Typed Concurrent Programming with Logic Variables

Martin Müller, **Joachim Niehren**, **Gert Smolka**
Programming Systems Lab,
Universität des Saarlandes and DFKI
D-66041 Saarbrücken, Germany
{mmueller,niehren,smolka}@ps.uni-sb.de

Abstract

We present a concurrent higher-order programming language called Plain and a concomitant static type system. Plain is based on logic variables and computes with possibly partial data structures. The data structures of Plain are procedures, cells, and records. Plain's type system features record-based subtyping, bounded existential polymorphism, and access modalities distinguishing between reading and writing.

1 Introduction

We present a concurrent higher-order programming language called Plain and a concomitant static type system. Plain is derived from the Oz programming model (OPM) [36], which formulates the essence of the programming language Oz [30]. OPM extends the concurrent constraint model [31, 17] with first-class procedures and first-class cells. Plain has been designed as an OPM-style language with the objectives that it should

- have a strong static type system with record-based subtyping and higher-order polymorphism, but
- retain most of OPM's expressiveness as far as concurrent, functional, and object-oriented programming are concerned.

Higher-order polymorphic types (in contrast to ML type schemes) are required for data structures that embed polymorphic procedures, for example to describe modules, and for cells that contain polymorphic data structures. They are also required to type check certain higher-order programming abstractions, for instance in the context of typed object-oriented programming; there, the combination of higher-order polymorphic types with subtyping is particularly convenient (for references and examples, see [1]).

As it turns out, both objectives can be met by simplifying OPM's store model such that it does not store equations between variables. Plain comes with a self-contained store model which replaces the more abstract model of a constraint store. The decision to not store variable equalities makes Plain a considerable restriction of OPM. But Plain does support partial data structures, namely records with embedded and possibly unbound logic variables. This retains the expressiveness of logic variables for synchronization and communication between concurrent threads [33], as well as the ease with which complex and possibly cyclic data structures can be built collaboratively by concurrent threads.

Plain's type system employs record-based subtyping, bounded existential polymorphism [5, 9], and access modalities (modes), which have been introduced for channels by Pierce and Sangiorgi [28]; we adapt their system to a language with logic variables. Modes for logic variables are essential to make the type system work. Neither logic variables nor procedures in constraint programming impose a static distinction between input and output, even though it is often made implicitly. However, this distinction is essential for any type system that provides a non-trivial order on types, such as the subtyping order and the instantiation order on polymorphic types: Outputs of a (more specific, smaller) subtype can be used as inputs of a (less specific, greater) supertype, and instantiation of polymorphic types must occur along the data flow [23]. (Note in passing that one can provide OPM with an ML-style polymorphic type system subject to Wright's restriction of polymorphic generalization [18, 39]. This system does not require static data flow information, but it rules out many higher programming abstraction, *e.g.*, in the object system.)

Static type checking in a system with ordered types requires that the data flow is statically known. In functional languages this is ensured by the restricted applicative syntax. It is not the case for OPM which relies on unification as a central computational concept. Plain solves this technical problem by replacing unification with a directed equation that we call assignment. Furthermore, Plain modifies the operational semantics of OPM such that it does not rely on unification anymore. In particular, such a modification was needed for the semantics of cells.

In summary, Plain shows under which preconditions standard approaches to strong typing carry over to OPM, and it marks a starting point from which strongly typed OPM-style languages can be developed.

Logic Variables and Channels. Plain is in size and expressiveness well-comparable with Pict [29], a recent concurrent programming language based on the π-calculus [19, 20]. Plain's type system is very close to Pict's and in fact inspired by it; Pict's type system in turn is based on type systems for functional programming (in particular, $F^{\omega}_{\leq}$; see [12, 6] and [29] for further references).

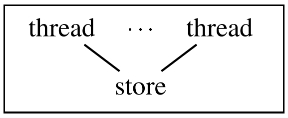
The design of Plain contributes to the comparison of the most prominent models for concurrent programming: based on constraints and on process calculi. The π-calculus is designed as a minimal base for concurrent computation. Its essential primitive is channel communication which can express concurrent versions of data structures and procedures. This minimality is intriguing from a foundational perspective, but of limited practical use. When designing a high-level languages, many basic programming abstractions must be encoded. The join calculus [11], a variant of the π-calculus is superior in this respect as it has direct support for a procedural form (the "join").

Following OPM, Plain provides essential programming primitives directly: First-class procedures, records, and cells. Addition of sequential composition and thread creation is straightforward. Due to logic variables, there is no need for a dedicated communication primitive. Once a logic variable is bound to a data structure it becomes indistinguishable from it. This is in contrast to channels which remain distinct from the data structure they receive. Channels and locks can be expressed in Plain as synchronized data structures. Our programming experience with Oz shows that concurrent threads typically communicate through custom-built synchronized objects, where the combination of data driven synchronization based on logic variables, sequential composition and locks proves essential [14]. Plain can conveniently express Pict programs as we illustrate by a simple encoding of channels in Plain. However, it requires major effort to express in Pict partial data structures and data driven synchronization through logic variables.

Overview. Section 2 presents untyped Plain, and Section 3 gives the type system and the type preservation result. Section 4 illustrates Plain by means of a well-typed polymorphic procedure that creates data structures modeling channels. Section 5 summarizes the changes of Plain with respect to OPM. Sections 6 and 7 mention further related work and some future work.

2 Plain

Computation in Plain is organized in terms of *threads* over a *shared store*. Each thread represents a (largely) independent concurrent computation with its own control structure. Threads proceed according to an interleaving semantics: That is, reduction steps are atomic and do not overlap in time. Concurrent threads communicate and synchronize with each other through logic variables in the store. Threads block automatically until all input variables are bound to data structures. The store binds variables to data structures (records, procedures, or cells) which may be partial and contain unbound variables. Variable bindings are never retracted. In contrast, the content of cells may be altered in order to support stateful computation. Blocking threads are woken when their input variables become bound. Since bindings are not retracted, this synchronisation condition is safe: If it holds true once, it will stay true forever. This also yields a simple *fairness condition*: Every reducible thread must eventually be reduced.



Syntax. We assume an infinite set Var of *variables* ranged over by x, y, z and an infinite set of *labels* ranged over by a, b, c. We write $\bar{x}$ for a finite sequence of variables, $\bar{a}$ for a finite sequence of labels, and $\bar{a}{:}\bar{y}$ for a finite sequence of pairs $a_1{:}y_1, \ldots, a_n{:}y_n$ (where $n \geq 0$). We will freely use an analogous sequence notation for other syntac categories.

The syntax of Plain's data structures and statements is given in Figure 1. A *data structure D* is a procedure, or a cell, or a record. A (monadic) *procedure* $(\mathsf{proc}(x)\, S)$ consists of a *formal argument* x and a *body* S. A *cell* $(\mathsf{cell}\; x)$ is a container with *contents* x. A *record* $\{\bar{a}{:}\bar{y}\}$ has *fields* $\bar{y}$ that can be accessed by pairwise distinct labels $\bar{a}$. The sets of structures Dat is the union of the sets of procedures Proc,

$$
\begin{array}{rcl}
D & ::= & (\mathsf{proc}(y)\,S) \;\mid\; (\mathsf{cell}\;y) \;\mid\; \{\bar{a}{:}\bar{y}\} \\
S & ::= & (\mathsf{local}(x)\,S) \;\mid\; \mathsf{skip} \;\mid\; S\,|\,S' \;\mid\; x{:=}y \;\mid\; x{:=}D \;\mid \\
 & & (x\,y) \;\mid\; (\mathsf{exch}\;x\;y\;(z)\;S) \;\mid\; (\mathsf{match}\;x\;(\{\bar{a}{:}\bar{y}\})\;S) \\
C & ::= & V\sigma[]\;S
\end{array}
$$

Figure 1: Syntax of Data Structures, Statements, and Configurations

$$
\begin{array}{l}
S_1\,|\,S_2 \wedge S_2\,|\,S_1 \qquad (S_1\,|\,S_2)\,|\,S_3 \wedge S_2\,|\,(S_2\,|\,S_3) \qquad \mathsf{skip}\,|\,S \wedge S \wedge S\,|\,\mathsf{skip} \\
\{\ldots a_1{:}y_1 \ldots a_2{:}y_2 \ldots\} \;\wedge\; \{\ldots a_2{:}y_2 \ldots a_1{:}y_1 \ldots\} \\
(\mathsf{match}\;x\;(\{\ldots a_1{:}y_1 \ldots a_2{:}y_2 \ldots\})\;S) \;\wedge\; (\mathsf{match}\;x\;(\{\ldots a_2{:}y_2 \ldots a_1{:}y_1 \ldots\})\;S)
\end{array}
$$

Figure 2: Structural Congruence

cells Cell, and records Rec.[1]

Amongst the *statements* S there is *declaration* $(\mathsf{local}(x)\,S)$, the *null statement* skip, and *parallel composition* $S_1\,|\,S_2$ of statements. Furthermore, there are two assignment forms: A *synchronized assignment* $x{:=}y$ of y to x, and a *data assignment* $x{:=}D$ of D to x. Next, there is *application* $(x\;y)$ of x with *actual paramenter* y, a *cell exchange* $(\mathsf{exch}\;x\;y\;(z)\;S)$ on x with new content y for some old content z, and a statement $(\mathsf{match}\;x\;(\{\bar{a}{:}\bar{y}\})\;S)$ *matching* x against the *pattern* $(\{\bar{a}{:}\bar{y}\})$. Note that both cell exchange and matching have a *continuation* S. (For conciseness, we leave out a case statement with multiple clauses.)

Procedures $(\mathsf{proc}(z)\,S)$, declarations $(\mathsf{local}(z)\,S)$ and exchange statements $(\mathsf{exch}\;x\;y\;(z)\;S)$ each bind z with scope S, and a clause $(\mathsf{match}\;x\;(\{\bar{a}{:}\bar{y}\})\;S)$ binds the *pattern variables* $\bar{y}$ with scope S. The set of *bound* and *free variables* in statements are defined accordingly and denoted with $fv(S)$ and $bv(S)$, respectively. Analogous notation is used for data structures D.

Figure 1 also defines *configurations* that describe the computation states of Plain. A configuration $V\sigma[]\;S$ consists of a store $V\sigma$ and a statement S. The *store* is a pair of a set of variables V and a function σ (defined below) that represents a set of bindings of variables in V to some structure D. We require $fv(S) \subseteq V$.

Structural Congruence. We identify statements S, data structures D, and configurations C up to consistent renaming of bound variables, and up to the structural congruence given in Figure 2. Parallel composition of statements is commutative and associative with neutral element skip. Records $\{\bar{a}{:}\bar{y}\}$ and patterns $(\{\bar{a}{:}\bar{y}\})$ are identified up to reordering of record fields.

Store. We assume an infinite set Nam of *names* and a function new that maps every finite set $\mathcal{N}$ of names to a fresh name $n \notin \mathcal{N}$. A *store function* σ is a finite partial function $\sigma : \mathsf{Var} \cup \mathsf{Nam} \rightharpoonup \mathsf{Nam} \cup \mathsf{Dat}$ with the properties:

$$\sigma(\mathsf{Var}) \subseteq \mathsf{Nam},\; \sigma(\mathsf{Nam}) \subseteq \mathsf{Dat}, \;\text{ and }\; \text{if } \sigma(x) \in \mathsf{Nam} \text{ then } \sigma(\sigma(x)) \text{ is defined.}$$

[1]The restriction to monadic procedures is for technical convenience and not a proper restriction in presence of records.

$$
\begin{array}{llll}
\hline
V\sigma[]\ (\mathsf{local}(x)\,S) & \rightarrow & V\cup\{x\}\ \sigma[]\ S & \text{if } x \notin V \\
V\sigma[]\ x{:=}y & \rightarrow & V\sigma,x{\mapsto}\sigma(y)\ []\ \mathsf{skip} & \text{if } y \in \mathcal{D}(\sigma) \\
V\sigma[]\ x{:=}D & \rightarrow & V\sigma,x{\mapsto}D\ []\ \mathsf{skip} & \\
V\sigma[]\ (x\,y) & \rightarrow & V\sigma[]\ S[y/z] & \text{if } \sigma(\sigma(x)) = (\mathsf{proc}(z)\,S) \\
V\sigma[]\ (\mathsf{exch}\,x\,y\,(z)\ S) & \rightarrow & V\sigma[(\mathsf{cell}\ y)/\sigma(x)]\ []\ S[z'/z] & \text{if } \sigma(\sigma(x)) = (\mathsf{cell}\ z') \\
V\sigma[]\ (\mathsf{match}\,x\,(\{\bar{a}{:}\bar{y}\})\ S) & \rightarrow & V\sigma[]\ S[\bar{z}/\bar{y}] & \text{if } \sigma(\sigma(x)) = \{\bar{a}{:}\bar{z}\ldots\} \\
\hline
\end{array}
$$

Figure 3: Operational Semantics of Plain

We denote the empty store as ε. Let $\mathcal{D}(\sigma)$ (*domain*) denote the set of variables for which $\sigma(x)$ is defined, and $\mathcal{R}(\sigma)$ (*range*) the sets of variables which occur free in a structure in the range of σ. A *store* $V\sigma$ consists of a set of variables V and a store function σ such that $\mathcal{D}(\sigma)\cup\mathcal{R}(\sigma)\subseteq V$. We say that x is *bound* by $V\sigma$ if $x{\in}\mathcal{D}(\sigma)$ and that x is *unbound* in $V\sigma$ if $x\in V\setminus\mathcal{D}(\sigma)$. If x is bound by $V\sigma$, we say that x is *bound to* the structure $\sigma(\sigma(x))$. Names are needed to model the *locations* of cells.[2]

Stores provide for two *extension* operations, written $\sigma,x{\mapsto}D$ and $\sigma,x{\mapsto}n$. Let $\mathcal{N}(\sigma)$ be the set of names n for which $\sigma(n)$ is defined, and denote with $\sigma[n/x]$ [*resp.*, $\sigma[D/n]$] the store which coincides with σ except that it maps x to n [*resp.*, n to D]. Then store extension is defined as follows.

$$
\sigma,x{\mapsto}D = \left\{\begin{array}{ll} \sigma & \text{if } x\in\mathcal{D}(\sigma) \\ \sigma[n/x][D/n] & \text{if } x\notin\mathcal{D}(\sigma), \\ & n = \mathsf{new}(\mathcal{N}(\sigma)) \end{array}\right. \qquad \sigma,x{\mapsto}n = \left\{\begin{array}{ll} \sigma & \text{if } x\in\mathcal{D}(\sigma) \\ \sigma[n/x] & \text{if } x\notin\mathcal{D}(\sigma) \end{array}\right.
$$

Both extension operations preserve the bindings in a store. Attempts to rebind a bound variable are ignored, and unbound variable are always bound to *new* names. To model cells, we will also use the (overriding) *substitution* $\sigma[D/\sigma(x)]$ directly. $\sigma[D/\sigma(x)]$ simultaneously affects all variables bound to the same name $\sigma(x)$. *E. g.*, if $\sigma=\varepsilon,n{\mapsto}(\mathsf{cell}\ y'),x{\mapsto}n,x'{\mapsto}n$ then $\sigma[(\mathsf{cell}\ y)/\sigma(x)]{=}\varepsilon,n{\mapsto}(\mathsf{cell}\ y),x{\mapsto}n,x'{\mapsto}n$: The bindings of both x and x' have been changed.

Operational Semantics. The operational semantics of Plain is defined in terms of a one-step reduction relation on configurations. *Reduction* $\rightarrow$ is the smallest binary relation on configurations that satisfies the rules in Figure 3 and is closed under the following rule.

$$
\frac{V\sigma[]\ S_1 \rightarrow V'\sigma'\ []\ S_2}{V\sigma[]\ S_1\,|\,S \rightarrow V'\sigma'\ []\ S_2\,|\,S}
$$

Reduction of a declaration $(\mathsf{local}(x)\,S)$ adds a new variable x to the store and reduces to S. A *variable assignment* $x{:=}y$ waits for y to be bound in the current store and then extends it by the binding of x to $\sigma(y)$. A *data assignment* $x{:=}D$ extends

[2]In this paper, we associate names with records and procedures only for homogeneity of presentation; they are not exploited. They are convenient, however, to support an untyped equality test at all data structures (compare `eq` in Scheme, and [36]).

the store by the binding of x to D without further preconditions. The following example illustrates the dynamic extension of the store.

$$\begin{array}{ll}\{x\}\varepsilon\,[]\ (\mathsf{local}(y)\,y:=\{a{:}x\}\,|\,x:=y) & \\ \quad\to\ \{x,y\}\varepsilon\,[]\ y:=\{a{:}x\}\,|\,x:=y & \\ \quad\to\ \{x,y\}\varepsilon,y\mapsto\{a{:}x\}\,[]\ x:=y & \\ \quad\to\ \{x,y\}\varepsilon,y\mapsto\{a{:}x\},x\mapsto\{a{:}x\}\,[]\ \mathsf{skip} & \end{array}$$

Note that this constructs a binding of x to $\{a{:}x\}$ which yields a cyclic record.

An application $(x\ y)$ can be reduced if the store binds x to a procedure. A cell exchange $(\mathsf{exch}\ x\ y\ (z)\ S)$ can be reduced if x is bound to a cell $(\mathsf{cell}\ z')$; then it alters the content z' of the cell to y and continues with $S[z'/z]$. *E. g.*, the following statement reads the contents from cell x and assigns it to y.

$$\begin{array}{l}\{x,y,y',\ldots\}\varepsilon,x\mapsto(\mathsf{cell}\ y')\,[]\ (\mathsf{exch}\ x\ y\ (z)\ (\mathsf{exch}\ x\ z\ (z')\ z':=z))\\ \quad\to\ \{x,y,y',\ldots\}\varepsilon,x\mapsto(\mathsf{cell}\ y)\,[]\ (\mathsf{exch}\ x\ y'\ (z')\ z':=y')\\ \quad\to\ \{x,y,y',\ldots\}\varepsilon,x\mapsto(\mathsf{cell}\ y')\,[]\ y:=z\end{array}$$

A matching statement $(\mathsf{match}\ x\ (\{\bar{a}{:}\bar{y}\})\ S)$ can be reduced if the store binds x to a record that *matches* the pattern $(\{\bar{a}{:}\bar{y}\})$; *i.e.*, that is of the form $\{\bar{a}{:}\bar{z}\ldots\}$. A special case of matching is field selection on records. For instance, let $\sigma{=}\varepsilon,y\mapsto\{a{:}x\ b{:}y\}$ and $V=\{x,y\}$ and consider:

$$\begin{array}{lll}V\sigma\,[]\ (\mathsf{match}\ y\ (\{b{:}z\})\ x:=z)) & \to & V\sigma\,[]\ x:=y\\ & \to & V\sigma,x\mapsto\{a{:}x\ b{:}y\}\,[]\ \mathsf{skip}\end{array}$$

Concurrency. Synchronization of concurrent threads on the presence of data structures in the store is illustrated by the following example.

$$\begin{array}{lll}\{x,y\}\varepsilon\,[]\ x:=y\,|\,y:=\{a{:}x\} & \to & \{x,y\}\varepsilon\,[]\ x:=y\,|\,y:=\{a{:}x\}\\ & \to & \{x,y\}\varepsilon,y\mapsto\{a{:}x\}\,[]\ x:=y\\ & \to & \{x,y\}\varepsilon,y\mapsto\{a{:}x\},x\mapsto\{a{:}x\}\,[]\ \mathsf{skip}\end{array}$$

Cells introduce indeterminism in Plain since the result of cell exchanges depends on the execution order. This indeterminism is useful for instance to model a number of concurrent agents competing for a single resource.

3 Typing and Subtyping

We present the type system of Plain. It is obtained by adaptation from Pict's type system [29] which in turn foots on a long tradition of type systems for functional languages (*e.g.*, see the overviews [7, 9]).

The communication of concurrent threads with each other through the shared store is mediated by logic variables. For this communication to work smoothly there must be consensus between the threads on the access protocols for the shared variables. These protocols include two kinds of information:

- Structural: "Which data structures may a variable be bound to?"

- Modal: "Is it legal to read from and/or write to a variable?"

Types are a means to describe such access protocols for variables.[3] Typical type assumptions include

x:?int**:** grants the right to read the variable x and guarantees that reading will yield an integer; denies write access to x, that is to bind x.

x:^int**:** grants the right to read integers from x *and* to write integers to x.

x:!$\{a{:}T_1, b{:}T_2\}$**:** grants the right to bind x to any record that has at least the features a and b, provided that their associated fields have types T_1 and T_2.

x:!(proc ?int)**:** grants the right to bind x to a procedure that can safely be applied to variables of type ?int.

x:?(proc $\exists\alpha.\{1{:}?\alpha\ 2{:}!\mathsf{int}\}$)**:** grants the right to read a procedure from x, and to apply it to all pairs of arguments of which the first provides read access, and the second one allows writing an integer.

We write $x{:}T$ for the *assumption* that variable x has type T. Type assumptions for multiple variables are grouped in *type environments* Γ. *Type checking* is protocol validation: namely, the process of verifying that a configuration C *respects* some type environment Γ at any time during reduction; we write this as $\Gamma \triangleright C$. *Subtyping* defines an order $\Gamma \leq \Gamma'$ on type environments such that C respects Γ whenever C respects Γ'; intuitively, this is the case if Γ describes the more specific protocol that allows more operations on the mentioned variables than Γ'. This order on type environments is obtained by lifting a corresponding subtyping order on types $T \leq T'$ pointwise to environments. Typical subtypings include:

?int $\leq$?num**:** reading numbers from a variable is less specific than reading integers. Hence, every variable respecting the protocol ?num will also respect the protocol ?int (provided that int $\leq$ num).

^$T \leq$?T**:** the protocol that grants read and write access to structures of type T is obviously respected by a statement that only reads from the variable.

The technical setup of our type system is as usual. We define a proof system for *judgements* $\Gamma \triangleright C$. If a judgment $\Gamma \triangleright C$ is derivable, then the configuration C is guaranteed to respect Γ. This *type safety* yields the static guarantee that none of the following *type errors* will occur during reduction of C.

$V\sigma[]\ (x\ y) \mid S$	where σ binds x but not to a procedure.
$V\sigma[]\ (\mathsf{exch}\ x\ y\ (z)\ S) \mid S'$	where σ binds x but not to a cell.
$V\sigma[]\ (\mathsf{match}\ x\ (\{\bar{a}{:}\bar{y}\})\ S) \mid S'$	where σ binds x but not to a record, or to a record that does not match the pattern $(\{\bar{a}{:}\bar{y}\})$.

[3] While these are only simple kinds of protocols, the types-as-protocols view nonetheless seems appropriate. complex protocol types in concurrent programming see, *e.g.*, [27].

$$P ::= (\mathsf{proc}\ T) \mid (\mathsf{cell}\ T) \mid \{\overline{a}{:}\overline{T}\} \mid B \mid \top \mid \alpha \mid \exists\alpha{\leq}P_1.P_2$$
$$T ::= \hat{\ }P \mid ?P \mid !P$$

Figure 4: Types

$P \leq P$	(REFL)	$\dfrac{P_1 \leq P_2 \quad P_2 \leq P_3}{P_1 \leq P_3}$	(TRANS)
$\dfrac{P_1 \leq P_2}{?P_1 \leq ?P_2}$	(READ)	$\dfrac{P_1 \leq P_2}{!P_2 \leq !P_1}$	(WRITE)
$\hat{\ }P \leq ?P$	(READSUB)	$\hat{\ }P \leq !P$	(WRITESUB)
$\dfrac{T \leq T'}{(\mathsf{proc}\ T') \leq (\mathsf{proc}\ T)}$	(PROCSUB)	$\dfrac{\overline{T} \leq \overline{T'}}{\{\overline{a}{:}\overline{T}\ldots\} \leq \{\overline{a}{:}\overline{T'}\}}$	(RECSUB)

Figure 5: Monomorphic Subtyping

Subtyping judgements $T_1 \leq T_2$ are defined by a second proof system.

Types. We assume an infinite set of *type variables* ranged over by α, and a set of *base types* ranged over by B that contains at least the types int and num. Figure 4 defines the abstract syntax of types. For technical reasons, we use *two* syntactic categories of *types* ranged over by P and T. If a distinction is necessary, we call P a *pretype*. A *mode* is read-only (?), write-only (!), or allows both reading and writing (ˆ). A *pretype* P is a type without its top-level mode.

The three leftmost pretypes are *procedure types* $(\mathsf{proc}\ T)$, *cell types* $(\mathsf{cell}\ T)$, and *record types* $\{\overline{a}{:}\overline{T}\}$. We require the labels of record types to be pairwise distinct and identify record types up to reordering of label-type pairs. A *monomorphic type* is a type constructed from the above mentioned constructs only. The only primitive monomorphic type is the empty record type $\{\}$.[4]

For polymorphic types, we assume an infinite set of (pre-) type variable ranged over by α. The three rightmost pretypes are *type variables* α, the *maximal type* top, and *polymorphic types* $\exists\alpha{\leq}P_1.P_2$ that support type abstraction by existential polymorphism [21, 9, 8]. Polymorphic types $\exists\alpha{\leq}P_1.P_2$ describe objects of "concrete" type $P_2[P_1'/\alpha]$ for some P_1' which one only knows to be a subtype of P_1. The type $\exists\alpha{\leq}P_1.P_2$ binds α with scope P_2. The type top is needed to express unbounded polymorphism in form of maximal type abstractions $\exists\alpha{\leq}\mathtt{top}.P$.

Subtyping. Subtyping on monomorphic types is the smallest relation satisfying the rules given in Figure 5. For two record types T and T', T is subtype of T' if T has at least the labels in T' and the corresponding fields of T and T' are in covariant subtype relation. A procedure type $(\mathsf{proc}\ T)$ is a subtype of $(\mathsf{proc}\ T')$ if $T' \leq T$, *i.e.*, if $(\mathsf{proc}\ T)$ is applicable to *more* arguments than $(\mathsf{proc}\ T')$. There is only trivial

[4] We have left out variant types, which are needed when conditionals with multiple clauses are added, and we have not discussed recursive types $\mu\alpha.P$. Extensions with both forms of types are standard though (*e.g.*, [7]).

$$\frac{\alpha \leq P \in \Gamma}{\Gamma \triangleright \alpha \leq P} \;(\textsc{Bnd}) \qquad \Gamma \triangleright P_2[P_1/\alpha] \leq \exists\alpha^{\leq}P_1.P_2 \;(\textsc{Abstr})$$

$$\Gamma \triangleright P \leq \mathsf{top} \;(\textsc{Top}) \qquad \frac{\Gamma \triangleright P_1 \leq P_1' \quad \Gamma, \alpha \leq P_1 \triangleright P_2 \leq P_2'}{\Gamma \triangleright \exists\alpha^{\leq}P_1.P_2 \leq \exists\alpha^{\leq}P_1'.P_2'} \;(\textsc{Poly})$$

Figure 6: Polymorphic Subtyping

$$\frac{x{:}T \in \Gamma}{\Gamma \triangleright x{:}T} \;(\textsc{Var}) \qquad \frac{\Gamma \triangleright x{:}T \quad \Gamma \triangleright T \leq T'}{\Gamma \triangleright x{:}T'} \;(\textsc{Sub})$$

$$\frac{\Gamma \triangleright x{:}!P \quad \Gamma \triangleright y{:}?P}{\Gamma \triangleright x{:=}y} \;(\textsc{Asgn1}) \qquad \frac{\Gamma \triangleright x{:}!P \quad \Gamma \triangleright D{:}?P}{\Gamma \triangleright x{:=}D} \;(\textsc{Asgn2})$$

$$\frac{\Gamma, y{:}T \triangleright S}{\Gamma \triangleright (\mathsf{proc}(y)\, S){:}?(\mathsf{proc}\ T)} \;(\textsc{Proc}) \qquad \frac{\Gamma \triangleright x{:}?(\mathsf{proc}\ T) \quad \Gamma \triangleright y{:}T}{\Gamma \triangleright (x\, y)} \;(\textsc{Appl})$$

$$\frac{\Gamma \triangleright y{:}T}{\Gamma \triangleright (\mathsf{cell}\ y){:}?(\mathsf{cell}\ T)} \;(\textsc{Cell}) \qquad \frac{\Gamma \triangleright \overline{y}{:}\overline{T}}{\Gamma \triangleright \{\overline{a}{:}\overline{y}\}{:}?\{\overline{a}{:}\overline{T}\}} \;(\textsc{Rec})$$

$$\frac{\Gamma \triangleright x{:}?(\mathsf{cell}\ T) \quad \Gamma \triangleright y{:}T \quad \Gamma, z{:}T \triangleright S}{\Gamma \triangleright (\mathsf{exch}\ x\ y\ (z)\ S)} \;(\textsc{Exch})$$

$$\frac{\Gamma \triangleright x{:}?\exists\overline{\alpha}^{\leq}\overline{P}.\{\overline{a}{:}\overline{T}\} \quad \Gamma, \overline{\alpha} \leq \overline{P}, \overline{y}{:}\overline{T} \triangleright S}{\Gamma \triangleright (\mathsf{match}\ x\ (\{\overline{a}{:}\overline{y}\})\ S)} \;(\textsc{Match})$$

Figure 7: Typing Expressions and Statements

subtyping for cells: Since a cell may be read to and written from, cell types must be non-variant. In the polymorphic case, types may have free variables; so subtyping is relative to an environment Γ which contains *subtype assumptions* $\alpha \leq P$ in addition to the type assumptions $x{:}T$. The *extension* of an environment Γ by $x{:}T$ is written as adjunction $\Gamma, x{:}T$. The notions $\Gamma, \alpha \leq P$ and Γ, Γ' are defined analogously. Subtyping for polymorphic types is defined via judgements $\Gamma \triangleright P \leq P'$ and $\Gamma \triangleright T \leq T'$ which state that Γ implies the subtype relations $P \leq P'$ and $T \leq T'$. We reinterpret all rules in Figure 5 such that their premises and conclusions share the same Γ and add the rules from Figure 6. Rule (Abstr) says that the *concrete type* $P_2[P_1/\alpha]$ is a subtype of the *abstract type* $\exists\alpha^{\leq}P_1.P_2$. Subtyping between two polymorphic types $\exists\alpha^{\leq}P_1.P_2$ and $\exists\alpha^{\leq}P_1'.P_2'$ follows from covariant subtyping $P_1 \leq P_1'$ and $P_2 \leq P_2'$.

Example 3.1 Assuming an additional primitive type $\mathsf{int} \leq \mathsf{top}$, we have $\{a{:}?\mathsf{int}\ b{:}!\mathsf{num}\} \leq \{a{:}?\mathsf{int}\ b{:}!\mathsf{int}\} \leq \exists\alpha^{\leq}\mathsf{int}.\{a{:}?\alpha\ b{:}!\alpha\} \leq \exists\alpha^{\leq}\mathsf{top}.\{a{:}?\alpha\ b{:}!\alpha\}$. □

Typing Rules Figure 7 defines typing of statements in terms of *judgements* of the form $\Gamma \triangleright x{:}T$, $\Gamma \triangleright D{:}T$, and $\Gamma \triangleright S$. The first two judgements mean that Γ ensures type T for x and D. The third one says that S is well-typed with respect to Γ.

Variables receive their type by lookup in the environment (VAR) and can be promoted along the subtyping order (SUB). (Due to this *subsumption* rule and the subtyping rule (PROCSUB), typing and subtyping recursively depend on each other). Assignments $x := y$ and $x := D$ require that x, y, and D have the same type up to the top-level mode; the mode describes the data flow of an assignment. The rules (PROC), (REC) and (CELL) are simple. For application $(x\ y)$, exchange $(\mathsf{exch}\, x\, y\, (z)\, S)$, or matching $(\mathsf{match}\, x\ (\{\overline{a}{:}\overline{y}\})\ S)$ to be well-typed, x must allow read access (rules (APPL), (EXCH), (MATCH)). The types of further arguments must match whatever the type of x requires. Rule (MATCH) is central to the polymorphic type system; given $(\mathsf{match}\, x\ (\{\overline{a}{:}\overline{y}\})\ S)$, it "opens" the polymorphic record type $\exists\overline{\alpha}{\le}\overline{P}.\{\overline{a}{:}\overline{T}\}$ of x within S, where one may assume the bounds $\overline{P}$ on the type variables $\overline{\alpha}$. We omit the trivial rules for parallel composition, declaration, and skip.

A statement S is called *well-typed* w.r.t. an environment Γ, written $\vdash \Gamma \triangleright S$, if $\Gamma \triangleright S$ is derivable by the inference system in Figures 5 – 7. A store σ is well-typed w.r.t. Γ iff $\vdash \Gamma \triangleright x := D$ for all x and D such that $\sigma(\sigma(x)) = D$. A configuration $V\sigma\,[]\ S$ is well-typed w.r.t. Γ if both σ and S are. A statement, store, or configuration is well-typed if it is well-typed w.r.t. some environment Γ.

Example 3.2 The polymorphic identity $(\mathsf{proc}(x)\,(\mathsf{match}\, x\ (\{1{:}x_1\ 2{:}x_2\})\ x_2 := x_1))$ can be typed as $(\mathsf{proc}\ ?\exists\alpha{\le}\mathsf{top}.(\mathsf{proc}\ \{1{:}?\alpha\ 2{:}!\alpha\}))$. Observe that not the procedure itself *has* a polymorphic type, but that it *takes* a polymorphic argument. □

Example 3.3 As a typical example for existential polymorphism consider:

$$y := (\mathsf{proc}(z)\,(\mathsf{match}\, z\ (\{1{:}z_1\ 2{:}z_2\})\ z_2 := z_1{+}1))\ |\ z := \{a{:}x\ b{:}y\}$$

We assume predefined the integers $1,2,\ldots$ with addition $+$, and we encode a binary procedure via a record $\{1{:}z_1\ 2{:}z_2\}$. Assume the type environment $x{:}?\mathsf{int}, y{:}?(\mathsf{proc}\ ?\{1{:}?\mathsf{int}\ 2{:}!\mathsf{int}\})$. Then S is well-typed with the further assumption $z{:}\hat{}\exists\alpha{\le}\mathsf{top}.\{a{:}?\alpha\ b{:}?(\mathsf{proc}\ ?\{?1{:}\alpha\ 2{:}!\mathsf{int}\})\}$; this type is obtained by type abstraction from $\hat{}\{a{:}?\mathsf{int}\ b{:}?(\mathsf{proc}\ ?\{1{:}?\mathsf{int}\ 2{:}!\mathsf{int}\})\}$. Full data encapsulation can be achieved by tight lexical scope: $(\mathsf{local}(x y)\, S)$. The type of z allows selection of fields at a and b but prohibits direct interaction with them due to the type variable α. But since the field types *share* this type variable α we can apply field b to field a to obtain an integer u' via $(\mathsf{match}\, z\ (\{a{:}u\ b{:}v\})\ (\mathsf{local}(w)\, w := \{1{:}u\ 2{:}u'\}\ |\ (v\ w)))$. □

Theorem 3.4 (Type Preservation) *If* $V\sigma\,[]\ S \to V'\sigma'\,[]\ S'$ *and* $\vdash \Gamma \triangleright V\sigma\,[]\ S$*, then there exists* Γ' *such that* $\vdash \Gamma, \Gamma' \triangleright V'\sigma'\,[]\ S'$.

Proof. The key lemma is that $\vdash \Gamma \triangleright S$ and $\vdash \Gamma(y) \le \Gamma(x)$ imply $\vdash \Gamma \triangleright S[y/x]$. □

Proposition 3.5 *A well-typed configuration does not contain a type error.*

Theorem 3.4 and Proposition 3.5 imply the absence of type errors in reductions of well-typed configurations (which, hence, "do not go wrong" [18]).

4 Encoding Channels

We express channels in Plain so as to embed Pict into Plain while preserving types. The full encoding is omitted for lack of space. For further examples see [22].

A *channel* for variables of type T is an abstract data type with two operations *put* and *get* of the following types:

$$put \;:\; ?(\mathsf{proc}\ T) \qquad get \;:\; ?(\mathsf{proc}\ ?(\mathsf{proc}\ T))$$

The *put* operation takes a variable of type T, puts it on ("sends it along") the channel, and then terminates. The *get* operation takes a variable of type $?(\mathsf{proc}\ ?T)$, that is, a reference to a procedure *cont* for arguments of type $?T$; then it takes ("receives") a variable from the channel and applies *cont* as a continuation to it. Combination of these operations in a record with fields *put* and *get* yields the following type of channel interfaces:

$$\mathsf{ch}(T) \;=_{def}\; \{g{:}?(\mathsf{proc}\ ?(\mathsf{proc}\ T))\ p{:}?(\mathsf{proc}\ T)\} \tag{1}$$

Now we proceed to implement a polymorphic procedure *newchan* that generates new channels for variables of arbitrary input type.

$$newchan \;:\; (\mathsf{proc}\ !\exists\alpha{\le}\mathsf{top}.\mathsf{ch}(?\alpha))$$

We implement a channel as a variable s_0 referring to a stream and two cells c_p and c_g as pointers into s. On creation, the stream is empty and both pointers refer to the first slot. On application of the procedure *put* to a variable z, the current content s_1 of c_p is replaced with a fresh variable s_2 and then s_1 is bound to $\{z\ s_2\}$. This advances the pointer *cput*. On application of *get* to a variable *cont*, the current content s_1 of c_g is replaced with a fresh variable s_2; then s_1 is matched against (z, s_3). When s_1 is bound to such a record, *cont* is applied to z, and then s_3, the tail of the stream, is assigned to s_2.

$$\boxed{\begin{array}{l}
newchan{:=}\\
\quad(\mathsf{proc}(chan)\ (\mathsf{local}(s_0\ c_p\ c_g\ put\ get)\\
\qquad c_p := (\mathsf{cell}\ s_0) \mid c_g := (\mathsf{cell}\ s_0)\\
\qquad \mid put := (\mathsf{proc}(z_1)\ \ (\mathsf{local}(s_2)\\
\qquad\qquad\qquad (\mathsf{exch}\ c_p\ s_2\ (s_1)\\
\qquad\qquad\qquad\quad s1 := \{1{:}z_1\ 2{:}s_2\})))\\
\qquad \mid get := (\mathsf{proc}(cont)\ \ (\mathsf{local}(s_2)\\
\qquad\qquad\qquad (\mathsf{exch}\ c_g\ s_2'\ (s_1')\\
\qquad\qquad\qquad\quad (\mathsf{match}\ s_1'\ (\{1{:}z_1'\ 2{:}s_3\})\\
\qquad\qquad\qquad\qquad s_2' := s_3 \mid (cont\ z_1)))))\\
\qquad \mid chan := \{p{:}put\ g{:}get\}
\end{array}}$$

The type (1) for *newchan* implies that *newchan* creates only channels for read-only variables (this suffices to formulate a type-correct embedding of Pict). Our implementation is, however, more general than this: We can put an unbound variable into a channel and assign to it on the get operation, thus effectively reversing the data flow. There are useful programs which employ this technique (*e.g.*, we can express *lazy streams*) and which require *newchan* to have the type $(\mathsf{proc}\ !\exists\alpha{\le}\mathsf{top}.\mathsf{ch}(!\alpha))$. In order to avoid code duplication we might want to have polymorphic types $\exists\beta{\le}T.P$ which abstract over *types including the mode*, instead of only pretypes. We then can show *newchan* to have the more general type

$$newchan : (\mathsf{proc}\ !\exists\beta{\le}\mathsf{top}.\mathsf{ch}(\beta))\ .$$

Remark. To type the channel encoding above we assume recursive types and define list types $M\mathsf{list}(T) = \mu\alpha.M\{T\ M\alpha\}$ for all $M \in \{?, !, \hat{}\}$.

5 Comparison to OPM

The essential changes of Plain with respect to OPM [36] are the absence of unification in the kernel language and the fact that cell exchange comes with a continuation. Both changes are needed in order to make the type system work.

Unification Issues. As a first approximation of unification imagine a (fictive) bidirectional assignment statement of the form $x :=: y$ that behaves either as $x := y$ or as $y := x$ but where it is only dynamically decided as which.

Since strong typing requires the types of variables to be known statically, the best typing rule for bidirectional equations $x :=: y$ that preserves type safety is

$$\frac{\Gamma \triangleright x{:}\hat{}P \qquad \Gamma \triangleright y{:}\hat{}P}{\Gamma \triangleright x :=: y} \quad (\textsc{Bidirect})$$

This does not exclude the possibility that P contains nested read and/or write modes. For example, the type $P = \{a{:}?\mathsf{int}\}$ is a useful type. Unification of complex data structures like records not only behaves as a dynamically directed form of assignment, it also traverses a given data structure recursively and generates additional unification tasks. Due to the recursive nature of unification, a strong typing rule for an equational constraint must be even more restricted than (BIDIRECT):

$$\frac{\Gamma \triangleright x{:}T \qquad \Gamma \triangleright y{:}T}{\Gamma \triangleright x{=}y} \quad T \text{ does not contain ? or !} \quad (\textsc{Unif})$$

In effect, this rule trivialises subtyping on the types of all expressions that may be involved in a unification. One looses virtually all subtyping in a language that makes unification the central operation. For this reason, Plain's basic equational form is the (directed) assignment $x := y$ instead of the equation $x{=}y$ as in OPM.

Cell Exchange. The cell exchange $(\mathsf{exch}\ x\ y\ z)$ in OPM makes use of unification $x{=}y$. Its operational semantics in Plain-style is

$$V\sigma[]\ (\mathsf{exch}\ x\ y\ z) \quad \to \quad V\sigma[(\mathsf{cell}\ y)/\sigma(x)]\,[]\ z'{=}z \qquad \text{if } \sigma(\sigma(x)) = (\mathsf{cell}\ z')$$

An immediate option to get better typing is to replace the equation by an assignment. However neither $z := z'$ nor $z' := z$ is preferred over the other. Both of them are needed. Plain's modified cell exchange $(\mathsf{exch}\ x\ y\ (z)\ S)$ defers to the continuation S the decision at which mode to use the old content of x. Another option is to have cells always hold records with some (arbitrary but fixed) feature a and to combine cell exchange with field selection at a. We decided against this option to keep cells independent of the other data structures.

6 Related Work

Earlier work that compared higher-order concurrent constraint programming and the π calculus includes the following. Smolka [35] presents OPM in form of the γ-calculus, and notices a close relation to the π-calculus. Niehren and Müller [26] prove this relation based on the ρ-calculus which extends the γ-calculus with a parametric constraint system. Niehren [24, 25] studies uniform confluence in the concurrent calculi mentioned above (γ, δ, ρ, π, join); he identifies a common sub-calculus of γ and π and proves that it can embed the eager and the call by need λ-calculus. Victor and Parrow [38] give an embedding of the γ-calculus into the π-calculus and prove its adequacy based on bisimulation semantics.

Modalities for logic variables have frequently been considered in (concurrent) logic programming, but either (i) in an untyped setting, (ii) such that modalities are not interleaved with structural type information, or (iii) with a different meaning: Modes have described the instantiation state of procedure arguments (such as ground, non-ground, free) directly before or before and after application [10, 4, 34], or guaranteed point-to-point communication [16, 32, 37]

In concurrent functional languages, synchronization based on futures [13] or I-structures [2] in Id comes close to data driven synchronization based on logic variables. In contrast to futures, logic variables are created independently from the thread that will bind it. In contrast to logic variables and similar to channels, both futures and I-structures require explicit operations to access the data. Id's M-structures [3] are related to cells but differ in that read and write operations on M-structures are not atomic. Reading an empty M-structure is a run-time error. Furthermore, M-structures have a specific synchronisation scheme independent from the one of I-structures. In contrast, cell exchange in OPM and Plain is unsynchronized (once the cell was created): Synchronization is complete left to producers and consumers of the exchanged logic variables.

7 Future Work

Future work will improve our prototype implementation of Plain, and will investigate whether and how it is possible to extend Plain by constraints and unification such that type safety can still be guaranteed. This is straightforward for constraints over flat domains like finite domain or finite set constraints, and is likely to require recursive type specifications for tree constraints. Alternatively, an interesting question is whether a strongly typed host language can coexist with (mainly) dynamically typed constraint extensions.

Acknowledgements. We grateful to David N. Turner for discussions on concurrent programming and Pict. This research has been supported by the BMBF (FKZ ITW 9601), the Esprit Working Group CCL II (EP 22457), and the DFG (SFB 378).

References

[1] M. Abadi and L. Cardelli. *A Theory of Objects*. Springer-Verlag, Berlin, 1996. Monographs in Computer Science.

[2] Arvind, R. S. Nikhil, and K. K. Pingali. I-structures: Data structures for parallel computing. *ACM Transactions on Progr. Lang. and Systems*, 11(4):598–632, 1989.

[3] P. S. Barth, R. S. Nikhil, and Arvind. M-structures: Extending a parallel, non-strict, functional language with state. In J. Hughes, ed, *5th ACM Conf. on Functional Programming and Computer Architecture*, *LNCS 523*, pp. 538–568. ACM Press, 1991.

[4] F. Bronsard, T. Lakshman, and U. Reddy. A framework of directionality for proving termination of logic programs. *JICS on Logic Programming*, pp. 321–335, 1992.

[5] L. Cardelli. A semantics of multiple inheritance. *Information and Computation*, 76(2/3):138–164, 1988.

[6] L. Cardelli. Structural subtyping and the notion of power type. In *Proceedings of the 15th ACM Symposium on Principles of Programming Languages*, 70–79. ACM Press, New York, Jan. 1988.

[7] L. Cardelli. Type systems. In A. B. Tucker, ed, *CRC Handbook of Computer Science and Engineering*. CRC Press LLC, 1996.

[8] L. Cardelli and X. Leroy. Abstract types and the dot notation. In *Proceedings of the IFIP Working Conf. Programming Concepts and Methods*, pp. 466–491, 1990.

[9] L. Cardelli and P. Wegner. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys*, 17(4), Dec. 1985.

[10] S. K. Debray and D. S. Warren. Automatic mode inference for prolog programs. *The Journal of Logic Programming*, 1986.

[11] C. Fournet and G. Gonthier. The reflexive cham and the join-calculus. In *23rd ACM Symp. on Principles of Programming Languages*, pp. 372–385. ACM Press, 1996.

[12] J.-Y. Girard. *Interprétation fonctionelle et élimination des coupures de l'arithmetique d'ordre supérieur*. Ph.D. thesis, Université de Paris VII, 1972.

[13] R. Halstead. Multilisp: A language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–538, 1985.

[14] M. Henz. *Objects in Oz*. Doctoral Dissertation. Universität des Saarlandes, Technische Fakultät, 66041 Saarbrücken, Germany, May 1997.

[15] J. Jaffar and M. Maher. Constraint Logic Programming: A Survey. *The Journal of Logic Programming*, 19/20:503–582, May-July 1994.

[16] A. Kleinman, Y. Moscowitz, A. Pnueli, and E. Shapiro. Communication with directed logic variables. In *18th ACM Symp. on Principles of Programming Languages*, pp. 221–232. ACM Press, New York, 1991.

[17] M. Maher. Logic semantics for a class of committed-choice programs. J.-L. Lassez, ed, *Int. Conf. on Logic Programming*, pp. 858–876. The MIT Press, 1987.

[18] R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Science*, 17:348–375, 1978.

[19] R. Milner. The polyadic π-calculus: A tutorial. In F. L. Bauer, W. Brauer, and H. Schwichtenberg, eds, *Proceedings of the 1991 Marktoberndorf Summer School on Logic and Algebra of Specification*, NATO ASI Series. Springer-Verlag, Berlin, 1993.

[20] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, I and II. *Information and Computation*, 100(1):1–40 and 41–77, Sept. 1992.

[21] J. C. Mitchell and G. D. Plotkin. Abstract Types have Existential Type. In *12^th ACM Symp. on Principles of Programming Languages*, pp. 37–51. ACM Press, 1985.

[22] M. Müller. *Set-based Failure Diagnosis for Concurrent Constraint Programming*. Doctoral Dissertation. Univ. des Saarlandes, Tech. Fak. 1997. In preparation.

[23] M. Müller. Polymorphic types for concurrent constraints. Tech. rep., Programming Systems Lab, Universität des Saarlandes, 1996. Available at `http://www.ps.uni-sb.de/~mmueller/papers/ptcc.ps.gz`.

[24] J. Niehren. Functional computation as concurrent computation. In *23^rd ACM Symp. on Principles of Programming Languages*, pp. 333–343, 1996. ACM Press.

[25] J. Niehren. Uniform Confluence in Concurrent Computation. submitted.

[26] J. Niehren and M. Müller. Constraints for free in concurrent computation. In K. Kanchanasut and J.-J. Lévy, eds, 1^st *Asian Computing Science Conference*, *LNCS 1023*, pp. 171–186. Springer-Verlag, Berlin, 1995.

[27] O. Nierstrasz. Regular types for active objects. In O. Nierstrasz and D. Tsichritzis, editors, *Object-oriented Software Construction*, pages 99–121. Prentice-Hall, Englewood Cliffs, N.J., 1995.

[28] B. Pierce and D. Sangiorgi. Typing and subtyping for mobile processes. In *IEEE Symposium on Logic in Computer Science*, pp. 376–385. IEEE Press, 1993.

[29] B. C. Pierce and D. N. Turner. Pict: A programming language based on the pi-calculus. In *Milner Festschrift*. The MIT Press, 1997.

[30] Programming Systems Lab. The Oz Programming System, 1997. Universität des Saarlandes: `http://www.ps.uni-sb.de/www/oz/`.

[31] V. A. Saraswat. *Concurrent Constraint Programming*. The MIT Press, 1993.

[32] V. A. Saraswat, K. Kahn, and J.-J. Levy. Janus: A step towards distributed constraint programming. In *North American Conference on Logic Programming*, 1990.

[33] E. Shapiro. The family of concurrent logic programming languages. *ACM Computing Surveys*, 21(3):413–510, Sept. 1989.

[34] G. Smolka. TEL (Version 0.9), report and user manual. SEKI-Report SR-87-17, FB Informatik, Universität Kaiserslautern, 1988.

[35] G. Smolka. A foundation for concurrent constraint programming. J.-P. Jouannaud, ed, *1^st Int. Conf. on Constraints in Computational Logics*, *LNCS 845*, pp. 50–72, 1994.

[36] G. Smolka. The Oz Programming Model. In J. van Leeuwen, ed, *Computer Science Today*, *LNCS 1000*, pp. 324–343. Springer-Verlag, Berlin, 1995.

[37] K. Ueda. Strong moding in concurrent logic/constraint programming. In *Proceedings of the 12^th International Conference on Logic Programming*, Kanagawa, Japan, 1995.

[38] B. Victor and J. Parrow. Constraints as processes. In U. Montanari and V. Sassone, eds, *7^th Int. Conf. on Concur. Theory*, *LNCS 1119*, pp. 389–405. Springer, 1996.

[39] A. Wright. Simple imperative polymorphism. *Lisp and Symbolic Computation*, 1994.