E\ then\ F\ else\ G$.

Committed-choices with more than one clause introduce indeterminism and hence destroy confluence, as one can see from the example

$$x \doteq 1 \ \wedge\ y \doteq 1 \ \wedge\ if\ (x \doteq 1\ then\ z \doteq 1)\ (y \doteq 1\ then\ z \doteq 2)\ else\ \top,$$

which can reduce to either $x \doteq 1 \wedge y \doteq 1 \wedge z \doteq 1$ or $x \doteq 1 \wedge y \doteq 1 \wedge z \doteq 2$. In general, committed-choices with more than one clause cannot be translated to first-order formulas such that the reduction rules amount to equivalence transformations.

4.3 Names

How can we extent Calculus A such that we can dynamically create new and unique names? The answer is surprisingly simple. First, we have to require that the constraint system comes with an infinite alphabet of distinguished constant symbols called **names** satisfying two conditions:

1. $\Delta \models \neg(a \doteq b)$ for every two distinct names a, b
2. $\Delta \models \phi \leftrightarrow \psi$ for every two first-order sentences ϕ, ψ over the signature of the constraint system such that ψ can be obtained from ϕ by permutation of names.

It is easy to see that the usual finite and rational tree constraint systems (taken over an infinite signature) satisfy these conditions for any set of constant symbols we decide to distinguish as names.

The following proposition says that names are different from any other value that can be uniquely described by a formula.

Proposition 4.1 *Let a constraint system with names satisfying requirements (1) and (2) be given. Moreover, let ϕ be a formula over the signature of the constraint system such that x is the only free variable of ϕ, and such that ϕ determines x, that is, $\Delta \models \exists! x\phi$. Then $\Delta \models \neg\phi[a/x]$ for every name a not occurring in ϕ.*

Proof. We prove the claim by contradiction. Suppose $\mathcal{A}$ is a model of the constraint system such that $\mathcal{A} \models \phi[a/x]$ for some name a not occurring in ϕ. Now let b be a name different from a that also does not occur in ϕ. Since $\phi[a/x]$ and $\phi[b/x]$ are sentences that are equal up to permutation of names, we know by requirement (2) that $\mathcal{A} \models \phi[b/x]$. Moreover, we know $\mathcal{A} \models \neg(a \doteq b)$ by requirement (1). Since we know $\mathcal{A} \models \exists! x\phi$ by assumption, we have a contradiction.

A small generalization of Calculus A will do the rest of the job: we allow quantification over names, that is, $\exists a E$ is considered a well-formed exoression; moreover, we provide the same congruence laws for quantification of names we already have

for quantification of variables, including renaming of quantified names. Of course, $\exists aE$ is not a formula of Predicate Logic and must not be thought of as existential quantification.

With this simple formal machinery in place, we can create new and unique names as follows:

$$newname(x) \implies \exists ax \doteq a.$$

That this construction indeed works can be seen from the congruences

$$\begin{aligned} newname(x) \wedge newname(y) \wedge E &\equiv \exists ax \doteq a \wedge \exists ay \doteq a \wedge E \\ &\equiv \exists a(x \doteq a \wedge \exists by \doteq b \wedge E) \\ &\equiv \exists a \exists b(x \doteq a \wedge y \doteq b \wedge E), \end{aligned}$$

which employ the Quantifier Mobility Law, renaming of names, and the assumption that the names a, b do not occur free in E. For this construction to work it is crucial that conjunction of expressions is not idempotent.[5] Hence

$$\exists ax \doteq a \wedge \exists ax \doteq a \equiv \exists a \exists b(x \doteq a \wedge x \doteq b) \equiv \exists a \exists b(\bot) \equiv \bot$$

does not imply $\exists ax \doteq a \equiv \bot$.

The above treatment of names, which we first published in [12], usually puzzles people a lot on first sight. It is related to the treatment of names in the π-calculus [21], even so the π-calculus does not distinguish between variables and names. We need this distinction because of the presence of constraints. A treatment of names similar to ours but in the context of an extended lambda calculus can be found in [26].

4.4 First-class Abstraction

The setup of Calculus A makes it straightforward to accommodate abstractions as first-class citizens. We just forget the defined predicate symbols and use names instead: abstraction now takes the form $a{:}\overline{x}/E$, and application becomes $a\overline{x}$. If we also allow applications of the form $x\overline{y}$ and assume the congruence law

$$x \doteq a \wedge E \equiv x \doteq a \wedge E[a/x] \qquad \text{if } a \text{ is free for } x \text{ in } E,$$

the higher-order programming techniques known from functional programming become available.

Example 4.2 *The following expression defines a function f that takes a predicate P as argument and returns a predicate Q, which holds iff its argument L is a list whose elements all satisfy P:*

$$\begin{aligned} f{:}P\,Q \,/\, \exists a\,(Q = a \wedge a{:}L \,/\, & \; if\ L \doteq nil\ then\ \top \\ & \; else\ \exists H \exists R(L \doteq H.R \wedge P\,H \wedge Q\,R)\ fi). \end{aligned}$$

[5] Incidentally, Linear Logic has a nonidempotent conjunction-like connective.

Generalized abstraction and application do not destroy the logical semantics of Calculus A (however, quantification of names does). By assuming a predicate symbol *apply* for every arity, we can translate generalized abstractions and applications to first-order formulas:

$$\begin{aligned} a{:}\overline{x}/E &\implies \forall \overline{x}\,(apply(a\overline{x}) \leftrightarrow E) \\ a\overline{x} &\implies apply(a\overline{x}) \\ x\overline{y} &\implies apply(x\overline{y}). \end{aligned}$$

Under this translation, the unfolding rule remains an equivalence transformation. From an operational point of view the congruence

$$\exists a(a{:}x/E) \quad \equiv \quad \top$$

seems reasonable: it allows throwing away abstractions that cannot be referred to anymore. This congruence will, for instance, enable the reduction of the conditional

$$if\ \exists a(a{:}x/E)\ then\ F\ else\ G\ fi \quad \to \quad F,$$

which otherwise would be irreducible. The Annulment Law of Calculus B subsumes the above congruence.

5 Calculus B

This section gives a self-contained definition of Calculus B.

5.1 Constraint Systems

Constraint systems as employed by Calculus B are based 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 formula over its signature a **constraint**. We use $\bot$ for the constraint that is always false, and $\top$ for the constraint

x, y, z	:		*variable*	
a, b, c	:		*name*	
ϕ	:		*constraint*	
u, v, w	::=	$x \mid a$		
E	::=	ϕ	*constraint*	
		$E_1 \wedge E_2$	*composition*	
		$\exists u\ E$	*declaration*	
		$a{:}\overline{x}/E$	*abstraction*	($\overline{x}$ *linear*)
		$u\overline{v}$	*application*	
		if D *else* E	*conditional*	
		or (D)	*disjunction*	
C	::=	E_1 *then* $E_2 \mid \exists u\ C$	*clause*	
D	::=	$C \mid \bot \mid D_1 \vee D_2$	*collection*	

Figure 7: Abstract syntax of Calculus B.

that is always true. Moreover, we will use the following relationships for constraints:

$$\begin{aligned} \phi \models\!\!\models_\Delta \psi \quad &:\Longleftrightarrow \quad \forall\,(\phi \leftrightarrow \psi) \text{ is valid in every model of } \Delta \\ \phi \models_\Delta \psi \quad &:\Longleftrightarrow \quad \phi \models\!\!\models_\Delta \phi \wedge \psi \\ \phi \textbf{ satisfiable} \quad &:\Longleftrightarrow \quad \phi \not\models_\Delta \bot. \end{aligned}$$

5.2 Syntax

The abstract syntax of Calculus B appears in Figure 7. It supposes that some constraint system is given, fixing infinite sets of variables, names and constraints.

We use $\overline{x}$ to denote a possibly empty sequence of variables. A sequence $\overline{x}$ is called **linear** if its elements are pairwise distinct.

An expression $a{:}\overline{x}/E$ represents a binding of the name a to the abstraction $\overline{x}/E$. For convenience, we call the entire expression $a{:}\overline{x}/E$ an "abstraction". We sometimes write $a{:}\alpha$, where $\alpha = \overline{x}/E$.

The syntactic category D represents multisets of clauses, where $\bot$ stands for the empty multiset and $\vee$ for multiset union.

We identify a conjunction $\phi_1 \wedge \phi_2$ of two constraints with the corresponding composition of constraints, and an existential quantification $\exists x\,\phi$ of a constraint ϕ with the corresponding declaration.

Calculus B has the following constructs for binding variables and names:

- A declaration $\exists u\, E$ binds u (a variable or a name) with scope E.
- An abstraction $a{:}\overline{x}/E$ binds its formal arguments $\overline{x}$ with scope E.
- A clausal declaration $\exists u\, C$ binds u (a variable or a name) with scope C.
- Quantification of constraints (as in Predicate Logic).

The **free variables** and **free names** of an expression are defined accordingly. We use $\mathcal{F}E$ to denote the set of variables and names occurring free in E.

5.3 Structural Congruence

A **congruence** is an equivalence relation on the expressions of Calculus B (i.e., the syntactic categories ϕ, E, C, and D) that is compatible with all syntactic combinators (e.g., if $E_1 \equiv E_1'$ and $E_2 \equiv E_2'$, then $E_1 \wedge E_2 \equiv E_1' \wedge E_2'$). The **structural congruence** "$E_1 \equiv E_2$" of Calculus B is defined as the least congruence satisfying the congruence laws in Figure 8.

The notation $E[u/x]$ stands for the expression that is obtained from E by replacing every free occurrence of x with u.

5.4 Reduction

The reduction relation of Calculus B is defined as the least relation "$E_1 \to E_2$" on expressions satisfying the structural reduction laws in Figure 9 and the reduction rules in Figure 10. An instance $E \to E'$ of the reduction relation expresses that E' can be obtained from E by one reduction step.

The structural reduction laws (Figure 9) say where the reduction rules (Figure 10) can be applied: everywhere but within abstractions, else constituents of conditionals, and then constituents of clauses. The first structural reduction law

$$\frac{E_1 \equiv E_2 \quad E_2 \to E_2' \quad E_2' \equiv E_3}{E_1 \to E_3}$$

says that the reduction rules can be applied modulo structural congruence, that is, an expression can be rewritten according to the congruence laws in Figure 8 before and after a reduction rule is applied.

The Unfolding Rule should be clear from Calculus A. The Failure Rule fails a local computation space, which means that the associated clause is discarded. The first rule for conditionals reduces the conditional with a clause whose guard is entailed (see Proposition 5.4). The second rule for conditionals reduces the conditional to the else constituent in case all clauses are failed. The first rule for disjunctions reduces a disjunction that has only one clause left (recall that failed clauses are discarded by the failure rule). The second rule reduces a disjunction that has no clause left to the constraint $\bot$. The third rule reduces a disjunction with an entailed clause whose body is the constraint $\top$ to $\top$.

Renaming

- $E_1 \equiv E_2$ if E_1 and E_2 are equal up to consistent renaming of bound variables and names

Composition and Collection

- $\wedge$ is associative, commutative and satisfies $E \wedge \top \equiv E$
- $\vee$ is associative, commutative and satisfies $D \vee \bot \equiv D$

Declaration

- $\exists u\, \exists v\ E \ \equiv\ \exists v\, \exists u\ E$
- $\exists u\, \exists v\ C \ \equiv\ \exists v\, \exists u\ C$
- $\exists u\ E_1 \wedge E_2 \ \equiv\ \exists u\ (E_1 \wedge E_2)$ if u does not occur free in E_2
- $\exists u\, E_1\ then\ E_2 \ \equiv\ \exists u\ (E_1\ then\ E_2)$ if u does not occur free in E_2

Simplification

- $\phi_1 \ \equiv\ \phi_2$ if $\phi_1 \models\!\!\!|_{\Delta}\ \phi_2$

Equality

- $x \doteq u \wedge E \ \equiv\ x \doteq u \wedge E[u/x]$ if u is free for x in E

Propagation

- $\pi \wedge if\ \exists \overline{u}\ (E_1\ then\ E_2) \vee D\ else\ E_3 \ \equiv\ \pi \wedge if\ \exists \overline{u}\ (\pi \wedge E_1\ then\ E_2) \vee D\ else\ E_3$
- $\pi \wedge or\ (\exists \overline{u}\ (E_1\ then\ E_2) \vee D) \ \equiv\ \pi \wedge or\ (\exists \overline{u}\ (\pi \wedge E_1\ then\ E_2) \vee D)$

  if π is a constraint or an abstraction with $\mathcal{F}\pi \cap \mathcal{F}\overline{u} = \emptyset$

Replication

- $a{:}\,\alpha \ \equiv\ a{:}\,\alpha \wedge a{:}\,\alpha$

Annulment

- $\exists \overline{x}\, \exists \overline{a}\, \exists \overline{b}\, (\phi \wedge \overline{a}{:}\,\overline{\alpha}) \ \equiv\ \top$ if $\exists \overline{x}\, \phi \models\!\!\!|_{\Delta}\ \top$

Figure 8: Congruence laws of Calculus B.

$$\frac{E_1 \equiv E_2 \quad E_2 \to E_2' \quad E_2' \equiv E_3}{E_1 \to E_3} \qquad \frac{E_1 \to E_1'}{E_1 \wedge E_2 \to E_1' \wedge E_2} \qquad \frac{E \to E'}{\exists u\, E \to \exists u\, E'}$$

$$\frac{D \to D'}{if\ D\ else\ E \to if\ D'\ else\ E} \qquad \frac{D \to D'}{or\,(D) \to or\,(D')}$$

$$\frac{D_1 \to D_1'}{D_1 \vee D_2 \to D_1' \vee D_2} \qquad \frac{E_1 \to E_1'}{E_1\ then\ E_2 \to E_1'\ then\ E_2} \qquad \frac{C \to C'}{\exists u\, C \to \exists u\, C'}$$

Figure 9: Structural reduction laws of Calculus B.

Unfolding

- $a\overline{u} \wedge a{:}\overline{x}/E \ \to\ E[\overline{u}/\overline{x}\,] \ \wedge\ a{:}\overline{x}/E$

  if $\overline{x}$ and $\overline{u}$ are of equal length and $\overline{u}$ is free for $\overline{x}$ in E

Failure

- $\exists\overline{u}\ (\bot \wedge E_1\ then\ E_2) \ \to\ \bot$

Conditional

- $if\ \exists\overline{u}\ (E_1\ then\ E_2) \vee D\ else\ E_3 \ \to\ \exists\overline{u}\ (E_1 \wedge E_2)$ if $\exists\overline{u}\ E_1 \equiv \top$
- $if\ \bot\ else\ E \ \to\ E$

Disjunction

- $or\,(\exists\overline{u}\ (E_1\ then\ E_2)) \ \to\ \exists\overline{u}\ (E_1 \wedge E_2)$
- $or\,(\bot) \ \to\ \bot$
- $or\,(\top\ then\ \top \vee D) \ \to\ \top$

Figure 10: Reduction rules of Calculus B.

Example 5.1 *Consider the expression*

$$\exists x\, \exists y\, (\exists a\, (x \doteq a) \ \wedge\ \exists a\, (y \doteq a) \ \wedge\ if\ x \doteq y\ then\ E_1\ else\ E_2)$$

and suppose that x and y are distinct variables that do not occur free in E_1 and E_2. Moreover, assume that a and b are two distinct names not occurring free in E_1 and E_2. We will show that this expression reduces in two steps to E_2.

First, we move the left declaration of the name 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_1 and E_2.

$$\equiv\ \exists a\, \exists x\, \exists y\, (x \doteq a \ \wedge\ \exists a\, (y \doteq a) \ \wedge\ if\ x \doteq y\ then\ E_1\ else\ E_2)$$

Next we apply the Equality Law to $x \doteq a$.

$$\equiv\ \exists a\, \exists x\, \exists y\, (x \doteq a \ \wedge\ \exists a\, (y \doteq a) \ \wedge\ if\ a \doteq y\ then\ E_1\ else\ E_2)$$

Now we move the declaration of x inside using the laws for composition and declaration (we exploit that x does not occur free in E_1 and E_2 and that x is different from y).

$$\equiv\ \exists a\, \exists y\, (\exists x\, (x \doteq a) \ \wedge\ \exists a\, (y \doteq a) \ \wedge\ if\ a \doteq y\ then\ E_1\ else\ E_2)$$

Since $\exists x\, (x \doteq a)$ is a constraint and $\exists x\, (x \doteq a) \models\!\!\!\models_\Delta \top$, we can delete $\exists x\, (x \doteq a)$ using the Simplification Law and the laws for compositions (in particular $E \wedge \top \equiv E$).

$$\equiv\ \exists a\, \exists y\, (\exists a\, (y \doteq a) \ \wedge\ if\ a \doteq y\ then\ E_1\ else\ E_2)$$

Next we rename the inner name a to the different name b using the Renaming Law.

$$\equiv\ \exists a\, \exists y\, (\exists b\, (y \doteq b) \ \wedge\ if\ a \doteq y\ then\ E_1\ else\ E_2)$$

This brings us in a position where we can eliminate $\exists b\, (y \doteq b)$ in the same way we did it before for $\exists a\, (x \doteq a)$.

$$\equiv\ \exists a\, \exists b\, (if\ a \doteq b\ then\ E_1\ else\ E_2)$$

Now, since $a \doteq b \models\!\!\!\models_\Delta \bot$, we obtain

$$\begin{aligned} &\equiv && \exists a\, \exists b\, (if\ \bot \wedge \bot\ then\ E_1\ else\ E_2) \\ &\to && \exists a\, \exists b\, (if\ \bot\ else\ E_2) \\ &\to && \exists a\, \exists b\, E_2 \end{aligned}$$

using the Simplification Law, the Failure Rule, and the second rule for the conditional. It remains to get rid of the declarations of the names a and b. This can be done using the Annulment Law together with the laws for compositions and declarations:

$$\equiv\ \exists a\, \exists b\, (\top \wedge E_2)\ \equiv\ (\exists a\, \exists b\, \top)\ \wedge\ E_2\ \equiv\ \top\ \wedge\ E_2\ \equiv\ E_2.$$

Example 5.2 *This example shows the reason for equipping Calculus B with the Replication Law. Consider the derivation*

$$\begin{array}{rl} & a{:}\alpha \ \wedge \ or\,(E_1 \ then \ E_2) \\ \equiv & a{:}\alpha \ \wedge \ or\,(a{:}\alpha \wedge E_1 \ then \ E_2) \\ \rightarrow & a{:}\alpha \ \wedge \ a{:}\alpha \ \wedge \ E_1 \ \wedge \ E_2 \\ \equiv & a{:}\alpha \ \wedge \ E_1 \ \wedge \ E_2. \end{array}$$

The first step is by the propagation law for disjunctions, the second step is by the first reduction rule for disjunctions, and the third step is by the Replication Law. Without the Replication Law it would be impossible to get rid of the second copy of the abstraction $a{:}\alpha$.

Example 5.3 *The Annulment Law reconciles first-class abstraction with deep guards. To see this, consider the reduction*

$$if \ \exists x \, \exists a \, (x \doteq a \ \wedge \ a{:}y / y \doteq x) \ then \ E_1 \ else \ E_2 \quad \rightarrow \quad E_1$$

which is justified by the first rule for conditionals and the fact that

$$\exists x \, \exists a \, (x \doteq a \ \wedge \ a{:}y / y \doteq x) \quad \equiv \quad \top$$

is an instance of the Annulment Law.

The next propostition says that conditionals can reduce with clauses whose guards are entailed.

Proposition 5.4 *Suppose $\phi_1 \models_\Delta \exists \overline{x} \, \phi_2$. Then*

$$\phi_1 \ \wedge \ if \ \exists \overline{x} \, (\phi_2 \ then \ E_1) \ else \ E_2 \quad \rightarrow \quad \phi_1 \ \wedge \ \exists \overline{x} \, (\phi_2 \ \wedge \ E_1).$$

Proof. Because of the Renaming Law we can assume without loss of generality that no variable in $\overline{x}$ occurs in ϕ_1. It suffices to show that there exists a constraint ϕ_3 such that $\phi_1 \wedge \phi_2 \models\!\!\models_\Delta \phi_1 \wedge \phi_3$ and $\exists \overline{x} \, \phi_3 \models\!\!\models_\Delta \top$ since

$$\begin{array}{rl} \phi_1 \ \wedge \ if \ \exists \overline{x} \, (\phi_2 \ then \ E_1) \ else \ E_2 & \equiv \quad \phi_1 \ \wedge \ if \ \exists \overline{x} \, (\phi_1 \wedge \phi_2 \ then \ E_1) \ else \ E_2 \\ & \equiv \quad \phi_1 \ \wedge \ if \ \exists \overline{x} \, (\phi_1 \wedge \phi_3 \ then \ E_1) \ else \ E_2 \\ & \equiv \quad \phi_1 \ \wedge \ if \ \exists \overline{x} \, (\phi_3 \ then \ E_1) \ else \ E_2 \\ & \rightarrow \quad \phi_1 \ \wedge \ \exists \overline{x} \, (\phi_3 \ \wedge \ E_1) \\ & \equiv \quad \exists \overline{x} \, (\phi_1 \ \wedge \ \phi_3 \ \wedge \ E_1) \\ & \equiv \quad \exists \overline{x} \, (\phi_1 \ \wedge \ \phi_2 \ \wedge \ E_1) \\ & \equiv \quad \phi_1 \ \wedge \ \exists \overline{x} \, (\phi_2 \ \wedge \ E_1). \end{array}$$

Let $\phi_3 := \phi_1 \rightarrow \phi_2$ (here $\rightarrow$ is implication, not reduction). Then $\phi_1 \wedge \phi_2 \models\!\!\models_\Delta \phi_1 \wedge \phi_3$ is obviously satisfied. Moreover, $\exists \overline{x} \, \phi_3 \models\!\!\models_\Delta \exists \overline{x} \, (\phi_1 \rightarrow \phi_2) \models\!\!\models_\Delta \phi_1 \rightarrow \exists \overline{x} \, \phi_2 \models\!\!\models_\Delta \top$ since $\phi_1 \models_\Delta \exists \overline{x} \, \phi_2$.

6 Constraint Communication

Calculus B provides for stream-based communication, which is the established form of communication in concurrent logic programming [30]. From a theoretical point of view, stream communication is nice since it comes for free, that is, without further primitives. From a practical point of view, we are however dissatisfied with both the expressivity and efficiency of stream-based communication. Streams and their problems are carefully discussed in [15], where a new communication mechanism, called ports, is proposed for use with AKL. Our search for a better form of communication for Oz finally led us to constraint communication [12, 32]. As we show in [12, 32], constraint communication introduces a notion of state that is fully compatible with logical constraints and concurrency.

We extend the abstract syntax of Calculus B with three new expressions called **communication tokens**:

$$\begin{array}{rcll} E & ::= & \ldots & \\ & | & a & a \textit{ is channel} \\ & | & u\,!\,v & \textit{put } u \textit{ on } v \\ & | & u?\,v & \textit{get } u \textit{ from } v. \end{array}$$

The semantics of the new primitives is given by the **communication rule**:

$$u\,!\,a \wedge v?\,a \wedge a \quad \rightarrow \quad u \doteq v \wedge a.$$

Moreover, we generalize the Annulment Law of Calculus B to

$$\exists \overline{x}\, \exists \overline{a}\, \exists \overline{b}\, \exists \overline{c}\, (\phi \ \wedge \ \overline{a}{:}\overline{\alpha} \ \wedge \ \overline{c} \ \wedge \ \overline{u}_1\,!\,\overline{c}_1 \ \wedge \ \overline{u}_2?\,\overline{c}_2) \ \equiv \ \top$$

$$\text{if } \exists \overline{x}\, \phi \models\!\!\!\models_\Delta \top, \text{ and } \overline{c}_1 \text{ and } \overline{c}_2 \text{ are disjoint and contained in } \overline{c}$$

so that it provides for the annulment of communication tokens. This formulation of the Annulment Law provides for a straightforward implementation of constraint communication.

An example of an instance of the generalized Annulment Law is

$$\exists x\, \exists a\, \exists c\, (a{:}\, y/x?\, c \ \wedge \ c \ \wedge \ a\,!\,c \ \wedge \ x \doteq a) \quad \equiv \quad \top.$$

The next two examples show typical usages of constraint communication. For a further discussion of its expressivity we refer the reader to [12, 32, 11].

Example 6.1 *We assume the constraint system $\mathcal{H}$ and a unary function symbol m. The expression*

$$a{:}\, x\, y \,/\, \exists z\ (z?\, x \ \wedge \ \textit{if } \exists u\, z \doteq m(u) \textit{ then } z\,!\, y \wedge a x\, y \textit{ else } \top)$$

defines a procedure a that takes two channels x, y as arguments and transfers messages from x to y. It is assumed that messages take the form $m(\ldots)$. The conditional synchronizes upon the arrival of a message on the input channel x.

Given the above abstraction, the expression $axz \wedge ayz$ merges two channels x and y into a channel z.

Example 6.2 *We assume the constraint system $\mathcal{H}$ and two constants* 1 *and* 2*. The expression*

$$\begin{aligned}\exists c\,(\,c \wedge 1\,!\,c \wedge a{:}x \,/\, \exists z(\quad & z?c \wedge \\ & \mathit{if}\ (z \doteq 1\ \mathit{then}\ x \doteq 1 \wedge 2\,!\,c) \\ & \vee\ (z \doteq 2\ \mathit{then}\ x \doteq 2 \wedge 1\,!\,c) \\ & \mathit{else}\ \top\,))\end{aligned}$$

defines a procedure a *that returns alternatingly the constants* 1 *and* 2*. Clearly,* a *is a procedure with state.*

Acknowledgements

Frequent discussions with Martin Müller, Joachim Niehren, Christian Schulte, Ralf Treinen helped to get things right. Torkel Franzen from SICS discovered a problem with an earlier version of the Annulment Law of Calculus B. Andreas Podelski read a draft and provided comments.

The research reported in this paper 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).

Remark

The Oz System and its documentation are available through anonymous ftp from `duck.dfki.uni-sb.de`.

References

[1] Hassan Aït-Kaci, Andreas Podelski, and Gert Smolka. A feature-based constraint system for logic programming with entailment. *Theoretical Computer Science*, 122, January 1994.

[2] Frédéric Benhamou and Alain Colmerauer, editors. *Constraint Logic Programming: Selected Research*. The MIT Press, Cambridge, Mass., 1993.

[3] Gérard Berry and Gérard Boudol. The chemical abstract machine. In *Proceedings of the 17th ACM Conference on Principles of Programming Languages*, pages 81–94, San Francisco, CA, January 1990.

[4] Weidong Chen, Michael Kifer, and David S. Warren. Hilog: A foundation for higher-order logic programming. *Journal of Logic Programming*, 15:187–230, 1993.

[5] K.L. Clark and S. Gregory. A relational language for parallel programming. In *Proc. of the ACM Conference on Functional Programming Languages and Computer Architecture*, pages 171–178, 1981.

[6] A. Colmerauer, H. Kanoui, and M. Van Caneghem. Prolog, theoretical principles and current trends. *Technology and Science of Informatics*, 2(4):255–292, 1983.

[7] Frank S. de Boer and Catuscia Palamidessi. A process algebra of concurrent constraint programming. In Krzysztof Apt, editor, *Proceedings of the Joint International Conference and Symposium on Logic Programming*, pages 463–477, Washington, USA, 1992. The MIT Press.

[8] Nachum Dershowitz and Jean-Pierre Jouannaud. Rewrite systems. In *Handbook of Theoretical Computer Science*, volume B, chapter 15. North–Holland, Amsterdam, Holland, 1990.

[9] S. Haridi and S. Janson. Kernel Andorra Prolog and its computation model. In D.H.D. Warren and P. Szeredi, editors, *Logic Programming, Proceedings of the 7th International Conference*, pages 31–48, Cambridge, MA, June 1990. The MIT Press.

[10] Seif Haridi, Sverker Janson, and Catuscia Palamidessi. Structural operational semantics for AKL. *Future Generation Computer Systems*, pages 409–421, 1992.

[11] 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. Available through anonymous ftp from `duck.dfki.uni-sb.de`.

[12] Martin Henz, Gert Smolka, and Jörg Würtz. Oz—a programming language for multi-agent systems. In Ruzena Bajcsy, editor, *13th International Joint Conference on Artificial Intelligence*, volume 1, pages 404–409, Chambéry, France, 30 August–3 September 1993. Morgan Kaufmann Publishers.

[13] Joxan Jaffar and Jean-Louis Lassez. Constraint logic programming. In *Proceedings of the 14th ACM Symposium on Principles of Programming Languages*, pages 111–119, Munich, Germany, January 1987.

[14] Sverker Janson and Seif Haridi. Programming paradigms of the Andorra kernel language. In Vijay Saraswat and Kazunori Ueda, editors, *Logic Programming, Proceedings of the 1991 International Symposium*, pages 167–186, San Diego, USA, 1991. The MIT Press.

[15] Sverker Janson, Johan Montelius, and Seif Haridi. Ports for objects. In *Research Directions in Concurrent Object-Oriented Programming*. The MIT Press, Cambridge, Mass., 1993.

[16] Jean-Louis Lassez, Michael J. Maher, and Kim Marriott. Unification revisited. In Jack Minker, editor, *Foundations of Deductive Databases and Logic Programming*. Morgan Kaufmann, Los Altos, CA, 1988.

[17] J. W. Lloyd. *Foundations of Logic Programming*. Symbolic Computation. Springer-Verlag, Berlin, Germany, second, extended edition, 1987.

[18] Michael J. Maher. Logic semantics for a class of committed-choice programs. In Jean-Louis Lassez, editor, *Logic Programming, Proceedings of the Fourth International Conference*, pages 858–876, Cambridge, MA, 1987. The MIT Press.

[19] Michael J. Maher. Complete axiomatizations of the algebras of finite, rational and infinite trees. In *Proceedings of the 3rd Annual Symposium on Logic in Computer Science*, pages 348–457, Edinburgh, Scotland, July 1988.

[20] Michael J. Maher. A logic programming view of CLP. In David S. Warren, editor, *Proceedings of the 10th DInternational Conference on Logic Programming*, pages 737–753, Budapest, Hungary, June 1993. The MIT Press.

[21] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, I. *Information and Computation*, 100(1):1–40, September 1992.

[22] Gopalan Nadathur and Dale Miller. An overview of λProlog. In Robert A. Kowalski and Kenneth A. Bowen, editors, *Proceedings of the Fifth International Conference and Symposium on Logic Programming*, pages 810–827, Seattle, Wash., 1988. The MIT Press.

[23] Lee Naish. Negation and quantifiers in NU-Prolog. In Ehud Shapiro, editor, *Proceedings of the Third International Conference on Logic Programming*, Lecture Notes in Computer Science, pages 624–634, London, 1986. Springer-Verlag.

[24] Joachim Niehren and Gert Smolka. Functional computation in a calculus of relational abstraction and application. Research Report RR-94-04, DFKI, D-66123 Saarbrücken, Germany, March 1994.

[25] R. A. O'Keefe. On the treatment of cuts in Prolog source-level tools. In *Symposium on Logic Programming*, pages 68–72. IEEE Computer Society, Technical Committee on Computer Languages, The Computer Society Press, July 1985.

[26] 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, Copenhagen, Denmark, June 1993. Has appeared as Document YALEU/DCS/RR-968, Department of Computer Science, Yale University, Conn.

[27] V. A. Saraswat. *Concurrent Constraint Programming Languages*. PhD thesis, School of Comp. Sc., Carnegie-Mellon University, Pittsburgh, CA, 1989.

[28] V.A. Saraswat and M. Rinard. Concurrent constraint programming. In *Proceedings of the 7th Annual ACM Symposium on Principles of Programming Languages*, pages 232–245, San Francisco, CA, January 1990.

[29] Vijay Saraswat. The category of constraint systems is Cartesian-closed. In *Proceedings, Seventh Annual IEEE Symposium on Logic in Computer Science*, pages 341–345, Santa Cruz, California, June 22–25 1992. IEEE Computer Society Press.

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

[31] Gert Smolka. Residuation and guarded rules for constraint logic programming. In Frédéric Benhamou and Alain Colmerauer, editors, *Constraint Logic Programming: Selected Research*, chapter 22, pages 405–419. The MIT Press, Cambridge, Mass., 1993.

[32] Gert Smolka, Martin Henz, and Jörg Würtz. Object-oriented concurrent constraint programming in Oz. Research Report RR-93-16, DFKI, Stuhlsatzenhausweg 3, D-66123 Saarbrücken, Germany, April 1993. Will appear in: P. van Hentenryck and V. Saraswat (eds.), Principles and Practice of Constraint Programming, The MIT Press, Cambridge, Mass.

[33] Gert Smolka and Ralf Treinen. Records for logic programming. In Krzysztof Apt, editor, *Proceedings of the Joint International Conference and Symposium on Logic Programming*, pages 240–254, Washington, USA, 1992. The MIT Press. Full version has appeared as Research Report RR-92-23, DFKI, Stuhlsatzenhausweg 3, 66123 Saarbrücken, Germany; will also appear in Journal of Logic Programming.