Polymorphic Types for Concurrent Constraints

Martin Müller*

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

Abstract. We investigate polymorphic typing for higher-order concurrent constraint programming. By instantiation of our cc framework ρ, we define the new cc calculus $\rho(x \Leftarrow y)$ with directed equational constraints. In combination with relational abstractions these allow to express functions, where data flow issues are factorised into the constraint system. Directed equations bring functional types into the reach of higher-order cc. We give a higher-order polymorphic type system for $\rho(x \Leftarrow y)$ and show that embeddings of λ into $\rho(x \Leftarrow y)$ preserve full System **F**-typability. This applies to an embedding of a λ calculus with call-by-need complexity.

Keywords. semantics of programming languages, concurrent constraint programming, functional programming, polymorphic types

1 Introduction

Concurrent constraint (cc) programming [21, 22, 15] has been shown to be a unifying programming model subsuming constraint [23], object-oriented [8, 24] and functional programming [24, 15]. Adequate embeddings of both call-by-value and call-by-need λ calculi into a cc calculus have been given and proved to preserve result, termination, and complexity [14].

Types for functional languages are well-understood, whereas cc languages are traditionally untyped. While monomorphic type systems are easily adapted, two complications arise in case of polymorphism:

1. cc-languages typically have a bidirectional data-flow and relational abstractions whose arguments may be input, output, or both.
2. cc languages separate declaration and assignment of logic variables. Similarly, reference cells in functional languages like ML separate declaration (creation) and assignment of locations.

Polymorphic type systems for functional languages crucially exploit the data flow inherent in λ calculi. For illustration, consider the typing rule (cont) from

* The research reported here has been supported by the Bundesminister für Bildung, Wissenschaft, Forschung und Technologie (FTZ-ITW-9105), the Esprit Project ACCLAIM (PE 7195), the Esprit Working Group CCL (EP 6028), and the DFG-Graduiertenkolleg Kognition at the Universität des Saarlandes.

Mitchell's containment-based version of System **F** [12, 3]:

$$(cont)\ \frac{\Gamma \triangleright M{:}S \quad T \supseteq S}{\Gamma \triangleright M{:}T} \qquad (app)\ \frac{\Gamma \triangleright M{:}\Pi\overline{\alpha}.S{\to}T \quad \Gamma \triangleright N{:}\Pi\overline{\alpha}.S}{\Gamma \triangleright MN{:}\Pi\overline{\alpha}.T}$$

By (cont), polymorphic types are specialised: e.g. $\Pi\gamma.(\gamma{\to}\gamma){\to}\beta \supseteq \Pi\alpha.\alpha{\to}\beta$. The syntax of λ guarantees that the type of an application MN is derived after the types of function M and argument N have been derived and appropriately specialised using (cont): Specialisation of polymorphic types strictly follows the *implicit* functional data flow.

$$\cfrac{\cfrac{\cfrac{\Gamma \triangleright M{:}R_1}{\vdots\ (cont)^*}}{\Gamma \triangleright M{:}\Pi\overline{\alpha}.S{\to}T} \quad \cfrac{\cfrac{\Gamma \triangleright N{:}R_2}{\vdots\ (cont)^*}}{\Gamma \triangleright N{:}\Pi\overline{\alpha}.S}}{\cfrac{\Gamma \triangleright MN{:}\Pi\overline{\alpha}.T}{\cfrac{\vdots\ (cont)^*}{\Gamma \triangleright MN{:}R_3}}}$$

In order to incorporate data flow information into cc, we present the calculus $\rho(x{\Leftarrow}y)$ as an instance of our higher-order cc scheme ρ [15]. The constraints of $\rho(x{\Leftarrow}y)$ are directed equations $x \Leftarrow y$, where $\Leftarrow$ denotes a reflexive and transitive relation. The ρ calculus strictly factorises higher-order abstraction and constraints. Since ρ is confluent for arbitrary constraint systems [15], $\rho(x{\Leftarrow}y)$ is confluent. Eager and lazy λ calculi have been adequatly embedded into ρ, and the latter has been shown to satisfy call-by-need complexity [1, 14]. These embeddings can be adapted to $\rho(x{\Leftarrow}y)$ [14]. Technically, the improved setup of ρ as compared to its predecessors [16, 24, 26] was crucial to obtain simple subject reduction proofs for $\rho(x{\Leftarrow}y)$.

The directed equation $x{\Leftarrow}y$ expresses a data flow from y to x; it allows to perform an application $x\,\overline{z}$ on x when y becomes bound to an abstraction, but not vice versa. Data flow is factorised into the constraint system and combines well with relational abstractions. In combination with an application, directed equations act as mode assertions and allow to recover functional data flow. For instance, the application xy within $\lambda x.\lambda y.xy$ becomes the relational $\rho(x{\Leftarrow}y)$ expression

$$\exists y' \exists z' (x\, y'\, z' \wedge z \Leftarrow z' \wedge y' \Leftarrow y)$$

Here, z explicitly refers to the overall result, $y' \Leftarrow y$ indicates y to be an input being specialised before application, and $z \Leftarrow z'$ analogously makes z an output.

The typing rules for $\rho(x{\Leftarrow}y)$ below, (SPEC) and (APP-1), are the relational counterparts to (spec) and (app). Rule (SPEC) allows to specialise types along the *explicit* data flow of $x \Leftarrow y$. Rule (APP-1) integrates (cont) to allow the type of an abstraction to be specialised before it is matched against its arguments.

$$\text{(SPEC)}\ \frac{\Gamma(x) \supseteq \Gamma(y)}{\Gamma \triangleright x \Leftarrow y} \qquad \text{(APP-1)}\ \frac{\Pi\overline{\alpha}.(S \times T) \supseteq \Gamma(x) \quad \Gamma(y,z) \equiv \Pi\overline{\alpha}.S, \Pi\overline{\alpha}.T}{\Gamma \triangleright x\, y\, z}$$

The system containing (SPEC) and (APP-1) is called **RF** (relational **F**). Unfortunately, (APP-1) is incorrect for $\rho(x{\Leftarrow}y)$ as is, due to the *separation of declaration and assignment* of logic variables: This property is shared with reference cells in ML (e.g., [9]) which also complicate polymorphic typing. However, **RF** is correct for a large class of $\rho(x{\Leftarrow}y)$-programs which contain the embedded λ expressions. In this sense, our embeddings of λ preserve full System **F**-typability.

By replacing (APP-1) with (APP-2) in $\mathbf{RF}$ we obtain system $\mathbf{R}$ which

$$\text{(APP-2)}\quad \frac{(S\times T)\supseteq\Gamma(x)\quad \Gamma, y{:}S, z{:}T \triangleright E}{\Gamma \triangleright x\ y\ z}$$

is correct for $\rho(x{\Leftarrow}y)$. By reference to a result by Wells we show that $\mathbf{R}$-typability is strictly weaker than $\mathbf{RF}$-typability.

Related Work. The main motivation for this work was to find out how much polymorphism can be provided for the higher-order cc-language Oz [27, 25]. Since ρ captures essential operational aspects of Oz, $\rho(x{\Leftarrow}y)$ indicates a possible extension of Oz if polymorphic typing is required.

The Oz context also lead us to $\mathbf{F}$-style polymorphism [3, 20] since higher-order polymorphic types seem to be required for the treatment of objects and modules. Nevertheless, many related results on polymorphic types have been obtained for ML-style polymorphism [11]:

1. The complications of polymorphic typing for reference cells or continuations in ML have attracted much attention [29, 9, 28, 4, 6, 5]. Leroy [9] observed that logic variables induce similar complications. The common feature of all these is the separation of declaration and assignment.
2. The restriction of $\mathbf{RF}$ to $\mathbf{R}$ corresponds to Wright's restriction of the standard Hindley/Milner polymorphism in presence of reference cells [32]: Wright observes that polymorphic generalisation is correct for let-bound abstractions even if their bodies contain reference cells.
3. Wright also notes that his restriction only rarely affects the typability of realistic programs [32]. In turn, this indicates that System $\mathbf{R}$ might be reasonable for $\rho(x{\Leftarrow}y)$ in practice.

The $\rho(x{\Leftarrow}y)$ calculus is closely and formally related to π [14], since the applicative core of π can express directed equations. Hence, our discussion could be reworked for π. We know of an embedding of λ into polyadic π by D. N. Turner in Curry style[2] and preserves $\mathbf{F}$-typing.[3] Given the relation of ρ and π, Turner's system should carry over to $\rho(x{\Leftarrow}y)$ as well. However, this embedding restricts sharing and does not capture call-by-need complexity for a lazy λ calculus. Leroy [10] discusses the relationship of polymorphism and sharing in an ML context.

Higher-order polymorphic types for non-functional calculi have been investigated for Pict, a programming language based on a polyadic π calculus [18]. We expect many results and experiences from Pict to be applicable to ρ and Oz (e.g., type declarations, pragmatics of type inference, or modes and subtyping).

The I-structures [2] of Id [17] can be seen as a restricted form of logic variables. Assignment to an I-structure resembles our directed equation. In Id, a run-time

[2] "Curry-style" calculi employ explicit type abstraction and type application as Girard's original presentation of System $\mathbf{F}$ [3]. We only discuss "Church-style" systems which do not treat types as syntactic parts of expressions but as their properties.

[3] D. N. Turner, personal communication (most probably, see [30]).

error is raised when an I-structure is assigned twice. Single directed assigment is sufficient to model functional computation in ρ. Multiple assignment of trivial values has received attention for the encoding of demand propagation [19, 13, 14].

Overview. Section 2 presents the calculus $\rho(x{\Leftarrow}y)$. Section 3 illustrates polymorphic typing for logic variables by example. Section 4 recovers full **F**-typing for the functional part of $\rho(x{\Leftarrow}y)$ while preserving call-by-need complexity for an embedding of the lazy λ calculus. Section 5 gives an **F**-style system for full bidirectional $\rho(x{\Leftarrow}y)$.

2 A Rho-Calculus with Directed Equations.

We assume an infinite set of variables ranged over by x, y, z, u, v, and w. Sequences of variables are written $\overline{x}$. The variables in a sequence $\overline{x}$ and its *length* are denoted by $\{\overline{x}\}$ and $\mathsf{len}(\overline{x})$, respectively. Sequences of other syntactic entities are treated alike. The expression syntax for $\rho(x{\Leftarrow}y)$ is defined by the grammar:[4]

$$\begin{array}{lcl} C, D, E, F & ::= & \phi \mid E \wedge F \mid \exists x E \mid x{:}\overline{y}/E \mid x\,\overline{y} \mid \mathsf{if}\ \phi\ \mathsf{then}\ E\ \mathsf{else}\ F\ \mathsf{fi} \mid \bullet \\ \phi, \psi & ::= & x \Leftarrow y \mid \phi \mathbin{\dot\wedge} \psi \mid \dot\top \end{array}$$

Contexts C, D and *expressions* E, F are likewise generated by this grammar, where contexts must contain exactly one hole $\bullet$ and expressions no hole at all. Expressions consist of *abstractions* $x{:}\overline{y}/E$ with *name* x and *formal arguments* $\overline{y}$, *applications* $x\,\overline{y}$ of an abstraction with name x and *actual arguments* $\overline{y}$, *constraints* ϕ and *conditionals* $\mathsf{if}\ \phi\ \mathsf{then}\ E\ \mathsf{else}\ F\ \mathsf{fi}$. As glue for expression there is *declaration* $\exists$ and *composition* $\wedge$. As primitive constraints we consider the trivial constraint $\dot\top$ and directed equations $x \Leftarrow y$ between variables. Constraints contain primitive constraints and are closed under conjunction $\dot\wedge$.[5]

The congruence $\equiv$ is the smallest relation on expressions which provides consistent renaming of bound variables (α), makes composition ($\wedge$) associative and commutative, satisfies the usual scope rules for declarations

$$\exists x \exists y E \;\equiv\; \exists y \exists x E \qquad\qquad \exists x E \wedge F \;\equiv\; \exists x (E \wedge F) \quad \text{if } x \notin \mathcal{FV}(F)$$

and allows equivalence transformations on constraints wrt. a theory Δ which fixes the semantics of constraints:

$$\phi \;\equiv\; \psi \qquad \textit{if } \phi \models\!\!\mid_{\Delta} \psi \textit{ and } \mathcal{FV}(\phi) = \mathcal{FV}(\psi)$$

The directed equation in $\rho(x{\Leftarrow}y)$ is axiomatised to be a reflexive and transitive, but not symmetric relation. That is, $\rho(x{\Leftarrow}y)$ comes with the following theory Δ_0:

$$\Delta_0 \;=\; \{\forall x (x \Leftarrow x), \quad \forall x \forall y \forall z (x \Leftarrow y \mathbin{\dot\wedge} y \Leftarrow z \rightarrow x \Leftarrow z)\}$$

[4] State is introduced into ρ by means of cells. We dropped cells from this presentation, because they do not add additional complexity to the typing issues.

[5] Note the distinction between composition of expressions and logical conjunction!

(A)	$\phi \wedge x{:}\overline{y}/E \wedge x'\,\overline{z} \;\rightarrow\; \phi \wedge x{:}\overline{y}/E \wedge E[\overline{z}/\overline{y}]$		$\phi \models_\Delta x' \Leftarrow x$
(C)	$\phi_1 \wedge \phi_2 \;\rightarrow\; \psi$		if $\phi_1 \dot\wedge \phi_2 \models\!\mid_\Delta \psi$
(Then)	$\phi \wedge$ if ψ then E else F fi $\;\rightarrow\; \phi \wedge E$		if $\phi \models_\Delta \psi$
(Else)	$\phi \wedge$ if ψ then E else F fi $\;\rightarrow\; \phi \wedge F$		if $\phi \models_\Delta \neg\psi$

Fig. 1. Axioms of Reduction in ρ

Reduction $\rightarrow$ is the smallest relation on expressions which contains all instances of the schemes in Figure 1 and is closed under the following rules:

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

By rule (C), constraints can be combined by transforming a composition into a conjunction. Rules (Then) and (Else) are standard for cc. A conditional may reduce in the context of a constraint ϕ if ϕ is strong enough to entail or disentail ψ (only proper extensions of $\rho(x{\Leftarrow}y)$ can make use of the else branch, since all constraints ϕ are satisfiable wrt. Δ_0). Rule (A) applies to $x{:}\overline{y}/E \wedge x'\,\overline{z}$ in a context ϕ entailing $x' \Leftarrow x$, which states that x is being assigned to x'.[6]

Expressions of the form $\phi \wedge x{:}\overline{u}/E \wedge y{:}\overline{v}/F$ are called *inconsistent* if $\phi \models_\Delta z \Leftarrow x \dot\wedge z \Leftarrow y$ for some z. Expressions which reduce to an inconsistency are called *non-admissible* and must be excluded to achieve confluence [13, 15]. Admissibility will not concern us here, but note that (i) there are static type systems to check that functional $\rho(x{\Leftarrow}y)$ programs as obtained by embedding λ are admissible, and (ii) a dynamic admissibility check can be realised using names [24, 15].

The Types. We deal with higher-order polymorphic types generated by:

$$S, T \;::=\; (\overline{S}) \mid \Pi\overline{\alpha}.S \mid \alpha$$

$(\overline{S})$ denotes the type of a relational abstraction allowing formal arguments of types $\overline{S}$. For convenience, we use functional notation as equivalent alternative to the relational one: $S \rightarrow T$ instead of $(S\ T)$. The variables $\overline{\alpha}$ are bound in $\Pi\overline{\alpha}.S$. The *free* variables of a type S is defined as usual and written $\mathcal{FV}(S)$. All *variables* in S are denoted by $\mathcal{V}(S)$. Polymorphic types are identified up to the smallest congruence satisfying consistent renaming of bound variables (α), $\Pi\alpha.\Pi\beta.S \equiv \Pi\beta.\Pi\alpha.S$, and $\Pi\alpha.S \equiv S$, provided $\alpha \notin \mathcal{FV}(S)$. The central relation on polymorphic types is the *containment* relation [12]:

$$\Pi\overline{\beta}.\overline{S}[\overline{T}/\overline{\alpha}] \supseteq \Pi\overline{\alpha}.\overline{S} \qquad \text{if } \{\overline{\beta}\} \cap \mathcal{FV}(\Pi\overline{\alpha}.\overline{S}) = \emptyset \text{ and } \overline{T} \text{ free for } \alpha \text{ in } \overline{S}$$

If $S \supseteq T$, we say T is a *generalisation* of S and S is a *specialisation* of T. Containment is a partial order on types modulo congruence. It is reflexive, transitive, and satisfies $\supseteq \cap \subseteq \,=\, \equiv$. An important property of containment is that

$$S \supseteq T \qquad \text{implies} \qquad \Pi\overline{\alpha}.S \supseteq \Pi\overline{\alpha}.T \tag{1}$$

[6] The minor change of using the side condition $\models_\Delta x' \Leftarrow x$ for rule (A) instead of $\models_\Delta x' {=} x$ as in the ρ calculus [15] does not affect any properties of ρ.

An *environment* Γ is a finite set $\overline{x}{:}\overline{T}$ of *type assumptions* (variable type pairs) where the $\overline{x}$ are pairwise disjoint. Equivalently, $\Gamma = \overline{x}{:}\overline{S}$ is treated as a partial function from variables to types with *domain* $\mathsf{dom}(\Gamma) = \{\overline{x}\}$ and *range* $\mathsf{ran}(\Gamma) = \{\overline{S}\}$. *Extension* of an environment Γ by a type assumption $x{:}S$ is written $\Gamma, x{:}S$ and defined only if $x \notin \mathsf{dom}(\Gamma)$. If $\Gamma = \overline{x}{:}\overline{S}$, then $\Pi\alpha.\Gamma$ is defined as $\overline{x}{:}\overline{\Pi\alpha.S}$.

3 Polymorphic Types and the Logic Variable

We consider λ-expressions with the usual syntax, where the let-expression is only used for the examples in this Section:

$$M, N \ ::= \ x \mid \lambda x.M \mid MN \mid \mathsf{let}\ \ x{=}M\ \ \mathsf{in}\ N$$

Example 1. Let $M;N$ abbreviate $\lambda\_.N\ M$ and consider the ML program:

$$\mathsf{let\ id = \lambda x.x\ in\ let\ b = \lambda y.(y\ true)\ in\ let\ i = \lambda y.(y\ 1)\ in\ (b\,id);(i\,d)} \qquad (2)$$

In relational calculi with symmetric equations only [13, 24, 26], Program (2) corresponds to the following:

$$\exists \mathsf{id}\,(\mathsf{id{:}x\,y/y{=}x} \wedge \exists \mathsf{b}\,(\mathsf{b{:}y\,z/y\ true\,z} \wedge \exists \mathsf{i}\,(\mathsf{i{:}y\,z/y\,1\,z} \wedge \exists \mathsf{i'}\,(\mathsf{i\ id\ i'} \wedge \mathsf{b\ id\ b'})))) \quad (3)$$

Program (2) is well-typed with $\mathsf{id}{:}\alpha{\to}\alpha$, $\mathsf{b}{:}(\mathsf{bool}{\to}\beta){\to}\beta$ and $\mathsf{i}{:}(\mathsf{int}{\to}\gamma){\to}\gamma$. A type checker for Program (3) cannot rely on a fixed data flow and must use the typing rules (SPEC) and (APP-ML) (properly restricted to ML-type schemes):

$$\text{(SPEC)}\ \frac{\Gamma(x){\approx}\Gamma(y)}{\Gamma \rhd x{=}y} \qquad\qquad \text{(APP-ML)}\ \frac{\Gamma(x) \equiv \Pi\overline{\alpha}.\overline{T} \quad \Gamma(\overline{y}) \equiv \overline{T}[\overline{S}/\overline{\alpha}]}{\Gamma \rhd x\,\overline{y}}$$

Under the assumption that id have type $\Pi\alpha.(\alpha\ \alpha)$, Program (3) is ill-typed. A number of additional "mode assertions" via directed equations make the program well-typed (some declarations have been dropped for brevity):

$$\begin{array}{l}\mathsf{id{:}x\,y/y \Leftarrow x \wedge b{:}y\,z/y\,true\,z \wedge i{:}y\,z/y\,1\,z} \wedge \\ \qquad\qquad \exists \mathsf{id'id''}\,(\mathsf{id' \Leftarrow id \wedge i\ id'\ i' \wedge id'' \Leftarrow id \wedge b\ id''\ b'})\end{array}$$

Example 2. This example has been adapted from Leroy's $\mathsf{functional\_ref}$ program [9], which he used to exemplify the difficulty of polymorphic typing for ML with reference cells:

$$\begin{array}{rl} & \mathsf{x{:}z/\exists r(z{:}v/r \Leftarrow v) \wedge x\,y\ \wedge y\,1\ \wedge y\,true} \\ \to & \mathsf{x{:}z/\exists r(z{:}v/r \Leftarrow v) \wedge \exists r(y{:}v/r \Leftarrow v \wedge r \Leftarrow q \wedge r \Leftarrow true)}\end{array}$$

Reduction of the application $x\,y$ binds y to a copy of $z{:}v/r \Leftarrow v$ with some variable r global to its body. While x deserves the type $\Pi\alpha.((\alpha))$, y may not get the polymorphic type $\Pi\alpha.(\alpha)$ since α refers to the *same* copy of r within each copy of z. That is, although y is known to be the output of $x\,y$, its type may not freely be generalised (compare $\mathsf{let}\ \ x{=}(\mathsf{let}\ \ r{=}\lambda y.y\ \mathsf{in}\ N)\ \mathsf{in}\ M\ \mathsf{in}\ \mathsf{ML}$).

Leroy [9] observes that logic variables and channels induce similar complications: Note that neither destructive assignment nor indeterminism are responsible for this, but the separation of declaration and assignment.

4 Recovering System F: Typing Functional Rho

Typing is defined in terms of *judgements* $\Gamma \triangleright M{:}T$ for λ stating that M is well-typed with type T under type assumptions Γ for $\mathcal{FV}(M)$. To state that $\mathcal{S}$ is derivable in a particular system T, we use *qualified judgements* $T \vdash \mathcal{S}$. An expression M is said to be T-*typable* if system T allows a derivation of $\Gamma \triangleright M{:}T$ for some Γ and T (the definitions adapt easily to $\rho(x{\Leftarrow}y)$ and judgements $\Gamma \triangleright E$).

$$(SPEC)\quad \frac{\Gamma(x) \supseteq \Gamma(y)}{\Gamma \triangleright x \Leftarrow y} \qquad \frac{\Gamma \triangleright E \quad \Gamma \triangleright F}{\Gamma \triangleright E \wedge F} \qquad \frac{\Gamma, x{:}S \triangleright E}{\Gamma \triangleright \exists x\, E}$$

$$\frac{\Gamma(x) \supseteq \Pi\overline{\alpha}.(\overline{S}) \quad \Gamma, \overline{y}{:}\overline{S} \triangleright E}{\Gamma \triangleright x{:}\overline{y}/E}\ \mathcal{FV}(\overline{S})\backslash\mathcal{FV}(\Gamma) = \{\overline{\alpha}\}$$

$$(\text{APP-1})\quad \frac{\Pi\overline{\alpha}.(\overline{S}) \supseteq \Gamma(x) \qquad \ldots \Gamma(y_i) \equiv \Pi\overline{\alpha}.S_i \ldots}{\Gamma \triangleright x\,\overline{y}} \qquad \frac{\Gamma \triangleright \phi \wedge E \wedge F}{\Gamma \triangleright \mathsf{if}\ \phi\ \mathsf{then}\ E\ \mathsf{else}\ F\ \mathsf{fi}}$$

Fig. 2. System **RF**: Recovering System **F** for Rho

Figure 2 presents the higher-order polymorphic type system **RF** for $\rho(x{\Leftarrow}y)$. In System **RF**, there are two variable positions which allow polymorphic specialisation and generalisation: The name position in an application $x\,\overline{y}$ and the left hand side of a directed equation (SPEC).

We show that subject reduction holds for **RF** and a large class of $\rho(x{\Leftarrow}y)$ expression which contains embedded λ expressions. **RF** is also shown to preserve full System **F**-typing for both the call-by-value and call-by-need embedding of λ.

System **RF** has the following usual properties:

Lemma 1. Congruence *If* $\mathbf{R} \vdash \Gamma \triangleright E$ *and* $E \equiv F$, *then* $\mathbf{R} \vdash \Gamma \triangleright F$.

Weakening *If* $x \notin \mathcal{FV}(E)$ *then* $\mathbf{R} \vdash \Gamma \triangleright E$, *if and only if* $\mathbf{R} \vdash \Gamma, x{:}S \triangleright E$.

Substitution-I *If* $\mathbf{R} \vdash \Gamma \triangleright E$, *then* $\mathbf{R} \vdash \Gamma[S/\alpha] \triangleright E$.

Substitution-II *If* $\Gamma(x) = \Gamma(y)$, *then* $\mathbf{RF} \vdash \Gamma \triangleright E$ *iff* $\mathbf{RF} \vdash \Gamma \triangleright E[x/y]$.

An important Lemma specific for **RF** is the following:

Lemma 2 Combination. $\mathbf{RF} \vdash \Gamma \triangleright \phi$ *and* $\phi \models\!\!\!\dashv_{\Delta} \psi$ *implies* $\mathbf{RF} \vdash \Gamma \triangleright \psi$.

Proof. Both $=$ and $\equiv$ on types are symmetric, transitive, and reflexive, and both $\Leftarrow$ and $\supseteq$ are transitive, and reflexive.

Figure 3 defines a partial function $\mathcal{O}$ on expressions, which computes the possible outputs of an expression. The domain of $\mathcal{O}$ defines the set of *input-admissible* expressions. In particular, an abstraction $x{:}\overline{y}/E$ is input-admissible, if all variables free in its body E ($\mathcal{FV}(E)\backslash\{\overline{y}\}$) are at most used for input ($\{\overline{y}\} \subseteq \mathcal{O}(E)$). By construction, the following lemma holds:

$\mathcal{O}(x \Leftarrow y) \;=\; \{x\}$	$\mathcal{O}(x\,\overline{y}\,) \;=\; \{\overline{y}\}$	$\mathcal{O}(x{:}\overline{y}/E) \;=\; \{x\}$ if $\{\overline{y}\} \subseteq \mathcal{O}(E)$
$\mathcal{O}(E_1 \wedge E_2) \;=\; \mathcal{O}(E_1) \cup \mathcal{O}(E_2)$	$\mathcal{O}(\exists x E) \;=\; \mathcal{O}(E)\backslash\{x\}$	

Fig. 3. Input-admissibility

Lemma 3. *If E is input-admissibile, then every subexpression of E and $\exists x\, E$ is.*

Input-admissibility is invariant under congruence and reduction in the sense:

Lemma 4. *If $E \equiv F$ then $\mathcal{O}(E) = \mathcal{O}(F)$, and if $E \to F$ then $\mathcal{O}(F) \subseteq \mathcal{O}(E)$*

Lemmata 5 and 6 state the key properties of input-admissibile $\rho(x{\Leftarrow}y)$ expressions which allow to prove subject reduction.

Lemma 5. *Assume $\Gamma, y{:}S \triangleright E$ and $y \in \mathcal{O}(E)$, then $\Gamma, y : \Pi\alpha.S \triangleright E$.*

Proof. The proof is by induction over the structure of E. If $E = x \Leftarrow y$, the claim follows from transitivity if $\supseteq$, if $E = x \Leftarrow z$ and $z{\neq}y$, (Weakening) applies. A similar argument holds for $y\,\overline{z}\,$. If $\Gamma, y{:}S \triangleright x{:}\overline{z}/E$, wlog. $y \notin \{\overline{z}\}$ and $y \notin \mathcal{O}(E)$. The claim follows from the fact that $\mathcal{FV}(\Gamma, y{:}\Pi\alpha.S) \subseteq \mathcal{FV}(\Gamma, y{:}S)$.

Lemma 6 Key Lemma. *If E is input-admissible, then $\mathbf{R} \vdash \Gamma \triangleright E$ implies $\mathbf{R} \vdash \Pi\alpha.\Gamma \triangleright E$.*

Proof. By induction over the structure of E. Cases $=$ and $\Leftarrow$ follow from Property (1). Case $\wedge$ and $\exists$ are immediate from the induction assumption. We show the abstraction case:

Abstraction. Assume $E \equiv x{:}\overline{y}/F$, $\mathbf{R} \vdash \Gamma \triangleright E$, and $\Gamma = \Gamma', x{:}T$. Hence there are $\overline{S}$ such that $\Gamma(x) \supseteq \Pi\overline{\alpha}.(\overline{S})$, $\Gamma, \overline{y}{:}\overline{S} \triangleright F$ and $\{\overline{\alpha}\} = \mathcal{FV}(\overline{S})\backslash\mathcal{FV}(\Gamma)$. Lemma 5 plus input-admissibility of E implies that $\Pi\beta.\Gamma', x{:}T, \overline{y}{:}\overline{S} \triangleright F$, thus $\Pi\beta.\Gamma', x{:}\Pi\beta.T \triangleright E$, either with property (1) or the fact that $\Pi\beta.S \supseteq S$ if $\beta \notin \mathcal{FV}(S)$.

Theorem 7 Subject Reduction. *If E is input-admissible and $E \to F$, then $\mathbf{RF} \vdash \Gamma \triangleright E$ implies $\mathbf{RF} \vdash \Gamma \triangleright F$.*

Proof. The proof is by induction over the context of a reduction. The most interesting cases are axioms (C) and (A).

Case (C) follows from Lemma 2: Importantly, the combination rule (C) [15] allows the strong syntactic invariant to maintain the data flow captured by $\Leftarrow$.

For case (A) assume $\Gamma \triangleright x\,\overline{y}\; \wedge\; x{:}\overline{z}/E$, which implies that $\Gamma, \overline{z}{:}\overline{S} \triangleright E$, $\Gamma(\overline{y}) = \overline{\Pi\overline{\alpha}.T}$, and $\Pi\overline{\alpha}.(\overline{S}) \supseteq \Gamma(x)$ where $\mathcal{FV}(\overline{S})\backslash\mathcal{FV}(\Gamma) = \{\overline{\alpha}\}$. Lemma 6 implies $\Pi\overline{\alpha}.\Gamma, \overline{z}{:}\overline{\Pi\overline{\alpha}.T} \triangleright E$ where $\Pi\overline{\alpha}.\Gamma \equiv \Gamma$ since $\{\overline{\alpha}\} \cap \mathcal{FV}(\Gamma) = \emptyset$. Hence $\Gamma \triangleright E[\overline{y}/\overline{z}]$ by Lemma 1 (Substitution-II).

Along with the observation that no well-typed expression contains an immediate type error, Theorem 7 yields correctness of System **RF** for input-admissible expressions in the sense that "well-typed programs do not go wrong" [11]:

Lemma 8 Safety. *Assume an expression* $E \equiv \exists\overline{x}(x{:}\overline{y}/E \wedge x\,\overline{z} \wedge F)$ *for some* $\overline{x}$ *and* F. *Then* $\mathsf{len}(\overline{y}) \neq \mathsf{len}(\overline{z})$ *implies that* E *is not* **R**-*typable.*

$$(var)\ \frac{\Gamma(x) = S}{\Gamma \rhd x{:}S} \qquad (lam)\ \frac{\Gamma, x{:}S \rhd M{:}T}{\Gamma \rhd \lambda x.M{:}\Pi\overline{\alpha}.S{\to}T}\ \{\overline{\alpha}\}{\cap}\mathcal{FV}(\Gamma){=}\emptyset$$

$$(cont)\ \frac{\Gamma \rhd M{:}S \quad T \supseteq S}{\Gamma \rhd M{:}T} \qquad (app)\ \frac{\Gamma \rhd M{:}\Pi\overline{\alpha}.S{\to}T \quad \Gamma \rhd N{:}\Pi\overline{\alpha}.S}{\Gamma \rhd MN{:}\Pi\overline{\alpha}.T}$$

Fig. 4. Containment Based Version of System **F**

Figure 4 shows Mitchell's version of System **F** [12]. Since Mitchell has proven equivalence to Girard's System **F** [12], we use **F** interchangeably for both systems. The application rule of **RF** (APP-1) is the straightforward adaptation of Mitchell's application rule (app) into the relational setting. In the functional context, (app) can be thought of as synthesising (*i*) a sequence of specialisation steps applied to function and argument in MN to get rid of the outermost quantifiers, (*ii*) the type-match between actual and formal arguments, and (*iii*) a sequence of generalisation steps applied to "the result" MN. In the relational setting of **RF**, this intuition no longer applies: All arguments are treated alike!

$$\begin{aligned} x =_{\mathsf{cbv}} MN &\equiv \exists y \exists z \exists x' (y =_{\mathsf{cbv}} M \wedge y\,z\,x' \wedge z =_{\mathsf{cbv}} N \wedge x \Leftarrow x') \\ x =_{\mathsf{cbv}} \lambda y.M &\equiv x{:}y\,z/z =_{\mathsf{cbv}} M \\ x =_{\mathsf{cbv}} y &\equiv x \Leftarrow y \end{aligned}$$

Fig. 5. Call-by-value λ in $\rho(x{\Leftarrow}y)$

Figure 5 recalls the embedding of the call-by-value λ calculus investigated in [14], with the slight modification of the auxiliary variable x' in the application case. This is necessary to account for **F**-derivations which apply the containment rule (cont) immediately after rule (app).[7] The corresponding call-by-need embedding is given in Figure 6, and Proposition states 14 that both are equivalent in terms of typing.

$$\frac{\dfrac{\cdots}{\Gamma \rhd MN{:}T}}{\Gamma \rhd MN{:}S}\ S \supseteq T$$

[7] In [13, 14], this auxiliary variable is avoided by explicit moding of functional abstractions which is not intended here. The adequacy proofs given in in [13] are not affected by this modification.

We can easily prove that embedding always yields input-admissible expressions. Hence, the Subject Reduction Theorem 7 holds for embedded λ expressions.

Lemma 9. *Expressions $x =_{\mathsf{cbv}} M$ are input-admissible.*

For the proof that the call-by-value embedding preserves **F**-types, we use Well's notion of a global type environment [31]: A *derivation* $\mathcal{D}$ in a type system T is a finite sequence of judgements $(\mathcal{S})_{i=1}^{n}$ such that for all j there is a rule in T with premises $\mathcal{S}_{k_1} \ldots \mathcal{S}_{k_m}$ and conclusion $\mathcal{S}_j$ where $1{\leq}k_1, \ldots, k_m{<}j$. A derivation $\mathcal{D} = (\Gamma_i \triangleright M_i{:}T_i)_{i=1}^{n}$ such that M_n is α-standardised induces a *global type environment* $\mathcal{G}(\mathcal{D})$ by $\mathcal{G}(\mathcal{D}) = \bigcup_{i=1}^{m} \Gamma_i$ (again, adaptation to $\rho(x{\Leftarrow}y)$ is straightforward).

Lemma 10. *For every expression M such that $z{=_{\mathsf{cbv}}}M$ is α-standardised, there is a unique mapping Λ_z^M from $\mathcal{V}(z =_{\mathsf{cbv}} M)$ to the subexpressions of M. If M and z are is understood, we drop the indices.*

Proposition 11. *Let an α-standardised M and $z \notin \mathcal{V}(M)$ be given, and assume a derivation $\mathcal{D}_1 = (\Delta_i{\triangleright}E_i)_{i=1}^{m}$ such that $E_m = z =_{\mathsf{cbv}} M$. Then there exists a derivation $\mathcal{D}_2 = (\Gamma_i{\triangleright}M_i{:}S_i)_{i=1}^{n}$ such that for all $x \in \mathcal{V}(E_m)$:*

1. *If $x \in \mathcal{V}(M)$, then $\mathcal{G}(\mathcal{D}_2)(x) = \mathcal{G}(\mathcal{D}_1)(x)$*
2. *If $x \notin \mathcal{V}(M)$, then there is some j, $1{\leq}j{\leq}n$ such that $\Gamma_j \triangleright \Lambda(x){:}\mathcal{G}(\mathcal{D}_2)(x)$.*

Proof. By a tedious but straightforward induction over the structure of the derivation $\mathcal{D}_1$.

Theorem 12. *Let M be given and assume $x \notin \mathcal{V}(M)$. Then* $\mathbf{F} \vdash \Gamma \triangleright M{:}S$ *if and only if* $\mathbf{RF} \vdash \Gamma, x{:}S \triangleright x =_{\mathsf{cbv}} M$.

Proof. The proof from right to left follows from Proposition 11. For the other direction it is proved by induction over **F**-derivations of $\Gamma \triangleright M{:}S$ that every sequence of applications of (cont) to a subterm of M corresponds to a single application of rule (SPEC) in the **RF**-derivation of $\Gamma, x{:}S \triangleright x =_{\mathsf{cbv}} M$.

Theorem 12 implies that typability in **F** can be reduced to typability in **RF**. With the undecidability result for System **F** of Wells [31] this yields the corollary:

Corollary 13. *The typability for System* **RF** *is undecidable.*

Call-by-need. Figure 6 presents our call-by-need embedding of λ [24]. Using logic variables to express sharing, this embedding preserves call-by-need complexity of the lazy λ calculus [14]. The embedding relies on the propagation of demand (see also [19]). Every variable z is associated with a trigger s, which is bound to a global variable z_0 when the evaluation of x is required (case 3). The

$$
\begin{aligned}
u =_{\mathsf{cbn}} MN &\equiv \exists yzsu'(y =_{\mathsf{cbn}} M \wedge y\ z\ s\ u' \wedge \mathsf{if}\ s \Leftarrow z_0\ \mathsf{then}\ z =_{\mathsf{cbn}} N\ \mathsf{fi} \wedge u \Leftarrow u')\\
u =_{\mathsf{cbn}} \lambda y.M &\equiv x{:}y\ s\ z/z =_{\mathsf{cbn}} M[y{\cdot}s/y]\\
x =_{\mathsf{cbn}} y{\cdot}s &\equiv x \Leftarrow y \wedge s \Leftarrow z_0\\
x =_{\mathsf{cbn}} y &\equiv x \Leftarrow y
\end{aligned}
$$

Fig. 6. Call-by-need λ in $\rho(x{\Leftarrow}y)$

first such binding to s triggers a conditional like $\mathsf{if}\ s \Leftarrow z_0\ \mathsf{then}\ z =_{\mathsf{cbn}} N\ \mathsf{fi}$ (case 1) and enables evaluation of the corresponding expression $z =_{\mathsf{cbn}} N$.

The embeddings of Figures 5 and 6 are equivalent wrt. type checking modulo an operation $\mathsf{N}(\cdot)$ on types, which satisfies $\mathsf{N}(\alpha) = \alpha$ and $\mathsf{N}(\Pi\overline{\alpha}.(S\ T)) = \Pi\overline{\alpha}.(\mathsf{N}(S)\ ()\ \mathsf{N}(T))$ (as well as $\mathsf{N}(\Pi\overline{\alpha}.(\overline{S})) = \Pi\overline{\alpha}.(\overline{\mathsf{N}(S)})$ if $\mathsf{len}(\overline{S}) \neq 2$).

Proposition 14. *Assume M and $z, z_0 \notin \mathcal{V}(M)$. Then $\mathbf{R} \vdash \Gamma \triangleright z =_{\mathsf{cbv}} M$ if and only if $\mathbf{R} \vdash \mathsf{N}(\Gamma), z_0{:}() \triangleright z =_{\mathsf{cbn}} M$.*

Proof. By structural induction over M, exploiting the structural similarity between both embeddings.

From Propositions 12 and 14 we obtain that **F**-typing is preserved for the call-by-need embedding:

Corollary 15. *Let M be given and assume $x \notin \mathcal{V}(M)$. Then $\mathbf{F} \vdash \Gamma \triangleright M{:}S$ if and only if $\mathbf{RF} \vdash \mathsf{N}(\Gamma), x{:}\mathsf{N}(S) \triangleright x =_{\mathsf{cbn}} M$.*

5 Polymorphic Typing for Bidirectional $\rho(x{\Leftarrow}y)$

System **RF** is not correct for full $\rho(x{\Leftarrow}y)$, due to the application rule. The responsible phenomenon has been discussed in Example 2 in Section 3. Instead, we discuss the slightly weaker System **R** which is defined just as System **RF** with the application rule (APP-1) replaced by the weaker (APP-2) in Figure 7.

$$
(APP\text{-}2)\quad \frac{(\overline{S}) \supseteq \Gamma(x) \quad \Gamma, \overline{y}{:}\overline{S} \triangleright E}{\Gamma \triangleright x\ \overline{y}}
$$

Fig. 7. Application Rule for Bidirectional $\rho(x{\Leftarrow}y)$

Lemmata such as 1 and 2 are easily proved for **R**. Due to the weaker application rule, Lemma 6 does *not* hold for **R**. Yet, subject reduction is simpler to prove:

Theorem 16. *$\mathbf{R} \vdash \Gamma \triangleright E$ and $E \to F$ imply $\mathbf{R} \vdash \Gamma \triangleright F$.*

Equally straightforward is the proof of a Lemma corresponding to Lemma 8, showing that **R**-typed expressions "do not go wrong". Proposition 12 implies that **R** is correct wrt. System **F**:

Corollary 17. *For M and $x \notin \mathcal{V}(M)$: If $\mathbf{R} \vdash \Gamma, x{:}S \triangleright x =_{\mathsf{cbv}} M$ then $\mathbf{F} \vdash \Gamma \triangleright M{:}S$.*

Intuitively, by the difference in the application rules, System **R** must be strictly weaker than System **F**. The following theorem formalises this observation:

Theorem 18. *There is an expression M such that M is typable in **F** but for no **F**-type S and $z \notin \mathcal{FV}(M)$, the type S is a **R**-type of $z =_{\mathsf{cbv}} M$.*

Proof. We prove first, that type *checking* in **R** is strictly weaker than in **RF** by considering: $M = \lambda y.\lambda x.y\,x$ and $R = (\Pi\overline{\alpha}.S \to T) \to (\Pi\overline{\alpha}.S \to \Pi\overline{\alpha}.T)$. Embedding yields:

$$\begin{array}{lcl} z =_{\mathsf{cbv}} M & \equiv & z{:}y\,y'/y'{:}x\,x'/\exists u \exists v \exists w (u \Leftarrow y \wedge u\,v\,w \wedge v \Leftarrow x \wedge x' \Leftarrow w) \\ R_{wit} & = & (\Pi\overline{\alpha}.(S\ T)\ (\Pi\overline{\alpha}.S\ \Pi\overline{\alpha}.T)) \end{array}$$

and we observe that $\mathbf{F} \vdash M{:}R_{wit}$ while $\mathbf{R} \not\vdash z =_{\mathsf{cbv}} M{:}R_{wit}$. The remainder of the proof follows from Proposition 11 and Wells' Theorem 4.12 [31] which implies Corollary 19. Wells shows the following Theorem, in order to show typability and type-checking for System **F** equivalent and undecidable [31]:

Wells' Theorem [31, 4.12] For every polymorphic type S there is a (λ) context $C^S[\bullet]$ such that for every expression M "the type of $\bullet$ within $C[\bullet]$ is forced into S": Assume S, let $C^S[\bullet]$ be such a context, and let $\mathcal{D} = (\Gamma_i \triangleright M_i{:}S_i)_{i=1}^m$ be any **F**-derivation which proves typability of $C^S[z]$ for some fresh variable z. Then the *global type environment* induced by $\mathcal{D}$ must map z into S, i.e. $\mathcal{G}(\mathcal{D})(z) = S$.

By an argument similar to the proof of Theorem 5.1 in [31], we obtain:

Corollary 19. *There is a λ-context $C[\bullet]$ such that $C[z\ \lambda y.\lambda x.y\,x]$ is **F**-typable if and only if the type of z will be forced into $R_{wit} \to T$ for some (irrelevant) T.*

Let $M = z\ \lambda y.\lambda x.y\,x$ and $C[\bullet]$ be such a context. Then there is a $\rho(x \Leftarrow y)$-context $D[\bullet]$ such that $u{=}C[M] \equiv D[v =_{\mathsf{cbv}} M]$ and $\Lambda_u^{C[M]}(v) = M$. By Proposition 11, every **RF**-derivation which proves typability of $D[v =_{\mathsf{cbv}} M]$ must force v into $(R\ T)$ for some T. But no such derivation can be a **R**-derivation, hence **R** is strictly weaker than **RF** and **F**.

Acknowledgements. I would like to thank Gert Smolka for discussions on simulating directed equations in γ and for encouraging my research into types. Joachim Niehren contributed helpful comments throughout this work. I also thank David N. Turner for discussions on polymorphism issues in general and on a preliminary version of this paper, and Joe Wells for commenting on his undecidability proof of **F**-typing.

References

1. Z. M. Ariola, M. Felleisen, J. Maraist, M. Odersky, and P. Wadler. A Call-By-Need Lambda Calculus. In *22^{th} POPL*. ACM Press, 1995.
2. Arvind, R. S. Nikhil, and K. K. Pingali. I-Structures: Data Structures for Parallel Computing. *ACM TOPLAS*, 11(4):598–632, 1989.
3. J.-Y. Girard, P. Taylor, and Y. Lafont. *Proofs and Types*. Cambridge University Press, Cambridge, England, 1989.
4. J. Greiner. Standard ML Weak Polymorphism Can Be Sound. Technical Report CMU-CS-93-160, Carnegie-Mellon University, 1993.
5. R. Harper and M. Lillibridge. Polymorphic Type Assignment and CPS Conversion. In *ACM SIGPLAN Workshop on Continuations*, April 1992. also Journal of LISP and Symbolic Computation.
6. M. Hoang, J.C. Mitchell, and R. Viswanathan. Standard ML-NJ Weak Polymorphism and Imperative Constructs. In *Proc. LICS*, pp. 15–25. IEEE Computer Society Press, June 1993.
7. G. Huet, editor. *Logical Foundations of Functional Programming*. UT Year of Programming Series. Addison-Wesley, Reading, MA, 1990.
8. K. M. Kahn, V. A. Saraswat, and J. Levy. Janus: A Step Towards Distributed Constraint Programming. In *Proc. NACLP*, Oct. 1990.
9. X. Leroy. *Polymorphic Typing of an Algorithmic Language*. PhD thesis, INRIA Rocquencourt, France, October 1992. Rapport de Recherche 1778.
10. X. Leroy. Polymorphism by Name for References and Continuations. In *20^{th} POPL*, pp. 220–231, January 1993.
11. Robin Milner. A Theory of Type Polymorphism in Programming. *Journal of Computer and System Science*, 17:348–375, 1978.
12. J. Mitchell. Type-Inference and Containment. In Huet [7], chapter 8. Part II.
13. J. Niehren. *Funktionale Berechnung in einem uniform nebenläufigen Kalkül mit logischen Variablen*. Doctoral Dissertation. Universität des Saarlandes, D-66041 Saarbrücken, Dec. 1994.
14. J. Niehren. Functional Computation as Concurrent Computation. Submitted, 1995. (`niehren@dfki.uni-sb.de`).
15. J. Niehren and M. Müller. Constraints for Free in Concurrent Computation. In *Asian Computer Science Conference*, 1995. LNCS, to appear. Preliminary version in *Proc. CCP 95*, 29–31 May, Venice, Italy, 1995.
16. J. Niehren and G. Smolka. A Confluent Relational Calculus for Higher-Order Programming with Constraints. In Jean-Pierre Jouannaud, editor, *Constraints in Computational Logics, LNCS 845*, München, Germany, 7–9 September 1994.
17. R. S. Nikhil. ID Language Reference Manual Version 90.1. Computation Structures Group Memo 284-2, MIT, July 1991.
18. B. C. Pierce and D. N. Turner. PICT User Manual. Technical Report to appear, LFCS, Edinburgh, 1995. Version 3.5g.
19. K. K. Pingali. Lazy Evaluation and the Logic Variable. Technical report, Cornell University, October 1987.
20. J. Reynolds. Towards a Theory of Type Structure. In *Proc. Colloque sur la Programmation, LNCS 19*, pp. 408–425. Springer-Verlag, 1974.
21. V. A. Saraswat. *Concurrent Constraint Programming Languages*. PhD thesis, School of Computer Science, Carnegie-Mellon University, Pittsburgh, CA, 1989.

22. V. A. Saraswat, M. Rinard, and P. Panangaden. Semantic Foundations of Concurrent Constraint Programming. In *18th ACM Symposium on Principles of Programming Languages*, pp. 333–352. ACM Press, January 1991.
23. C. Schulte, G. Smolka, and J. Würtz. Encapsulated Search and Constraint Programming in Oz. In *Proc. PPCP*, *LNCS 874*, pp. 134–150, Orcas Island, Washington, USA, 2-4 May 1994. Springer, Berlin, Germany.
24. G. Smolka. A Foundation for Concurrent Constraint Programming. In J.-P. Jouannaud, editor, *Constraints in Computational Logics*, *LNCS 845*, pp. 50–72, München, Germany, 7–9 Sep. 1994.
25. G. Smolka. An Oz primer. DFKI Oz documentation series, DFKI, Stuhlsatzenhausweg 3, D-66123 Saarbrücken, 1995.
26. G. Smolka. The Definition of Kernel Oz. In Andreas Podelski, editor, *Constraints: Basics and Trends*, *LNCS 910*, pp. 251–292. Springer, Berlin, Germany, 1995.
27. G. Smolka and R. Treinen, editors. *DFKI Oz Documentation Series.* DFKI, Stuhlsatzenhausweg 3, D-66123 Saarbrücken, 1995.
28. J.-P. Talpin and P. Jouvelot. The Type and Effect Discipline. In *IEEE Symposium on Logic in Computer Science.* IEEE Computer Society Press, 1992.
29. M. Tofte. Type Inference for Polymorphic References. *Information and Computation*, pp. 1–34, 1990.
30. D. N. Turner. *Types and Polymorphism in the Pi-Calculus.* PhD thesis, LFCS, Edinburgh, 1995. In Preparation.
31. J.B. Wells. Typability and Type Checking in the Second-Order λ-Calculus are Equivalent and Undecidable. In *Proc. LICS*, pp. 176–185, 1994.
32. A. K. Wright. Simple Imperative Polymorphism. *Lisp and Symbolic Computation*, 1994.

A Symmetric Equations and Cells

Symmetric equations and cells can be added to $\rho(x{\Leftarrow}y)$ by extending the syntax of expressions and constraints:

$$E ::= \ldots \mid x{:}\, y \mid x\langle y\, z\rangle \qquad \phi ::= \ldots \mid x{=}y$$

and the theory Δ_0. Symmetric equations should be interpreted as an equivalence relation implying directed equality:

$$\begin{aligned}\Delta_1 = \Delta_0 \cup \{\, & \forall x \forall y(x{=}y \rightarrow x \Leftarrow y),\ \forall x(x{=}x), \\ & \forall x \forall y \forall z(x{=}y \mathbin{\dot\wedge} y{=}z \rightarrow x{=}z),\ \forall x \forall y(x{=}y \rightarrow y{=}x)\}\end{aligned}$$

Systems **RF** or **R** need to be extended by one of the equivalent typing rules:

$$\frac{\Gamma(x) \equiv \Gamma(y)}{\Gamma \triangleright x{=}y} \qquad \frac{\Gamma \triangleright x \Leftarrow y \quad \Gamma \triangleright y \Leftarrow x}{\Gamma \triangleright x{=}y}$$

The definition of $\mathcal{O}$ straightforwardly extends with the additional clause:

$$\mathcal{O}(x{=}y) = \{x, y\}$$

A *cell* x hosting a variable y is written $x{:}\, y$. An *exchange* expression $x\langle y\, z\rangle$ binds z to the current value of the cell and replaces it by the new value y. The standard reduction rule for cells is (Cell-1) [24, 15] which performs the binding via a symmetric equation:

$$\text{(Cell-1)} \quad \phi \wedge x{:}\, y \wedge x'\langle u\, v\rangle \ \rightarrow\ \phi \wedge x{:}\, u \wedge v{=}y \quad \text{if } \phi \models_\Delta x' \Leftarrow x$$

This rule corresponds to the following pair of typing rules:

$$\frac{\Gamma(x){=}\langle S\rangle \quad \Gamma(y) = S}{\Gamma \triangleright x{:}\, y} \qquad \frac{\Gamma(x){=}\langle S\rangle \quad \Gamma(y){=}S \quad \Gamma(z){=}S}{\Gamma \triangleright x\langle y\, z\rangle}$$

According to these, a cell can host only values whose types have the same polymorphic structure up to $\equiv$. The additional clauses for $\mathcal{O}$ reflect the undirectedness of (Cell-1):

$$\mathcal{O}(x{:}\, y) \ = \ \{\} \qquad \mathcal{O}(x\langle y\, z\rangle) \ = \ \{z\}$$

Given a directed equation, the alternative rule (Cell-2) seems attractive:

$$\text{(Cell-2)} \quad \phi \wedge x{:}\, y \wedge x'\langle u\, v\rangle \ \rightarrow\ \phi \wedge x{:}\, u \wedge v \Leftarrow y \quad \text{if } \phi \models_\Delta x' \Leftarrow x$$

The indeterminism of a cell with type $\langle S\rangle$ is reflected by the requirement that $S \supseteq T$ and $R \supseteq S$ for *all* types T and R of inputs to x or outputs from x, respectively. Reduction (Cell-2) comes with the somewhat more liberal pair of typing rules:

$$\frac{\Gamma(x){=}\langle S\rangle \quad S \supseteq \Gamma(y)}{\Gamma \triangleright x{:}\, y} \qquad \frac{\Gamma(x){=}\langle S\rangle \quad S \supseteq \Gamma(y) \quad \Gamma(z) \supseteq S}{\Gamma \triangleright x\langle y\, z\rangle}$$

and the more permissive input-admissibility according to $\mathcal{O}$:

$$\mathcal{O}(x{:}\, y) \ = \ \{y\} \qquad \mathcal{O}(x\langle y\, z\rangle) \ = \ \{y, z\}$$

The correctness of **RF** for input-admissible expressions holds unchanged for expressions containing symmetric equations and cells with either reduction.

This article was processed using the LaTeX macro package with LLNCS style