Approaches to Polymorphism in Classical Sequent Calculus

Alexander J. Summers and Steffen van Bakel

Department of Computing, Imperial College London
180 Queen's Gate, London SW7 2AZ, U.K.
{ajs300m,svb}@doc.ic.ac.uk

Abstract. $\mathcal{X}$ is a relatively new calculus, invented to give a Curry-Howard correspondence with Classical Implicative Sequent Calculus. It is already known to provide a very expressive language; embeddings have been defined of the λ-calculus, Bloo and Rose's $\lambda\mathbf{x}$, Parigot's $\lambda\mu$ and Curien and Herbelin's $\lambda\mu\tilde{\mu}$.
We investigate various notions of polymorphism in the context of the $\mathcal{X}$-calculus. In particular, we examine the first class polymorphism of System F, and the shallow polymorphism of ML. We define analogous systems based on the $\mathcal{X}$-calculus, and show that these are suitable for embedding the original calculi.
In the case of shallow polymorphism we obtain a more general calculus than ML, while retaining its useful properties. A type-assignment algorithm is defined for this system, which generalises Milner's $\mathcal{W}$.

1 Introduction

Polymorphism is a powerful aspect of most modern programming languages. It is a mechanism for allowing a program to be applied with various different types for its inputs (or outputs), and so allows flexibility and reuse of code. For example, in a polymorphic system, the identity function might be given the type $\forall X.(X \rightarrow X)$, where the $\forall$-bound type variable X ranges over all types. This correctly expresses that the identity may be typed with $A \rightarrow A$ for any and all formulas A. The rules for type-assignment typically allow this type to be *instantiated* several different times, so that it would be acceptable for the identity function to be applied to both an integer and a list in the same program.

$\mathcal{X}$ is based on the work of [5] and [9], and has since been further studied in [10]. Like the $\lambda\mu$-calculus of Parigot [7], it has been designed to have a Curry-Howard correspondence with Classical Logic. Unlike most existing calculi in this field (which, like $\lambda\mu$ are typically based on a Natural Deduction formulation of logic), $\mathcal{X}$ corresponds to a Classical Sequent Calculus. The particular sequent calculus is defined by Urban [9].

In this paper we investigate various notions of polymorphism based on the logical $\forall$ connective, in the context of the $\mathcal{X}$-calculus. We examine the first class polymorphism of System F, and the shallow polymorphism of ML. We define analogous systems based on the $\mathcal{X}$-calculus, and show that these systems are suitable for encoding System F and ML. In the case of shallow polymorphism we present a more general calculus than ML, and show that all the useful properties of ML still hold.

2 The $\mathcal{X}$-Calculus

In this section we will give a brief presentation of the $\mathcal{X}$-calculus; a more detailed description is given in [10]. We present here the syntax and reduction rules, and aim to give an intuition of how the calculus behaves.

Although $\mathcal{X}$ provides a rather different computational behaviour to calculi based on the λ-calculus, it has been shown that it can faithfully encode many such calculi, including λ-calculus, λx, and the $\lambda\mu$-calculus [10]. These calculi incorporate variable-symbols and (with the exception of λx) rely on an implicit concept of substitution to perform the basic computational steps. $\mathcal{X}$ on the other hand features two separate categories of 'connectors', *plugs* and *sockets*, that act as input and output channels, and is defined without any notion of substitution.

Definition 1 ($\mathcal{X}$-Terms). The terms of the $\mathcal{X}$-calculus are defined by the following syntax, where x, y range over the infinite set of *sockets* and α, β over the infinite set of *plugs* (sockets and plugs together form the set of *connectors*).

$$\begin{array}{rcll} P, Q & ::= & \langle x.\alpha \rangle & \textit{capsule} \\ & | & \widehat{y}P\widehat{\beta}\cdot\alpha & \textit{export} \\ & | & P\widehat{\beta}\,[y]\,\widehat{x}Q & \textit{mediator} \\ & | & P\widehat{\alpha}\dagger\widehat{x}Q & \textit{cut} \end{array}$$

The $\widehat{\cdot}$ symbolises that the connector underneath is bound in the circuit; notions of *free* and *bound* connectors are defined as usual. We will use $\mathit{fp}(P)$ to denote the free plugs of P, and similarly $\mathit{fs}(P)$ for free sockets.

The notion of reduction on $\mathcal{X}$-terms corresponds to the process of *cut elimination* on sequent calculus proofs. As such, the reduction rules define how cuts may be eliminated from an $\mathcal{X}$-term. If a cut binds a connector which occurs several times in the corresponding subterm, it may not be immediately eliminated, but rather must seek out each of these occurrences and make a copy of itself for each one. For example, if there are many occurrences of x in the term Q then the term $P\widehat{\alpha}\dagger\widehat{x}Q$ can reduce by 'pushing' the cut into the structure of Q, and making a cut between a copy of P and each x found. This process of 'pushing' is correctly referred to as propagation, in this example right-propagation (since we propagate the cut into the right-hand term). Once the cut reaches a level where a single α and x are immediately introduced in its two subterms, a *logical rule* specifies how the two subterms can communicate with one another through the cut. For example, a cut between an export and a mediator allows the body of the function from the export to be inserted between the two subterms of the mediator.

Definition 2 (Logical Rules). The logical rules are presented by:

$$\begin{array}{lrcll} (\textit{cap}): & \langle y.\alpha\rangle\widehat{\alpha}\dagger\widehat{x}\langle x.\beta\rangle & \to & \langle y.\beta\rangle & \\ (\textit{exp}): & (\widehat{y}P\widehat{\beta}\cdot\alpha)\widehat{\alpha}\dagger\widehat{x}\langle x.\gamma\rangle & \to & \widehat{y}P\widehat{\beta}\cdot\gamma & \alpha \notin \mathit{fs}(P) \\ (\textit{med}): & \langle y.\alpha\rangle\widehat{\alpha}\dagger\widehat{x}(P\widehat{\beta}\,[x]\,\widehat{z}Q) & \to & P\widehat{\beta}\,[y]\,\widehat{z}Q & x \notin \mathit{fs}(P,Q) \\ (\textit{exp-med}): & (\widehat{y}P\widehat{\beta}\cdot\alpha)\widehat{\alpha}\dagger\widehat{x}(Q\widehat{\gamma}\,[x]\,\widehat{z}R) & \to & \left\{\begin{array}{l} Q\widehat{\gamma}\dagger\widehat{y}(P\widehat{\beta}\dagger\widehat{z}R) \\ (Q\widehat{\gamma}\dagger\widehat{y}P)\widehat{\beta}\dagger\widehat{z}R \end{array}\right\} & \begin{array}{l}\alpha \notin \mathit{fs}(P),\\ x \notin \mathit{fs}(Q,R)\end{array} \end{array}$$

The first three logical rules above specify a renaming (reconnecting) procedure, whereas the last rule specifies the basic computational step: it links the exportation of a function, available on the plug α, to an adjacent mediator via the socket x (the resulting cuts may be bracketed either way, as shown).

A key element of the cut-elimination procedure of [9] is that cuts which are propagated to the left or right are marked as such.

Definition 3 (Active Cuts). The syntax is extended with two *flagged* or *active* cuts:

$$P ::= \ldots \mid P_1 \widehat{\alpha} \not\leftarrow \widehat{x} P_2 \mid P_1 \widehat{\alpha} \not\rightarrow \widehat{x} P_2$$

We define two *cut-activation* rules.

$$\begin{array}{l} (\textit{act}\text{-L}) : \; P\widehat{\alpha} \dagger \widehat{x}Q \;\rightarrow\; P\widehat{\alpha} \not\leftarrow \widehat{x}Q \;\; \textit{if } P \textit{ does not introduce } \alpha \\ (\textit{act}\text{-R}) : \; P\widehat{\alpha} \dagger \widehat{x}Q \;\rightarrow\; P\widehat{\alpha} \not\rightarrow \widehat{x}Q \;\; \textit{if } Q \textit{ does not introduce } x \end{array}$$

where: P *introduces* x**:** Either $P = Q\widehat{\beta}\,[x]\,\widehat{y}R$ and $x \notin \mathit{fs}(Q, R)$, or $P = \langle x.\alpha \rangle$.
P *introduces* α**:** Either $P = \widehat{x}Q\widehat{\beta}{\cdot}\alpha$ and $\alpha \notin \mathit{fp}(Q)$, or $P = \langle x.\alpha \rangle$.

An activated cut is processed by 'pushing' it systematically through the syntactic structure of the circuit in the direction indicated by the tilting of the dagger. Whenever an active cut meets a circuit exhibiting the connector it is trying to communicate with, a new (inactive) cut is 'deposited', representing an attempt to communicate at this level. The pushing of the active cut continues until the level of capsules is reached, where it is either deactivated or destroyed. Once again, the inactive cut can reduce via a logical rule, or pushing can continue in the other direction. This behaviour is expressed by the following propagation rules.

Definition 4 (Propagation Rules).
Left Propagation:

$$\begin{array}{rrll} (\not\leftarrow\dagger): & \langle y.\alpha \rangle \widehat{\alpha} \not\leftarrow \widehat{x}P \;\rightarrow\; \langle y.\alpha \rangle \widehat{\alpha} \dagger \widehat{x}P & \\ (\not\leftarrow \textit{cap}): & \langle y.\beta \rangle \widehat{\alpha} \not\leftarrow \widehat{x}P \;\rightarrow\; \langle y.\beta \rangle & \beta \neq \alpha \\ (\not\leftarrow \textit{exp-outs}): & (\widehat{y}Q\widehat{\beta}{\cdot}\alpha)\widehat{\alpha} \not\leftarrow \widehat{x}P \;\rightarrow\; (\widehat{y}(Q\widehat{\alpha} \not\leftarrow \widehat{x}P)\widehat{\beta}{\cdot}\gamma)\widehat{\gamma} \dagger \widehat{x}P, & \gamma \textit{ fresh} \\ (\not\leftarrow \textit{exp-ins}): & (\widehat{y}Q\widehat{\beta}{\cdot}\gamma)\widehat{\alpha} \not\leftarrow \widehat{x}P \;\rightarrow\; \widehat{y}(Q\widehat{\alpha} \not\leftarrow \widehat{x}P)\widehat{\beta}{\cdot}\gamma, & \gamma \neq \alpha \\ (\not\leftarrow \textit{med}): & (Q\widehat{\beta}\,[z]\,\widehat{y}R)\widehat{\alpha} \not\leftarrow \widehat{x}P \;\rightarrow\; (Q\widehat{\alpha} \not\leftarrow \widehat{x}P)\widehat{\beta}\,[z]\,\widehat{y}(R\widehat{\alpha} \not\leftarrow \widehat{x}P) & \\ (\not\leftarrow \textit{cut}): & (Q\widehat{\beta} \dagger \widehat{y}R)\widehat{\alpha} \not\leftarrow \widehat{x}P \;\rightarrow\; (Q\widehat{\alpha} \not\leftarrow \widehat{x}P)\widehat{\beta} \dagger \widehat{y}(R\widehat{\alpha} \not\leftarrow \widehat{x}P) & \end{array}$$

Right Propagation:

$$\begin{array}{rrll} (\not\rightarrow\dagger): & P\widehat{\alpha} \not\rightarrow \widehat{x}\langle x.\beta \rangle & \rightarrow\; P\widehat{\alpha} \dagger \widehat{x}\langle x.\beta \rangle & \\ (\not\rightarrow \textit{cap}): & P\widehat{\alpha} \not\rightarrow \widehat{x}\langle y.\beta \rangle & \rightarrow\; \langle y.\beta \rangle, & y \neq x \\ (\not\rightarrow \textit{exp}): & P\widehat{\alpha} \not\rightarrow \widehat{x}(\widehat{y}Q\widehat{\beta}{\cdot}\gamma) & \rightarrow\; \widehat{y}(P\widehat{\alpha} \not\rightarrow \widehat{x}Q)\widehat{\beta}{\cdot}\gamma & \\ (\not\rightarrow \textit{med-outs}): & P\widehat{\alpha} \not\rightarrow \widehat{x}(Q\widehat{\beta}\,[x]\,\widehat{y}R) & \rightarrow\; P\widehat{\alpha} \dagger \widehat{z}((P\widehat{\alpha} \not\rightarrow \widehat{x}Q)\widehat{\beta}\,[z]\,\widehat{y}(P\widehat{\alpha} \not\rightarrow \widehat{x}R)), & \\ & & & z \textit{ fresh} \\ (\not\rightarrow \textit{med-ins}): & P\widehat{\alpha} \not\rightarrow \widehat{x}(Q\widehat{\beta}\,[z]\,\widehat{y}R) & \rightarrow\; (P\widehat{\alpha} \not\rightarrow \widehat{x}Q)\widehat{\beta}\,[z]\,\widehat{y}(P\widehat{\alpha} \not\rightarrow \widehat{x}R), & z \neq x \\ (\not\rightarrow \textit{cut}): & P\widehat{\alpha} \not\rightarrow \widehat{x}(Q\widehat{\beta} \dagger \widehat{y}R) & \rightarrow\; (P\widehat{\alpha} \not\rightarrow \widehat{x}Q)\widehat{\beta} \dagger \widehat{y}(P\widehat{\alpha} \not\rightarrow \widehat{x}R) & \end{array}$$

The symmetry of the cut can be seen by these rules - it may (depending on the conditions on the activation rules) be propagated to the left or right, making copies of the right or left term respectively. Right-propagation is reminiscent of *substitution* of terms for term-variables; left-propagation $P\widehat{\alpha} \leftslash \widehat{x}Q$ then is its dual: it expresses the connection of the continuation Q, accessible via x, to all the 'calls' α in P.

We write $\rightarrow$ for the (reflexive, transitive, compatible) reduction relation generated by the logical, propagation and activation rules. The reduction relation $\rightarrow$ is not confluent; this comes in fact from the critical pair that activates a cut $P\widehat{\alpha} \dagger \widehat{x}Q$ in two ways if P does not introduce α and Q does not introduce x.

Definition 5 ([10]). The interpretation of lambda terms into circuits of $\mathcal{X}$ via the plug α, $[\![M]\!]^{\lambda}_{\alpha}$, is defined by:

$$\begin{array}{rcll} [\![x]\!]^{\lambda}_{\alpha} & = & \langle x.\alpha \rangle & \\ [\![\lambda x.M]\!]^{\lambda}_{\alpha} & = & \widehat{x}[\![M]\!]^{\lambda}_{\beta}\widehat{\beta}\cdot\alpha, & \beta \textit{ fresh} \\ [\![MN]\!]^{\lambda}_{\alpha} & = & [\![M]\!]^{\lambda}_{\gamma}\widehat{\gamma} \dagger \widehat{x}([\![N]\!]^{\lambda}_{\beta}\widehat{\beta}\,[x]\,\widehat{y}\langle y.\alpha\rangle), & x, y, \beta, \gamma \textit{ fresh} \end{array}$$

In [10] it is shown that this interpretation respects (CBN/CBV) reduction and typeability.

Notice that every sub-circuit of $[\![M]\!]^{\lambda}_{\alpha}$ has exactly one free plug. This can be seen as an explicit notation for the output of the lambda term (outputs are not explicitly labelled in λ-calculus).

3 Type Assignment for $\mathcal{X}$

The notion of type assignment on $\mathcal{X}$ that we present in this section is the basic implicative system for Classical Logic. The Curry-Howard property is easily achieved.

Definition 6 (Types and Contexts).

1. The set of types $\mathcal{T}_C$, ranged over by A, B, is defined over a set of *atomic types* $\mathcal{V} = \{\varphi_1, \varphi_2, \varphi_3, \ldots\}$ by the grammar:

$$A, B ::= \varphi \mid A{\rightarrow}B$$

   These types are normally known as *Curry types*.
2. A *context of sockets* Γ is a mapping from sockets to types, denoted as a finite set of *statements* $x{:}A$, such that the *subjects* of the statements (the sockets) are distinct. We write $\Gamma, x{:}A$ for $\Gamma \cup \{x{:}A\}$. When writing a context as $\Gamma, x{:}A$, we indicate that either Γ is not defined on x or contains the same statement $x{:}A$. We write $\Gamma\backslash x$ for the context from which the statement concerning x, if any, has been removed. *Contexts of plugs* Δ, and the notations $\alpha{:}A, \Delta$ and $\Delta\backslash\alpha$ are defined in a similar way.
3. A pair $\langle \Gamma; \Delta \rangle$ is usually referred to simply as a *context*, and is a shorthand for the sequent $\Gamma \vdash \Delta$.

The notation $\Gamma \vdash \Delta$ will still usually be used when discussing sequents.

Definition 7 (Typing for $\mathcal{X}$).

1. *Type judgements* are expressed via a ternary relation $P \vdots\ \Gamma \vdash \Delta$, where Γ is a context of *sockets* and Δ is a context of *plugs*, and P is an $\mathcal{X}$-term. We say that P is the *witness* of this judgement.
2. *Type assignment* is defined by the following sequent calculus:

$$(\textit{cap}) : \frac{}{\langle y.\alpha \rangle \vdots\ \Gamma, y{:}A \vdash \alpha{:}A, \Delta} \quad (\textit{med}) : \frac{P \vdots\ \Gamma \vdash \alpha{:}A, \Delta \quad Q \vdots\ \Gamma, x{:}B \vdash \Delta}{P\widehat{\alpha}\ [y]\ \widehat{x}Q \vdots\ \Gamma, y{:}A{\to}B \vdash \Delta}$$

$$(\textit{exp}) : \frac{P \vdots\ \Gamma, x{:}A \vdash \alpha{:}B, \Delta}{\widehat{x}P\widehat{\alpha}{\cdot}\beta \vdots\ \Gamma \vdash \beta{:}A{\to}B, \Delta} \quad (\textit{cut}) : \frac{P \vdots\ \Gamma \vdash \alpha{:}A, \Delta \quad Q \vdots\ \Gamma, x{:}A \vdash \Delta}{P\widehat{\alpha} \dagger \widehat{x}Q \vdots\ \Gamma \vdash \Delta}$$

We write $P \vdots\ \Gamma \vdash \Delta$ if there exists a derivation that has this judgement in the bottom line.

Notice that, in $P \vdots\ \Gamma \vdash \Delta$, Γ and Δ carry the types of the free connectors in P, as unordered sets. By the Curry-Howard correspondence, P represents a proof of the sequent $\Gamma \vdash \Delta$, so P is actually a witness to this sequent being derivable in the logic. Moreover, there is no notion of a single type for P itself, instead the derivable statement shows the consistency between the free connectors of P.

It is important to note that the typing rules include a notion of implicit contraction (just as the original sequent rules do); if a new statement is introduced on the bottom line of a rule, but it was already present in the context, then it is simply merged. We do not consider duplicate statements, as we consider contexts to be unordered sets.

We have the following result:

Theorem 8 (Witness Reduction [10]). *If $P \vdots\ \Gamma \vdash \Delta$, and $P \to Q$, then $Q \vdots\ \Gamma \vdash \Delta$.*

Also, the standard notion of Curry type assignment on lambda terms and the notion of type assignment on $\mathcal{X}$ defined above are strongly linked:

Theorem 9 ([10]). *If $\Gamma \vdash_\lambda M : A$, then $[\![M]\!]^\lambda_\alpha \vdots\ \Gamma \vdash \alpha{:}A$.*

In [11] a notion of *principal contexts* (principal typings, in the language of [12]) is defined by providing an algorithm *pC* that, given an $\mathcal{X}$-term P, returns a context $\langle \Gamma; \Delta \rangle$, with the following properties:

Theorem 10 (Soundness and Completeness of *pC*).

1. *Soundness: If $pC(P) = \langle \Gamma; \Delta \rangle$, then $P \vdots\ \Gamma \vdash \Delta$.*
2. *Completeness: If $P \vdots\ \Gamma \vdash \Delta$, then there exist Γ_p and Δ_p, and a substitution S such that $pC(P) = \langle \Gamma_p; \Delta_p \rangle$, and $(S\ \Gamma_p) \subseteq \Gamma$ and $(S\ \Delta_p) \subseteq \Delta$.*

4 System F in $\mathcal{X}$

In this section, we will examine the System F approach to polymorphism, and how it may be incorporated into the $\mathcal{X}$-calculus. We will present System F, and show it can be expressed in an $\mathcal{X}$ setting, by giving an explicit encoding into a variant of the $\mathcal{X}$-calculus. We will show that typings and reductions are preserved by this encoding.

4.1 System F

System F (also known as the Polymorphic λ-calculus) was invented independently by Jean-Yves Girard [4] and John C. Reynolds [8]. We will give here a short overview of its main definitions, based largely on those of [3].

Definition 11 (System F Types). The types of System F (ranged over by A, B) are defined over an infinite set of *atomic types* (ranged over by φ), and one of type variable-symbols (ranged over by X, Y), in the following way:

$$A, B ::= \varphi \mid X \mid A \to B \mid \forall X.A$$

A type is *well-formed* if and only if it contains no free type variable-symbols (i.e. every such symbol X appears under a $\forall X$ binder). It is useful to consider types modulo some kind of alpha-conversion, for example we would like to identify the types $\forall X.(X \to X)$ and $\forall Y.(Y \to Y)$. From here on we will assume this.

Definition 12 (System F). The terms of System F (à la Church) are defined over an infinite set of typed term variable-symbols, $\{x^A, y^B, \ldots\}$, where A, B can be any System F type. They are defined by the following syntax:

$$M, N ::= x^A \mid \lambda x^A.M^1 \mid MN \mid \Lambda\varphi.M^2 \mid MA$$

[1]: *if x^B appears free in M, then $B = A$.*
[2]: *φ does not appear in the type of a free term variable of M.*

The syntax as described above is in fact rather too liberal; a notion of *well-formed terms* will be employed, which insists that terms must have a well-formed type. It is simple to derive the type of a particular System F term (unique, modulo alpha conversion) from the type information within the syntax. We will write $M{:}_F A$ to denote that A is the type of the term M.

Definition 13 (Type Derivation in System F). The procedure of type derivation is defined as follows:

$$\frac{}{x^A{:}_F A}\,(Ax) \qquad \frac{M{:}_F B}{\lambda x^A.M{:}_F A \to B}\,(\to\mathcal{I}) \qquad \frac{M{:}_F A \to B \qquad N{:}_F A}{(MN){:}_F B}\,(\to\mathcal{E})$$

$$\frac{M{:}_F A}{\Lambda\varphi.M{:}_F \forall X.A[X/\varphi]}\,(\forall\mathcal{I}) \qquad \frac{M{:}_F \forall X.A}{(MB){:}_F A[B/X]}\,(\forall\mathcal{E})$$

For example, we would not consider the term $(x^A y^A)$ to be well-formed, since (according to the rules above) it does not have a type. In addition, we would not consider the term $\lambda x^X.x^X$ to be well-formed (its type is $X \to X$ where X is free). As an example of a well-formed term, the identity function would be represented in System F by the term $\Lambda\varphi.\lambda x^\varphi.x^\varphi$, which has the type $\forall X.(X \to X)$. From here onwards we will assume terms are well-formed unless otherwise stated.

Definition 14 (System F Reductions). There are two reduction rules:

$$\begin{array}{rcl}(\lambda x^A.M)\ N & \to_{\mathrm{F}} & M[N/x^A] \\ (\Lambda\varphi.M)\ A & \to_{\mathrm{F}} & M[A/\varphi]\end{array}$$

In general, we will write $\to_{\mathrm{F}}$ for the reflexive, transitive, compatible closure of the relation generated by these rules.

System F à la Church possesses a Curry-Howard correspondence with the '$\forall, \to$'-fragment of Intuitionistic Natural Deduction. Since each term carries only one type, the correspondence between terms and proofs is in fact one-to-one.

To illustrate the polymorphism in this system, we can find a typeable term analogous with the lambda term $(\lambda z.zz)(\lambda x.x)$. The term we would use is

$$(\lambda z^{\forall Z.(Z\to Z)}.\Lambda\varphi_1.((z^{\forall Z.(Z\to Z)}\ (\varphi_1\to\varphi_1))(z^{\forall Z.(Z\to Z)}\ \varphi_1)))(\Lambda\varphi_2.\lambda x^{\varphi_2}.x^{\varphi_2})$$

4.2 Typed Polymorphic $\mathcal{X}$

One possible method of introducing polymorphism to $\mathcal{X}$ is to go back to the sequent calculus rules, and encode the quantifier rules there into the syntax of a typed version of $\mathcal{X}$. This gives typed $\mathcal{X}$-terms which naturally carry polymorphic types. This approach is analogous to that of System F, where the original implicative calculus (typed λ-calculus) is extended with representations of the '$\forall$' rules.

The $\forall$-rules from the sequent calculus are as follows:

$$\frac{\Gamma, A[B/X] \vdash \Delta}{\Gamma, \forall X.A \vdash \Delta}\ (\forall\mathcal{L}) \qquad \frac{\Gamma \vdash A, \Delta}{\Gamma \vdash \forall X.A[X/\varphi], \Delta}\ (\forall\mathcal{R})^*$$

** if φ does not occur in Γ, Δ.*

Notice that (as is typical of the sequent calculus rules) quantifiers are only introduced (and not eliminated), but may be introduced on the left of a sequent (which approximately corresponds to elimination in a Natural Deduction setting).

We introduce two new terms, representing the rules $(\forall\mathcal{L})$ and $(\forall\mathcal{R})$, and give a typed version of the existing syntax.

Definition 15 (Typed Polymorphic $\mathcal{X}$). The terms of Typed Polymorphic $\mathcal{X}$ (hereafter denoted by $\mathcal{X}^\forall$) are defined by the following syntax:

$$\begin{array}{rcl} P, Q & ::= & \langle x^A.\alpha^A\rangle \ \mid\ \widehat{y}^A P\widehat{\beta}^B\cdot\alpha^{A\to B} \ \mid\ P\widehat{\beta}^A\,[y^{A\to B}]\,\widehat{x}^B Q \\ & \mid & P\widehat{\alpha}^A \dagger \widehat{x}^A Q \ \mid\ P\widehat{\alpha}^A{}_{\lhd_{\varphi}}\beta^{\forall X.A[X/\varphi]\ (*)} \ \mid\ y^{\forall X.A}{}_{\rhd_B}\widehat{x}^{A[B/X]}Q \end{array}$$

$^{(*)}$ *φ does not appear in the type of a free connector of P, except (possibly) α^A.*

The notation $\lhd$ is chosen to indicate the *generalisation* of the output α, whereas the symbol $\rhd$ denotes the corresponding *instantiation*. Although instantiation is really a 'Natural Deduction way' of considering this mechanism (where there is an elimination rule to do the job), the concept still makes sense in reading the term from left to right, since this means reading the $(\forall\mathcal{L})$ rule from the bottom upwards.

Notice that no process of derivation is required in determining the type (context) of an $\mathcal{X}^\forall$ term - the context may be immediately formed by taking a statement for each free connector in the term, with the type it has there. This is because outputs are labelled as well as inputs, so all of the pertinent information is present in the term. For example, the $\mathcal{X}^\forall$ term $\widehat{y^A}\langle x^B.\beta^B\rangle\widehat{\beta^B}\cdot\alpha^{A\to B}$ would be given the context $x{:}B \vdash \alpha{:}A{\to}B$. We write $P :\cdot\ \Gamma \vdash_\forall \Delta$ to indicate that $\Gamma \vdash \Delta$ is the context for the $\mathcal{X}^\forall$ term P.

It is straightforward to convert the original $\mathcal{X}$ reduction rules into their typed versions. The extra rules required to deal with the new syntax constructs are given in Appendix A. We will write $\to_\forall$ for the reduction relation for $\mathcal{X}^\forall$.

We have the following result:

Theorem 16 (Witness Reduction for $\mathcal{X}^\forall$). *For all $\mathcal{X}^\forall$-terms P, Q, if $P :\cdot\ \Gamma_P \vdash_\forall \Delta_P$, and $P \to_\forall Q$ and $Q :\cdot\ \Gamma_Q \vdash_\forall \Delta_Q$, then $\Gamma_Q \subseteq \Gamma_P$ and $\Delta_Q \subseteq \Delta_P$.*

So our new formulation of the calculus is well-behaved with respect to the type-assignment proposed.

It is possible to encode System F à la Church into $\mathcal{X}^\forall$, just as $\mathcal{X}$ can encode the original λ-calculus. The interpretation is based on the translation from Natural Deduction to Sequent Calculus proofs, as originally given in [2].

The interpretation function takes as input a System F term and a plug α (used to represent the output in the resulting $\mathcal{X}^\forall$ term) and returns the corresponding $\mathcal{X}^\forall$ term. It makes use of the derivation of the (unique) type of a System F term, of Definition 13.

Definition 17 (Encoding System F à la Church). The interpretation of System F into $\mathcal{X}^\forall$, via the plug α is defined recursively by:

$$\llbracket x^A \rrbracket^\forall_\alpha = \langle x^A.\alpha^A\rangle$$

$$\llbracket \lambda x^A.M \rrbracket^\forall_\alpha = \widehat{x}^A P\widehat{\beta}^B \cdot \alpha^C \quad \text{where } M :_F B,\ P = \llbracket M \rrbracket^\forall_\beta,\ C = A{\to}B$$

$$\llbracket MN \rrbracket^\forall_\alpha = P\widehat{\beta}^C \dagger \widehat{x}^C(Q\widehat{\gamma}^A\,[x^C]\,\widehat{y}^B\langle y^B.\alpha^B\rangle) \quad \text{where } M :_F A{\to}B,\ N :_F A,\ P = \llbracket M \rrbracket^\forall_\beta,\ Q = \llbracket N \rrbracket^\forall_\gamma,\ C = A{\to}B$$

$$\llbracket \Lambda\varphi.M \rrbracket^\forall_\alpha = P\widehat{\beta}^A \underset{\varphi}{\lhd} \alpha^B \quad \text{where } M :_F A,\ P = \llbracket M \rrbracket^\forall_\beta,\ B = \forall X.A[X/\varphi]$$

$$\llbracket MB \rrbracket^\forall_\alpha = P\widehat{\beta}^C \dagger \widehat{x}^C(x^C \underset{B}{\rhd} \widehat{y}^D\langle y^D.\alpha^D\rangle) \quad \text{where } M :_F \forall X.A,\ P = \llbracket M \rrbracket^\forall_\beta,\ C = \forall X.A,\ D = A[B/X]$$

The following results show that we can simulate System F faithfully.

Theorem 18. *1. If $M \to_{\mathrm{F}} N$ then $\llbracket M \rrbracket^\forall_\alpha \to_\forall \llbracket N \rrbracket^\forall_\alpha$.*
2. If $M{:}_F A$ then there exists a Γ such that $\llbracket M \rrbracket^\forall_\alpha :\cdot\ \Gamma \vdash_\forall \alpha{:}A$.

4.3 Untyped Polymorphic $\mathcal{X}$

As an alternative to $\mathcal{X}^\forall$, it is possible to work in the style of System F à la Curry and deal with the original syntax of $\mathcal{X}$ while allowing polymorphism to be represented only

in the type system. This is essentially achieved by employing System F types, and by adding the following two type assignment rules to those standard for $\mathcal{X}$.

$$\frac{P :\cdot \Gamma, x{:}A[B/X] \vdash_{\mathrm{p}} \Delta}{P :\cdot \Gamma, x{:}\forall X.A \vdash_{\mathrm{p}} \Delta}\ (\forall\mathcal{L}) \qquad \frac{P :\cdot \Gamma \vdash_{\mathrm{p}} \alpha{:}A, \Delta}{P :\cdot \Gamma \vdash_{\mathrm{p}} \alpha{:}\forall X.A[X/\varphi], \Delta}\ (\forall\mathcal{R})^*$$

**if φ does not occur in Γ, Δ.*

We encode System F à la Curry by the usual encoding of the λ-calculus syntax into $\mathcal{X}$, as given in Definition 5. This encoding respects typeability and reductions.

5 Shallow Polymorphism

In this section, we will examine the style of polymorphism commonly associated with ML, that of *shallow polymorphism*. We will show that a shallow polymorphic type assignment can be naturally defined on $\mathcal{X}$-terms without the need to extend the syntax (in contrast to the case of the λ-calculus). We will show that ML can be encoded into $\mathcal{X}$, and that using this new type-assignment, typings and reductions are preserved. We will discuss the notions of principal types and typings [12] with respect to our shallow polymorphic version of $\mathcal{X}$, and present a type inference algorithm in the style of the algorithm $\mathcal{W}$ of [6].

ML [6] is a calculus based upon the λ-calculus, which uses a different approach to System F for admitting polymorphism. To obtain decidability of type assignment, it permits only *shallow polymorphism*, which means that types are allowed to contain the $\forall$ symbol only on the outside of their structure.

The syntax of the λ-calculus is extended the construct $\mathtt{let}\ x = M\ \mathtt{in}\ N$ which (along with its typing rule) is designed to give a workaround for the situation when an application $(\lambda x.N)M$ would be untypeable, whereas the reduct $N[M/x]$ can be typed. The typing rule for $\mathtt{let}$ allows M to be given a shallow polymorphic type, and for this type to be used for x when trying to derive a type for N. This way, it may be that several instances of the polymorphic type are used for different occurrences of x within N.

Definition 19 (ML Expressions). The set $\mathcal{L}_{\mathrm{ML}}$ of *ML expressions* is defined by:

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

The construct $\mathtt{Fix}\, g.M$ is included to allow recursion in the calculus. For simplicity in our discussions of polymorphism we choose to study the subset of ML expressions without $\mathtt{Fix}$, and from hereon will consider ML expressions only within this subset.

Definition 20 (ML Reductions). The reduction rules in ML are as follows:

$$\begin{aligned} (\lambda x.N)M &\ \rightarrow_{\mathrm{ML}}\ N[M/x] \\ (\mathtt{let}\ x = M\ \mathtt{in}\ N) &\ \rightarrow_{\mathrm{ML}}\ N[M/x] \end{aligned}$$

The typing rules for $\mathtt{let}$ provide the polymorphism in this system - it is allowed for each of the occurrences of x in N to be given a different instance of a polymorphic type found for M. This is in contrast to the usual way in which the term $(\lambda x.N)M$ would be treated, which would allow only *one* Curry type to be used for the variable x.

Definition 21 (Generic Types [6]).
The set of *generic types* is built from the usual Curry types by allowing $\forall$ quantifiers to be built on the outside. We will use A, B to range over the usual Curry types, and ψ to range over generic types, as defined below.

$$\begin{array}{ll} A ::= \varphi \mid X \mid (A \to B) & \textit{Curry types} \\ \psi ::= A \mid (\forall X.\psi) & \textit{generic types} \end{array}$$

As in the discussions in the previous section, we distinguish between atomic types φ and type variable symbols X (whereas Milner chooses not to), and again consider only types with no free type-variable symbols to be well-formed.

Definition 22 ([1]). *ML-type assignment* and *ML-derivations* are defined by the following deduction system.

$$(ax): \quad \frac{}{\Gamma \vdash_{\text{ML}} x : \psi}\ (x{:}\psi \in \Gamma) \qquad (\texttt{let}): \frac{\Gamma \vdash_{\text{ML}} M_1 : \psi \quad \Gamma, x{:}\psi \vdash_{\text{ML}} M_2 : B}{\Gamma \vdash_{\text{ML}} (\texttt{let}\ x = M_1\ \texttt{in}\ M_2) : B}$$

$$(\to I): \quad \frac{\Gamma, x{:}A \vdash_{\text{ML}} M : B}{\Gamma \vdash_{\text{ML}} \lambda x.M : A \to B} \qquad (\to E): \frac{\Gamma \vdash_{\text{ML}} M_1 : A \to B \quad \Gamma \vdash_{\text{ML}} M_2 : A}{\Gamma \vdash_{\text{ML}} M_1 M_2 : B}$$

$$(\forall I): \frac{\Gamma \vdash_{\text{ML}} M : \psi}{\Gamma \vdash_{\text{ML}} M : \forall X.\psi[X/\varphi]}\ (*) \qquad (\forall E): \quad \frac{\Gamma \vdash_{\text{ML}} M : \forall X.\psi}{\Gamma \vdash_{\text{ML}} M : \psi[B/X]}$$

**If φ is not free in Γ.*

Notice that generic types ψ may not be used in the $(\to I)$ or $(\to E)$ rules - this reflects the fact that $\forall$-symbols may not appear inside an arrow type. However, when x is a variable not occurring under an abstraction, the rules allow more freedom - if x has a polymorphic type in the basis then the use of the (ax) and $(\forall E)$ rules allows a different instance of this type to be chosen each time x is used.

Although ML admits less polymorphism than System F does, it has the advantage of being very practical - not only is type assignment in ML decidable (in contrast to System F), but it has a principal type property. Milner presents an algorithm (called $\mathcal{W}$) that takes as input a pair of (*basis*, *term*) and returns a pair of (*substitution*, *type*), representing the most general typing for the term (if one exists) using a substitution instance of the basis.

6 ML in $\mathcal{X}$

The key to the use of polymorphism in ML is in the `let` construct, which is interpreted as a substitution both syntactically (according to its reduction rule) and semantically (see [6]). The polymorphism present in the $(\texttt{let})$-rule essentially gives a way of typing the substitution about to take place, such that the multiple occurrences of the name to replace need not all be typed in the same way. The `let`-construct is a necessary extension to the syntax for a shallow polymorphic approach (short of allowing polymorphism to be used directly with abstractions and applications, which leads to System F), since there is nothing in the syntax of the λ-calculus to represent these substitutions.

In the $\mathcal{X}$-calculus, there is a construct already present which can be seen to represent substitution. The cut $P\widehat{\alpha} \dagger \widehat{x}Q$ can, depending on the structure of P and Q, be seen to represent the substitution of P for the x's in Q, or symmetrically the substitution of Q for α's in P.

A subtle problem occurs in defining a shallow polymorphic type assignment, which motivates a relaxation of Definition 6 to allow multiple statements in a context with the same subject. The main reason for this is in the manipulation of quantified types, when we wish to take several instances of a type in the same derivation. It should be noted that in a sequent calculus setting, instances are taken on the same side of the sequent as the quantified type appeared (see the $\forall\mathcal{L}$ rule of Definition 23 below). We wish many such instances to be available (to make full use of the polymorphism in the system), and this causes us a difficulty, since all must be types for the same connector. For example the (*med*)-rule

$$(\textit{med}) : \frac{P \vdots \Gamma \vdash \alpha{:}A, \Delta \quad Q \vdots \Gamma, x{:}B \vdash \Delta}{P\widehat{\alpha}\,[y]\,\widehat{x}Q \vdots \Gamma, y{:}A{\rightarrow}B \vdash \Delta}$$

(which adds a type for y to the context Γ) would be expressed awkwardly: assume $y{:}C$ already occurs in Γ, then, given the polymorphic character of types, we can accept that $A{\rightarrow}B$ and C are different, as long as they are all instances of the same quantified type. In other words, we can assume that $y{:}\forall\varphi.D \in \Gamma$, and ask that $A{\rightarrow}B$ can be obtained from D by instantiation. This would give a complicated side-condition to the rule.

Instead, we choose to relax Definition 6, in that we now allow multiple statements in a context with the same subject. However, in order to retain soundness, we insist that whenever the rules (*exp*), (*med*) and (*cut*) are employed, the connectors mentioned in the top line of the rule (which are bound in the construction of the respective terms) have a unique statement in the rule. This enforces that all the types for a connector disappear from the contexts when the connector is bound. We also insist that a derivation is not complete unless the subjects of the statements in the final sequent are unique (so the relaxation is only usable temporarily within a derivation). As a consequence of these restrictions, if several statements with the same subject (but different types) are used in a derivation, it will be necessary for the $\forall$ rules to be applied until the types of these statements match, and they are contracted into a single statement. Until this takes place, it will be impossible to either bind the connective concerned, or complete the derivation.

Definition 23 (Shallow Polymorphic Type Assignment for $\mathcal{X}$). The shallow polymorphic type assignment for $\mathcal{X}$ is defined by the following rules (where ψ represents a generic type of Definition 21):

$$(\textit{cap}) : \frac{}{\langle y.\alpha\rangle \vdots \Gamma, y{:}\psi \vdash_{\mathrm{SP}} \alpha{:}\psi, \Delta} \qquad (\textit{med}) : \frac{P \vdots \Gamma \vdash_{\mathrm{SP}} \alpha{:}A, \Delta \quad Q \vdots \Gamma, x{:}B \vdash \Delta}{P\widehat{\alpha}\,[y]\,\widehat{x}Q \vdots \Gamma, y{:}A{\rightarrow}B \vdash_{\mathrm{SP}} \Delta}\,(^2)$$

$$(\textit{exp}) : \frac{P \vdots \Gamma, x{:}A \vdash_{\mathrm{SP}} \alpha{:}B, \Delta}{\widehat{x}P\widehat{\alpha}{\cdot}\beta \vdots \Gamma \vdash_{\mathrm{SP}} \beta{:}A{\rightarrow}B, \Delta}\,(^1) \qquad (\textit{cut}) : \frac{P \vdots \Gamma \vdash_{\mathrm{SP}} \alpha{:}\psi, \Delta \quad Q \vdots \Gamma, x{:}\psi \vdash_{\mathrm{SP}} \Delta}{P\widehat{\alpha} \dagger \widehat{x}Q \vdots \Gamma \vdash_{\mathrm{SP}} \Delta}\,(^3)$$

$$(\forall\mathcal{L}) : \frac{P \vdots \Gamma, x{:}\psi[B/X] \vdash_{\mathrm{SP}} \Delta}{P \vdots \Gamma, x{:}\forall X.\psi \vdash_{\mathrm{SP}} \Delta} \qquad (\forall\mathcal{R}) : \frac{P \vdots \Gamma \vdash_{\mathrm{SP}} \alpha{:}\psi, \Delta}{P \vdots \Gamma \vdash_{\mathrm{SP}} \alpha{:}\forall X.\psi[X/\varphi], \Delta}\,(^4)$$

[1]: if $x \notin \Gamma$ and $\alpha \notin \Delta$. [2,3]: if $x \notin \Gamma$ and $\alpha \notin \Delta$. [4]: if φ does not occur in Γ, Δ.

We include a notion of implicit contraction in the above rules (as for the type system presented in Section 3), so that if a derivation rule introduces a statement which was already present in the context, it is simply merged.

Notice that generic types are not used in the (*exp*) or (*med*) rules. This enforces the restriction that the $\forall$-symbol may not appear to the left of an '$\rightarrow$' in a type, and is similar to the way the ($\rightarrow I$) and ($\rightarrow E$) rules are treated in ML.

We have the following result:

Theorem 24 (Witness Reduction). *If $P :\cdot\ \Gamma \vdash_{\text{SP}} \Delta$, and $P \rightarrow Q$, then $Q :\cdot\ \Gamma \vdash_{\text{SP}} \Delta$.*

Using our previous observation concerning the fact that `let` and a cut both explicitly represent a substitution, we define an encoding of the language of ML into $\mathcal{X}$.

Definition 25 (Encoding ML in $\mathcal{X}$).

$$\begin{aligned}
\llbracket x \rrbracket_{\alpha}^{\text{ML}} &= \langle x.\alpha \rangle \\
\llbracket \lambda x.M \rrbracket_{\alpha}^{\text{ML}} &= \widehat{x} \llbracket M \rrbracket_{\beta}^{\text{ML}} \widehat{\beta} \cdot \alpha \\
\llbracket MN \rrbracket_{\alpha}^{\text{ML}} &= \llbracket M \rrbracket_{\beta}^{\text{ML}} \widehat{\beta} \dagger \widehat{y} (\llbracket N \rrbracket_{\gamma}^{\text{ML}} \widehat{\gamma}\ [y]\ \widehat{z} \langle z.\alpha \rangle) \\
\llbracket \texttt{let}\ x = M\ \texttt{in}\ N \rrbracket_{\alpha}^{\text{ML}} &= \llbracket M \rrbracket_{\beta}^{\text{ML}} \widehat{\beta} \dagger \widehat{x} \llbracket N \rrbracket_{\alpha}^{\text{ML}}
\end{aligned}$$

where y, z, β, γ are fresh connectors.

We have the following results for our encoding:

Theorem 26. *1.* $\llbracket M \rrbracket_{\beta}^{\text{ML}} \widehat{\beta} \dagger \widehat{x} \llbracket N \rrbracket_{\alpha}^{\text{ML}} \rightarrow \llbracket (N[M/x]) \rrbracket_{\alpha}^{\text{ML}}$.
2. If $M \rightarrow_{\text{ML}} N$ *then* $\llbracket M \rrbracket_{\alpha}^{\text{ML}} \rightarrow \llbracket N \rrbracket_{\alpha}^{\text{ML}}$.
3. If $\Gamma \vdash_{\text{ML}} M : \psi$ *then* $\llbracket M \rrbracket_{\beta}^{\text{ML}} :\cdot\ \Gamma \vdash_{\text{SP}} \beta{:}\psi$.

In fact, the converse of part 3 also holds if we restrict the right-context in our $\mathcal{X}$ typing judgement to contain only a statement for β (any other information would be redundant since β is the only free plug in such a term). This implies that the possible typings for M in ML and $\llbracket M \rrbracket_{\beta}^{\text{ML}}$ in shallow-polymorphic $\mathcal{X}$ are essentially the same. Since Wells proves in [12] that ML does not in general have principal typings (i.e. when the basis of assumptions is unspecified, there is no pair of basis and type which represents all other possible typings), this immediately implies that the same is the case of our shallow polymorphic version of $\mathcal{X}$.

On the other hand, it is well known that a notion of principal types for ML terms exists (as presented by Milner), with respect to a fixed basis Γ. We can define principal typings in our shallow polymorphic version of $\mathcal{X}$, with respect to a given context $\langle \Gamma; \Delta \rangle$ which gives a type to the free connectors in a term. Notice that such a context provides types for the outputs as well as the inputs.

We define an algorithm, based on the $\mathcal{W}$ algorithm of [1], which takes as input an $\mathcal{X}$-term and a context $\langle \Gamma; \Delta \rangle$, and produces as output a substitution S, giving the most general solution to the problem of typing the term with a (substitution) instance of $\langle \Gamma; \Delta \rangle$. We require that no types in Δ contain the $\forall$ symbol - the intention is that Γ provides any known licence to use polymorphism in the type search. In defining this algorithm (which we will name $\mathcal{W}^{\mathcal{X}}$), we require the following definition.

Definition 27 (∀-closure). The ∀-*closure* of type ψ with respect to a context $\langle \Gamma; \Delta \rangle$, is defined by: ∀-*closure* $\psi\ \langle \Gamma; \Delta \rangle = \forall X_1 \ldots \forall X_n.(\psi[X_i/\varphi_i])$ where $\varphi_1, \ldots, \varphi_n$ are the atomic types occurring in ψ but not in $\langle \Gamma; \Delta \rangle$.

We are now in a position to define our type-inference algorithm.

Definition 28 ($\mathcal{W}^{\mathcal{X}}$). The procedure $\mathcal{W}^{\mathcal{X}} :: \langle \mathcal{X}, \langle \Gamma; \Delta \rangle \rangle \to \mathcal{S}$ is defined by:

$$\mathcal{W}^{\mathcal{X}}(\langle x.\alpha \rangle, \langle \Gamma; \Delta \rangle) = S$$
where $A = \textit{instance}\ x\ \Gamma$
$B = \textit{instance}\ \alpha\ \Delta$
$S = \textit{unify}\ A\ B$

$$\mathcal{W}^{\mathcal{X}}(\widehat{x}P\widehat{\alpha}\cdot\beta, \langle \Gamma; \Delta \rangle) = S_2 \circ S_1$$
where $\varphi_1 = \textit{fresh}$
$\varphi_2 = \textit{fresh}$
$S_1 = \mathcal{W}^{\mathcal{X}}(P, \langle \Gamma \cup x{:}\varphi_1; \Delta \cup \alpha{:}\varphi_2 \rangle)$
$A = (S_1\ \varphi_1)$
$B = (S_1\ \varphi_2)$
$C = \textit{instance}\ \beta\ (S_1\ \Delta)$
$S_2 = \textit{unify}\ C\ A{\to}B$

$$\mathcal{W}^{\mathcal{X}}(P\widehat{\alpha}\ [y]\ \widehat{x}Q, \langle \Gamma; \Delta \rangle) = S_3 \circ S_2 \circ S_1$$
where $\varphi_1 = \textit{fresh}$
$\varphi_2 = \textit{fresh}$
$S_1 = \mathcal{W}^{\mathcal{X}}(P, \langle \Gamma; \Delta \cup \alpha{:}\varphi_1 \rangle)$
$S_2 = \mathcal{W}^{\mathcal{X}}(Q, (S_1\ \langle \Gamma \cup x{:}\varphi_2; \Delta \rangle))$
$A = (S_2 \circ S_1\ \varphi_1)$
$B = (S_2 \circ S_1\ \varphi_2)$
$C = \textit{instance}\ y\ (S_2 \circ S_1\ \Gamma)$
$S_3 = \textit{unify}\ C\ A{\to}B$

$$\mathcal{W}^{\mathcal{X}}(P\widehat{\alpha}\,\dagger\,\widehat{x}Q, \langle \Gamma; \Delta \rangle) = S_2 \circ S_1$$
where $\varphi = \textit{fresh}$
$S_1 = \mathcal{W}^{\mathcal{X}}(P, \langle \Gamma; \Delta \cup \alpha{:}\varphi \rangle)$
$\psi = \forall\textit{-closure}\ (S_1\ \varphi)\ (S_1\ \langle \Gamma; \Delta \rangle)$
$S_2 = \mathcal{W}^{\mathcal{X}}(Q, \langle (S_1\ \Gamma) \cup x{:}\psi; (S_1\ \Delta) \rangle)$

where *instance* is a mapping that takes the type associated to the given connective in the given context and replaces all ∀-bound type-variables by fresh atomic types. Note that since we prohibit Δ from containing the ∀ symbol, our uses of *instance* on a plug α merely extract the type for α from the context.

In order to reason about this context being truly principal, we need a notion of 'more general' for quantified types.

Definition 29 (Generic Instance). A type scheme $\psi = \forall X_1 \ldots \forall X_m.A$ has a *generic instance* $\psi' = \forall Y_1 \ldots \forall Y_n.A'$ if there exists a type B such that:

1. There exist types $B_1, \ldots, B_m$ with $B = A[B_i/X_i]$.
2. There exist atomic types $\varphi_1, \ldots, \varphi_n$ such that $A' = B[Y_i/\varphi_i]$, and the φ_i are not free in ψ.

We write $\psi' \preceq \psi$ in this case, read "ψ' is a generic instance of ψ".
We extend this notion to contexts, by defining $\preceq$ on contexts to be the least preorder such that:

$$\psi' \preceq \psi \Rightarrow \langle \Gamma; \Delta, \alpha{:}\psi' \rangle \preceq \langle \Gamma; \Delta, \alpha{:}\psi \rangle$$
$$\psi' \preceq \psi \Rightarrow \langle \Gamma, x{:}\psi; \Delta \rangle \preceq \langle \Gamma, x{:}\psi'; \Delta \rangle$$

∀-closure is extended to right-contexts by taking the closure of each statement.

Definition 30 (∀-closure for Contexts). We define the closure of a context $\langle \Gamma; \Delta \rangle$ by

$$\begin{array}{c} \textit{closure}\ \langle \Gamma; \Delta \rangle = \langle \Gamma; \Delta' \rangle \\ \text{where}\ \ \Delta' = \{\alpha{:}\psi' \mid \alpha{:}\psi \in \Delta \text{ and } \psi' = \forall\textit{-closure}\ \psi\ \langle \Gamma; \Delta \backslash \alpha \rangle\} \end{array}$$

We can now define our notion of principal contexts for shallow polymorphic $\mathcal{X}$, by running the algorithm $\mathcal{W}^{\mathcal{X}}$, applying the resulting substitution, and taking the closure of the result.

Definition 31 (Principal Contexts for Shallow Polymorphic $\mathcal{X}$). Given any $\mathcal{X}$-term P, and a context $\langle \Gamma; \Delta \rangle$ which provides types for exactly the free connectors in P, we define the *shallow polymorphic principal context* for P with respect to a $\langle \Gamma; \Delta \rangle$ by:

$$\begin{array}{rl} \textit{sppc}\,(P, \langle \Gamma; \Delta \rangle) = & \textit{closure}\,((S\ \langle \Gamma; \Delta \rangle)) \\ & \text{where } S = \mathcal{W}^{\mathcal{X}}(P, \langle \Gamma; \Delta \rangle) \end{array}$$

Notice that *sppc* may or may not succeed, depending on whether the call to $\mathcal{W}^{\mathcal{X}}$ does. The following result justifies this definition.

Theorem 32 (Soundness and Completeness of *sppc*). *Given an $\mathcal{X}$-term P and an initial context $\langle \Gamma_1; \Delta_1 \rangle$,*

1. *If* sppc $(P, \langle \Gamma_1; \Delta_1 \rangle)$ *succeeds and* $\langle \Gamma; \Delta \rangle =$ sppc $(P, \langle \Gamma_1; \Delta_1 \rangle)$ *then* $P :\cdot\ \Gamma \vdash_{\text{SP}} \Delta$.
2. *If $\langle \Gamma_2; \Delta_2 \rangle$ is an instance of $\langle \Gamma_1; \Delta_1 \rangle$ (i.e. can by obtained from the latter by substitution), and is such that $P :\cdot\ \Gamma_2 \vdash_{\text{SP}} \Delta_2$ then:*
   (a) sppc $(P, \langle \Gamma_1; \Delta_1 \rangle)$ *succeeds.*
   (b) If $\langle \Gamma; \Delta \rangle =$ sppc $(P, \langle \Gamma_1; \Delta_1 \rangle)$ *then there is a substitution S such that $\langle \Gamma_2; \Delta_2 \rangle \preceq (S\ \langle \Gamma; \Delta \rangle)$.*

In summary, we have shown that in our shallow polymorphic formulation of $\mathcal{X}$ we can faithfully simulate ML reductions, we have decideable type-assignment (at least as strong as that of ML) and principal typings with respect to a fixed basis of assumptions (in the style of $\mathcal{W}$). While we retain all these useful properties, our calculus is more general than ML because of its basis on Classical Sequent Calculus. We can give typeable programs which have no analogue in ML (for example, we can give a program that has Pierce's Law as a type), and can treat terms with multiple outputs. Furthermore, since cut elimination is well-known to be non-confluent, we can simulate non-determinism, a feature not present in ML. The precise computational content of these various extensions is the subject of ongoing research.

7 Future Work

We are interested in investigating further useful programming features in the context of $\mathcal{X}$, like recursion.

At present, we only model polymorphism in the same style that ML does, in generalising an output on the left of a cut and then taking instances for the various inputs on

the right. Since $\mathcal{X}$ is a very symmetric calculus, a natural idea to explore is the use of polymorphism in the opposite direction: to generalise the input of the right-hand term, and take instances for the (possibly several) outputs on the left. In logical terms, this can be seen as introducing the $\exists$ connective to the system, and investigations into a system based on this observation are ongoing. While a 'dual' notion of polymorphism is fairly straightforward to define (where we allow $\exists$ in the type system instead of $\forall$), it is more complicated to allow both kinds of polymorphism at once. This is a promising line of future work, and is expected to yield a very powerful decideable type system for $\mathcal{X}$.

Acknowledgements We would like to thank Jayshan Raghunandan, Pierre Lescanne, Dragisa Zunic, Dorian Gaertner and the (anonymous) referees for their generous feedback and discussions on the subject of this paper.

References

1. L. Damas and R. Milner. Principal type-schemes for functional programs. In *Proceedings 9$^{\text{th}}$ ACM Symposium on Principles of Programming Languages*, pages 207–212, 1982.
2. Gerhard Gentzen. Untersuchungen über das logische schließen. *Mathematische Zeitschrift*, 39:176–210, 405–431, 1935.
3. J.-Y. Girard, Y. Lafont, and P. Taylor. *Proofs and Types*. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1989.
4. Jean-Yves Girard. Une extension de l'interprétation de Gödel à l'analyse et son application à l'élimination des coupures dans l'analyse et la théorie des types. In *Proceedings of the Second Scandinavian Logic Symposium (Univ. Oslo, Oslo, 1970)*, volume 63, pages 63–92. North-Holland, 1971.
5. Stéphane Lengrand. Call-by-value, call-by-name, and strong normalization for the classical sequent calculus. In Bernhard Gramlich and Salvador Lucas, editors, *Electronic Notes in Theoretical Computer Science*, volume 86. Elsevier, 2003.
6. R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.
7. M. Parigot. An algorithmic interpretation of classical natural deduction. In *Proc. of Int. Conf. on Logic Programming and Automated Reasoning, LPAR'92*, volume 624 of *Lecture Notes in Computer Science*, pages 190–201. Springer-Verlag, 1992.
8. J.C. Reynolds. Towards a Theory of Type Structures. In B. Robinet, editor, *Proceedings of 'Colloque sur la Programmation'*, Paris, France, volume 19 of *Lecture Notes in Computer Science*, pages 408–425. Springer-Verlag, 1974.
9. Christian Urban. *Classical Logic and Computation*. PhD thesis, University of Cambridge, October 2000.
10. S. van Bakel, S. Lengrand, and P. Lescanne. The language $\mathcal{X}$: circuits, computations and classical logic. In *Proc. 10th Italian Conf. on Theoretical Computer Science (ICTCS'05)*, volume 3701 of *Lecture Notes in Computer Science*, pages 66–80. Springer-Verlag, 2005.
11. S. van Bakel, J. Raghunandan, and A. Summers. Term Graphs and Principal Types for $\mathcal{X}$. Unpublished, `http://www.doc.ic.ac.uk/~svb/Research`, 2005.
12. J. B. Wells. The essence of principal typings. In *Proc. 29th Int'l Coll. Automata, Languages, and Programming*, volume 2380 of *LNCS*, pages 913–925. Springer-Verlag, 2002.

A New Reduction Rules for $\mathcal{X}^{\forall}$

The new key logical rule which is introduced, is the following:

$$\begin{array}{rl}(\textit{poly}): & (P\widehat{\alpha}^{A}\!\!\underset{\varphi}{\triangleleft}\beta^{\forall X.A[X/\varphi]})\widehat{\beta}^{\forall X.A[X/\varphi]} \dagger \widehat{y}^{\forall X.A[X/\varphi]}(y^{\forall X.A[X/\varphi]}\!\!\underset{B}{\triangleright}\widehat{x}^{A[B/\varphi]}Q) \\ \rightarrow & (P[B/\varphi])\widehat{\alpha}^{A[B/\varphi]} \dagger \widehat{x}^{A[B/\varphi]}Q \quad \textit{if } \beta \notin \mathit{fp}(P), y \notin \mathit{fs}(Q)\end{array}$$

This rule is complex, more so than the existing logical rules, because of the need to account for the instantiation of the polymorphic type involved. It is necessary for a type substitution to be made throughout P, in order for the type of the output α to be able to agree with the type of the input x. This is where the instantiation happens - once a copy of P has been propagated to meet a single term Q with which it can communicate, an appropriate instance of P is taken. Of course, many such copies of P may have been made by this stage, and it is the polymorphic type for α which allows each of these to be instantiated in different (and possibly non-unifiable) ways.

We omit the type information from the following rules for brevity. However, except in the case of the *poly* rule already described, the types are not necessary for understanding the operation of the rules.

Logical Rules

$$\begin{array}{rl}(\textit{gen}): & (P\widehat{\beta}\triangleleft\alpha)\widehat{\alpha} \dagger \widehat{x}\langle x.\gamma\rangle \rightarrow P\widehat{\beta}\triangleleft\gamma \quad \alpha \notin \mathit{fp}(P) \\ (\textit{inst}): & \langle y.\alpha\rangle\widehat{\alpha} \dagger \widehat{x}(x\triangleright\widehat{z}P) \rightarrow y\triangleright\widehat{z}P \quad x \notin \mathit{fs}(P) \\ (\textit{poly}): & (P\widehat{\beta}\triangleleft\alpha)\widehat{\alpha} \dagger \widehat{x}(x\triangleright\widehat{y}Q) \rightarrow P\widehat{\beta} \dagger \widehat{y}Q \quad \alpha \notin \mathit{fp}(P), x \notin \mathit{fs}(Q)\end{array}$$

Activation Rules

$$\begin{array}{rl}(\textit{act}\text{-L}): & P\widehat{\alpha} \dagger \widehat{x}Q \rightarrow P\widehat{\alpha} \not\leftarrow \widehat{x}Q \textit{ if } P \textit{ does not introduce } \alpha \\ (\textit{act}\text{-R}): & P\widehat{\alpha} \dagger \widehat{x}Q \rightarrow P\widehat{\alpha} \not\rightarrow \widehat{x}Q \textit{ if } Q \textit{ does not introduce } x\end{array}$$

where:

P *introduces* x**:** $P = Q\widehat{\beta}\,[x]\,\widehat{y}R$ and $x \notin \mathit{fs}(Q,R)$, or $P = x\triangleright\widehat{y}Q$ and $x \notin \mathit{fs}(Q)$, or $P = \langle x.\alpha\rangle$.

P *introduces* α**:** $P = \widehat{x}Q\widehat{\beta}\cdot\alpha$ and $\alpha \notin \mathit{fp}(Q)$, or $P = Q\widehat{\beta}\triangleleft\alpha$ and $\alpha \notin \mathit{fp}(Q)$, or $P = \langle x.\alpha\rangle$.

Left Propagation

$$\begin{array}{rll}(\not\leftarrow\textit{gen-outs}): & (Q\widehat{\beta}\triangleleft\alpha)\widehat{\alpha} \not\leftarrow \widehat{x}P \rightarrow ((Q\widehat{\alpha} \not\leftarrow \widehat{x}P)\widehat{\beta}\triangleleft\gamma)\widehat{\gamma} \dagger \widehat{x}P, & \gamma \textit{ fresh} \\ (\not\leftarrow\textit{gen-ins}): & (Q\widehat{\beta}\triangleleft\gamma)\widehat{\alpha} \not\leftarrow \widehat{x}P \rightarrow (Q\widehat{\alpha} \not\leftarrow \widehat{x}P)\widehat{\beta}\triangleleft\gamma, & \gamma \neq \alpha \\ (\not\leftarrow\textit{inst}): & (y\triangleright\widehat{z}Q)\widehat{\alpha} \not\leftarrow \widehat{x}P \rightarrow y\triangleright\widehat{z}(Q\widehat{\alpha} \not\leftarrow \widehat{x}P) & \end{array}$$

Right Propagation

$$\begin{array}{rll}(\not\rightarrow\textit{gen}): & P\widehat{\alpha} \not\rightarrow \widehat{x}(Q\widehat{\beta}\triangleleft\gamma) \rightarrow (P\widehat{\alpha} \not\rightarrow \widehat{x}Q)\widehat{\beta}\triangleleft\gamma & \\ (\not\rightarrow\textit{inst-outs}): & P\widehat{\alpha} \not\rightarrow \widehat{x}(x\triangleright\widehat{y}Q) \rightarrow P\widehat{\alpha} \dagger \widehat{z}(z\triangleright\widehat{y}(P\widehat{\alpha} \not\rightarrow \widehat{x}Q)), & z \textit{ fresh} \\ (\not\rightarrow\textit{inst-ins}): & P\widehat{\alpha} \not\rightarrow \widehat{x}(z\triangleright\widehat{y}R) \rightarrow z\triangleright\widehat{y}(P\widehat{\alpha} \not\rightarrow \widehat{x}R), & z \neq x\end{array}$$