Pure Type Systems in Rewriting Logic: Specifying Typed Higher-Order Languages in a First-Order Logical Framework

Mark-Oliver Stehr[1][*] and José Meseguer[2]

[1] Universität Hamburg
Fachbereich Informatik - TGI
22527 Hamburg, Germany
`stehr@informatik.uni-hamburg.de`
[2] University of Illinois at Urbana-Champaign
Computer Science Department
Urbana, IL 61801, USA
`meseguer@cs.uiuc.edu`

Dedicated to the memory of Ole-Johan Dahl

Abstract. The logical and operational aspects of *rewriting logic* as a logical framework are tested and illustrated in detail by representing *pure type systems* as object logics. More precisely, we apply *membership equational logic*, the equational sublogic of rewriting logic, to specify pure type systems as they can be found in the literature and also a new variant of pure type systems with explicit names that solves the problems with closure under α-conversion in a very satisfactory way. Furthermore, we use rewriting logic itself to give a formal operational description of type checking, that directly serves as an efficient type checking algorithm. The work reported here is part of a more ambitious project concerned with the development of the *open calculus of constructions*, an equational extension of the calculus of constructions that incorporates rewriting logic as a computational sublanguage.

This paper is a detailed study on the ease and naturalness with which a family of higher-order formal systems, namely *pure type systems (PTSs)* [6,50], can be represented in the first-order logical framework of rewriting logic [36]. PTSs generalize the λ-cube [1], which already contains important calculi like $\lambda{\rightarrow}$ [12], the systems F [23,43] and Fω [23], a system λP close to the logical framework LF [24], and their combination, the calculus of constructions CC [16]. PTSs are considered to be of key importance, since their generality and simplicity makes them an ideal basis for representing higher-order logics, either via the propositions-as-types interpretation [21], or via their use as a higher-order logical framework in the spirit of LF [24,20] or Isabelle [39].

---

[*] Currently visiting University of Illinois at Urbana-Champaign, Computer Science Department Urbana, IL 61801, USA, e-mail: `stehr@cs.uiuc.edu`

Exploiting the fact that *rewriting logic (RWL)* and its *membership equational sublogic (MEL)* [10] have initial and free models, we can define the representation of PTSs as a *parameterized theory* in the framework logic; that is, we define in a single parametric way all the representations for the infinite family of PTSs. Furthermore, the representational versatility of RWL, and of MEL, are also exercised by considering four different representations of PTSs at different levels of abstraction, from a more abstract textbook version in which terms are identified up to α-conversion, to a more concrete version with a calculus of names and explicit substitutions, and with a type checking inference system that can in fact be used as a reasonably efficient implementation of PTSs by executing the representation in the Maude language [13,14].

This case study complements earlier work [31,32], showing that rewriting logic has good properties as a logical framework to represent a wide range of logics, including linear logic, Horn logic with equality, first-order logic, modal logics, sequent-based presentations of logics, and so on. In particular, representations for the λ-calculus, and for binders and quantifiers have already been studied in [32], but this is the first systematic study on the representation of *typed* higher-order systems. One property shared by all the above representations, including all those discussed in this paper, is that what might be called the *representational distance* between the logic being formalized and its rewriting logic representation is virtually zero. That is, both the syntax and the inference system of the object logic are directly and faithfully mirrored by the representation. This is an important advantage both in terms of understandability of the representations, and in making the use of encoding and decoding functions unnecessary in a so-called adequacy proof.

Besides the directness and naturalness with which logics can be represented in a framework logic, another important quality of a logical framework is the *scope* of its applicability; that is, the class of logics for which faithful representations preserving relevant structure can be defined. Typically, we want representations that both preserve and reflect provability; that is, something is a theorem in the original logic if and only if its translation can be proved in the framework's representation of the logic. Such mappings go under different names and differ in their generality; in higher-order logical frameworks representations are typically required to be *adequate* mappings [20], and in the theory of general logics more liberal, namely *conservative* mappings of entailment systems [35], are studied. In this paper, we we further generalize conservative mappings to the notion of a sound and complete full *correspondence of sentences* between two entailment systems. In fact, all the representations of PTSs that we consider are correspondences of this kind. Sound and complete full correspondences are systematically used not only to state the correctness of the representations of PTSs at different levels of abstraction, but also to relate those different levels of abstraction, showing that the more concrete representations correctly implement their more abstract counterparts.

A systematic way of comparing the scopes of two logical frameworks $\mathcal{F}$ and $\mathcal{G}$ is to exhibit a sound and complete full correspondence $\mathcal{F} \rightsquigarrow \mathcal{G}$, representing

$\mathcal{F}$ in $\mathcal{G}$. In view of this quite general concept, it is important to add that the *representational distance*, which we informally define as the complexity of this correspondence, is an important measure of the quality of the representation. Since such correspondences form a category, and therefore compose, this then shows that the scope of $\mathcal{G}$ is *at least as general* as that of $\mathcal{F}$. Since PTSs include the system λP, close to the logical framework LF, and the calculus of constructions CC, the results in this paper indicate that the scope of rewriting logic is at least as general as that of those logics. Furthermore, since there are no adequate mappings from linear logic to LF in the sense of [20], but there is a conservative mapping of logics from linear logic to rewriting logic [32], this seems to indicate that the LF methodology together with its rather restrictive notion of adequate mapping is more specialized than the rewriting logic approach.

In this paper we will be concerned with PTSs as formal systems represented inside informal set theory, or inside another formal system such as rewriting logic or its membership equational sublogic. For formal systems in general, and for PTSs in particular, there is not a single canonical presentation. Instead each presentation is tailored for specific purposes. For example, there are different formulations of PTSs with different sets of rules, but the same sets, or related sets, of derivable sentences. Furthermore, presentations can be more or less abstract, e.g. concerning the treatment of names, or concerning the degree of operationality. It is needless to say that the use of some general terminology is highly desirable in this situation to deal with these issues in a systematic way. To this end, we follow the general logics methodology [35] to use an abstract logical metatheory, which is concerned with formal systems *and* their relationships, together with a particular formal system as a logical framework, namely rewriting logic. Regarding general logics terminology, we furthermore found that the notion of correspondences between sentences that generalizes the idea of maps of entailment systems is a simple a useful tool to structure our results.

In summary, we think that, besides the more technical contributions to PTSs discussed in Section 5, the key contributions of this paper are threefold. First, as already mentioned, the expressiveness of RWL and its MEL sublogic as logical frameworks is tested and demonstrated by showing how a well-known family of typed higher-order logics, that are themselves frequently used for logical framework purposes, are naturally represented. But this brings along with it a second important consequence: our representation maps *suggest fruitful generalizations of PTSs*, in which higher-order reasoning is seamlessly integrated with equational and rewriting logic reasoning. The need for such multiparadigm integrations of equational logic and type theory is clearly recognized by many researchers, because of the restrictive notions of equality and computation in traditional λ-calculi. Specifically, as further explained in Section 5.1, an integration of a typed higher-order λ-calculus with MEL and RWL, namely the *open calculus of constructions (OCC)* [48], has been developed by the first author as a natural extension and generalization of the ideas presented here. It is worth pointing out that the *executability* of the representation maps has made possible the development of a prototype for OCC in Maude which has been used in a wide range

of examples concerned with programming, specification and interactive theorem proving [48]. A third and final consideration is that our representation maps have another important advantage: since MEL and RWL theories have *initial models*, theories with initial semantics can be endowed with *inductive reasoning principles*. It is indeed such an initial (or free extension) semantics that is used in all our representations of PTSs. This means that we can not only *simulate* PTSs in MEL or RWL using our representations, but we can also *reason about* the metalogical properties of such systems using induction. Different approaches to metalogical reasoning are touched upon in Section 5.2. These include the use of a higher-order logic such as OCC as a metalogic to reason about formalisms represented in its MEL or RWL sublogic, and the use of a reflective metalogical framework such as RWL, which is discussed at greater length in [4].

1   Preliminaries

1.1   Entailment Systems

In the following sections we are concerned with a variety of different interrelated formal systems that can all be viewed as entailment systems, a notion defined in [35] as a main component of general logics. Since the notion of entailment system is more general than what is needed for the purposes of the present paper, we work with *unary* entailment systems over a fixed signature. A *unary entailment system* $(\mathbf{Sen}, \vdash)$ is a set of *sentences* $\mathbf{Sen}$, together with a unary *entailment predicate* $\vdash \subseteq \mathbf{Sen}$.

In [35] maps between sentences are used to relate different logics. Here we introduce a more general notion of morphism, namely a correspondence between sentences of different entailment systems. Let $(\mathbf{Sen}, \vdash)$, $(\mathbf{Sen}', \vdash')$ be unary entailment systems. A *correspondence of sentences* between $(\mathbf{Sen}, \vdash)$ and $(\mathbf{Sen}', \vdash')$ is a relation $\curvearrowright\ \subseteq \mathbf{Sen} \times \mathbf{Sen}'$. Given such a correspondence $\curvearrowright$, we say that $\curvearrowright$ is *sound* iff for all $\phi \curvearrowright \phi'$, $\vdash' \phi'$ implies $\vdash \phi$. Similarly, we say that $\curvearrowright$ is *complete* iff for all $\phi \curvearrowright \phi'$, $\vdash \phi$ implies $\vdash' \phi'$. Moreover, $\curvearrowright$ is called *total* iff for each $\phi' \in \mathbf{Sen}'$ there is a ϕ such that $\phi \curvearrowright \phi'$. Correspondences compose in the obvious relational way, giving rise to a category $\mathbf{CEnt}$. Often a correspondence of sentences $\curvearrowright\ \subseteq \mathbf{Sen} \times \mathbf{Sen}'$ takes the form of a function $\alpha : \mathbf{Sen} \longrightarrow \mathbf{Sen}'$, but in principle it can also take the form of a function $\alpha : \mathbf{Sen}' \longrightarrow \mathbf{Sen}$ in the opposite direction. Indeed, a *map of entailment systems* $\alpha : \mathbf{Sen}' \longrightarrow \mathbf{Sen}$ in the sense of [35] gives rise to a sound correspondence $\alpha^{-1} \subseteq \mathbf{Sen} \times \mathbf{Sen}'$, and if α is a *conservative* map then α^{-1} is a sound and complete correspondence.

1.2   Rewriting Logic and Membership Equational Logic

A rewrite theory is a triple $\mathcal{R} = (\Sigma, E, R)$, with Σ a signature of function symbols, E a set of equations, and R a set of (possibly conditional) rewrite rules of the form $t \longrightarrow t'$ (with t and t' Σ-terms) which are applied *modulo* the equations E. *Rewriting logic* (RWL) has then a deductive system to infer all possible

rewrites provable in a given rewrite theory [36]. Since an equational theory (Σ, E) can be regarded as a rewrite theory $(\Sigma, E, \emptyset)$ with no rules, equational logic is a sublogic of rewriting logic. In fact, rewriting logic is parameterized by the choice of its underlying equational logic, which can be unsorted, many-sorted, and so on.

In this paper, and in the design of the Maude language, we have chosen *membership equational logic* (MEL) [37,11] as the underlying equational logic. Membership equational logic is quite expressive. It has sorts, subsorts, overloading of function symbols, and can express partiality very directly by defining membership in a sort by means of equational conditions. The atomic sentences are equalities $t = t'$ and memberships $t : s$, with s a sort, and general sentences are Horn clauses on the atoms. Both membership equational logic and rewriting logic have initial and free models [36,37]. We denote by **MEL** $\subseteq$ **RWL** the sublogic inclusion from membership equational logic into rewriting logic.

Logics can be naturally represented as rewrite theories by defining the formulas, or other proof-theoretic structures such as sequents, as elements of appropriate sorts in an abstract data type specified by an equational theory (Σ, E). Then, each inference rule in the logic can be axiomatized as a, possibly conditional, rewrite rule, giving rise to a representation as a rewrite theory (Σ, E, R). Alternatively, we can exploit the rich sort structure of membership equational logic to represent the inference rules of a logic not as rewrite rules, but as Horn clauses H expressing membership in an adequate sort of derivable sentences, leading to a membership equational logic representation of the form $(\Sigma, E \cup H)$. In this paper we will use both forms of representations for different versions of PTSs.

2   Overview and Main Results

In Section 3 we show how the definition of PTSs can be formalized in MEL. The approach we use is not only less specialized than the one used in a higher-order logical framework like LF [24] or Isabelle [39], but it has also more explanatory power, since we explain higher-order calculi in terms of a first-order system with a simpler semantics, and our representations have initial (or, more generally, free extension) models supporting metalogical inductive reasoning about the PTSs thus represented.

In order to make the specification of PTSs more concrete, we introduce in Section 3.5 the notion of *uniform pure type systems (UPTSs)* [46,49,48], that do not abstract from the treatment of names but use CINNI [47,48], a generic the first-order calculus of names and substitutions. UPTSs solve the problem of closure under α-conversion, that has been discussed by Pollack in [40], in a simple and elegant way. Again, a MEL specification of UPTSs is given that directly formalizes the informal definition.

As an intermediate step we employ *optimized UPTSs (OUPTSs)* which are introduced in Section 3.6. OUPTSs have an explicit judgement for well-typed contexts, and can be seen as a refinement of UPTSs towards a more efficient implementation of type checking.

Last but not least, we describe how the meta-operational view of an important class of OUPTSs, namely type checking and type inference, can be expressed as a transition system and can likewise be formalized in rewriting logic. The result of this formalization is an executable specification of *rewriting-based OUPTSs (ROUPTSs)* that is sound w.r.t. the logical specification given before in a very obvious way.

Formally, these different presentations of PTSs are families of unary entailment systems parameterized by *PTS signatures*. We use the notation $\mathbf{PTS}_S$, $\mathbf{UPTS}_S$, $\mathbf{OUPTS}_S$ and $\mathbf{ROUPTS}_S$ to denote the entailment systems of PTSs, UPTSs, OUPTSs, and ROUPTSs, respectively, associated with a PTS signature S.

For appropriate PTS signatures S we obtain a chain of sound and complete total correspondences

$$\mathbf{PTS}_S \curvearrowright \mathbf{UPTS}_S \curvearrowright \mathbf{OUPTS}_S \curvearrowright \mathbf{ROUPTS}_S.$$

Actually, we have two different kinds of connections between the first two entailment systems, leading to two different correspondences of the form $\mathbf{PTS}_S \curvearrowright \mathbf{UPTS}_S$. By composing three correspondences of the form above we finally arrive at a sound and complete total correspondence

$$\mathbf{PTS}_S \curvearrowright \mathbf{ROUPTS}_S$$

which shows the equivalence of the high-level specification of PTSs with the implementation of a type checker.

The deductive system of RWL induces a unary entailment system $\mathbf{RWL}$ with sentences of the form $\mathcal{R} \vdash \phi$, where $\mathcal{R}$ is a rewrite theory and ϕ is an equation, a membership or a rewrite. In this chapter we abstract from rewrite proofs, so that we use the term rewrite to refer to a sentence of the form $M \to M'$ and we define $\mathcal{R} \vdash M \to M'$ to be derivable iff $\mathcal{R} \vdash P : M \to M'$ is derivable for some rewrite proof P in the deductive system of RWL. Likewise, MEL induces a unary entailment system $\mathbf{MEL}$ obtained by restricting $\mathcal{R}$ to MEL theories and ϕ to equations or memberships.

The entailment systems $\mathbf{PTS}_S$, $\mathbf{UPTS}_S$, $\mathbf{OUPTS}_S$ and $\mathbf{ROUPTS}_S$ can be easily specified in membership equational logic or in rewriting logic. Specifically, we have the following sound and complete total correspondences:

$$\mathbf{PTS}_S \curvearrowright \mathbf{MEL}$$

$$\mathbf{UPTS}_S \curvearrowright \mathbf{MEL}$$

$$\mathbf{OUPTS}_S \curvearrowright \mathbf{MEL}$$

$$\mathbf{ROUPTS}_S \curvearrowright \mathbf{RWL}$$

In all cases the *representational distance* between the formal system and its representation is practically zero, that is, both the syntax and the inference system of each version of PTSs have direct and faithful representations in the framework logic.

The first correspondence is the representation of PTSs in MEL given in Section 3. Let $\overline{\mathbf{PTS}_S}$ be the MEL specification of $\mathbf{PTS}_S$. Then, for all PTS judgements ϕ of $\mathbf{PTS}_S$ and possible representations ϕ' of ϕ in MEL, the sentence $\overline{\mathbf{PTS}_S} \vdash \phi'$ is derivable in MEL iff the judgement ϕ is derivable in $\mathbf{PTS}_S$. This defines a sound and complete total correspondence of the form $\mathbf{PTS}_S \curvearrowright \mathbf{MEL}$. We are concerned with a correspondence rather than a function, due to the fact that PTSs abstract from names, but in the MEL representation names are part of the description of terms, although by adding appropriate equations an equivalent abstraction can be achieved in MEL at the semantic level.

In the remaining three systems $\mathbf{UPTS}_S$, $\mathbf{OUPTS}_S$, and $\mathbf{ROUPTS}_S$ we do not abstract from names. Hence, the three associated representational correspondences actually take the form of functions, i.e., with each judgement of the type system we can associate a unique sentence in MEL or RWL, respectively. For the presentation of PTSs we follow [52], which can be seen as an informal presentation of the machine-checked formalization [34].

3   The Metalogical View of PTSs

A *PTS signature* is a triple $(\mathcal{S}, \mathcal{A}, \mathcal{R})$ where $\mathcal{S}$ is a set of *sorts*, $\mathcal{A} \subseteq \mathcal{S} \times \mathcal{S}$ is the set of *axioms*, and $\mathcal{R} \subseteq \mathcal{S} \times \mathcal{S} \times \mathcal{S}$ is the set of *rules*. The sorts of a PTS signature are used as types of types and are therefore often referred to as *universes*. We use S to range over PTS signatures, and for the following we fix an arbitrary PTS signature S.

In PTSs there is no a priori distinction between terms and types. *PTS terms* are defined by the following syntax with binders:

$$X \,|\, (M\ N) \,|\, [X:A]M \,|\, \{X:A\}M \,|\, s$$

Here, and in the following, s ranges over $\mathcal{S}$; M, N, A, B, T range over terms; and X ranges over names. We should add that in $[X:A]M$ and $\{X:A\}M$ the name X is bound in M, and we assume that α-*convertible* terms, i.e. terms that are equal up to renaming of bound variables, are identified.

Formally this identification can be achieved by different means: the definition of PTS terms as equivalence classes modulo α-equivalence, or a representation based on de Bruijn indices are two possibilities. For the following it is important to keep in mind that the choice of particular names for bound variables is part of the informal notation (for readability) but is not reflected in PTS terms.

A *PTS context* is a list of *declarations*, each of the form $X:A$. A declaration $X:A$ *declares* a name X of type A. A context is *simple* if it declares each identifier at most once. In the following, Γ ranges over PTS contexts.

PTS typing judgements are of the form $\Gamma \vdash M:T$, and *derivability*, i.e. the set of *derivable typing judgements*, is defined by the formal system given by the following inference rules:

$$\frac{}{[] \vdash s_1 : s_2} \quad (s_1, s_2) \in \mathcal{A} \tag{Ax}$$

$$\frac{\Gamma \vdash A : s}{\Gamma, X : A \vdash X : A} \quad X \notin \Gamma \tag{Start}$$

$$\frac{\Gamma \vdash M : A \quad \Gamma \vdash B : s}{\Gamma, X : B \vdash M : A} \quad X \notin \Gamma \tag{Weak}$$

$$\frac{\Gamma \vdash A : s_1 \quad \Gamma, X : A \vdash B : s_2}{\Gamma \vdash \{X : A\}B : s_3} \quad (s_1, s_2, s_3) \in \mathcal{R} \tag{Pi}$$

$$\frac{\Gamma \vdash A : s_1 \quad \Gamma, X : A \vdash M : B \quad \Gamma, X : A \vdash B : s_2}{\Gamma \vdash [X : A]M : \{X : A\}B} \quad (s_1, s_2, s_3) \in \mathcal{R} \tag{Lda}$$

$$\frac{\Gamma \vdash M : \{X : A\}B \quad \Gamma \vdash N : A}{\Gamma \vdash (M \ N) : [X:=A]B} \tag{App}$$

$$\frac{\Gamma \vdash M : A \quad \Gamma \vdash B : s}{\Gamma \vdash M : B} \quad A \equiv_\beta B \tag{Conv}$$

Here we write $X \notin \Gamma$ iff there is no $X : A \in \Gamma$ for any A, and we denote by $[X := N]M$ the standard (capture-free) substitution of all free occurrences of X in M by N. In the last rule, $\equiv_\beta$ is the usual notion of β-convertibility, which contains α-convertibility (this is trivially satisfied in this presentation). Observe that the side conditions ensure that we can only derive *simple judgements*, i.e. judgements with simple contexts. We say that T is a *type* in the context Γ iff $T \in \mathcal{S}$ or $\Gamma \vdash T : s$ for some $s \in \mathcal{S}$. Furthermore, M is said to be an *element* of type T in the context Γ iff $\Gamma \vdash M : T$, in which case we also say that M is *well-typed* in Γ.

As an example, we can instantiate PTSs by

$$\mathcal{S} = \{\texttt{Prop}, \texttt{Type}\},$$
$$\mathcal{A} = \{(\texttt{Prop}, \texttt{Type})\},$$
$$\mathcal{R} = \{(\texttt{Prop}, \texttt{Prop}, \texttt{Prop}),$$
$$(\texttt{Prop}, \texttt{Type}, \texttt{Type}),$$
$$(\texttt{Type}, \texttt{Prop}, \texttt{Prop}),$$
$$(\texttt{Type}, \texttt{Type}, \texttt{Type})\}$$

to obtain the calculus of constructions.

This presentation of PTSs is rather abstract for two reasons: firstly, we are working modulo α-conversion, i.e., we identify α-convertible terms, and secondly, we are concerned with an inductive definition of a *set* of derivable judgements, but *not* with an *algorithm* to verify derivability of a given judgement.

Mathematically the abstract presentation has an important benefit: It allows us to reason about PTSs metalogically, without assuming anything about the concrete realization of names. This leads to very general results [1,51] and frees proofs from unnecessary technical details.

Closure under α-conversion is the property that derivability of $\Gamma \vdash M : A$ and $M \equiv_\alpha M'$ implies derivability of $\Gamma \vdash M' : A$. Of course, this property trivially holds for PTSs as presented above, since $\equiv_\alpha$ is the identity. To state a stronger property we extend α-conversion $\equiv_\alpha$ from terms to judgements such that $\Gamma \vdash M : A \equiv_\alpha \Gamma' \vdash M' : A'$ iff $\Gamma' \vdash M' : A'$ and $\Gamma \vdash M : A$ are equal up to consistent renaming of variables. Then we have the following

Lemma 31 (Strong Closure under α-Conversion for PTSs)
Let M, A, M', A' be PTS terms and Γ, Γ' be PTS contexts. If the PTS judgement $\Gamma \vdash M : A$ is derivable in **PTS**$_S$ and $\Gamma \vdash M : A \equiv_\alpha \Gamma' \vdash M' : A'$, then $\Gamma' \vdash M' : A'$ is derivable in **PTS**$_S$.

Proof Sketch.   By induction over derivations of $\Gamma \vdash M : A$.   □

The previous Lemma is equivalent to the statement that the following rule is admissible in PTSs:

$$\frac{\Gamma \vdash M : A}{\Gamma' \vdash M' : A'} \quad \text{if } \Gamma \vdash M : A \equiv_\alpha \Gamma' \vdash M' : A' \tag{Rename}$$

3.1   PTSs in Membership Equational Logic

In the following specifications, given in Maude syntax, we use the algebraic semantics of MEL for representing PTSs exactly as given above; a more operational version suited for use as an implementation is discussed in Section 4.2.

First, notice that we plan to describe not a single type system but the *infinite family* of PTSs parameterized by PTS signatures which define sorts, axioms and rules. All such PTS signatures can be formalized as models of a single parameter theory that can be specified in Maude as follows:

```
fth PTS-SIG is
sorts Sorts Axioms Axioms? Rules Rules? .
subsort Axioms < Axioms? .
subsort Rules < Rules? .
op (_,_) : Sorts Sorts -> Axioms? .
op (_,_,_) : Sorts Sorts Sorts -> Rules? .
endfth
```

As an example, the PTS signature of CC is given by the following functional module:

```
fmod CC-SIG is

sorts Sorts Axioms Axioms? Rules Rules? .
subsort Axioms < Axioms? .
subsort Rules < Rules? .
op (_,_) : Sorts Sorts -> Axioms? .
op (_,_,_) : Sorts Sorts Sorts -> Rules? .
```

```
op Prop : -> Sorts .
op Type : -> Sorts .

mb (Prop,Type) : Axioms .
mb (Prop,Prop,Prop) : Rules .
mb (Prop,Type,Type) : Rules .
mb (Type,Prop,Prop) : Rules .
mb (Type,Type,Type) : Rules .

endfm
```

PTSs can then be specified as a functional module parameterized by the theory `PTS-SIG`. Since functional modules have an initial (in this case free) model semantics, this formalization of PTSs is in fact a parameterized inductive definition that captures in a precise model-theoretic way the inductive character of PTS rules.

```
fmod PTS[S :: PTS-SIG] is
```

First we define the sort `Trm` of terms as an algebraic data type. Notice that we distinguish between a sort of names `Qid`, that are used in places where a variable is *declared*, and a sort of variables `Var`, that are used to *refer* to an already declared variable.

```
sorts Var Trm .
subsort Qid < Var .
subsort Var < Trm .
subsort Sorts < Trm .
op __ : Trm Trm -> Trm .
op [_:_]_ : Qid Trm Trm -> Trm .
op {_:_}_ : Qid Trm Trm -> Trm .

vars s s1 s2 s3 : Sorts .
vars X Y Z : Qid .
vars A B M N O P Q R T A' B' M' N' T' : Trm .
```

The usual deterministic version of capture-free substitution can be naturally defined in MEL as demonstrated in [32]. An important point is that we do not want to restrict ourselves to a particular choice of fresh names, since this would make the specification overly concrete. This can be accomplished by leaving unspecified the deterministic function for choosing fresh variables such that the actual function varies with the choice of the model; for details we refer to [32]. Here we only give the signature for set membership, free variables and the substitution function:

```
op _in_ : Qid QidSet -> Bool .
op FV : Trm -> QidSet .
op [_:=_]_ : Qid Trm Trm -> Trm .
```

We can use the substitution operator $[\_:=\_]\_$ to semantically identify terms that are α-convertible (we refer to the induced equality as α-equality) by means of the following equations.

```
ceq [X : A] M = [Y : A] ([X := Y] M) if not(Y in FV(M)) .
ceq {X : A} M = {Y : A} ([X := Y] M) if not(Y in FV(M)) .
```

We next define the binary relation of β-convertibility, which is used in the Conv rule of PTSs. The following (conditional) memberships, together with the initiality condition, define β-conversion as the smallest congruence (w.r.t. the term constructors) containing β-reduction.

```
sorts Convertible Convertible? .
subsort Convertible < Convertible? .

op _<->_ : Trm Trm -> Convertible? .

mb  M <-> M : Convertible .

cmb M <-> N : Convertible
    if N <-> M : Convertible .

cmb P <-> R : Convertible if
    P <-> Q : Convertible and Q <-> R : Convertible .

cmb (M N) <-> (M' N') : Convertible if
    M <-> M' : Convertible and N <-> N' : Convertible .

cmb ([X : A] M) <-> ([X : A'] M') : Convertible if
    A <-> A' : Convertible and M <-> M' : Convertible .

cmb ({X : A} B) <-> ({X : A'} B') : Convertible if
    A <-> A' : Convertible and B <-> B' : Convertible .

mb  (([X : A] M) N) <-> ([X := N] M) : Convertible .
```

The judgements of PTSs are of the form $\Gamma \vdash M : A$. We next define the syntax of contexts and judgements. Also, we define the function `_in_` used in the side conditions of some PTS rules.

```
sorts Context Judgement .
op emptyContext : -> Context .
op _:_ : Qid Trm -> Context .
op _,_ : Context Context -> Context [assoc id : emptyContext] .

var G : Context .
```

```
op _|-_:_ : Context Trm Trm -> Judgement .

op _in_ : Qid Context -> Bool .
eq X in emptyContext = false .
eq X in (G,(Y : A)) = (X in G) or (X == Y) .
```

We are now ready to define the inference rules. Semantically, the inference rules define an inductive subset of *derivable judgements*. The derivability predicate is usually implicit in informal reasoning, where $\Gamma \vdash M : A$ refers either to the judgement itself or to the fact that it is derivable.

```
sort Derivable .
subsort Derivable < Judgement .

cmb (emptyContext |- s1 : s2) : Derivable if (s1,s2) : Axioms .

cmb (G,(X : A) |- X : A) : Derivable if
    (G |- A : s) : Derivable /\ not(X in G) .

cmb (G,(X : B) |- M : A) : Derivable if
    (G |- M : A) : Derivable /\
    (G |- B : s) : Derivable /\ not(X in G) .

cmb (G |- {X : A} B : s3) : Derivable if
    (G |- A : s1) : Derivable /\
    (G,(X : A) |- B : s2) : Derivable /\ (s1,s2,s3) : Rules .

cmb (G |- [X : A] M : {X : A} B) : Derivable if
    (G |- A : s1) : Derivable /\
    (G,(X : A) |- M : B) : Derivable /\
    (G,(X : A) |- B : s2) : Derivable /\ (s1,s2,s3) : Rules .

cmb (G |- (M N) : [X := A] B) : Derivable if
    (G |- M : {X : A} B) : Derivable /\
    (G |- N : A) : Derivable .

cmb (G |- M : B) : Derivable if
    (G |- M : A) : Derivable /\
    (G |- B : s) : Derivable /\ A <-> B : Convertible .
```

```
endfm
```

In this formalization we have avoided any arbitrary encoding of syntax with binders that would require nontrivial justifications. Also, we have seen that the first-order framework is sufficiently powerful to represent PTSs without making any commitments. In particular, there was no need to change the syntax nor the rules of PTSs to obtain a faithful representation.

3.2   Taking Names Seriously

Although the abstract treatment of names in PTSs leads to a general metatheory that can be used as a high-level theoretical basis for quite different implementations of PTSs, there is a price to pay, in that an abstract view necessarily limits the expressivity of the theory. In the case of PTSs, properties involving names cannot be expressed. Indeed, we often need a more concrete representation with more specialized results to deal, for example, with the implementation of a formal system, or with tools that use the formal system in an essential way. Also in the context of reasoning about a formal system, a more concrete specification that is computationally meaningful can have considerable advantages for the partial automation of metatheoretic proofs in logics with computational sublanguages.

However, as soon as *we take names seriously, i.e., we give up the identification of α-convertible terms*, and interpret the inference rules literally, we encounter at least two problems first discussed in [40] under the title "closure under α-conversion".[3]

The *first problem* is that the set of derivable judgements is not closed under α-conversion. For instance, adapting an example given for $\lambda{\to}$ in [40], we cannot derive a judgement of the form

$$A : \mathtt{Prop}, P : \{Z : A\}\mathtt{Prop} \vdash [X : A][X : P\ X]A : \{X : A\}\{X : P\ X\}\mathtt{Prop},$$

say in CC, although the α-equivalent version

$$A : \mathtt{Prop}, P : \{Z : A\}\mathtt{Prop} \vdash [X : A][Y : P\ X]A : \{X : A\}\{Y : P\ X\}\mathtt{Prop},$$

where some bound variables are distinct can be derived.

A *second difficulty* pointed out in [40] is that we want to derive

$$A : \mathtt{Prop}, P : \{Z : A\}\mathtt{Prop} \vdash [X : A][X : P\ X]X : \{X : A\}\{Y : P\ X\}(P\ X),$$

but we should *not* be able to derive

$$A : \mathtt{Prop}, P : \{Z : A\}\mathtt{Prop} \vdash [X : A][X : P\ X]X : \{X : A\}\{X : P\ X\}(P\ X).$$

However, we cannot derive the first judgement, since the name X in the conclusion of the Lda rule is the same on both sides of the colon.

To tackle the first problem, Pollack proposed a type system $\vdash_{lt}$, a variation of $\lambda{\to}$. It uses a more liberal notion of context that allows multiple declarations of the same name, the one most recently introduced being visible inside the judgement. Unfortunately, he did not pursue this direction further because of the second difficulty, which appears in the context of PTSs with dependent types but is not present in $\lambda{\to}$. Concerning $\vdash_{lt}$, he remarks "I don't think we can do the same for PTS."

The solution finally discussed in [40] is the solution employed in the *constructive engine* [25] used in proof assistants such as LEGO [41] and COQ [2]. The

---

[3] The problem with closure under α-conversion also remains unsolved in [30], where a system with dependent types is presented that does not enjoy this property.

idea is to use a hybrid naming scheme which employs distinct names for *global variables* declared in the context of a judgement, and a de Bruijn representation of terms with bound *local variables*. Clearly, PTSs based on such a hybrid naming scheme are a correct implementation of (abstract) PTSs as described above. More precisely, PTSs using the hybrid naming scheme can be seen as particular models of the MEL specification of PTSs in the sense that the corresponding model is isomorphic to the one given by the appropriately instantiated functional module PTS. Nevertheless, an approach which maintains a distinction between global and local variables appears not to be very uniform, complicating formal metatheoretic proofs and type checking. Of course, scaling up Pollack's $\vdash_{lt}$ to PTSs would be much more satisfying, and this is the direction we pursue in the following.

3.3   Indexed Names and Named Indices

We believe that the root of the second difficulty discussed above is that the traditional notion of binding used in logic and in programming reveals an undesirable property, which may be called *accidental hiding*, if the language is refined in the most direct way, i.e., by just giving up identification by α-conversion.

Consider for instance the formula

$$\forall X \, . \, (A \, \wedge \, \forall Y \, . \, (B \Rightarrow \forall X \, . \, C(X)))$$

with distinct names X and Y, where $C(X)$ is a formula that contains X free. Each occurrence of X in $C(X)$ is *captured* by the inner $\forall$ quantifier, so that the outermost $\forall$ quantifier is hidden from the viewpoint of $C(X)$. Indeed there is no way to refer to the outermost $\forall$ quantifier within $C(X)$.

Hence, we are faced with the following problem: a calculus without α-equality is not only less abstract, which is an unavoidable consequence of giving up identification by α-conversion, but also, depending on the (accidental) choice of names, visibility of (bound) variables may be restricted. It is important to emphasize that visibility is not restricted in the original calculus with α-equality, since renaming can be performed *tacitly* at any time.

Clearly, this phenomenon of hiding that occurs in the example above is undesirable[4], because it is not present in the original calculus with α-equality. It is merely an accident caused by giving up identification by α-conversion without adding a compensating flexibility to the language.

This suggests tackling this general problem by migrating to a more flexible syntax, where we express a binding constraint by annotating each name X with an index $i \in \mathbb{N}$, written X_i, that indicates how many X-binders should be skipped before we reach the one that X_i refers to. For instance we write

$$\forall X \, . \, (A \, \wedge \, \forall Y \, . \, (B \Rightarrow \forall X.C(X_0)))$$

to express that X_0 is bound by the inner $\forall$, and

$$\forall X \, . \, (A \, \wedge \, \forall Y \, . \, (B \Rightarrow \forall X.C(X_1)))$$

---

[4] Of course, in general hiding is important but it is not an issue of binding; it should be treated independently.

meaning that X_1 is bound by the outermost $\forall$. To make the language a conservative extension of the traditional notation, we can identify X and X_0.

In fact, the use of indexed names is equivalent to a representation introduced by Berkling [7,8] in the context of λ-calculus[5] which is why we refer to the notation based on indexed names also as Berkling's notation. As indicated by the example above we use Berkling's representation not (only) for λ-calculus but as the core syntax of CINNI, the *Calculus of Indexed Names and Named Indices* which is generic in the sense that it can be instantiated for a wide range of object languages with different binding constructs. For a detailed treatment of CINNI, its metatheoretic properties, and its relation to other calculi we refer to [47,48].

Obviously, there is some similarity to a notation based on de Bruijn indices [18]. But notice that there is an essential difference: the index m in the occurrence X_m is *not* the number of binders to be skipped; it states that we have to skip m binders for the particular name X, *not* counting binders for other names. Still a formal relationship to de Bruijn's notation can be established: if we restrict ourselves to terms that *contain only a single name X*, then we can replace each X_i by the index i without loss of information and we arrive at de Bruijn's purely indexed notation.[6] In other words, if we restrict the available names to a single one, we obtain de Bruijn's notation as a very special case. In this sense, Berkling's representation can be formally seen as a proper generalization of de Bruijn's notation. Pragmatically, however, the relationship to de Bruijn's syntax plays only a minor role, since a typical user will exploit the dimension of names much more than the dimension of indices. Hence, in practice the notation can be used as a standard named notation, with the additional advantage that accidental hiding and weird renaming[7] are avoided.

The pragmatic advantage of Berkling's notation is that it can be used to reduce the distance between the formal system and its implementation: it can be directly employed by the user who wants to think in terms of names, so that the need for a translation between an internal representation (e.g. using de Bruijn indices) and a user-friendly syntax (e.g. using ordinary names) disappears completely. As far as we know the CINNI substitution calculus is the first calculus of explicit substitutions which combines named and index-based representations and hence provides a link between these two worlds of explicit substitution calculi.

Usually, this translation is not considered to be a problem, and indeed in the case of terms, where all parts are known or accessible, solutions are straightforward. However, it is clear that this gap is not desirable: consider, for example, a tactic-based theorem prover where the user is confronted with an internal representation which reflects the theory only in a very indirect way. More seriously, the translation between internal and external representations becomes unworkable, or at least requires certain restrictions, as soon as we use terms containing

---

[5] An indexed variable X_i is represented in Berkling's representation as $\#^i X$ where $\#$ is the so-called unbinding operation.

[6] With the slight difference that de Bruijn's indices start at 1 instead of 0.

[7] See the discussion on weird renaming in the next section.

metavariables, holes or placeholders, which are useful for many applications in-
cluding unification algorithms and representation of incomplete proofs.

3.4   Explicit Substitutions

In the previous section we discussed Berkling's first-order representation for ex-
pressions, which contains the conventional named notation as well as de Bruijn's
indexed notation as special cases. The most important operation to be performed
on such terms represented in this way is capture-free substitution. Therefore, we
now present the CINNI substitution calculus, a first-order calculus that can be
seen as an (operational) refinement of an external (i.e. metalevel) substitution
function such as the one given in [8].

Strictly speaking, CINNI is a family of explicit substitution calculi, parame-
terized by the syntax (including information about binding) of the language we
want to represent. Here we present the instantiation of this substitution calculus
for the untyped λ-calculus. λ-terms in CINNI syntax are:

$$X_m \mid (M\ N) \mid [X]M$$

As a motivation for the substitution calculus given below, consider the fol-
lowing example of a β-reduction step in the traditional λ-calculus with distinct
names X and Y, again taking names literally, i.e. not presupposing identification
by α-conversion:

$$(([X][Y]X)Y) \rightarrow [Z]Y$$

Clearly, Z must be a name different from Y to avoid capturing. Unfortunately,
there is no canonical choice if all names should be treated as being equal. We call
this phenomenon *weird renaming* of bound variables. It is actually a combination
of two undesirable effects: (1) names that have been carefully chosen by the user
have to be changed, and (2) the enforced choice of a new name collides with the
right of names to be treated as equal citizens.

These effects are avoided in the CINNI calculus, when instantiated to the
λ-calculus. CINNI is specified by the first-order equational theory given below.
Indeed, the only operation assumed on names is equality. CINNI has also an
operational semantics viewing equations as rewrite rules. Apart from the two
basic kinds of substitutions, namely *simple* substitutions $[X{:=}M]$, and *shift* sub-
stitutions $\uparrow_X$, substitutions can be *lifted* using $\Uparrow_X S$, where the variable S ranges
over substitutions.

$$
\begin{array}{ll}
[X{:=}M]\ X_0\ =\ M & \Uparrow_X S\ X_0\ =\ X_0 \\
[X{:=}M]\ X_{m+1}\ =\ X_m & \Uparrow_X S\ X_{m+1}\ =\ \uparrow_X (S\ X_m) \\
[X{:=}M]\ Y_n\ =\ Y_n \text{ if } X \neq Y & \Uparrow_X S\ Y_n\ =\ \uparrow_X (S\ Y_n) \text{ if } X \neq Y
\end{array}
$$

$$
\begin{array}{ll}
\uparrow_X X_m\ =\ X_{m+1} & S\ (MN)\ =\ (SM)(SN) \\
\uparrow_X Y_n\ =\ Y_n \text{ if } X \neq Y & S\ ([X]M)\ =\ [X](\Uparrow_X S\ M)
\end{array}
$$

The CINNI calculus can be instantiated to various object languages with different binding operators to give a more concrete treatment of their associated formal systems. The only equations specific to the syntax of the language are the structural equations. Here, the last two equations in the right column are the structural equations for the λ-calculus. In a similar way, CINNI can be instantiated to other object languages such as Abadi and Cardelli's ς-calculus or Milner's π-calculus [47,48].

Now we can define β-*reduction* by the rule

$$([X]N)M \rightarrow_\beta [X{:=}M]N.$$

Notice that weird renaming of bound variables as in the previous example is avoided with the new notion of β-reduction which yields[8]

$$(([X][Y]X_0)Y_0) \rightarrow_\beta ([Y]Y_1)$$

As another application of substitution, consider *renaming of a bound variable* X by $\bullet$ as in the following rule of α-*reduction*:

$$([X]N) \rightarrow_\alpha ([\bullet][X{:=}\bullet] \uparrow_\bullet N) \text{ if } X \neq \bullet$$

where $\bullet$ is an arbitrary but fixed name. Using this rule every CINNI term can be reduced to a nameless α-*normal form* which is essentially its de Bruijn index representation. For terms M,N we use $M \equiv_\alpha N$ to denote that M and N are equal up to renaming of bound variables.

Just as Berkling's notation contains de Bruijn's indexed notation as a very special case, the instantiation of CINNI for the λ-calculus reduces to the calculus $\lambda\upsilon$ of explicit substitutions proposed by Pierre Lescanne [27,28,5], but only in the degenerate case where we only use a single name. It is noteworthy that $\lambda\upsilon$ is the smallest known indexed substitution calculus enjoying good theoretic properties like confluence[9] and preservation of strong normalization. It seems that its simplicity is inherited by CINNI although in practice the dimension of names will be much more important than the dimension of indices. Hence, we tend to think of CINNI more as a substitution calculus with names than as one with indices.

3.5   Uniform Pure Type Systems

The application of CINNI to PTSs can be seen as Pollack's $\vdash_{lt}$ scaled up to PTSs. In contrast to the hybrid approach to PTSs adopted in the constructive engine [25] and in the PTS formalization given in [34] based on an idea from [15], both

---

[8] One might argue that even the change of variable indices constitutes a form of renaming, but an important point is that only references to previously introduced names are affected rather than the binding occurrences themselves.

[9] In fact, we have confluence on terms without metavariables [47,48], but this is sufficient for the approach to type checking/inference presented in this paper, since all metavariables will eventually become instantiated.

distinguishing between global and local variables, we use indexed names *uniformly*. This suggests defining *uniform pure type systems (UPTSs)* by modifying PTSs in three steps:

First, PTS terms are generalized to UPTS terms in the way explained before, i.e., *UPTS terms* are defined by the first-order CINNI syntax:

$$X_m \,|\, (M\ N) \,|\, [X : A]M \,|\, \{X : A\}M \,|\, s$$

As a second step, we adapt the syntax-dependent part of the CINNI calculus to UPTS terms:

$$S\ s\ =\ s$$
$$S\ (MN)\ =\ (SM)(SN)$$
$$S\ ([X : A]M)\ =\ [X : (S\ A)](\Uparrow_X S\ M)$$
$$S\ (\{X : A\}M)\ =\ \{X : (S\ A)\}(\Uparrow_X S\ M)$$

The third and final step is to define the derivable typing judgements. Since we do not want to identify α-convertible terms, this is a fundamental change in the formal system. However, a careful inspection of the typing rules *under the new reading* shows that only minor changes in the rules Start and Weak are needed. The new rules are:

$$\frac{\Gamma \vdash A : s}{\Gamma, X : A \vdash X_0\ : \uparrow_X A} \tag{Start}$$

$$\frac{\Gamma \vdash M : A \quad \Gamma \vdash B : s}{\Gamma, X : B \vdash \uparrow_X M\ : \uparrow_X A} \tag{Weak}$$

It might appear that the UPTSs we have defined above are a specialization of PTSs, since we have committed ourselves to a particular representation of names. But this is not the full truth, because on the other hand we have described a generalization of PTSs where multiple declarations of the same name are admitted in a well-typed context. Notice that in both rules above we have dropped the side condition $X \notin \Gamma$, which means that we have completely eliminated the need for these side conditions in UPTSs. We would also like to point out, that, in particular, we have not touched the Lda rule: the only place where α-conversion comes into play is in the Conv rule, where $\equiv_\beta$ subsumes α- and β-conversion, just as in the original PTSs.

Finally, we describe how these changes are reflected in the MEL specification, that is how UPTSs can be represented by modifying the previous specification.

First, instead of using names as variables we use indexed names. So we replace `subsort Qid < Var` by

```
op _{_} : Qid Nat -> Var .
```

Second, instead of conventional substitution `[_:=_]_`, we use CINNI for UPTS terms:

```
sort Subst .

op [_:=_] : Qid Trm -> Subst .
op [shift_] : Qid -> Subst .
op [lift__] : Qid Subst -> Subst .
op __ : Subst Trm -> Trm .

var  S : Subst .
vars n m : Nat .

eq  ([X := M] (X{0})) = M .
eq  ([X := M] (X{suc(m)})) = (X{m}) .
ceq ([X := M] (Y{n})) = (Y{n}) if X =/= Y .

eq  ([shift X] (X{m})) = (X{suc(m)}) .
ceq ([shift X] (Y{n})) = (Y{n}) if X =/= Y .

eq  ([lift X S] (X{0})) = (X{0}) .
eq  ([lift X S] (X{suc(m)})) = [shift X] (S (X{m})) .
ceq ([lift X S] (Y{m})) = [shift X] (S (Y{m})) if X =/= Y .

eq  (S s) = s .
eq  (S (M N)) = ((S M) (S N)) .
eq  S ([X : A] M) = [X : (S A)] ([lift X S] M) .
eq  S ({X : A} M) = {X : (S A)} ([lift X S] M) .
```

Third, conversion now explicitly contains α-conversion, something that was implicit in the equality of the previous specification:

```
mb  [X : A] M <->
    [Y : A] ([X := Y{0}] [shift Y] M) : Convertible .

mb  {X : A} M <->
    {Y : A} ([X := Y{0}] [shift Y] M) : Convertible .
```

Finally, the new versions of Start and Weak are:

```
cmb (G,(X : A) |- X{0} : [shift X] A) : Derivable if
    (G |- A : s) : Derivable .

cmb (G,(X : B) |- [shift X] M : [shift X] A) : Derivable if
    (G |- M : A) : Derivable /\
    (G |- B : s) : Derivable .
```

Again, we can see that the representational distance between the mathematical presentation of UPTSs and their MEL specification is practically zero. In particular, the equational nature of the CINNI substitution calculus is directly captured by the MEL specification.

UPTSs are more liberal than PTS, since a derivable judgement $\Gamma \vdash M : A$ may contain multiple declarations of the same name in Γ. However, the set of derivable judgements $\Gamma \vdash M : A$ of PTS can be recovered as the set of derivable UPTS judgements $\Gamma \vdash_1 M : A$ generated by adding the following rule:

$$\frac{\Gamma \vdash M : A}{\Gamma \vdash_1 M : A} \quad \text{if } \Gamma \text{ is simple} \tag{Simple}$$

The representation of judgements $\Gamma \vdash_1 M : A$ together with this rule in MEL is straightforward, and we omit it here and in all the following formalizations for the sake of brevity.

To state the following results we proceed as for PTSs: We extend α-conversion $\equiv_\alpha$ from terms to judgements, so that $\Gamma \vdash M : A \equiv_\alpha \Gamma' \vdash M' : A'$ iff $\Gamma' \vdash M' : A'$ and $\Gamma \vdash M : A$ are equal up to consistent renaming of declared *and* bound variables. Then we have the following

Lemma 32 (Strong Closure under α-Conversion for UPTSs)
Let M, A, M', A' be UPTS terms and Γ, Γ' be UPTS contexts. If the UPTS judgement $\Gamma \vdash M : A$ is derivable in $\mathbf{UPTS}_S$ and $\Gamma \vdash M : A \equiv_\alpha \Gamma' \vdash M' : A'$, then $\Gamma' \vdash M' : A'$ is derivable in $\mathbf{UPTS}_S$.

Proof Sketch.   By induction over derivations of $\Gamma \vdash M : A$.   □

It is noteworthy that a weak form of this lemma using α-conversion on terms instead of judgements, i.e. the special case where $\Gamma = \Gamma'$, cannot be proved directly by induction. The induction would fail for the rules Pi and Lda, since a declared variable X becomes a local variable.

As for PTSs the previous lemma is equivalent to the admissibility of the following rule in UPTSs:

$$\frac{\Gamma \vdash M : A}{\Gamma' \vdash M' : A'} \quad \text{if } \Gamma \vdash M : A \equiv_\alpha \Gamma' \vdash M' : A' \tag{Rename}$$

Using the terminology introduced in Section 1.1 for entailment systems, each of the following two propositions establishes a sound and complete total correspondence of the form $\mathbf{PTS}_S \curvearrowright \mathbf{UPTS}_S$, where S is an arbitrary PTS signature.

Proposition 33 (Soundness and Completeness of UPTSs I)
For all PTS terms M,A and PTS contexts Γ, if the PTS judgement $\Gamma \vdash_1 M : A$ is derivable in $\mathbf{UPTS}_S$ then $\Gamma \vdash M : A$ is derivable in $\mathbf{PTS}_S$ and vice versa.[10]
Proof Sketch.
First observe that each PTS rule is a UPTS rule if we restrict ourselves to simple judgements. In particular, the side conditions $X \in \Gamma$ in the PTS rules Start and Weak imply that the shift substitution in the corresponding UPTS rules can be eliminated.

($\Rightarrow$) Given a UPTS derivation of a simple judgement $\Gamma \vdash M : A$, each occurrence of a UPTS inference rule can be replaced as follows: First the original

---

[10] Here we make use of the convention, introduced in Section 3.3, that ordinary terms (here PTS terms) can be seen as CINNI terms (here UPTS terms).

premises are converted into suitable simple PTS form by virtue of Rename. Then the corresponding inference rule for PTSs is applied (which is also a UPTS rule according to the observation above). Finally, the conclusion in simple PTS form is converted back to the original conclusion in UPTS form, again using Rename. After transforming the entire derivation in this way all intermediate UPTS judgements which are not PTS judgements, i.e. the original premises and original conclusions, can be removed, and the result is still a UPTS derivation. Also, the resulting derivation corresponds to a derivation in PTSs extended by the admissible rule Rename.

($\Leftarrow$) According to the observation above, each application of a PTS rule can be seen as an application of the corresponding UPTS rule. Furthermore, each implicit α-conversion step that is possible in PTSs can be simulated by Rename, which is an admissible rule in UPTSs.

$\square$

In other words, UPTSs are conservative over PTSs. A slightly weaker but more comprehensive correspondence of the form $\mathbf{PTS}_S \curvearrowright \mathbf{UPTS}_S$ can be given modulo renaming of variables:

Proposition 34 (Soundness and Completeness of UPTSs II)
For all UPTS terms M,A, PTS terms M',A', UPTS contexts Γ and simple PTS contexts Γ' with $\Gamma \vdash M : A \equiv_\alpha \Gamma' \vdash M' : A'$, if the UPTS judgement $\Gamma \vdash M : A$ is derivable in $\mathbf{UPTS}_S$ then $\Gamma' \vdash M' : A'$ is derivable in $\mathbf{PTS}_S$ and vice versa.

Proof Sketch.
($\Rightarrow$) Let $\Gamma \vdash M : A$ be derivable in $\mathbf{UPTS}_S$ and let $\Gamma \vdash M : A \equiv_\alpha \Gamma' \vdash M' : A'$. By Proposition 32 (strong α-closure) $\Gamma' \vdash M' : A'$ and therefore $\Gamma' \vdash_1 M' : A'$ are derivable in $\mathbf{UPTS}_S$. So by Proposition 33 $\Gamma' \vdash M' : A'$ is derivable in $\mathbf{PTS}_S$.

($\Leftarrow$) Let $\Gamma' \vdash M' : A'$ be derivable in $\mathbf{PTS}_S$ and let $\Gamma' \vdash M' : A' \equiv_\alpha \Gamma \vdash M : A$. By Proposition 33, $\Gamma' \vdash M' : A'$ is derivable in $\mathbf{UPTS}_S$, and by Proposition 32 (strong α-closure) $\Gamma \vdash M : A$ is derivable in $\mathbf{UPTS}_S$ too.

$\square$

The last proposition implies that, concerning judgements of the form $\Gamma \vdash M : A$, PTSs and UPTSs are equivalent modulo α-conversion. Hence all (metatheoretic) results about PTSs [22] apply to UPTSs after appropriate renaming.

Another consequence of the last proposition is that the new form of judgement $\Gamma \vdash_1 M : A$ is not necessary to ensure soundness, and could therefore be dropped. Sometimes, however, focusing on judgements of the form $\Gamma \vdash_1 M : A$ instead of the more general form $\Gamma \vdash M : A$ is more convenient, e.g. to formulate the weakening/thinning lemma [22,52], since simple contexts can be treated as sets of declarations.

3.6   A Conservative Optimization

The presentations of PTSs and UPTSs given above maintain a good economy in the number of rules and are therefore well-suited for metatheoretic (inductive)

reasoning. The judgement $\Gamma \vdash M : A$ implicitly subsumes another judgement $\Gamma \Vdash$, stating that Γ is a well-typed context. Since in practice checking contexts is as important as checking types, we switch to a conservative extension of UPTSs (similar to an optimization for PTSs mentioned in [52]) that is not biased towards any of the two forms of judgement. From a practical point of view, the addition of a separate judgement for well-typed contexts can be seen as an optimization which avoids rechecking contexts in each subderivation. We will refer to this optimized version as *optimized UPTSs (OUPTSs)* and the entailment system will be denoted by **OUPTS**. The only modifications we need are described below. In addition to the *main typing judgement*, which is written now as $\Gamma \Vdash M : A$ (stating that M is an element of the type T in Γ), we use *context typing judgements* of the form $\Gamma \Vdash$ meaning that Γ is a well-typed context, and *relative typing judgements* of the form $\Gamma \vdash M : A$ meaning that M is an element of type A if Γ is well-typed. Furthermore, we add the following rules:

$$\frac{}{[] \Vdash} \qquad \qquad \text{(Ctxt1)}$$

$$\frac{\Gamma \Vdash \quad \Gamma \vdash A : s}{\Gamma, X : A \Vdash} \qquad \qquad \text{(Ctxt2)}$$

$$\frac{}{\Gamma \vdash X_m : lookup(\Gamma, X_m)} \quad \text{if } lookup(\Gamma, X_m) \text{ is defined} \qquad \text{(Lookup)}$$

$$\frac{\Gamma \Vdash \quad \Gamma \vdash M : A}{\Gamma \Vdash M : A} \qquad \qquad \text{(Main)}$$

where $lookup(\Gamma, X_m)$ is a partial function defined by:

$$
\begin{aligned}
lookup((\Gamma, X : A), X_0) &= \uparrow_X A \\
lookup((\Gamma, X : A), X_{m+1}) &= \uparrow_X lookup(\Gamma, X_m) \\
lookup((\Gamma, X : A), Y_m) &= \uparrow_X lookup(\Gamma, Y_m) \text{ if } X \neq Y
\end{aligned}
$$

Then we replace Ax and Simple by:

$$\frac{}{\Gamma \vdash s_1 : s_2} \quad (s_1, s_2) \in \mathcal{A} \qquad \qquad \text{(Ax)}$$

$$\frac{\Gamma \Vdash M : A}{\Gamma \Vdash_1 M : A} \quad \text{if no variable is declared in } \Gamma \text{ more than once.} \qquad \text{(Simple)}$$

respectively, and we remove the rules Start and Weak, since they are admissible rules in the new system. The system we have just obtained is similar to the system $\vdash_{vtyp}$, $\vdash_{vcxt}$ presented in [52], but here we are concerned with UPTSs instead of PTSs. Another minor difference is that we make use of an explicit lookup function. As before, we do not need any freshness side conditions thanks to CINNI.

Again, the representation in MEL is quite direct. It nicely illustrates the mixed specification style using equations and memberships, and also the representation of partial functions such as *lookup*.

```
sort Trm? .
subsort Trm < Trm? .

op  lookup : Context Var -> Trm? .
eq  lookup(G,(X : A), X{0}) = [shift X] A .
eq  lookup(G,(X : A), X{suc(m)}) = [shift X] lookup(G,X{m}) .
ceq lookup(G,(X : A), Y{m}) = lookup(G,Y{m}) if (X =/= Y) .

op  _||- : Context -> Judgement .
op  _|-_:_ : Context Trm Trm -> Judgement .
op  _||-_:_ : Context Trm Trm -> Judgement .

mb  (emptyContext ||-) : Derivable .

cmb (G,(X : A) ||-) : Derivable if
    (G ||-) : Derivable /\ (G |- A : s) : Derivable .

cmb (G |- X{m} : lookup(G,X{m})) : Derivable if
    lookup(G,X{m}) : Trm .

cmb (G ||- M : A) : Derivable if
    (G ||-) : Derivable /\ (G |- M : A) : Derivable .

cmb (G |- s1 : s2) : Derivable if (s1,s2) : Axioms .
```

OUPTSs are equivalent to UPTSs, i.e., there is a sound and complete total correspondence of the kind $\mathbf{UPTS}_S \curvearrowright \mathbf{OUPTS}_S$ for arbitrary PTS signatures S, in the following sense:

Proposition 35 (Soundness and Completeness of OUPTSs)
Let M, A be UPTS terms, and let Γ be a UPTS context. If the judgement $\Gamma \Vdash M : A$ ($\Gamma \Vdash_1 M : A$) is derivable in $\mathbf{OUPTS}_S$, then $\Gamma \vdash M : A$ ($\Gamma \vdash_1 M : A$) is derivable in $\mathbf{UPTS}_S$ and vice versa.

Proof Sketch. It is easy to adopt the proof of the similar lemma 23 in [52] to our setting. The main change is that we are using UPTSs instead of PTSs here. A minor point is that we are using an explicit *lookup* function.   □

4   The Meta-operational View of PTSs

PTSs can not only be equipped with a logical semantics, e.g. via the proposition-as-types interpretation, but, more fundamentally, PTSs are usually equipped with an operational semantics, defined by an internal notion of functional computation, such as β-reduction. The operational view of PTSs is concerned with their internal notion of computation, but here we are interested in the *meta-operational view*, which deals with the question of how to embed PTSs in a formal

system with an operational semantics, so that typical computational tasks like type checking and type inference become possible by exploiting the operational semantics of the metalanguage. In the following we employ for this purpose the efficiently executable sublanguage of rewriting logic that is supported by the Maude engine.

First, we introduce several well-known classes of PTS signatures, giving rise to corresponding PTSs that are practically interesting and enjoy particularly good properties.

Definition 41   A PTS signature S is *decidable* iff: (1) $\mathcal{S}$ is denumerable, (2) $\mathcal{A}$ and $\mathcal{R}$ are decidable, and (3) for all $s_1, s_2 \in \mathcal{S}$ the predicates $\exists s_2' : (s_1, s_2') \in \mathcal{A}$ and $\exists s_3' : (s_1, s_2, s_3') \in \mathcal{R}$ are decidable.

Decidability is a reasonable requirement to ensure that type inference and type checking do not become undecidable because of a too complex PTS signature.

Definition 42   A PTS signature S is *functional* iff (1) $(s_1, s_2) \in \mathcal{A}$ and $(s_1, s_2') \in \mathcal{A}$ implies $s_2 = s_2'$, and (2) $(s_1, s_2, s_3) \in \mathcal{R}$ and $(s_1, s_2, s_3') \in \mathcal{R}$ implies $s_3 = s_3'$.

In functional PTS signatures, the relations $\mathcal{A}$ and $\mathcal{R}$ can be viewed as partial functions $\mathcal{A} : \mathcal{S} \rightsquigarrow \mathcal{S}$ and $\mathcal{R} : \mathcal{S} \times \mathcal{S} \rightsquigarrow \mathcal{S}$. Functionality ensures that every term has a unique type modulo $\equiv_\beta$ [22]. The class of functional PTSs[11] includes, for example, all systems of the λ-cube.

Definition 43   A PTS signature S is *full* iff for all $s_1, s_2 \in \mathcal{S}$ there is an s_3 such that $(s_1, s_2, s_3) \in \mathcal{R}$. A PTS signature S is *semi-full* iff $(s_1, s_2, s_3) \in \mathcal{R}$ implies that for each s_2' there is an s_3' such that $(s_1, s_2', s_3') \in \mathcal{R}$.

Full PTSs allow us to form dependent types $\{X : A\}B$ very liberally, by avoiding those restrictions on the sorts of A and B that are imposed by the side condition $(s_1, s_2, s_3) \in \mathcal{R}$ of the Pi rule. As an example, CC is a full PTS.

Definition 44   Given a PTS signature S, a *top sort* is a sort s such that there is no sort s' with $(s, s') \in \mathcal{A}$. The set of top sorts is denoted by $\mathcal{S}_{\text{top}}$.

To avoid inessential technicalities in our presentation, we will later focus on PTS signatures without top sorts, which introduce some kind of nonuniformity in the set of sorts. Just as $\mathcal{R}$ can be seen as a function $\mathcal{R} : \mathcal{S} \times \mathcal{S} \to \mathcal{S}$ in full PTS signatures, $\mathcal{A}$ can be viewed as a function $\mathcal{A} : \mathcal{S} \to \mathcal{S}$ in functional PTS signatures without top sorts.

Semi-full PTSs have the nice property that we can get rid of the third premise in Lda by replacing it with the following rule:

$$\frac{\Gamma \vdash A : s_1 \quad \Gamma, X : A \vdash M : B}{\Gamma \vdash [X : A]M : \{X : A\}B} \quad (s_1, s_2, s_3) \in \mathcal{R} \text{ and } B \notin \mathcal{S}_{\text{top}} \qquad \text{(Lda')}$$

---

[11] The attributes for PTS signatures are naturally lifted to the corresponding PTSs.

The premises together with the side conditions in Lda' imply that $\{X : A\}B$ is a type (cf. rule Pi). Indeed, as explained in [52] in the context of PTSs, replacing Lda by Lda' does not change the set of derivable judgements in semi-full UPTSs.

For full UPTSs without top sorts we can completely eliminate the side conditions in the rule Lda', and we obtain Lda" without changing the set of derivable judgements:

$$\frac{\Gamma \vdash A : s \quad \Gamma, X : A \vdash M : B}{\Gamma \vdash [X : A]M : \{X : A\}B} \tag{Lda"}$$

Our example PTS signature of CC at the beginning of Section 3 has Type as a top sort. However, it is straightforward to extend CC by an infinite universe hierarchy yielding a PTS without top sorts. Our example PTS signature of CC has Type as a top sort. However, it is straightforward to extend CC by an infinite universe hierarchy yielding a PTS without top sorts.

Together with the introduction of UPTSs in the previous section, we have now (following the corresponding arguments for PTSs in [52]) three families of inference systems which only differ in the choice of the rule Lda. For a full PTS signature S without top sorts all of them define the same unary entailment system, which is denoted by **UPTS**$_S$.

In the remainder of this paper we will present a standard type checking algorithm for a class of UPTSs using rewriting logic as a formal specification language. In spite of some unsolved theoretical questions such as the expansion postponement problem, efficient algorithms for the important classes of functional PTSs and semi-full PTSs (satisfying appropriate decidability and normalization properties) have been presented in [52]. In order to avoid excessive technical details and to make clear the general way we use rewriting logic to represent type checking algorithms, we restrict ourselves in the following to UPTSs that are decidable, normalizing (w.r.t. β-reduction), functional, full, and without top sorts. The class of UPTSs that are decidable, normalizing, functional and semi-full can be treated along the same lines (using the rule Lda' instead of Lda").

The use of UPTSs instead of PTSs is motivated by our desire to obtain a *formal executable representation* that takes names seriously and makes type checking simpler and more uniform. The approach is different from the constructive engine [25] and its presentation in [40] that employs named global variables and a de Bruijn representation for local variables. It is also different from [15], [52] and the formalization [34] that distinguish between two unrelated sets of global names and local names.

4.1   Uniform PTSs in Membership Equational Logic

The standard way to implement type checking, which goes back to [33] and [25], is to cast the inference rules into an equivalent syntax-directed inductive definition, and to define a type-inference function on the basis of this new system. Formally and technically this could be done in the executable sublanguage of MEL or in any other functional programming language, but the use of MEL is

attractive, since it allows us to formulate the logical and operational versions of PTSs in a single uniform language with a simple semantics, which in particular does not presuppose higher-order constructs, but is used to explain them in more elementary terms. Also, data structures and functions of the specification can be directly used in the implementation.

In our setting there is another reason why MEL is more natural than the use of a (higher-order) functional programming language: the equational specification of the calculus of substitutions presented above is naturally equipped with an operational semantics just by viewing the equations as rewrite rules. By contrast, in a functional programming language that is not based on equational rewriting, the substitution calculus has to be *encoded*, which essentially means that a (specialized) rewrite engine for this calculus has to be implemented in the functional language itself and, what is even more cumbersome, this engine has to be explicitly invoked when needed. In this sense, a specification/programming style based on rewriting is more abstract and closer to mathematical practice for applications of this kind than a (higher-order) functional programming approach.

Using the specification of the above substitution calculus, a purely equational executable specification of a type checker for UPTSs with decidable type checking can be written in MEL using standard equational/functional programming techniques. The core of this specification consists of a type-inference function

```
op type : Context Trm -> Trm? .
```

that computes a type for each term which is well-typed in the given context. The function can be defined in a way similar to the one given in [45], but using CINNI, instead of abstracting from the treatment of names.

Thanks to CINNI, freshness conditions are avoided. Therefore, an implementation based on this specification appears to be more elegant than that of the constructive engine with its hybrid treatment of names. As an additional advantage, multiple declarations of the same name are naturally admitted in contexts if we use judgements $\Gamma \Vdash M : A$. However, it is also easy to disallow these more general contexts if desired by implementing simple judgements $\Gamma \Vdash_1 M : A$.

Instead of discussing this purely equational approach in more detail, we present an alternative approach in the following section that exploits features of rewriting logic that are beyond equational and functional languages. Our experience shows that this alternative approach scales up well to more complex type theories, e.g. extensions of UPTSs such as OCC (see Section 5.1) in a more satisfactory way than the purely functional and equational approaches to type checking.

4.2   Uniform PTSs in Rewriting Logic

As shown by an extensive collection of examples in [31,32], rewriting logic can be used as a logical framework that can naturally represent inference systems of different kinds in a logically and operationally satisfying way. In the present

section we view a type checker as a particular inference system. In contrast to a (higher-order) functional programming approach that would require us to *encode* the inference system in terms of a type checking function, the rewriting logic approach offers the advantage that inference rules can be expressed directly, namely, as rewrite rules. We will in fact make use of a type inference system expressed as a collection of rewrite rules that transform a conjunction of judgements into a simplified form, in the style of *constraint solving systems*. This yields a rewrite system that is efficiently executable, while still maintaining a close correspondence to the logical specification of UPTSs.

The rewriting logic specification represents *rewriting-based OUPTSs (ROUPTSs)* and is able to perform type checking, i.e. to decide derivability of judgements of the form $\Gamma \Vdash$, $\Gamma \vdash M : A$, and $\Gamma \Vdash M : A$, for the class of decidable, normalizing, functional, full and PTS signatures without top sorts discussed before. As in PTSs, type checking is reduced to type inference, that is, to solving incomplete queries of the form $\Gamma \vdash M$ ->: ?.

Instead of giving an informal account we directly discuss the formal specification in rewriting logic.

First, we exploit our assumption that the PTS signature is decidable, functional, full and without top sorts, which means that the relations $\mathcal{A}$ and $\mathcal{R}$ can be specified by equationally-defined functions `Axioms` and `Rules`:

```
fth FPTS-SIG is
sort Sorts .
op Axioms : Sorts -> Sorts .
op Rules : Sorts Sorts -> Sorts .
endfth
```

As usual for syntax-directed approaches following the ideas of [33] and [25] we "invert" the inference rules in order to obtain a goal-directed algorithm from the inductive definition. In contrast to a purely equational and functional approach, the rewriting logic specification we aim at has a rewrite transition system as a model, and can therefore be seen as an operational generalization of the equational and functional paradigms. In contrast to [52] and [40], the type-checking algorithm itself receives a direct formal status as a transition system, which is a good basis for reasoning formally about operational properties and especially about its correctness.

The inductive definition of UPTSs can be seen as a static description of a set of judgements that we would like to equip with a dynamic structure. More precisely, a *(static) logical implication*

$$A_1 \wedge \ldots \wedge A_n \Rightarrow B$$

can be seen as an *inference rule* or *(dynamic) state transition* refining a goal B into subgoals $A_1, \ldots, A_n$, and can be directly represented as a rewrite rule

$$B \rightarrow A_1 \wedge \ldots \wedge A_n$$

in rewriting logic. Each state consists of a finite set of subgoals that remain to be solved.

The static description can be seen as inducing the following invariant that our dynamic system should always satisfy: for each state, the empty set of goals is reachable iff the logical interpretation of the state is true.

Although the inference rules of a formal system typically take the form of Horn clauses that can be operationally refined to rewrite rules, there may be functional and equational parts (e.g. auxiliary functions or substitution calculi) that are more naturally expressed in the MEL fragment. It is this mix of different paradigms that allows us to express the type-checking algorithm in a way that is very close to the logical specification.

We discuss below the rewriting logic specification of the UPTS type checker in some detail. Instead of an equational theory introduced by the `fmod` keyword, the specification takes the form of a rewrite theory, introduced by the `mod` keyword, that has a transition system as its initial semantics:

```
mod PTS[S :: FPTS-SIG] is
```

We reuse most components of the functional module defined before, but in addition to the *typing judgement*

```
op _|-_:_ : Context Trm Trm -> Judgement .
```

we add the following *auxiliary judgements*:

```
op _Sort : Trm -> Judgement .
op (_,_,_)Rule : Trm Trm Trm -> Judgement .
op _=_ : Trm Trm -> Judgement .
op _<->_ : Trm Trm -> Judgement .
op _|-_->:_ : Context Trm Trm -> Judgement .
op _|-(_->:_)(_->:_)->:_ : Context Trm Trm Trm Trm Trm ->
                           Judgement .
```

Recall that, in our setting of PTS signatures without top sorts, T is a type in Γ iff $\Gamma \vdash T : s$. Presupposing that Γ is a well-typed context and A,B are types in Γ, the meaning of the auxiliary judgements is the following: The judgement A `Sort` means that there is an $s \in \mathcal{S}$ such that $A \equiv_\beta s$. The judgement (A, B, s) `Rule` means that there are $s_1, s_2 \in \mathcal{S}$ such that $A \equiv_\beta s_1$, $B \equiv_\beta s_2$ and $(s_1, s_2, s) \in \mathcal{R}$. The judgement A `<->` B just means that $A \equiv_\beta B$. The judgement $\Gamma \vdash M$ `->:` T means that M has an *inferred type* T in Γ. Regarding this refinement of typing judgements we only assume that $\Gamma \vdash M$ `->:` T implies $\Gamma \vdash M : T$, and conversely that $\Gamma \vdash M : T$ implies that $T \equiv_\beta T'$ and $\Gamma \vdash M$ `->:` T' for some T'. Furthermore, the judgement $\Gamma \vdash ((M$ `->:` $S)(N$ `->:` $T))$ `->:` U abbreviates $\Gamma \vdash M$ `->:` S, $\Gamma \vdash N$ `->:` T, and $\Gamma \vdash (M\ N)$ `->:` U. Finally, the judgement $M = N$ just means that M and N are equal terms.

In order to express intermediate goals or queries, like $\Gamma \vdash M$ `->:` ?, that are present in the operational refinement but not in the abstract presentation, we extend terms by explicit metavariables:

```
sort MetaVar .
subsort MetaVar < Trm .
op ? : Qid -> MetaVar .
var MV : MetaVar .
```

In ROUPTSs we use the weak head normal form, calculated by the following function whnf, to check if two terms are convertible, and in particular if a term is convertible to the form s or $\{X : A\}M$. We also use sorts WhNf and WhReducible containing terms in weak head normal form and weak head reducible terms, respectively.

```
sort WhNf WhReducible .
subsort WhNf < Trm .

subsort Sorts < WhNf .
subsort Var < WhNf .
mb  ([X : A] M) : WhNf .
mb  ({X : A} B) : WhNf .
mb  (s N) : WhNf .
mb  (X{m} N) : WhNf .
cmb ((P Q) N) : WhNf if (P Q) : WhNf .
mb  (({X : A} M) N) : WhNf .

subsort WhReducible < Trm .

mb  (([X : A] M) N) : WhReducible .
cmb (M N) : WhReducible if M : WhReducible .

op  whnf : Trm -> Trm? .

ceq whnf(M) = M if M : WhNf .
eq  whnf(([X : A] M) N) = whnf([X := N] M) .
ceq whnf(M N) = whnf(whnf(M) N) if M : WhReducible .
```

A configuration is a conjunctive set of judgements that have to be solved or verified by the type checker. We represent a set of judgements as a list. This allows us to solve goals in a well-defined order, a fact that we exploit later in this section.

```
sort JudgementList .

op emptyJudgementList : -> JudgementList .
subsort Judgement < JudgementList .
op __ : JudgementList JudgementList -> JudgementList
        [assoc id: emptyJudgementList] .

var JS : JudgementList .
```

```
sort Configuration .

op {{_}} : JudgementList -> Configuration .
```

Replacement of metavariables by terms (that is, textual replacement) has the obvious definition, not spelled out here, except for its syntax:

```
op <_:=_>_ : MetaVar Trm Trm -> Trm .
op <_:=_>_ : MetaVar Trm Subst -> Subst .
op <_:=_>_ : MetaVar Trm Context -> Context .
op <_:=_>_ : MetaVar Trm Judgement -> Judgement .
op <_:=_>_ : MetaVar Trm JudgementList -> JudgementList .
```

It is used only in the following *equality elimination rule*, that instantiates a metavariable throughout the entire configuration if it is uniquely determined by an equality:

```
rl [Subst] : {{ (MV = A) JS }} => {{ < MV := A > JS }} .
```

A rule like this is typical of a constraint-based programming approach, and indeed the configuration can be seen as a set of constraints that should be simplified using the subsequent rules [32].

In addition to simplification of constraints by rewrite rules, simplification by equational rewriting also plays a major role in our approach. As an example, the judgement of convertibility between normalizing terms can be checked using `whnf` as follows. In order to avoid redundant reductions we reduce the general problem to a check of convertibility between weak head normal forms (which are treated by the first five rules below). In the case of binders we perform renaming to equalize names.

```
rl  [Conv1] :  {{ (s <-> s) JS }} => {{ JS }} .

rl  [Conv2] :  {{ (X{m} <-> X{m}) JS }} => {{ JS }} .

crl [Conv3] :  {{ ((M N) <-> (M' N')) JS }} =>
               {{ (M <-> M') (N <-> N') JS }}
               if (M N) : WhNf /\ (M' N') : WhNf .

rl  [Conv4] :  {{ ({X : A} T <-> {Y : A'} T') JS }} =>
               {{ (A <-> A')
                  ([X := Y{0}] [shift Y] T <-> T') JS }} .

rl  [Conv5] :  {{ ([X : A] M <-> [Y : A'] M') JS }} =>
               {{ (A <-> A')
                  ([X := Y{0}] [shift Y] M <-> M') JS }} .
```

```
crl [Conv6] :   {{ (M <-> N) JS }} =>
                {{ (whnf(M) <-> N) JS }}
                if M : WhReducible .

crl [Conv7] :   {{ (M <-> N) JS }} =>
                {{ (M <-> whnf(N)) JS }}
                if N : WhReducible .
```

We use two auxiliary judgements to formalize side conditions:

```
rl  [Sort] :    {{ (s Sort) JS }} => {{ JS }} .

rl  [Rule] :    {{ ((s1,s2,MV) Rule) JS }} =>
                {{ (MV = Rules(s1,s2)) JS }} .
```

Each inference rule of OUPTSs gives rise to a rewrite rule obtained by reversing the direction of inference:

```
rl  [Ax] :      {{ (G |- s ->: MV) JS }} =>
                {{ (MV = Axioms(s)) JS }} .

crl [Lookup] :  {{ (G |- X{m} ->: MV) JS }} =>
                {{ (MV = lookup(G,X{m})) JS }}
                if lookup(G,X{m}) .

rl  [Pi] :      {{ (G |- {X : A} B ->: MV) JS }} =>
                {{ (G |- A ->: ?(NEW1)) (?(NEW1) Sort)
                   (G,(X : A) |- B ->: ?(NEW2))
                   ((?(NEW1), ?(NEW2), MV) Rule) JS }} .

rl  [Lda] :     {{ (G |- [X : A] M ->: MV) JS }} =>
                {{ (G |- A ->: ?(NEW1)) (?(NEW1) Sort)
                   (G,(X : A) |- M ->: ?(NEW2))
                   (MV = {X : A} ?(NEW2)) JS }} .

rl  [App1] :    {{ (G |- (M N) ->: MV) JS }} =>
                {{ (G |- M ->: ?(NEW1)) (G |- N ->: ?(NEW2))
                   (G |- (M ->: ?(NEW1))(N ->: ?(NEW2)) ->: MV)
                   JS }} .

rl  [App2] :    {{ (G |- (M ->: {X : A} B)(N ->: A') ->: MV)
                   JS }} =>
                {{ (A <-> A') (MV = [X := N] B) JS }} .
```

The terms `?(NEW1)` and `?(NEW2)` above denote fresh metavariables. Hence rewriting has to be controlled by a simple *strategy*, that *constraints* the possible

rewrites by instantiating the variables `NEW1` and `NEW2` only with fresh names each time a rule is applied. Notice that, in contrast to ordinary variables, where names are taken seriously, we abstract from (i.e. we do not care about) metavariable names, since they do not have a formal status inside UPTSs, but belong instead to the metalevel which is partially made explict in the operational refinement.[12]

According to the explanations given before, the new judgements have certain closure properties w.r.t. $\equiv_\beta$. The following simplification rules allow us to work with (partially) normalized judgements in the inference rules:

```
crl [Norm1] :   {{ (T Sort) JS }} =>
                {{ (whnf(T) Sort) JS }}
                if T : WhReducible .

crl [Norm2] :   {{ ((A,B,T) Rule) JS }} =>
                {{ ((whnf(A),B,T) Rule) JS }}
                if A : WhReducible .

crl [Norm3] :   {{ ((A,B,T) Rule) JS }} =>
                {{ ((A,whnf(B),T) Rule) JS }}
                if B : WhReducible .

crl [Norm4] :   {{ (G |- (M ->: A)(N ->: B) ->: T) JS }} =>
                {{ (G |- (M ->: whnf(A))(N ->: B) ->: T) JS }}
                if A : WhReducible .
```

This completes the definition of the *type-inference* system for judgements of the form $\Gamma \vdash M$ `->:` A. *Type checking* is reduced to type inference in the standard way, that is, $\Gamma \vdash M : A$ is verified by first checking if A is a type in Γ, and if this is the case we then check if A and the inferred type of M are convertible. Exploiting the fact that in PTSs without top sorts each type is contained in some sort, this can be specified by the rule

```
rl  [Aux] :     {{ (G |- M : A) JS }} =>
                {{ (G |- A ->: ?(NEW1)) (?(NEW1) Sort)
                   (G |- M ->: ?(NEW2)) (?(NEW2) <-> A) JS }} .
```

This rule can be slightly optimized by using an adaption of Lemma 3 from [42], which allows us to omit the goal (`?(NEW1) Sort`) on the right hand side, since it is implied by the remaining goals.

Finally, we add rules to check the context typing judgement and the main typing judgement:

---

[12] By a further refinement of the present specification we can obtain a system with takes even metavariables seriously, but this is not necessary for the purposes of this paper.

```
rl  [Ctxt1] : {{ (emptyContext ||-) JS }} => {{ JS }} .

rl  [Ctxt2] : {{ (G,(X : A) ||-) JS }} =>
              {{ (G ||-) (G |- A ->: ?(NEW))
                 (?(NEW) Sort) JS }} .

rl  [Main] :  {{ (G ||- M : A) JS }} =>
              {{ (G ||-) (G |- M : A) JS }} .
endm
```

Again we have omitted the straightforward rule corresponding to Simple, which allows us to check derivability of typing judgements $\Gamma \Vdash_1 M : A$ that disallow multiple occurrences of the same variable in Γ.

To verify a judgement J we start with an initial configuration $\{\{J\}\}$. Either this configuration can be reduced to $\{\{\texttt{emptyJudgementList}\}\}$, meaning that the judgement has been proved, or the final configuration contains unsolved goals giving an informative indication of an error.

Notice that we have not only used inductive definitions to specify PTSs and UPTSs logically, but that, in addition, the operational version of UPTSs given by the rewrite rules above is an inductive definition of a labeled transition system which gives us a more refined view of the type-checking process.

The most important property of a type checker is soundness, i.e., each judgement that has been verified should be derivable in the type system. In fact the formal system has been defined in such a way that the soundness of each of the rewrite rules above relative to OUPTSs can be verified by straightforward inspection of the rules using the meaning of all auxiliary judgements given earlier.

More precisely, let S range over decidable, normalizing, functional, full PTS signatures without top sorts. We denote by **ROUPTS**$_S$ the entailment system in which sentences are rewrites of the form $\{\{JS\}\} \longrightarrow \{\{JS'\}\}$ and such a rewrite is derivable iff it is derivable in the rewrite theory that has been presented above. Then the next proposition gives a sound and complete total correspondence **OUPTS**$_S \curvearrowright$ **ROUPTS**$_S$.

Proposition 45 (Soundness and Completeness of ROUPTSs)
Let M,A be UPTS terms, let Γ be a UPTS context, and let J be one of the judgements $\Gamma \Vdash$, $\Gamma \Vdash M : A$, or $\Gamma \Vdash_1 M : A$. If the rewrite $\{\{J\}\} \longrightarrow$ $\{\{\texttt{emptyJudgement}$
$\texttt{List}\}\}$ is derivable in **ROUPTS**$_S$, then J is derivable in **OUPTS**$_S$ and vice versa.

Proof Sketch. The soundness part follows from the simple observation that for each ROUPTS rewrite rule the right hand side together with its possible condition implies the left hand side under the intended logical interpretation given earlier. The completeness part can be obtained by adapting the inductive proof of Lemma 29 in [52]: Instead of the conventional notion of terms and substitution we have to use CINNI syntax with explicit substitutions, and instead of of PTSs we have to use OUPTSs. $\square$

Executability in the following proposition means that the structural equations are implementable and the remaining equations and membership axioms satisfy the standard variable restriction [13,14]. Since we are interested in completeness of the operational semantics of rewriting logic for the specific goals relevant in our application, we also verify a number of sufficient conditions that are further explained in [37,11,13].

Proposition 46 (Executability of ROUPTSs)
The ROUPTS specification is executable, sort-preserving, equationally confluent, and coherent. Furthermore, the underlying membership equational theory is partially terminating in the sense that all membership, equational, and reduction goals, satisfying the condition that `whnf` is applied only to representations of weak head normalizing UPTS terms, are terminating.

Proof Sketch.   Sort-preservation can be easily checked by inspection of each equation. To verify confluence observe that the entire equational specification is orthogonal and has three subspecifications: (1) the specification of explicit substitutions `[_:=_]`, `[shift_]`, `[lift__]`, and their application `__`, (2) the specification of metavariable substitution `<_:=_>_`, (3) the specification of `whnf`, and (4) the specification of `lookup`. Orthogonality of (2) and (4) is obvious, because there are no critical pairs, and orthogonality of (1) and (3) follows from the fact that critical pairs can be eliminated by a simple transformation, because their conditions are unsatisfiable. Furthermore, there are no critical pairs between (1), (2), (3), and (4), so that we can conclude that the membership equational theory is orthogonal and hence confluent. Similarly, coherence of the entire rewrite theory follows from the absence of critical pairs between equations and rules.

Finally, we show partial termination of the membership equational theory, that is termination of all membership and reduction goals under the condition of the proposition. Termination of membership goals M : `WhNf` and M : `WhReducible` follows by structural induction over the terms M. For the remaining termination proof we again exploit orthogonality of our specification, which implies that it is sufficient to prove termination under an innermost reduction strategy [38]. We use the following strategy: Given a reduction goal M or G, we repeat the following two steps as long as applicable: (a) We reduce it to normal form w.r.t. (1) if this form has not been reached yet, and then (b) we select an arbitrary innermost occurrence of `whnf` or `lookup` and apply one the equations from (3) or (4), respectively. Termination of this strategy follows from termination of Step (a), which holds according to the strong normalization property of CINNI proved in [47,48], and from the fact that `whnf` and `lookup` are either eliminated in Step (b) or replaced by corresponding occurrences with smaller measures. For `whnf`(M) the measure is the minimal number of β-reduction steps necessary to reach the weak head normal form from M, and for `lookup`$(G, X\{m\})$ the measure is the length of the context G.   □

A remarkable property of our specification is that it can be executed efficiently in the sense that we do not need an exhaustive search to verify whether $\{\{J\}\} \longrightarrow \{\{$`emptyJudgementList`$\}\}$ is derivable in **ROUPTS**$_S$. Instead, we

can use a simple execution strategy, i.e. a strategy without backtracking, and there is no additional restriction on the strategy beyond the freshness requirement for metavariables mentioned before. In fact, this is a consequence of confluence and partial termination of the rewrite part of our specification, which is stated in the following proposition.[13]

Proposition 47 (Confluence and Termination of ROUPTSs)
The ROUPTS specification is rewrite-confluent and partially terminating in the sense that all rewrite goals $\{\{J\}\} \to ?$, where J is one of the judgements $\Gamma \Vdash$, $\Gamma \Vdash M : A$, or $\Gamma \Vdash_1 M : A$ with UPTS terms M, A and a UPTS context Γ, are terminating.

Proof Sketch.   Confluence of rewrite rules follows from an analysis of (conditional) critical pairs. In fact, there is only a single nontrivial critical pair generated by the overlapping rules `Conv6` and `Conv7`. Termination follows from structural induction over terms using the fact that `whnf` is only applied to terms M for which the goals

$$(G \mathrel{||-}) \ (G \mathrel{|-} M : \text{?(NEW)}) \ (\text{?(NEW) Sort})$$

have been already verified for some context G. As a consequence, M is well-typed in $\mathbf{ROUPTS}_S$, and by the chain of soundness results given in Propositions 45, 35, and 34, we conclude that M is α-equivalent to a well-typed term in $\mathbf{PTS}_S$, and hence strongly normalizing.   □

5   Final Remarks

In this paper we have given presentations of PTSs at different levels of abstraction. Moreover, we have discussed very natural representations of these systems in MEL or RWL. Both, abstractions and representations are uniformly captured by the general notion of correspondence between entailment systems. Our treatment is guided by the general logics methodology, which explores the space of formal systems by using a particular formal system, in this case rewriting logic, as a logical framework. Our representations of PTSs range from an abstract textbook representation in membership equational logic to a more refined operational representation for a subclass of PTSs in the executable sublanguage of rewriting logic.

Apart from its methodological aspect concerned with the use of rewriting logic as a logical framework to represent higher-order languages, this paper contains a more technical contribution, namely uniform pure type systems, a new variant of PTSs that provides a solution to the known problem with closure under α-conversion in systems with dependent types. Our solution is inspired by earlier work of Pollack, who first pointed out the difficulty to obtain closure

---

[13] Confluence modulo renaming of metavariables would be sufficient in practice, but it happens that, due to the deterministic nature of our specification, we have confluence here in the strongest sense.

under α-conversion if names are taken seriously. By instantiating our operational representation of PTSs, our approach directly leads to an executable prototype of the type theory in Maude. In our view the potential of this approach is by no means confined to formal representations and prototyping, but we think that it provides an interesting alternative to the implementation of type theories and typed higher-order logics, which are traditionally conducted using functional programming languages such as ML.

5.1   The Open Calculus of Constructions

We furthermore would like to point out that the techniques presented in this paper have been applied in the development of the *open calculus of constructions* (OCC) [48], an extension of the calculus of constructions that incorporates rewriting logic and its membership equational logic as a computational sublanguage. Although OCC deviates quite considerably from the prevailing, more conservative line of research in the context of the calculus of constructions, it can be seen as a possible realization of the early ideas in [26] on a marriage of these different paradigms.

OCC is a monomorphic type theory with dependent types and universes that is considerably more liberal than the calculus of constructions and several extensions such as the extended calculus of constructions [29] and the calculus of inductive constructions [17], or the calculus of algebraic constructions [9], but it maintains its core feature, which is also shared by all the remaining PTSs, namely that type checking is ultimately based on a notion of computation. Similar to PTSs, OCC is a family of type theories parameterized by a universe hierarchy, but we have imposed the requirement that impredicative universes can only appear at the bottom of this hierarchy. All other universes are predicative and hence form a monomorphic Martin-Löf-style type theory.

Different from the calculus of constructions, OCC is an *open* type theory in the sense that it is based on an *open computational system*, which can be specified by the user within the bounds provided by its logic. The computational system is of similar flexibility as that of membership equational and rewriting logic, which means in particular that restrictive operational properties, such as confluence and normalization, are in general not enforced by syntactic means. Generalizing the operational semantics of membership equational logic and rewriting logic, OCC supports conditional equations, conditional assertions, and conditional rewrite rules together with an operational semantics based on a combination of conditional rewriting modulo structural equations and exhaustive goal-oriented proof search.

Since OCC contains a higher-order equational programming/specification language with dependent types, and simultaneously a higher-order logic with dependent types by virtue of the propositions-as-types interpretation, our approach can be regarded as a marriage between the first-order paradigm of executable specification languages, such as equational and rewriting logic, and the higher-order paradigm, used in functional programming languages and higher-order logic proof assistants. Without excluding alternative models, we have equipped

OCC with a classical set-theoretic semantics, because it best reflects the pre-vailing practice in mathematics and computer science and also facilitates formal interoperability with many existing classical logic theorem provers.

It is remarkable that in spite of its logical and computational expressiveness, OCC is a rather minimalistic system based on the combination of only two key features: dependent types, and a computational system based on conditional rewriting modulo equations. Therefore, it can also be regarded as a natural higher-order generalization of rewriting logic. A key rationale behind the design of OCC is that an underlying computationally powerful system, like that of MEL and RWL, can increase the degree of automation in theorem proving already at the level of the formal system, rather than delegating the issue of automation entirely to the metalevel by means of tactics. This point is especially important for type theories in the line of PTSs, where type checking does not involve the use of tactics, but is based on the operational semantics of the type theory itself.

Using the techniques developed in this paper we have mapped the higher-order case to the first-order case, and, not surprisingly, we have employed the Maude rewriting engine and its reflective capabilities, to develop an experimen-tal prototype of OCC based on this mapping. The prototype, which can be used as a programming/specification language and as a interactive proof assistant, has been a valuable tool to explore the applications of OCC already in the early phase of its development, and has made possible the study of a wide range of very different examples in various application domains [48]. In summary, our ex-perience indicates that OCC opens a promising new research direction, which we hope will contribute to the long-term goal of a unified language for programming, specification and interactive theorem proving.

5.2   Conclusions

To sum up the main points of this paper, we have shown how a given first-order framework can very naturally and directly express powerful higher-order frame-works and how binders and substitution can be handled in a fully satisfactory way by purely algebraic means. We have also indicated that all this is not only of theoretical interest, but that as a fruit of this study, a proof assistant based on a new framework like OCC that combines the best of higher-order frameworks with the computational flexibility of MEL and RWL has been obtained.

Although we are interested in more complex applications such as OCC and its meta-theoretic properties, in this paper we have focused mainly on PTSs and have emphasized the representational aspects. We believe that choices of formal representation are important in their own right, and a major issue in applying a framework logic like MEL and RWL and in ascertaining the practical value of a logical framework. Apart from the benefit of executability that our last spec-ification of UPTSs enjoys, a formal specification provides the basis for *formal* metatheoretic proofs. Indeed, MEL and RWL together with their initial model semantics provide a very general notion of *equational inductive definitions*, a fact that we exploited for representing several formal systems in this paper. We feel that our work is very much in the spirit of Feferman's first-order approach

of finitary inductive systems [19], but by using equational and rewriting logic our approach puts a particular emphasis on executability. In fact, an important benefit of our use of rewriting logic, compared with informal presentations of algorithms by means of (possibly formal) inductive definitions of derivable judgements, is that the algorithms receive a clear formal status as (labeled) transition systems, which is the basis to express and reason about operational properties such as confluence and termination in a formally rigorous way.

The general problem of carrying out metatheoretic proofs, soundness and completeness proofs being typical examples, often involves the development of useful induction principles on the basis of possibly different but related presentations of the formal system. Such induction principles can be formulated either using an *internal approach*, e.g. by using a formal system such as OCC, which contains the framework logic as a sublogic in a suitable sense, or using an *external approach*, such as the one adopted in Twelf [44], where an external first-order logic is added on top of a higher-order logical framework for inductive reasoning about the representations. In a certain sense similar to the latter, but avoiding its hybrid character, one can instead use a *reflective approach* (cf. the approach to reflective metalogical frameworks presented in [3,4]), which introduces induction principles at the metalevel of the representation in a reflective framework such as rewriting logic.

6   Acknowledgements

References

1. H. P. Barendregt. Lambda-calculi with types. In S. Abramsky, D. M. Gabbay, and T. S. E. Maibaum, editors, *Background: Computational Structures*, volume 2 of *Handbook of Logic in Computer Science*. Claredon Press, Oxford, 1992.
2. B. Barras, S. Boutin, C. Cornes, J. Courant, Y. Coscoy, D. Delahaye, D. de Rauglaudre, J. C . Filliatre, E. Giménez, H. Herbelin, G. Huet, H. Laulhère, C. Muñoz, C. Murthy, C. Parent-Vigouroux, P. Loiseleur, C. Paulin, A. Saïbi, and B. Werner. The Coq Proof Assistent Reference Manual, Version 6.3.1, Coq Project. Technical report, INRIA, 1999. http://logical.inria.fr/.

3. D. Basin, M. Clavel, and J. Meseguer. Reflective metalogical frameworks. In *LFM'99: Workshop on Logical Frameworks and Meta-languages, Paris, France, September 28, 1999, Proceedings*, 1999. http://plan9.bell-labs.com/who/felty/LFM99/.

4. D. Basin, M. Clavel, and J. Meseguer. Rewriting logic as a metalogical framework. In S. Kapoor and S. Prasad, editors, *Twentieth Conference on the Foundations of Software Technology and Theoretical Computer Science, New Delhi, India, December 13–15, 2000, Proceedings*, volume 1974 of *Lecture Notes in Computer Science*, pages 55–80. Springer-Verlag, 2000.

5. Z. Benaissa, D. Briaud, P. Lescanne, and J. Rouyer-Degli. λv, a calculus of explicit substitutions which preserves strong normalisation. *Journal of Functional Programming*, 6(5):699–722, September 1996.

6. S. Berardi. Towards a mathematical analysis of the Coquand-Huet calculus of constructions and other systems in Barendregt's cube. Technical report, Carnegie Mellon University and Universita di Torino, 1988.

7. K. J. Berkling. A symmetric complement to the lambda-calulus. Interner Bericht ISF-76-7, GMD, St. Augustin, Germany, 1976.

8. K. J. Berkling and E. Fehr. A consistent extension of the lambda-calculus as a base for functional programming languages. *Information and Control*, 55:89–101, 1982.

9. F. Blanqui, J.-P. Jouannaud, and M. Okada. The calculus of algebraic constructions. In *Rewriting Techniques and Applications*, Lecture Notes in Computer Science. Springer-Verlag, 1999.

10. A. Bouhoula, J.-P. Jouannaud, and J. Meseguer. Specification and proof in membership equational logic. In M. Bidoit and M. Dauchet, editors, *TAPSOFT'97: Theory and Practice of Software Development, 7th International Joint Conference CAAP/FASE, Lille, Frace, April 1997, Proceedings*, volume 1214 of *Lecture Notes in Computer Science*. Springer-Verlag, 1997.

11. A. Bouhoula, J.-P. Jouannaud, and J. Meseguer. Specification and proof in membership equational logic. *Theoretical Computer Science*, 236:35–132, 2000.

12. A. Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5(1), 1940.

13. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. Quesada. *Maude: Specification and Programming in Rewriting Logic*. SRI International, January 1999. http://maude.csl.sri.com.

14. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. Quesada. A tutorial on maude. http://maude.csl.sri.com, March 2000.

15. T. Coquand. An algorithm for testing conversion in type theory. In G. Huet and G. D. Plotkin, editors, *Logical Frameworks*. Cambridge University Press, 1991.

16. T. Coquand and G. Huet. The calculus of constructions. *Information and Computation*, 76(2/3):95–120, 1988.

17. T. Coquand and C. Paulin-Mohring. Inductively defined types. In *COLOG-88, International Conference on Computer Logic, Tallinn, USSR, December 1988, Proceedings*, volume 417 of *Lecture Notes in Computer Science*. Springer-Verlag, 1990.

18. N. G. de Bruijn. Lambda calculus with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. In *Proceedings Kninkl. Nederl. Akademie van Wetenschappen*, volume 75(5), pages 381–392, 1972.

19. S. Feferman. Finitary inductive systems. In R. Ferro, editor, *Proceedings of Logic Colloquium '88, Padova, Italy, August 1988*, pages 191–220. North-Holland, 1988.

20. P. Gardner. *Representing Logics in Type Theory.* PhD thesis, University of Edinburgh, 1992.
21. H. Geuvers. *Logics and Type Systems.* PhD thesis, University of Nijmegen, 1993.
22. H. Geuvers and M.-J. Nederhof. A modular proof of strong normalization for the calculus of constructions. *Journal of Functional Programming*, 1(2):155–189, April 1991.
23. J. Y. Girard. *Interpretation fonctionelle et elimination des coupures dans l'arithmetique d'ordre superieur.* PhD thesis, Université Paris VII, 1972.
24. R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. In *Second Annual Symposium on Logic in Computer Science, Ithaca, New York, 22–25 June 1987, Proceedings*, pages 193–204. IEEE, 1987.
25. G. Huet. The constructive engine. In R. Narasimhan, editor, *A Perspective in Theoretical Computer Science.* World Scientific, 1989.
26. J.-P. Jouannaud. Membership equational logic, calculus of inductive constructions, and rewrite logic. In *International Workshop on Rewriting Logic and its Applications Abbaye des Prémontrés at Pont-à-Mousson, France, September 1998, Proceedings*, volume 15 of *Electronic Notes in Theoretical Computer Science.* Elsevier, 1998. http://www.elsevier.nl/locate/entcs/volume15.html.
27. P. Lescanne. From $\lambda\sigma$ to λv, a journey through calculi of explicit substitutions. In Hans Boehm, editor, *Conference Record of POPL'94: 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Portland, Oregon, January 17–21, 1994*, pages 60–69. ACM, 1994.
28. P. Lescanne and J. Rouyer-Degli. The calculus of explicit substitutions λv. Technical Report RR-2222, INRIA-Lorraine, January 1994.
29. Z. Luo. *Computation and Reasoning: A Type Theory for Computer Science.* International Series of Monographs on Computer Science. Oxford University Press, 1994.
30. L. Magnussen. *The Implementation of ALF – A Proof Editor based on Martin-Löf's Monomorphic Type Theory with Explicit Substitutions.* PhD thesis, University of Göteborg, Department of Computer Science, 1994.
31. N. Martí-Oliet and J. Meseguer. General logics and logical frameworks. In D. Gabbay, editor, *What is a Logical System?*, pages 355–392. Oxford University Press, 1994.
32. N. Martí-Oliet and J. Meseguer. Rewriting logic as a logical and semantic framework. In *RWLW'96, First International Workshop on Rewriting Logic and its Applications Asilomar Conference Center, Pacific Grove, CA, USA, September 3-6, 1996, Proceedings*, volume 4 of *Electronic Notes in Theoretical Computer Science.* Elsevier, 1996. http://www.elsevier.nl/locate/entcs/volume4.html. To appear in D. M. Gabbay, F. Guenthner, (eds.), *Handbook of Philosophical Logic* (2nd edition), Kluwer Academic Publishers.
33. P. Martin-Löf. An intuitionistic theory of types. Technical report, University of Stockholm, 1972.
34. J. McKinna and R. Pollack. Pure type systems formalized. In M. Bezem and J. F. Groote, editors, *Typed Lambda Calculi and Applications, International Conference on Typed Lambda Calculi and Applications, TLCA '93, Utrecht, The Netherlands, March 16–18, 1993, Proceedings*, volume 664 of *Lecture Notes in Computer Science.* Springer-Verlag, 1993.
35. J. Meseguer. General logics. In H.-D. Ebbinghaus et al., editors, *Logic Colloquium'87, Granada, Spain, July 1987, Proceedings*, pages 275–329. North-Holland, 1989.

36. J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96:73–155, 1992.
37. J. Meseguer. Membership algebra as a logical framework for equational specification. In F. Parisi-Presicce, editor, *Recent Trends in Algebraic Development Techniques, 12th International Workshop, WADT'97, Tarquinia, Italy, June 3–7, 1997, Selected Papers*, volume 1376 of *Lecture Notes in Computer Science*, pages 18 – 61. Springer-Verlag, 1998.
38. M. J. O'Donnell. Computing in systems descibed by equations. In *Fundamentals of Computation Theory, International Conference, Poznañ-Kornik, Poland September 19–23, 1977, Proceedings*, volume 58 of *Lecture Notes in Computer Science*. Springer-Verlag, 1977.
39. L. C. Paulson. *Isabelle*, volume 828 of *Lecture Notes in Computer Science*. Springer Verlag, 1994.
40. R. Pollack. Closure under alpha-conversion. In H. Barendregt and T. Nipkow, editors, *Types for Proofs and Programs: International Workshop TYPES'93, Nijmegen, May 1993, Selected Papers*, volume 806 of *Lecture Notes in Computer Science*, pages 313–332. Springer-Verlag, 1993.
41. R. Pollack. *The Theory of LEGO: A Proof Checker for the Extended Calculus of Constructions*. PhD thesis, University of Edinburgh, 1994.
42. R. Pollack. A verified typechecker. In M. Dezani-Ciancaglini and G. D. Plotkin, editors, *Second International Conference on Typed Lambda Calculi and Applications, Edinburgh, UK, April 10–12, 1995*, volume 902 of *Lecture Notes in Computer Science*. Springer-Verlag, 1995.
43. J. Reynolds. Towards a theory of type structure. In *Programming Symposium, Paris*, volume 19 of *Lecture Notes in Computer Science*. Springer-Verlag, 1974.
44. C. Schürmann and F. Pfenning. Automated theorem proving in a simple meta-logic for LF. In C. Kirchner and H. Kirchner, editors, *Automated Deduction – CADE-15, 15th International Conference on Automated Deduction, Lindau, Germany, July 5–10, 1998, Proceedings*, volume 1421 of *Lecture Notes in Artificial Intelligence*, pages 286–300. Springer-Verlag, 1998.
45. P. G. Severi. *Normalization in Lambda Calculus and its relation to Type Inference*. PhD thesis, Eindhoven University of Technology, 1996.
46. M.-O. Stehr. CINNI - A New Calculus of Explicit Substitutions and its Application to Pure Type Systems. Manuscript, CSL, SRI-International, Menlo Park, CA, USA, 1999.
47. M.-O. Stehr. CINNI – A Generic Calculus of Explicit Substitutions and its Application to λ-, σ- and π-calculi. In K. Futatsugi, editor, *The 3rd International Workshop on Rewriting Logic and its Applications Kanazawa City Cultural Hall, Kanzawa Japan, September 18–20, 2000, Proceedings*, volume 36 of *Electronic Notes in Theoretical Computer Science*, pages 71 – 92. Elsevier, 2000. http://www.elsevier.nl/locate/entcs/volume36.html. Extended version at http://www.csl.sri.com/˜stehr.
48. M.-O. Stehr. Programming, Specification, and Interactive Theorem Proving — Towards a Unified Language based on Equational Logic, Rewriting Logic, and Type Theory. Doctoral Thesis, Universität Hamburg, Fachbereich Informatik, Germany, 2002. http://www.sub.uni-hamburg.de/disse/810/.
49. M.-O. Stehr and J. Meseguer. Pure type systems in rewriting logic. In *LFM'99: Workshop on Logical Frameworks and Meta-languages, Paris, France, September 28, 1999, Proceedings*, 1999. http://plan9.bell-labs.com/who/felty/LFM99/.
50. J. Terlouw. Een nadere bewijstheoretische analyse van GSTTs. Manuscript, University of Nijmegen, The Netherlands, 1989.

51. L. S. van Benthem Jutting. Typing in pure type systems. *Information and Computation*, 105:30–41, 1993.
52. L. S. van Benthem Jutting, J. McKinna, and R. Pollack. Checking algorithms for pure type systems. In H. Barendregt and T. Nipkow, editors, *Types for Proofs and Programs: International Workshop TYPES'93, Nijmegen, May 1993, Selected Papers*, volume 806 of *Lecture Notes in Computer Science*, pages 19–61. Springer-Verlag, 1993.