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) [27], which formulates the essence of the programming language Oz [23]. OPM extends the concurrent constraint model [24, 12] with first-class procedures and first-class cells. The concurrent constraint model merges two lines of research in logic programming, addressing concurrency [25] and constraints [10]. 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 expressivity 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. This provides 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, 22], a recent concurrent programming language based on the π-calculus [14, 15, 9]. 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, cells, sequential composition, and thread creation. 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 the combination of sequential composition, locks, logic variables and synchronization by blocking proves essential. Since most programming techniques from OPM carry over to Plain, we restrict ourselves to few programming examples in this paper and refer to [27] for many 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.*, [11, 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 [26] 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 [28] give an embedding of the γ-calculus into the π-calculus and prove its adequacy based on bisimulation semantics.

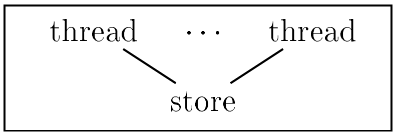
The paper is organized as follows. Section 2 presents Plain and specifies its abstract syntax and operational semantics. Section 3 presents the type system and the type preservation theorem. Section 4 exemplifies the language by discussing a well-typed polymorphic procedure that creates data structures modeling channels.

$$
\begin{array}{lcl}
D & ::= & (\mathsf{proc}(y)\ S) \ \ | \ \ (\mathsf{cell}\ y) \ \ | \ \ \{\overline{a}{:}\overline{y}\} \\
S & ::= & \mathsf{skip} \ \ | \ \ (\mathsf{local}(x)\ S) \ \ | \ \ x := y \ \ | \ \ x := D \ \ | \\
 & & (x\ y) \ \ | \ \ (\mathsf{exch}\ x\ y\ (z)\ S) \ \ | \ \ (\mathsf{case}\ x\ C_1 \ldots C_n) \ \ | \\
 & & S; S' \ \ | \ \ (\mathsf{thread}\ S) \\
C & ::= & ((\overline{a}{:}\overline{y})\ \ S) \\
R & ::= & S \ \ | \ \ R \,|\, R'
\end{array}
$$

Figure 1: Syntax of Plain

2 Plain

The Computation Model. Plain organizes a computation into a number of *threads* computing over a *shared store*. Each thread represents 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. Each thread maintains a stack of statements and tries to execute the topmost statement on its stack. 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.



Plain's store binds variables to data structures, which are procedures, cells, or records. Data structures may contain variables. Typically, only some of these variables are bound by the store. Thus, Plain's store accomodates for partial data structures. Once a binding is made in the store, it cannot be retracted. However, cell bindings may be altered. Statements synchronize on the property that the store binds all their input variables. Since bindings cannot be retracted, a synchronized statement remains reducible once it has become reducible. 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 $\overline{x}$ for a finite sequence of variables, $\overline{a}$ for a finite sequence of labels, and $\overline{a}{:}\overline{y}$ 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 as given in Figure 1 is built from variables and labels. There are four syntactical categories, (data) structures D, statements S, clauses C, and pools R. The first three categories define the syntax of Plain programs, while pools are created on reduction, *i.e.*, dynamically.

There are several variable binders used in expressions of the syntactic categories in this paper. We identify expressions up to consistent renaming of bound variables and assume once and for all that all bound variables in an expression are pairwise distinct and distinct from all its free variables. The set of *free variables* in an expression X of some syntactic category is denoted with $\mathcal{FV}(X)$.

Data Structures. A *structure* D is a procedure, a cell, or a record. A (monadic) *procedure* $(\mathsf{proc}(x)\ S)$ consists of a *formal argument* x and a *body* S where x is bound with scope S. A *cell* $(\mathsf{cell}\ x)$ is a container with *contents* x, which may be exchanged by reduction. A *record* $\{\overline{a}{:}\overline{y}\}$ has *fields* $\overline{y}$ that can be accessed by pairwise distinct labels $\overline{a}$. We identify records up to reordering of label-field pairs, for instance $\{a_1{:}y_1\ a_2{:}y_2\} = \{a_2{:}y_2\ a_1{:}y_1\}$. The sets of structures Dat is the union of the sets of procedures Proc, cells Cell, and records Rec.

Store. A *store* $V\sigma$ consists of a finite set of variables V and a function σ that represents a set of bindings of variables in V to some structure D. For a formal definition, 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 $\sigma : \mathsf{Var} \cup \mathsf{Nam} \rightharpoonup \mathsf{Nam} \cup \mathsf{Dat}$ with the properties:

1. $\sigma(\mathsf{Var}) \subseteq \mathsf{Nam} \cup \mathsf{Rec}$ and $\sigma(\mathsf{Nam}) \subseteq \mathsf{Proc} \cup \mathsf{Cell}$.
2. if $\sigma(x) \in \mathsf{Nam}$ then $\sigma(\sigma(x))$ is defined.

We denote the empty store as ε. The *variable domain* $\mathcal{D}(\sigma)$ of σ is the set of all variables x for which $\sigma(x)$ is defined. The *variable range* $\mathcal{R}(\sigma)$ of σ is the set of all variables that occur free in a structure in the range of σ. A *store* $V\sigma$ is a pair of a set of variables V and a σ such that $\mathcal{D}(\sigma) \cup \mathcal{R}(\sigma) \subseteq V$. We say that x is *bound* by the store $V\sigma$ if $x \in \mathcal{D}(\sigma)$ and that x is *unbound* in $V\sigma$ if $x \in V \setminus \mathcal{D}(\sigma)$. There are three kinds of bindings. If $\sigma(x)$ is a name then σ *binds* x to $\sigma(\sigma(x))$, which is a procedure or a cell with name $\sigma(x)$. If $\sigma(x)$ is a record then σ binds x to $\sigma(x)$. Names for procedures are not needed in this paper but we use them as in OPM [27].

A store allows for two *tell* operations that add new variable bindings. Let $\mathcal{N}(\sigma)$ be the set of names n such that $\sigma(n)$ is defined. The tell operations are defined as follows:

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

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

The symbol $[\cdot/\cdot]$ stands for the substitution operator that replaces its first argument for the second one. Note that the tell operations $x{\mapsto}D$ and $x{\mapsto}n$ preserves all bindings of a store since attempts to rebind a bound variable are ignored.

A store allows for a *substitution* operation $\sigma[(\mathsf{cell}\ y)/\sigma(x)]$ if x is bound to a cell in σ. This operation alters the content of the cell $\sigma(\sigma(x))$ but not its name $\sigma(x)$. For instance, if $\sigma = \varepsilon, n{\mapsto}(\mathsf{cell}\ y'), x{\mapsto}n, x'{\mapsto}n$ then $\sigma[(\mathsf{cell}\ y)/\sigma(x)] = \varepsilon, n{\mapsto}(\mathsf{cell}\ y), x{\mapsto}n, x'{\mapsto}n$. Since σ binds x and x' to cells with the same name n, an application of the substitution $[(\mathsf{cell}\ y)/\sigma(x)]$ to σ alters both the bindings of x and x'.

Statements. The abstract syntax of *statement* S is given in Figure 1. In the first line, we have the null statement skip, a *declaration* $(\mathsf{local}(x)\ S)$, 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 paramenter* y, a *cell exchange* $(\mathsf{exch}\ x\ y\ (z)\ S)$ on x with new content y for some old content z and continuation S, and a *case statement* $(\mathsf{case}\ x\ C_1 \ldots C_n)$ for x with clauses $C_1, \ldots, C_n$. A *clause* $((\overline{a}{:}\overline{y})\quad S)$ consists of a *pattern* $(\overline{a}{:}\overline{y})$ and a continuation S. The third line of statements contains *sequential composition* $S; S'$ and *thread creation* $(\mathsf{thread}\ S)$.

We assume that skip is a left unit with respect to sequential composition, *i.e.*, we identify statements up to the equations $\mathsf{skip}; S = S$. Case statements are identified up to reordering of clauses. A declaration $(\mathsf{local}(x)\ S)$ binds x with scope S, a cell exchange $(\mathsf{exch}\ x\ y\ (z)\ S)$ binds z with scope S, and a clause $((\overline{a}{:}\overline{y})\quad S)$ binds the pattern variables $\overline{y}$ with scope S.

Operational Semantics. The operational semantics of Plain is defined in terms of a one-step reduction relation $\rightarrow$ on configurations describing computation states. A *configuration* $V\sigma\,[]\;\; R$ consists of a store $V\sigma$ and a *pool* R such that $\mathcal{FV}(R) \subseteq V$. A pool R is a *parallel composition* of statements. We identify pools up to associativity and commutativity of parallel composition, i.e. up to the equations $R_1 \mid R_2 = R_2 \mid R_1$ and $(R_1 \mid R_2) \mid R_3 = R_2 \mid (R_2 \mid R_3)$.

Sequential reduction $\rightarrow_1$ and *concurrent reduction* $\rightarrow$ are defined as the smallest binary relations on configurations satisfying the rules in Figure 2.

Sequential Computation. Sequential reduction $\rightarrow_1$ fully defines the operational semantics of statements S that do not contain thread creation and thus a complete sequential language with lexical scoping.

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 and then tells a binding of x to

Sequential Computation

$$V\sigma\,[]\ \ (\mathsf{local}(x)\ S)\ \ \rightarrow_1\ \ V{\cup}\{x\}\ \sigma\,[]\ \ S \qquad \text{if } x \notin V$$

$$V\sigma\,[]\ \ x := y\ \ \rightarrow_1\ \ V\sigma, x{\mapsto}\sigma(y)\,[]\ \ S \qquad \text{if } y \in \mathcal{D}(\sigma)$$

$$V\sigma\,[]\ \ x := D\ \ \rightarrow_1\ \ V\sigma, x{\mapsto}D\,[]\ \ \mathsf{skip}$$

$$V\sigma\,[]\ \ (x\ y)\ \ \rightarrow_1\ \ V\sigma\,[]\ \ S[y/z] \qquad \text{if } \sigma(\sigma(x)) = (\mathsf{proc}(z)\ S)$$

$$V\sigma\,[]\ \ (\mathsf{exch}\ x\ y\ (z)\ S)\ \ \rightarrow_1\ \ V\sigma[(\mathsf{cell}\ y)/\sigma(x)]\,[]\ \ S[z'/z] \qquad \text{if } \sigma(\sigma(x)) = (\mathsf{cell}\ z')$$

$$V\sigma\,[]\ \ (\mathsf{case}\ x\ ((\overline{a}{:}\overline{y})\ \ S)\ C_2 \ldots C_n)\ \ \rightarrow_1\ \ V\sigma\,[]\ \ S[\overline{z}/\overline{y}] \qquad \text{if } \sigma(x) = \{\overline{a}{:}\overline{z}\ldots\}$$

$$\frac{V\sigma\,[]\ \ S_1 \rightarrow_1 V'\sigma'\,[]\ \ S_2'}{V\sigma\,[]\ \ S_1; S_2 \rightarrow_1 V'\sigma'\,[]\ \ S_1'; S_2}$$

Concurrent Computation

$$V\sigma\,[]\ \ (\mathsf{thread}\ S_1); S_2\ \ \rightarrow\ \ V\sigma\,[]\ \ S_1 \mid S_2$$

$$\frac{V\sigma\,[]\ \ S \rightarrow_1 V'\sigma'\,[]\ \ S'}{V\sigma\,[]\ \ S \mid R \rightarrow V'\sigma'\,[]\ \ S' \mid R}$$

Figure 2: Operational Semantics of Plain

$\sigma(y)$ to the store. A *data assignment* $x := D$ tells a binding of x to D to the store. The closure rule for sequential reduction $\rightarrow_1$ says that a thread is reduced from the left to right. In the following examples, new variables and variable bindings are told to the store.

$$\begin{array}{ll} \{x\}\varepsilon\,[]\ \ (\mathsf{local}(y)\ y := \{a{:}x\}; x := y) & \\ \qquad \rightarrow_1 & \{x,y\}\varepsilon\,[]\ \ y := \{a{:}x\}; x := y \\ \qquad \rightarrow_1 & \{x,y\}\varepsilon, y{\mapsto}\{a{:}x\}\,[]\ \ x := y \\ \qquad \rightarrow_1 & \{x,y\}\varepsilon, y{\mapsto}\{a{:}x\}, x{\mapsto}\{a{:}x\}\,[]\ \ \mathsf{skip} \end{array}$$

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

An application $(x\ y)$ can be executed 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 above cell to y and continues with $S[z'/z]$. For instance, we assume the store functions $\sigma = \varepsilon, x{\mapsto}(\mathsf{cell}\ y)$ and $\sigma' = \varepsilon, x{\mapsto}(\mathsf{cell}\ y')$ and the variable set $V = \{x, y, y'\}$.

$$\begin{array}{rl} V\sigma' \,[]\; y := x; (\mathsf{exch}\ x\ y\ (z)\ (\mathsf{exch}\ x\ z\ (z')\ \mathsf{skip})) & \\ \to_1 & V\sigma', y{\mapsto}\sigma'(x)\,[]\; (\mathsf{exch}\ x\ y\ (z)\ (\mathsf{exch}\ x\ z\ (z')\ \mathsf{skip})) \\ \to_1 & V\sigma, y{\mapsto}\sigma'(x)\,[]\; (\mathsf{exch}\ x\ y'\ (z')\ \mathsf{skip}) \\ \to_1 & V\sigma', y{\mapsto}\sigma'(x)\,[]\; \mathsf{skip} \end{array}$$

A case statement ($\mathsf{case}\ x\ C_1 \ldots C_n$) can be reduced if the store binds x to a record $\{\overline{a}{:}\overline{z}\}$ and if $\{\overline{a}{:}\overline{z}\}$ matches one the pattern in the clauses C_i. A record $\{\overline{a}{:}\overline{z}\}$ *matches* all patterns of the form $(\overline{a}{:}\overline{y}\ldots)$. In particular, the case statement provides for field selection on records. For instance, let $\sigma = \varepsilon, y{\mapsto}\{a{:}x\ b{:}y\}$ and $V = \{x, y\}$.

$$\begin{array}{rcl} V\sigma\,[]\; (\mathsf{case}\ y\ (b{:}z)\ x := z)) & \to_1 & V\sigma\,[]\; x := y \\ & \to_1 & V\sigma, x{\mapsto}\{a{:}x\ b{:}y\}\,[]\; \mathsf{skip} \end{array}$$

Records can be used for encoding polyadic procedures in Plain. For instance, let $\overline{a}$ be a sequence of pairwise distinct elements.

$$\begin{array}{rcl} x := (\mathsf{proc}(\overline{y})\ S) & \stackrel{\text{def}}{=} & x := (\mathsf{proc}(z)\ (\mathsf{case}\ z\ ((\overline{a}{:}\overline{y})\ \ S))) \\ (x\ \overline{y}) & \stackrel{\text{def}}{=} & (\mathsf{local}(z)\ z := \{\overline{a}{:}\overline{y}\}; (x\ z)) \end{array}$$

Concurrent Computation. Concurrent reduction $\to$ is defined by lifting sequential reduction $\to_1$ to configurations with concurrent threads. Note that concurrent composition is not allowed as an operator in statements S. Instead, concurrent threads have to be created explicitely by using the statement ($\mathsf{thread}\ S$).

Concurrent threads in Plain are able to synchronize on the presence of data structures in the store as illustrated by the following example.

$$\begin{array}{rl} \{x, y\}\varepsilon\,[]\; (\mathsf{thread}\ x := y; S); y := \{a{:}x\} & \\ \to & \{x, y\}\varepsilon\,[]\; x := y; S \mid y := \{a{:}x\} \\ \to & \{x, y\}\varepsilon, y{\mapsto}\{a{:}x\}\,[]\; x := y; S \mid \mathsf{skip} \\ \to & \{x, y\}\varepsilon, y{\mapsto}\{a{:}x\}, x{\mapsto}\{a{:}x\}\,[]\; S \mid \mathsf{skip} \end{array}$$

Cells introduce indeterministic to concurrent computation in Plain since the result of cell exchanges depend on the order in that they are executed. This indeterminism is useful. For instance, we might wish to describe concurrent agents that compete for a single resource without determining the winning agent statically. We can do so by putting the recource into a cell and representing the agents as threads with a cell exchange on top. The thread executing its cell exchange first will get the resource.

$$
\begin{array}{rl}
P ::= & (\mathsf{proc}\ T) \ \mid\ (\mathsf{cell}\ T) \ \mid\ \{\overline{a}{:}\overline{T}\} \ \mid \\
& \alpha \ \mid\ \mathsf{top} \ \mid\ \exists\alpha{\le}P_1.P_2 \\
T ::= & \hat{}\,P \ \mid\ ?P \ \mid\ !P
\end{array}
$$

Figure 3: Types

3 Typing and Subtyping

In this section we present the type system of Plain. It is obtained by adaptation from Pict's type system [22] which in turn foots on a long tradition of type systems for functional languages. The goal of our type system is to distinguish a class of Plain statements that can be reduced without ever leading to a configuration containing a type error.

Type Errors. A configuration contains a *type error* if it has one of the following forms:

- $V\,\sigma\,[]\ \ (x\ y); S \mid R$ where σ binds x but not to a procedure.
- $V\,\sigma\,[]\ \ (\mathsf{exch}\ x\ y\ (z)\ S_1); S_2 \mid R$ where σ binds x but not to a cell.
- $V\,\sigma\,[]\ \ (\mathsf{case}\ x\ C_1 \ldots C_n); S' \mid R$ where σ binds x but not to a record or to a record that does not match any of the patterns in the clauses $C_1, \ldots, C_n$.

Types. Figure 3 defines two syntactic categories of *types* ranged over by P and T. If a distinction is necessary, we also call a type P a *pretype*. Types T are pretypes tagged with an access modality that we call mode for short. A *mode* is one of read (?), write (!), and both (ˆ). The first line of the abstract syntax of pretypes provides for *procedure types* ($\mathsf{proc}\ T$), *cell types* ($\mathsf{cell}\ T$), and *record types* $\{\overline{a}{:}\overline{T}\}$. We require that the labels of record types are pairwise distinct and identify record types up to reordering of label-type pairs. A *monomorphic type* is a type constructed from the above mentionned constructs only. The only primitive monomorphic type is the empty record type $\{\}$.

For polymorphic types, we assume an infinite set of type variable ranged over by α. The second line of the abstract syntax of pretypes contains type variables α, the maximal type top, and *package types* $\exists\alpha{\le}P_1.P_2$ that support type abstraction by existential polymorphism [16, 6, 5]. A *package type* $\exists\alpha{\le}P_1.P_2$ describes objects of "concrete" type $P_2[P_1'/\alpha]$ for some unknown P_1' which is a subtype of P_2 (subtyping is defined below). The type $\exists\alpha{\le}P_1.P_2$ binds α with scope P_2. The type top is

$$P \leq P \quad (\textsc{Refl}) \qquad \frac{P_1 \leq P_2 \quad P_2 \leq P_3}{P_1 \leq P_3} \quad (\textsc{Trans})$$

$$\frac{P_1 \leq P_2}{?P_1 \leq ?P_2} \quad (\textsc{Read}) \qquad \frac{P_1 \leq P_2}{!P_2 \leq !P_1} \quad (\textsc{Write})$$

$$\hat{}P \leq ?P \quad (\textsc{ReadSub}) \qquad \hat{}P \leq !P \quad (\textsc{WriteSub})$$

$$\frac{T \leq T'}{(\mathsf{proc}\ T') \leq (\mathsf{proc}\ T)} \quad (\textsc{ProcSub}) \qquad \frac{\overline{T} \leq \overline{T'}}{\{\overline{a}{:}\overline{T}\ \ldots\} \leq \{\overline{a}{:}\overline{T'}\}} \quad (\textsc{RecSub})$$

Figure 4: Monomorphic Subtyping

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

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

Figure 5: Polymorphic Subtyping

needed to express unbounded polymorphism in form of maximal type abstractions $\exists\alpha{\leq}\mathsf{top}.P$.

Subtyping. Types restrict the syntactic positions in which variables and structures may occur in a statement. A type T is called a *subtype* of another one T', written $T \leq T'$, if expressions of type T may be replaced for structures of type T' in well-typed programs. Subtyping on monomorphic types is defined as the smallest relation on types satisfying the inference rules for relation $\leq$ given in Figure 4.

A record type T is subtype of another one T' if T has at least the labels in T' and the corresponding fields of T and T' are in covariant subtype relationship.[1] 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 fewer arguments than ($\mathsf{proc}\ T$). There is only trivial subtyping for cells: Since a cell may be read to and written from, cell types must be non-variant.

In the polymorphic case where types may have free variables, subtyping is relative to an environment Γ which contains a set of *subtype assumptions* $\alpha{\leq}P$. To define

[1] A less permissive subtyping on records could enforce T and T' to have exactly the same fields. In Pict, these orders are tied to different kinds of data structures, namely tuples and records. We prefer to see tuples as special records and to have *two types* with a different subtyping order for just *one kind of data structure*.

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

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

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

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

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

$$\frac{\Gamma \triangleright x{:}?\exists\overline{\alpha}{\le}\overline{P}.\{\overline{a}{:}\overline{T}\} \quad \Gamma, \overline{\alpha}{\le}\overline{P}, \overline{y}{:}\overline{T} \triangleright S}{\Gamma \triangleright (\mathsf{case}\ x\ ((\overline{a}{:}\overline{y})\ \ S))} \;(\textsc{Case})$$

Figure 6: Typing Statements

polymorphic subtyping, we need judgements $\Gamma \triangleright P \le P'$ and $\Gamma \triangleright T \le T'$ meaning that Γ implies the subtype relations $P \le P'$ and $T \le T'$. We reinterpret all rules in Figure 4 such that their premisses and conclusions share the same Γ and add the rules from Figure 5. Rule (Abstr) says that the concrete type $P_2[P_1/\alpha]$ is a subtype of the abstract package $\exists\alpha{\le}P_1.P_2$. Subtyping between two polymorphic types $\exists\alpha{\le}P_1.P_2$ and $\exists\alpha{\le}P_1'.P_2'$ follows from covariant subtyping $P_1 \le P_1'$ and $P_2 \le P_2'$.

Example 3.1 Assuming an additional primitive type $\mathsf{int} \le \mathsf{top}$, we obtain the subtypings $\{a{:}?\mathsf{int}\ b{:}?\mathsf{int}\} \le \{a{:}?\mathsf{int}\} \le \exists\alpha{\le}\mathsf{int}.\{a{:}?\alpha\} \le \exists\alpha{\le}\mathsf{top}.\{a{:}?\alpha\}$. □

Environments and Judgements. An *environment* Γ is a set of subtype assumptions $\alpha{\le}P$ and of *type assumptions* $x{:}T$. An *extension* of an environment Γ by an assumption $x{:}T$ is written as adjunction $\Gamma, x{:}T$. The notions $\Gamma, \alpha{\le}P$ and Γ, Γ' are defined analogously.

For typing statements, we need *judgements* of the form $\Gamma \triangleright x{:}T$, $\Gamma \triangleright D{:}T$, and $\Gamma \triangleright S$. The first (second) judgement means that Γ ensures type T for x (resp. D). The third judgement says that S is well-typed under the assumption of Γ. The derivation of these judgements needs the subtyping judgements $\Gamma \triangleright P \le P'$ and $\Gamma \triangleright T \le T'$, whose inference rules are discussed above.

Typing Statements. Figure 6 defines typing rules for statements. Variables receive their type by lookup in the environment (Var) and can be promoted along the subtyping order (Sub). Assignments $x := y$ and $x := D$ require that x, y, and D have the same type up to their top-level mode which describes the data flow of an assignment. The rules (Proc), (Rec) and (Cell) are simple. For an application $(x\ y)$, an exchange ($\mathsf{exch}\ x\ y\ (z)\ S$), or a conditional ($\mathsf{case}\ x\ ((\overline{a}{:}\overline{y})\ \ S)$) to be well-typed (rules (Appl), (Exch), (Case)), x must allow read access. The types of further arguments must match the requirements of the type of x. The rule (Case) is central for the treatment of polymorphism; given ($\mathsf{case}\ x\ ((\overline{a}{:}\overline{y})\ \ S)$), it intuitively "opens" the polymorphic record type $\exists\overline{\alpha}{\leq}\overline{P}.\{\overline{a}{:}\overline{T}\}$ of x in the clause body S. Within S only the bound $\overline{P}$ may be assumed about the type variables $\overline{\alpha}$. The rules for sequential composition, declaration, thread creation, and skip are omitted since trivial.

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 4, 5, 6. A store σ is well-typed w.r.t. Γ iff $\vdash \Gamma \triangleright x := D$ for all x and D such that x is bound to D in σ. A pool is well-typed well-typed w.r.t. Γ if all its threads are well-typed w.r.t. Γ. A configuration $V\sigma\,[]\ \ R$ is well-typed w.r.t. Γ if both σ and R are. A statement, store, pool, or configuration is well-typed if it is well-typed w.r.t. some environment Γ.

Example 3.2 The polymorphic identity ($\mathsf{proc}(x)$ ($\mathsf{case}\ x\ ((i{:}x_1\ o{:}x_2)\ x_2 := x_1)$)) can be typed as ($\mathsf{proc}\ ?\exists\alpha{\leq}\mathsf{top}.\{i{:}?\alpha\ \ o{:}!\alpha\}$). Observe that not the procedures has a polymorphic type itself but that it takes a polymorphic argument. □

Example 3.3 As a typical example for existential polymorphism we consider the statement S equal to

$$(\mathsf{local}(x\ y)\ x := 1; y := (\mathsf{proc}(z_1\ \ z_2)\ z_2 := z_1{+}1); z := \{a{:}x\ b{:}y\})$$

Here, we use a predefined addition function $+$ on integers and assume the binary procedure y encoded with a binary record as argument of y (as defined above). S is well-typed if we assume the following types where we freely omit the labels of the argument record type. $\mathtt{x}{:}?\mathsf{int}$, $\mathtt{y}{:}?(\mathsf{proc}\ \ ?\{?\mathsf{int}\ !\mathsf{int}\})$, $\mathtt{z}{:}\hat{}\exists\alpha{\leq}\mathsf{top}.\{a{:}?\alpha\ \ b{:}?(\mathsf{proc}\ \ ?\{?\alpha\ \ !\mathsf{int}\})\}$. The type of $\mathtt{z}$ is obtained by type abstraction from $\hat{}\{a{:}?\mathsf{int}\ b{:}?(\mathsf{proc}\ ?\{?\mathsf{int}\ !\mathsf{int}\})\}$. Full data encapsulation is achieved by the tight lexical scope of x and y. The type of $\mathtt{z}$ allows selection of field at a and b but prohibits direct interaction with them due to the type variable α. But since the types of the fields share α we can apply field b to field a to obtain an integer $\mathtt{w}$ via ($\mathsf{case}\ z\ ((a{:}u\ b{:}v)\ (\mathsf{local}(w)\ w := \{u\ u'\}; (v\ w)))$). □

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

Proof. The proof can be reduced to the following Lemma 3.5. □

Lemma 3.5 *If* $\vdash \Gamma \triangleright S$ *and* $\vdash \Gamma(y) \leq \Gamma(x)$ *then* $\vdash \Gamma \triangleright S[y/x]$.

Theorem 3.4 and the following Proposition imply the absence of type errors in reductions of well-typed configurations (which, hence, "do not go wrong" [13]).

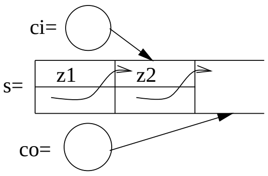
Proposition 3.6 *A well-typed configuration does not contain a type error.*

Extensions We have left out sum types, which are needed for typing case statements with multiple clauses. We have also not discussed recursive types $\mu\alpha.P$. Extensions with both kind of types are standard though (see, *e.g.*, [4]). As our channel encoding in the next section illustrates, we might also want package types $\exists\beta{\leq}T.P$ abstracting over types variables β instead of pretype variables α.

4 Encoding Channels

In this section 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 an input and an asynchronous output operation on channels. The output operation puts a variable into the slot pointed to by `co` and advances `co`. The input 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);
(case chan ({i:inp o:out}
 (local(z1 z2 z3 cont)
  (out z1); (out z2);
  cont:=(proc(y) y:=z3);
  (inp cont); S)))
```

We are seeking for a procedure newchan that binds its argument to a new a channel with the operations described above. For instance, the statement to the left should reduce to `z1:=z3;`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 s0, ci, co, inp, and out. The stream is modelled as an incomplete, initially empty list referred to by s0. The pointers co and ci are implemented by cells with initial content s0. Input and output operations are realized as unary procedures inp and out wrapped in a record {i:inp o:out} and bound to the argument chan of newchan. The application of out to a variable z1 creates a new slot for z1 in the stream: For a fresh variable s2, the content of the cell co is assigned the new record {z1 s2} and replaced with s2. The application of inp first replaces the content s1′ of ci with a fresh variable s2′. It waits for s2′ to be bound to a record {z1′ s3} (*i.e.*, to point to a filled slot) and then "advances" ci by assigning s3 to s2′.

```
newchan:=
  (proc(chan)
   (local(s0 co ci out inp)
     co:=(cell s0);
     ci:=(cell s0);
     out:=(proc(z1)
            (local(s2)
             (exch co s2 (s1)
               s1:={z1 s2})));
     inp:=(proc(cont)
            (local (s2)
             (exch ci s2′ (s1′)
              (case s1′ ({z1′ s3}
               (thread s2′:=s3);
                (cont z1))))));
     chan:={o:out i:inp}))
```

Typing. Assuming recursive types, we define list types $M\mathsf{list}(T) = \mu\alpha.M\{T\ \ M\alpha\}$ for $M\in\{?,!,\hat{}\}$, and abbreviate the type $\mathsf{ch}(T)$ of channels for variables of type T as $\{\mathtt{i}{:}?(\mathsf{proc}\ ?(\mathsf{proc}\ T))\ \ \mathtt{o}{:}?(\mathsf{proc}\ T)\}$. Now, the procedure newchan can be shown to have type $(\mathsf{proc}\ !\exists\alpha{\le}\mathsf{top}.\mathsf{ch}(?\alpha))$. Our implementation is, however, more general than this: On output, we can put an unbound variable in the channel and assign to it on input, thus effectively reversing the data flow. This indicates that in Plain we might also want package types $\exists\beta{\le}T.P$ abstracting over *types* instead of pretypes. We then could show that newchan has type $(\mathsf{proc}\ !\exists\beta{\le}\mathsf{top}.\mathsf{ch}(\beta))$. To see this, assume the following types for the variables in our encoding.

$$\begin{array}{lll} \texttt{inp} : ?(\mathsf{proc}\ ?(\mathsf{proc}\ \beta)) & \texttt{out} : ?(\mathsf{proc}\ \beta) & \texttt{ci}, \texttt{co} : \hat{}(\mathsf{cell}\ \hat{}\mathsf{list}(\beta)) \\ \texttt{s0} : \hat{}\mathsf{list}(\beta) & \texttt{s2}', \texttt{s1} : !\mathsf{list}(\beta) & \texttt{s1}', \texttt{s2}, \texttt{s3} : ?\mathsf{list}(\beta) \\ \texttt{cont} : ?(\mathsf{proc}\ \beta) & \texttt{z1}, \texttt{z1}' : \beta & \end{array}$$

Finally note that Plain's subtyping rules induce Pict's channel subtyping: Channel types $\{\texttt{i}{:}?(\mathsf{proc}\ ?(\mathsf{proc}\ T))\ \texttt{o}{:}?(\mathsf{proc}\ T)\}$ can be restricted to input or output separately (due to record subtyping), and $T \leq T'$ implies $\mathsf{ich}(T) \leq \mathsf{ich}(T')$ and $\mathsf{och}(T') \leq \mathsf{och}(T)$ where $\mathsf{ich}(T) \stackrel{def}{=} \{\texttt{i}{:}?(\mathsf{proc}\ ?(\mathsf{proc}\ T))\}$ and $\mathsf{och}(T) \stackrel{def}{=} \{\texttt{o}{:}?(\mathsf{proc}\ T)\}$.

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*, pages 321–335, 1992.

[2] L. Cardelli. A Semantics of Multiple Inheritance. Number 173 in Lecture Notes in Computer Science, pages 51–67. Springer-Verlag, Berlin, Germany, 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 Notataion. In *Proc. IFIP Working Conference on Programming Concepts and Methods*, pages 466–491, Apr. 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] M. Henz, G. Smolka, and J. Würtz. Object-Oriented Concurrent Constraint Programming in Oz. In V. Saraswat and P. V. Hentenryck, editors, *Principles and Practice of Constraint Programming*, chapter 2, pages 27–48. The MIT Press, Cambridge, MA, 1995.

[9] K. Honda and N. Yoshida. On Reduction-Based Semantics. In R. K. Shyamasundar, editor, *13th Conference on Foundations of Software Technology and Theoretical Computer Science*, Bombay, India, Dec. 1993.

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

[11] A. Kleinman, Y. Moscowitz, A. Pnueli, and E. Shapiro. Communication with Directed Logic Variables. In *18th ACM Symposium on Principles of Programming Languages*, pages 221–232, 1991.

[12] M. J. Maher. Logic Semantics for a Class of Committed-Choice Programs. In J.-L. Lassez, editor, *International Conference on Logic Programming*, pages 858–876. The MIT Press, Cambridge, MA, 1987.

[13] R. Milner. A Theory of Type Polymorphism in Programming. *Journal of Computer and System Science*, 17:348–375, 1978.

[14] R. Milner. The Polyadic π-Calculus: A Tutorial. In F. L. Bauer, W. Brauer, and H. Schwichtenberg, editors, *Proceedings of the 1991 Marktoberndorf Summer School on Logic and Algebra of Specification*, NATO ASI Series. Springer-Verlag, Berlin, Germany, 1993.

[15] 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.

[16] J. Mitchell and G. Plotkin. Abstract Types have Existential Type. In *12th ACM Symposium on Principles of Programming Languages*, pages 37–51, Jan. 1985.

[17] M. Müller. Polymorphic Types for Concurrent Constraints. Available at `http://www.ps.uni-sb.de/` in `~mmueller/papers/ptcc.ps.gz`, 1996.

[18] J. Niehren. Functional Computation as Concurrent Computation. In *23rd ACM Symposium on Principles of Programming Languages*, pages 333–343, St. Petersburg Beach, Florida, Jan. 1996.

[19] J. Niehren and M. Müller. Constraints for Free in Concurrent Computation. In K. Kanchanasut and J.-J. Lévy, editors, *Asian Computing Science Conference*, number 1023 in Lecture Notes in Computer Science, pages 171–186, Pathumthani, Thailand, 11–13 Dec. 1995. Springer-Verlag, Berlin, Germany.

[20] B. Pierce and D. Sangiorgi. Typing and Subtyping for Mobile Processes. In *IEEE Symposium on Logic in Computer Science*, pages 376–385. IEEE Computer Society Press, June 1993.

[21] B. C. Pierce and D. N. Turner. Concurrent Objects in a Process Calculus. In *Theory and Practice of Parallel Programming (TPPP)*, Lecture Notes in Computer Science, Sendai, Japan, Nov. 1994. Springer-Verlag, Berlin, Germany.

[22] B. C. Pierce and D. N. Turner. Pict: A Programming Language Based on the Pi-Calculus, May 1996. Unpublished Draft.

[23] Programming Systems Lab, Universität des Saarlandes. The Oz Programming System, 1996. `http://www.ps.uni-sb.de/www/oz/`.

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

[25] E. Shapiro. The Family of Concurrent Logic Programming Languages. *ACM Computing Surveys*, 21(3):413–510, Sept. 1989.

[26] G. Smolka. A Foundation for Concurrent Constraint Programming. In J.-P. Jouannaud, editor, *Conference on Constraints in Computational Logics*, volume 845 of *Lecture Notes in Computer Science*, pages 50–72, München, Germany, 7–9 Sept. 1994.

[27] G. Smolka. The Oz Programming Model. In J. van Leeuwen, editor, *Computer Science Today*, Lecture Notes in Computer Science, vol. 1000, pages 324–343. Springer-Verlag, Berlin, Germany, 1995.

[28] B. Victor and J. Parrow. Constraints as Processes. In *7^{th} International Conference on Concurrency Theory*, Lecture Notes in Computer Science, 1996.