

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

This paper presents 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.

Plain is derived from the Oz programming model (OPM) [28], which formulates the essence of the programming language Oz [24]. OPM extends the concurrent constraint model [25,11] with first-class procedures and first-class cells. The concurrent constraint model merges two lines of research in logic programming, addressing concurrency [26] and constraints [9]. Plain has been designed as an OPM-style language with the following goals:

- Replace OPM's abstract notion of constraint store with a self-contained store model that does not require complex operations such as unification.
- Have a static type system with record-based subtyping and bounded existential polymorphism.
- Retain most of OPM's expressiveness as far as concurrent, functional, and object-oriented programming are concerned.

Plain's device for connecting and synchronizing the actors of a computation are logic variables. Logic variables can be seen as once only communication channels that become referentially transparent with the data structure they eventually receive. Basic synchronization comes for free since all actors block until their input variables are bound to data structures. In a language based with sequential composition and threads, logic

variables can provide for a smooth transition from sequential to concurrent code. Compound data structures can be built in an incremental fashion by designating components with variables that will be bound to data structures by concurrent or subsequent computation steps. Last but not least, logic variables are the ideal substrate for constraint programming, an evolving methodology for solving combinatorial problems through constraint propagation and informed search. It seems feasible to reformulate OPM's constraint extensions such that they can coexist with Plain's static type discipline.

Plain should be seen in competition with Pict [21,23], a recent concurrent programming language based on the π -calculus [13,14,8]. The π -calculus is designed as a minimal base for concurrent computation. Its essential primitive is channel communication which can express concurrent versions of procedures and data structures. While this minimality is intriguing from a foundational perspective, we doubt that it provides a good base for designing high-level languages. Since common programming abstractions must be regained through sometimes involved codings, the minimality of the π -calculus is lost, as exemplified by Pict.

Following OPM, Plain provides essential programming primitives directly: Records, first-class procedures, and cells. Sequential composition and thread creation can be added straightforwardly. Due to the presence of logic variables, there is no need for a dedicated communication primitive. Channels and locks can be expressed with the primitives and appear as synchronized data structures. Our programming experience with Oz shows that concurrent threads typically communicate through custom-built synchronized objects. Here logic variables and synchronization by blocking in combination with sequential composition and locks prove essential. Since most programming techniques from OPM carry over to Plain, we restrict ourselves to few programming examples in this paper and refer to [28] for others.

Plain can conveniently express Pict programs. However, many Plain programs need major rewriting to be expressed in Pict. This is due to the presence of logic variables and the concomitant synchronization techniques. If an operation needs information about a variable, it automatically blocks until this information is available in the store. Once a logic variable is bound to a data structure it is referentially transparent with respect to this data structure. This is in contrast to channels which remain distinct from the data structure they receive.

Plain's type system is inspired by Pict's type system and turns out to be surprisingly similar. It employs record-based subtyping and bounded existential polymorphism [2,6]. It also uses access modalities to make polymorphism and subtyping work in the presence of logic variables (see also [17]). Logic variables do not impose a static distinction between input and output. Such a distinction is however made implicitly in well-typed programs. The distinction is essential since outputs of a more restrictive type can be used as inputs of a more permissive type. Pierce and Sangiorgi [20] introduce access modalities for channels. Modalities and directionality have been considered in logic programming (*e.g.*, [10,1,7]) but either in an untyped setting or such that modalities are not interleaved with structural type information.

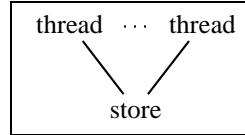
In contrast to Pict, Plain is defined as an untyped language and Plain programs do not contain explicit type information. Plain's type system is a proof system that establishes a class of Plain programs and configurations as well-typed. We show that the set of well-typed configurations is closed under reduction (*i.e.*, application of execution steps). Well-typed configurations exclude erroneous situations like the selection of nonexisting record fields. This paper is not concerned with algorithmic techniques for proving Plain programs to be well-typed. Such techniques are essential in practice. We expect that the type inference techniques used for Pict (based on work of Cardelli [3]) carry over to Plain.

Plain is a further contribution to connecting concurrent constraint programming in general and OPM in particular with other programming models. Smolka [27] presents OPM in form of the γ -calculus, which provides for a direct comparison with the π -calculus. Niehren and Müller [19] introduce the ρ -calculus, a technical variant of the γ -calculus contributing a modular factorization of constraints and actors. Niehren [18] identifies a common subcalculus of γ and π and proves that it can embed the eager and the call by need λ -calculus. Victor and Parrow [29] give an embedding of the γ -calculus into the π -calculus and prove its adequacy based on bisimulation semantics.

Plan of the Paper. The paper is organized as follows. Section 2 presents Plain and specifies its abstract syntax and operational semantics. Section 3 presents the type system and the type preservation theorem. Section 4 exemplifies the language by discussing a well-typed polymorphic procedure that creates data structures modeling channels. Section 5 explains which changes to OPM were needed to make the type system work.

2 Plain

The Computation Model. Plain organizes a computation into a number of *threads* computing over a *shared store*. Each thread is a computational task to be performed and represents, when sequential composition $S_1;S_2$ is added, the control structure of a sequential computation. The store is the place where all data structures reside. Threads are connected to the data structures in the store through variables. Threads wait automatically until the store contains enough information about the input variables of the next reduction step. Plain assumes interleaving semantics, that is, reduction steps are atomic and do not overlap in time.

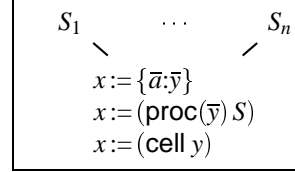


The Store Model. Plain's store binds variables to data structures, which are records, procedures, or cells. Data structures may contain variables. Typically, only some of these variables are bound. Thus, Plain's store accomodates for partial data structures.

$D ::= (\text{proc}(y) S) \mid (\text{cell } y) \mid \{\bar{a}:\bar{y}\}$
$S ::= (\text{local}(x) S) \mid \text{skip} \mid S_1 S_2 \mid x := y \mid x := D \mid$ $(x y) \mid (\text{exch } x y (z) S) \mid (\text{match } x ((\bar{a}:\bar{y}) S))$
$C ::= V\sigma \parallel S$

Fig. 1. Syntax of Data Structures, Statements, and Configurations

Once a binding is done in the store, it cannot be retracted. In contrast, the content of cell bindings may be altered to support stateful computation. Binding a variable in the store may wake up a statement (*i.e.*, turn it reducible again) which waits for its input variables to become bound. Since bindings cannot be retracted, reducibility is a monotonic property. This yields a straightforward *fairness condition*: Every reducible thread must eventually be reduced.



Syntactic Preliminaries. 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 $a_1:y_1, \dots, a_n:y_n$ (where $n \geq 0$). We will freely use an analogous sequence notation for other syntac categories.

The syntax of Plain as given in Figure 1 is built from variables and labels. There are three syntactic categories, (data) structures D , statements S , and configurations C . Data structures and statements define the syntax of Plain programs, while configurations describe the computation states.

Data Structures. A *structure* D is a procedure, or a cell, or a record. A (monadic) *procedure* $(\text{proc}(x) S)$ consists of a *formal argument* x and a *body* S where x is bound with scope S . A *cell* $(\text{cell } x)$ is a container with *contents* x , which may be exchanged by reduction. 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 , cells Cell , and records Rec .

Statements. In the first line of *statements* S we have the *declaration* $(\text{local}(x) S)$, *null statement* skip , and *parallel composition* $S_1 \mid S_2$ 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 . The second line contains an *application* $(x y)$ of x with *actual parameter* y , a *cell exchange* $(\text{exch } x y (z) S)$ on x with new content y for some old content z , and a statement $(\text{match } x ((\bar{a}:\bar{y}) S))$ *matching* x against the *pattern* $(\bar{a}:\bar{y})$. Note that both cell exchange and matching have a *continuation* S . (A conditional form has been left out for conciseness)

Configurations. A *configuration* $V\sigma \parallel S$ consists of a store $V\sigma$ and a statement S . The *store* is a pair of a set of variables $V \subseteq \text{Var}$ and a function σ that represents a set of bindings of variables in V to some structure D . We require that always $fv(S) \subseteq V$ holds.

$$\begin{aligned}
(S, |, \text{skip}) & \text{ is a commutative monoid} \\
\{\dots a_1:y_1 \dots a_2:y_2 \dots\} & \equiv \{\dots a_2:y_2 \dots a_1:y_1 \dots\} \\
(\text{match } x ((\dots a_1:y_1 \dots a_2:y_2 \dots) S)) & \equiv (\text{match } x ((\dots a_2:y_2 \dots a_1:y_1 \dots) S))
\end{aligned}$$

Fig. 2. Structural Congruence

Structural Congruence. A declaration $(\text{local}(z) S)$ and a cell exchange $(\text{exch } x y (z) S)$ bind z with scope S , and a clause $((\bar{a}:\bar{y}) S)$ 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 and configurations C).

We identify statements S , data structures D , and configurations C up to consistent re-naming of bound variables and assume once and for all that bound variables are pairwise distinct and distinct from the free variables. Further, we identify S , D , and C up to the structural congruence given in Figure 2. Parallel composition of statements is commutative, associative, and has the neutral element skip . This is, statements are identified up to the equations $S_1 | S_2 \equiv S_2 | S_1$, $(S_1 | S_2) | S_3 \equiv S_2 | (S_1 | S_3)$, and $\text{skip} | S \equiv S \equiv S | \text{skip}$. 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 of names to a name that is not contained in it. A *store function* σ is a finite partial function $\sigma : \text{Var} \cup \text{Nam} \rightarrow \text{Nam} \cup \text{Dat}$ with the properties:

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

We denote the empty store as ε . Let $D(\sigma)$ (*domain*) denote the set of variables for which $\sigma(x)$ is defined, and $R(\sigma)$ (*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 σ such that $D(\sigma) \cup R(\sigma) \subseteq V$. We say that x is *bound* by the store $V\sigma$ if $x \in D(\sigma)$ and that x is *unbound* in $V\sigma$ if $x \in V \setminus D(\sigma)$. If x is bound by $V\sigma$ then we say that x is *bound to* the structure $\sigma(\sigma(x))$ with name $\sigma(x)$. Names model the *location* of a cell. Names for records and procedures are not needed in this paper; but they can support for an untyped equality test at all data structures¹, and they make the presentation more homogeneous.

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

$$\sigma, x \mapsto D = \begin{cases} \sigma & \text{if } x \in D(\sigma) \\ \sigma[n/x][D/n] & \text{if } x \notin D(\sigma), \\ & n = \text{new}(N(\sigma)) \end{cases} \quad \sigma, x \mapsto n = \begin{cases} \sigma & \text{if } x \in D(\sigma) \\ \sigma[n/x] & \text{if } x \notin D(\sigma) \end{cases}$$

Note that both extension operations preserve the bindings in a store. Attempts to rebind a bound variable are ignored, and an unbound variable is always bound to a *new* name.

¹ See OPM [28] and compare eq in Scheme

$V\sigma \parallel (\text{local}(x) S)$	$\rightarrow V \cup \{x\} \sigma \parallel S$	if $x \notin V$
$V\sigma \parallel x := y$	$\rightarrow V\sigma, x \mapsto \sigma(y) \parallel \text{skip}$	if $y \in D(\sigma)$
$V\sigma \parallel x := D$	$\rightarrow V\sigma, x \mapsto D \parallel \text{skip}$	
$V\sigma \parallel (x y)$	$\rightarrow V\sigma \parallel S[y/z]$	if $\sigma(\sigma(x)) = (\text{proc}(z) S)$
$V\sigma \parallel (\text{exch } x y (z) S)$	$\rightarrow V\sigma[(\text{cell } y)/\sigma(x)] \parallel S[z'/z]$	if $\sigma(\sigma(x)) = (\text{cell } z')$
$V\sigma \parallel (\text{match } x ((\bar{a}:\bar{y}) S))$	$\rightarrow V\sigma \parallel S[\bar{z}/\bar{y}]$	if $\sigma(\sigma(x)) = \{\bar{a}:\bar{z} \dots\}$

Fig. 3. Operational Semantics of Plain

To model cells, we also use *substitution* on stores in the form $\sigma[D/\sigma(x)]$ to simultaneously alter the binding of all variables x' with $\sigma(x') = \sigma(x)$. For instance, if $\sigma = \varepsilon, n \mapsto (\text{cell } y'), x \mapsto n, x' \mapsto n$ then $\sigma[(\text{cell } y)/\sigma(x)] = \varepsilon, n \mapsto (\text{cell } y), x \mapsto n, x' \mapsto n$ where the bindings of both variables 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 defined as the smallest binary relation on configurations which satisfies the rules in Figure 3 and is closed under the following rule.

$$\frac{V\sigma \parallel S_1 \rightarrow V'\sigma' \parallel S_2}{V\sigma \parallel S_1 \mid S \rightarrow V'\sigma' \parallel S_2 \mid S}$$

Reduction of a declaration $(\text{local}(x) S)$ 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 the store by the binding of x to D without further preconditions. The following example illustrates the dynamic extension of the store.

$$\begin{aligned} \{x\}\varepsilon \parallel (\text{local}(y) y := \{a:x\} \mid x := y) \\ &\rightarrow \{x, y\}\varepsilon \parallel y := \{a:x\} \mid x := y \\ &\rightarrow \{x, y\}\varepsilon, y \mapsto \{a:x\} \parallel x := y \\ &\rightarrow \{x, y\}\varepsilon, y \mapsto \{a:x\}, x \mapsto \{a:x\} \parallel \text{skip} \end{aligned}$$

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 $(\text{exch } x y (z) S)$ can be reduced if x is bound to a cell $(\text{cell } z')$; then it alters the content z' of the above cell to y and continues with $S[z'/z]$. For instance, we assume $\sigma = \varepsilon, x \mapsto (\text{cell } y)$, $\sigma' = \varepsilon, x \mapsto (\text{cell } y')$, and $V = \{x, y, y'\}$.

$$\begin{aligned} V\sigma' \parallel y := x \mid (\text{exch } x y (z) (\text{exch } x z (z') \text{ skip})) \\ &\rightarrow V\sigma', y \mapsto \sigma'(x) \parallel (\text{exch } x y (z) (\text{exch } x z (z') \text{ skip})) \\ &\rightarrow V\sigma, y \mapsto \sigma'(x) \parallel (\text{exch } x y' (z') \text{ skip}) \\ &\rightarrow V\sigma', y \mapsto \sigma'(x) \parallel \text{skip} \end{aligned}$$

A matching statement $(\text{match } x ((\bar{a}:\bar{y}) S))$ can be reduced if the store binds x to a record $\{\bar{b}:\bar{z}\}$ which matches the pattern $(\bar{a}:\bar{y})$. A pattern $(\bar{a}:\bar{y})$ is *matched* by all records

which have at least fields \bar{a} , *i.e.*, by all records of the form $\{\bar{a}:\bar{z} \dots\}$. A special case of matching is field selection on records. For instance, let $\sigma = \varepsilon, y \mapsto \{a:x \ b:y\}$ and $V = \{x, y\}$.

$$\begin{aligned} V\sigma \parallel (\text{match } y \ (b:z) \ x := z) &\rightarrow V\sigma \parallel x := y \\ &\rightarrow V\sigma, x \mapsto \{a:x \ b:y\} \parallel \text{skip} \end{aligned}$$

Example 1. As is well-known, records can be used to encode polyadic procedures. For instance, fix infinitely many distinct labels a_1, a_2, \dots and define for all n :

$$\begin{aligned} x := (\text{proc}(y_1 \dots y_n) S) &\stackrel{\text{def}}{=} x := (\text{proc}(z) (\text{match } z \ ((a_1:y_1 \dots a_n:y_n) S))) \\ (x \ \bar{y}) &\stackrel{\text{def}}{=} (\text{local}(z) \ z := \{a_1:y_1 \dots a_n:y_n\} \mid (x \ z)) \end{aligned}$$

Concurrency. Concurrent threads $S_1 \mid S_2$ in Plain can synchronize on the presence of data structures in the store as illustrated by the following example.

$$\begin{aligned} \{x, y\} \varepsilon \parallel x := y \mid y := \{a:x\} &\rightarrow \{x, y\} \varepsilon \parallel x := y \mid y := \{a:x\} \\ &\rightarrow \{x, y\} \varepsilon, y \mapsto \{a:x\} \parallel x := y \mid \text{skip} \\ &\rightarrow \{x, y\} \varepsilon, y \mapsto \{a:x\}, x \mapsto \{a:x\} \parallel \text{skip} \end{aligned}$$

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 we can place the resource in a cell and have each agent execute a cell exchange. The first agent to reduce will get the resource.

3 Typing and Subtyping

In this section we present the type system of Plain. It is obtained by adaptation from Pict's type system [23] which in turn foots on a long tradition of type systems for functional languages (*e.g.*, see the overviews [4,6]).

Types are Protocols. The communication between concurrent threads and the store in a configuration is mediated through logic variables. For this communication to take work nicely there must be consensus between the threads on the access protocols on the shared variables. These protocol include structural information (“which data structures may a variable be bound to?”) and modality information (“is it legal to read from and/or write to a variable?”). Types are a means to describe such access protocols for variables. We write $x:T$ for the *assumption* that variable x has type T . Typical type assumptions include

$x:?\text{int}$	reading from variable x will yield an integer; writing is not allowed.
$x:\text{!int}$	the variable x may be bound to an integer; reading is prohibited.
$x:\text{^int}$	both reading and writing integers from and to x is allowed.
$x:?\{\bar{a}:\bar{T}\}$	reading from variable x will yield a record which matches the pattern $(\bar{a}:\bar{y})$ (labels other than \bar{a} may or may not be present); selection of the fields at \bar{a} will yield variables with types \bar{T} , respectively.
$x:\{\bar{a}:\bar{y}\}$	the variable x may be bound to any record provided it has at least labels \bar{a} such that the corresponding fields have types \bar{T} .

Finite sets of type assumptions for distinct variables x_1, \dots, x_n are called *type environments*, written $x_1:T_1, \dots, x_n:T_n$. Protocol validation is formalized by a *type system* which answers the question whether or not a configuration $V\sigma \sqcup S$ respects a type environment Γ at any time during reduction, written $\Gamma \triangleright C$. *Subtyping* defines an order $\Gamma \leq \Gamma'$ on type environments such that C respects Γ whenever C respects Γ' ; this order is obtained by lifting a corresponding order on types $T \leq T'$ pointwise to environments. Typical subtypings include:

$?\text{num} \leq ?\text{int}$	if only integers are read from a variable then, since integers are numbers, the protocol to read only numbers is also respected.
$\text{^}T \leq ?T$	if only structures of type T are read from a variable, then, since reading and writing includes read-only, the protocol to read from or write to this variable is also respected.

The technical setup of our type system is as usual. We define a proof system for *judgements* $\Gamma \triangleright C$. If a judgement $\Gamma \triangleright C$ 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 .

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

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

Types. Figure 4 defines the abstract syntax of types. For technical reasons, we use *two* syntactic categories of *types* ranged over by P and T , respectively. If a distinction is necessary, we call P a *pretype*. A *mode* is either read-only ($?$), write-only ($!$), or allows both reading and writing (^). A *pretype* P is a type with its top-level mode stripped off.

The three leftmost pretypes are *procedure types* ($\text{proc } T$), *cell types* ($\text{cell } T$), and *record types* $\{\bar{a}:\bar{T}\}$. 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 $\{\}$.

$P ::= (\text{proc } T) \mid (\text{cell } T) \mid \{\bar{a}:\bar{T}\} \mid \alpha \mid \text{top} \mid \exists\alpha \leq P_1.P_2$
$T ::= \wedge P \mid ?P \mid !P$

Fig. 4. Types

$P \leq P$	(REFL)	$\frac{P_1 \leq P_2 \quad P_2 \leq P_3}{P_1 \leq P_3}$	(TRANS)
$\frac{P_1 \leq P_2}{?P_1 \leq ?P_2}$	(READ)	$\frac{P_1 \leq P_2}{!P_2 \leq !P_1}$	(WRITE)
$\wedge P \leq ?P$	(READSUB)	$\wedge P \leq !P$	(WRITESUB)
$\frac{T \leq T'}{(\text{proc } T') \leq (\text{proc } T)}$	(PROCSUB)	$\frac{\bar{T} \leq \bar{T}'}{\{\bar{a}:\bar{T} \dots\} \leq \{\bar{a}:\bar{T}'\}}$	(RECSUB)

Fig. 5. Monomorphic Subtyping

$\frac{\alpha \leq P \in \Gamma}{\Gamma \triangleright \alpha \leq P}$	(BND)	$\Gamma \triangleright P_2[P_1/\alpha] \leq \exists\alpha \leq P_1.P_2$	(ABSTR)
$\Gamma \triangleright P \leq \text{top}$	(TOP)	$\frac{\Gamma \triangleright P_1 \leq P'_1 \quad \Gamma, \alpha \leq P_1 \triangleright P_2 \leq P'_2}{\Gamma \triangleright \exists\alpha \leq P_1.P_2 \leq \exists\alpha \leq P'_1.P'_2}$	(POLY)

Fig. 6. Polymorphic Subtyping

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 [15,6,5]. Polymorphic types $\exists\alpha \leq P_1.P_2$ describe objects of “concrete” type $P_2[P'_1/\alpha]$ for some unknown P'_1 which is known 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 $\exists\alpha \leq \text{top}.P$.²

Monomorphic Subtyping. Subtyping on monomorphic types is the smallest relation on types 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.³ A procedure type $(\text{proc } T)$ is a subtype of $(\text{proc } T')$ if $T' \leq T$, i.e., if $(\text{proc } T)$ is applicable to *more* arguments than $(\text{proc } T')$. There is only

² This *bounded* existential polymorphism can express *partially abstract* types which are particularly useful in object-oriented programming ([22]; see also [16] for references).

³ This rule could be replaced by two rules, which express the addition of fields and the covariant specialization, respectively. If one disallows the addition of fields one obtains what is often called “tuple subtyping”, e.g., in [23].

$\frac{x:T \in \Gamma}{\Gamma \triangleright x:T}$	(VAR)	$\frac{\Gamma \triangleright x:T \quad \Gamma \triangleright T \leq T'}{\Gamma \triangleright x:T'}$	(SUB)
$\frac{\Gamma \triangleright x:!P \quad \Gamma \triangleright y:?P}{\Gamma \triangleright x:=y}$	(ASGN1)	$\frac{\Gamma \triangleright x:!P \quad \Gamma \triangleright D:?P}{\Gamma \triangleright x:=D}$	(ASGN2)
$\frac{\Gamma, y:T \triangleright S}{\Gamma \triangleright (\text{proc}(y) S):?(\text{proc } T)}$	(PROC)	$\frac{\Gamma \triangleright x:?(\text{proc } T) \quad \Gamma \triangleright y:T}{\Gamma \triangleright (x y)}$	(APPL)
$\frac{\Gamma \triangleright y:T}{\Gamma \triangleright (\text{cell } y):?(\text{cell } T)}$	(CELL)	$\frac{\Gamma \triangleright \bar{y}:\bar{T}}{\Gamma \triangleright \{\bar{a}:\bar{y}\}:? \{\bar{a}:\bar{T}\}}$	(REC)
$\frac{\Gamma \triangleright x:?(\text{cell } T) \quad \Gamma \triangleright y:T \quad \Gamma, z:T \triangleright S}{\Gamma \triangleright (\text{exch } x y (z) S)}$		(EXCH)	
$\frac{\Gamma \triangleright x:?\exists \bar{\alpha} \leq \bar{P}. \{\bar{a}:\bar{T}\} \quad \Gamma, \bar{\alpha} \leq \bar{P}, \bar{y}:\bar{T} \triangleright S}{\Gamma \triangleright (\text{match } x ((\bar{a}:\bar{y}) S))}$		(MATCH)	

Fig. 7. Typing Statements

trivial subtyping for cells: Since a cell may be read to and written from, cell types must be non-variant.

Environments. In the polymorphic case where types may have free variables, 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.

Polymorphic Subtyping. 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 corresponding *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 2. Assuming an additional primitive type $\text{int} \leq \text{top}$, we obtain the subtyping chain $\{a:? \text{int } b:? \text{int}\} \leq \{a:? \text{int}\} \leq \exists \alpha \leq \text{int}. \{a:? \alpha\} \leq \exists \alpha \leq \text{top}. \{a:? \alpha\}$.

Judgements. Figure 7 defines the typing for 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 , respectively. The third judgement says that S is well-typed under the assumption of Γ .

Typing Statements. 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 their

top-level mode; the mode describes the data flow of an assignment. The rules (PROC), (REC) and (CELL) are simple. For an application ($x\ y$), an exchange ($\text{exch } x\ y\ (z\ S)$), or a matching ($\text{match } x\ ((\bar{a}:\bar{y})\ S)$) to be well-typed (rules (APPL), (EXCH), (MATCH)), x must allow read access. The types of further arguments must match the requirements by the type of x . The rule (MATCH) is central for the polymorphic type system; given ($\text{match } x\ ((\bar{a}:\bar{y})\ S)$), it “opens” the polymorphic record type $\exists \bar{\alpha} \leq \bar{P}. \{\bar{a}:\bar{T}\}$ of x within S . Within S only the bounds \bar{P} may be assumed about the type variables $\bar{\alpha}$. The trivial rules for parallel, declaration, and skip are omitted.

Well-typedness. 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, 6, and 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 $V\sigma \parallel S$ 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. The polymorphic identity ($\text{proc}(x)\ (\text{case } x\ ((i:x_1\ o:x_2)\ x_2 := x_1))$) can be typed as ($\text{proc } \exists \alpha \leq \text{top}. \{i:\alpha\ o:\alpha\}$). Observe that not the procedures has a polymorphic type itself but that it takes a polymorphic argument.

Example 4. As a typical example for existential polymorphism consider:

$$S \stackrel{\text{def}}{=} (\text{local}(x\ y)\ x := 1 \mid y := (\text{proc}(z)\ (\text{match } z\ ((z_1\ z_2)\ z_2 := z_1 + 1))) \mid z := \{a:x\ b:y\})$$

Here, we assume predefined the integer 1 and the addition function $+$ on integers, and we encode a binary procedure via a record $\{z_1\ z_2\}$ from which we freely omit the labels (see Example 1). S can be shown well-typed assuming the types $x:\text{int}$, $y:?\text{proc } ?\{?\text{int } !\text{int}\}$, $z:\exists \alpha \leq \text{top}. \{a:?\alpha\ b:?\text{proc } ?\{?\alpha\ !\text{int}\}\}$. The type of z is obtained by type abstraction from $\wedge \{a:?\text{int } b:?\text{proc } ?\{?\text{int } !\text{int}\}\}$. Full data encapsulation is achieved by the tight lexical scope of x and y . 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 types of the fields *share* this type variable α we can apply field b to field a to obtain an integer u' via ($\text{match } z\ ((a:u\ b:v)\ (\text{local}(w)\ w := \{u\ u'\}; (v\ w)))$).

Theorem 5 (Type Preservation). *If $V\sigma \parallel S \rightarrow V'\sigma' \parallel S'$ and $\vdash \Gamma \triangleright V\sigma \parallel S$, then there exists Γ' such that $\vdash \Gamma, \Gamma' \triangleright V'\sigma' \parallel S'$.*

Proof. The proof can be reduced to the following Lemma 6. □

Lemma 6. *If $\vdash \Gamma \triangleright S$ and $\vdash \Gamma(y) \leq \Gamma(x)$ then $\vdash \Gamma \triangleright S[y/x]$.*

Theorem 5 and the Proposition 7 below imply the absence of type errors in reductions of well-typed configurations (which, hence, “do not go wrong” [12]).

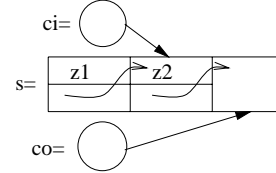
Proposition 7. *A well-typed configuration does not contain a type error.*

Extensions. 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., [4]).

4 Encoding Channels

We give an encoding of channels in Plain. Based on this encoding it is possible to encode Pict into Plain such that types are preserved. We have omitted this encoding for lack of space.

A *channel* is an infinite list s of slots along with two pointers ci and co into it. Slots can be empty and are filled with variables during computation. Once filled, a slot can never turn empty. On creation, the list s is empty and both ci and co point to the first slot. There is a get and an asynchronous put operation on channels. The put operation inserts a variable into the slot pointed to by co and advances co . The get operation takes a continuation $cont$, waits for the slot pointed to by ci to contain a variable z , and then applies $cont$ to z .



```
(newchan chan) |
(match chan (g:get p:put)
  (local z1 z2 z3 cont
    (put z1) | (put z2) |
    cont := (proc(y) z3 := y) |
    (get cont) | S))
```

We need a procedure *newchan* that binds its argument to a new channel with the specified operations. For instance, the statement to the left should reduce to $z1 := z3 \mid S$ over the store depicted in the previous paragraph. An implementation of such a procedure *newchan* in Plain is presented below.

An application of *newchan* declares the variables s_0 , ci , co , put , and get . The stream is modeled as an incomplete, initially empty list referred to by s_0 . The pointers co and ci

are implemented by cells with initial content s_0 . Put and get operations are realized as unary procedures *put* and *get* wrapped in a record $(p:put\ g:get)$ and bound to the argument *chan* of *newchan*. Applying *put* to a variable z_1 creates a new slot for z_1 in the stream: For a fresh variable s_2 , the content of the cell *co* is assigned the new record $(z_1\ s_2)$ and replaced with s_2 . Applying *get* first replaces the content s'_1 of *ci* with a fresh variable s'_2 . It waits for s'_2 to be bound to a record $(z'_1\ s_3)$ (i.e., to point to a filled slot) and then “advances” *ci* by assigning s_3 to s'_2 .

```

newchan:=
(proc(chan)
  (local(s0 co ci put get)
    co := (cell s0) | ci := (cell s0) |
    put := (proc(z1)
      (local(s2)
        (exch co s2 (s1)
          s1 := {z1 s2})))) |
    get := (proc(cont)
      (local(s2)
        (exch ci s'_2 (s'_1)
          (match s'_1 (z'_1 s3)
            s'_2 := s3 | (cont z1)))))) |
    chan := {p:put g:get}
  )
)

```

Typing. To type the channel encoding above we assume recursive types and define list types $Mlist(T) = \mu\alpha.M\{T\ M\alpha\}$ for all $M \in \{?, !, ^\}$. Now the type of channels for variables of type T can be defined as

$$ch(T) \stackrel{def}{=} \{i:?(proc\ ?(proc\ T))\ o:?(proc\ T)\}$$

Given this definition, the channel creation procedure *newchan* has type

$$newchan : (proc\ !\exists\alpha\leq top.ch(?\alpha))$$

This type implies that *newchan* creates no channels other than for read-only variables, and it 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 and which require *newchan* to have the type $(proc\ !\exists\alpha\leq top.ch(!\alpha))$. In order to avoid code duplication we might want to have polymorphic types $\exists\beta\leq T.P$ which abstract over *types* instead of *pretypes*. We then can show *newchan* to have type

$$newchan : (proc\ !\exists\beta\leq top.ch(\beta))$$

To see this, assume the following types for the variables in our encoding.

$$\begin{array}{lll}
get : ?(proc\ ?(proc\ \beta)) & put : ?(proc\ \beta) & ci, co : ^{(cell\ ^list(\beta))} \\
s_0 : ^list(\beta) & s'_2, s_1 : !list(\beta) & s'_1, s_2, s_3 : ?list(\beta) \\
cont : ?(proc\ \beta) & z_1, z'_1 : \beta &
\end{array}$$

Finally note that Plain’s subtyping rules induce Pict’s channel subtyping: Channel types $\{i:?(proc\ ?(proc\ T))\ o:?(proc\ T)\}$ can be restricted to input or output separately (due to record subtyping), and, given the abbreviations

$$ich(T) \stackrel{def}{=} \{i:?(proc\ ?(proc\ T))\} \quad \text{and} \quad och(T) \stackrel{def}{=} \{o:?(proc\ T)\},$$

the relation $T \leq T'$ implies $ich(T) \leq ich(T')$ and $och(T') \leq och(T)$.

5 Comparison to OPM

The essential changes of Plain with respect to OPM [28] are the absence of unification and the fact that cell exchange comes with a continuation. Both changes are motivated by and necessary in order to make the type system work.

Unification. Note that the operational service of unification is to *direct equations dynamically*. To see this assume Plain to be extended by unification in terms of an (undirected) equation $x=y$ and consider two typical configurations.

$$Vx=\{\} \mid x=y \mid (\text{match } y \mid () \mid S)) \qquad Vy=\{\} \mid x=y \mid (\text{match } x \mid () \mid S))$$

Both configurations reduce in two steps to $Vx=\{\}, y=\{\} \mid S$ where the equation $x=y$ in the left configuration, has effectively assigned (the value $\{\}$ of) x to y , and in the right configuration assigned y to x .

Since strong typing requires the types of variables to be known statically, the best possible typing rule for equations $x=y$ without losing type preservation is

$$\frac{\Gamma \triangleright x:\hat{P} \quad \Gamma \triangleright y:\hat{P}}{\Gamma \triangleright x=y}$$

which trivializes subtyping. Hence we replaced equations $x=y$ by assignment in Plain. (Future work will investigate whether and how it is possible to put constraints and unification back in without losing type safety at all, or such that type safety can be proved for program fragments adhering to a certain discipline. It is likely that this question has different answers for different constraint systems.)

Cell Exchange. The cell exchange ($\text{exch } x \ y \ z$) in OPM makes use of equations $x=y$. In Plain-style, its operational semantics would appear as

$$V\sigma \mid (\text{exch } x \ y \ z) \rightarrow V\sigma[(\text{cell } y)/\sigma(x)] \mid z'=z \quad \text{if } \sigma(\sigma(x)) = (\text{cell } z')$$

An immediate option is to replace the equation by an assignment. However neither $z:=z'$ nor $z':=z$ is the preferred candidate since both are needed. Plain's modified cell exchange ($\text{exch } x \ y \ (z) \ S$) defers the decision at which mode to use the old content of x to the continuation S .

Another interesting option (in analogy to the encoding of polyadic procedures) is to have cells always hold records with some label a and to combine cell exchange with field selection at a :

$$(\text{exch } x \ y \ z) \mid (\text{match } z \mid ((\bar{a}:\bar{z}') \mid S))$$

Summary. Plain exemplifies the following argument. If we want a non-trivial *subtyping* (which is particularly interesting for typed object-oriented programming), then we need a statically directed form of equation, viz. *assignment*, and we must adapt the computational primitives such that they do not use the undirected form. Second, since

OPM does not fix *input/output* arguments of procedures *syntactically*, we need to express *modalities* in the type system. This paper has shown that Pict's type system carries over under these preconditions.

Acknowledgements. We grateful to David N. Turner for discussions on concurrent programming and Pict. The research reported in this paper has been supported by the Bundesminister für Bildung, Wissenschaft, Forschung, und Technologie (FKZ ITW 9601), the Esprit Working Group CCL II (EP 22457), and the Deutsche Forschungsgemeinschaft (SFB 378).

References

1. F. Bronsard, T. Lakshman, and U. S. Reddy. A Framework of Directionality for Proving Termination of Logic Programs. In *International Conference and Symposium on Logic Programming*, pp. 321–335, 1992.
2. L. Cardelli. A Semantics of Multiple Inheritance. In G. Kahn, D. MacQueen, and G. Plotkin, eds., *Semantics of Data Types*, LNCS 173, pp. 51–67. Springer, 1984. Full version in *Information and Computation*, 76(2/3):138–164, 1988.
3. L. Cardelli. An Implementation of $F_{<}$. Technical Report 97, Digital Systems Research Center, Feb. 1993.
4. L. Cardelli. Type Systems. In *CRC Handbook of Computer Science and Engineering*. CRC, 1996. to appear.
5. L. Cardelli and X. Leroy. Abstract Types and the Dot Notation. In *Proc. IFIP Working Conference on Programming Concepts and Methods*, pp. 466–491, 1990.
6. L. Cardelli and P. Wegner. On Understanding Types, Data Abstraction, and Polymorphism. *ACM Computing Surveys*, 17(4), Dec. 1986.
7. I. Foster. Strand and PCN: Two Generations of Compositional Programming Languages. Preprint MCS-P354-0293, Argonne National Laboratories, 1993.
8. K. Honda and N. Yoshida. On Reduction-Based Semantics. In R. K. Shyamasundar, ed., *13th Conference on Foundations of Software Technology and Theoretical Computer Science*, Bombay, India, Dec. 1993.
9. J. Jaffar and M. J. Maher. Constraint Logic Programming: A Survey. *The Journal of Logic Programming*, 19/20:503–582, May–July 1994.
10. A. Kleinman, Y. Moscovitz, A. Pnueli, and E. Shapiro. Communication with Directed Logic Variables. In *18th ACM Symposium on Principles of Programming Languages*, pp. 221–232, 1991.
11. M. J. Maher. Logic Semantics for a Class of Committed-Choice Programs. In J.-L. Lassez, ed., *International Conference on Logic Programming*, pp. 858–876. The MIT Press, Cambridge, MA, 1987.
12. R. Milner. A Theory of Type Polymorphism in Programming. *Journal of Computer and System Science*, 17:348–375, 1978.
13. R. Milner. The Polyadic π -Calculus: A Tutorial. In F. L. Bauer, W. Brauer, and H. Schwichtenberg, ed., *Proc. of the 1991 Marktoberdorf Summer School on Logic and Algebra of Specification*, NATO ASI Series. Springer, 1993.
14. 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.
15. J. Mitchell and G. Plotkin. Abstract Types have Existential Type. In *12th ACM Symposium on Principles of Programming Languages*, pp. 37–51, Jan. 1985.

16. J. C. Mitchell and K. Fisher. The Development of Type Systems for Object-oriented Languages. 1996. Background and Reference Material for John C. Mitchell's Sendai Lecture during TACS '94.
17. M. Müller. Polymorphic Types for Concurrent Constraints. Available at <http://www.ps.uni-sb.de/~mmueller/papers/ptcc.ps.gz>, 1996.
18. J. Niehren. Functional Computation as Concurrent Computation. In *23rd ACM Symposium on Principles of Programming Languages*, pp. 333–343, Florida, 1996.
19. J. Niehren and M. Müller. Constraints for Free in Concurrent Computation. In K. Kanchanasut and J.-J. Lévy, eds., *Asian Computing Science Conference*, LNCS 1023, pp. 171–186, Thailand, 1995. Springer.
20. B. Pierce and D. Sangiorgi. Typing and Subtyping for Mobile Processes. In *IEEE Symposium on Logic in Computer Science*, pp. 376–385, June 1993.
21. B. C. Pierce and D. N. Turner. Concurrent Objects in a Process Calculus. In *Theory and Practice of Parallel Programming (TPPP)*, LNCS, 1994. Springer.
22. B. C. Pierce and D. N. Turner. Simple Type-Theoretic Foundations For Object-Oriented Programming. *The Journal of Functional Programming*, 4(2):207–247, Apr. 1994.
23. B. C. Pierce and D. N. Turner. Pict: A Programming Language Based on the Pi-Calculus, May 1996. To appear in *Milner Festschrift*, The MIT Press, 1997.
24. Programming Systems Lab, Universität des Saarlandes. The Oz Programming System, 1996. <http://www.ps.uni-sb.de/www/oz/>.
25. V. A. Saraswat. *Concurrent Constraint Programming*. The MIT Press, Cambridge, MA, 1993.
26. E. Shapiro. The Family of Concurrent Logic Programming Languages. *ACM Computing Surveys*, 21(3):413–510, Sept. 1989.
27. G. Smolka. A Foundation for Concurrent Constraint Programming. In J.-P. Jouannaud, ed., *1st International Conference on Constraints in Computational Logics*, LNCS 845, pp. 50–72, 1994. Springer.
28. G. Smolka. The Oz Programming Model. In J. van Leeuwen, ed., *Computer Science Today*, LNCS 1000, pp. 324–343. Springer, 1995.
29. B. Victor and J. Parrow. Constraints as Processes. In U. Montanari and V. Sassone, eds., *7th International Conference on Concurrency Theory*, LNCS 1119, pp. 389–405, Springer 1996.