# Pure Type Systems in Rewriting Logic

Mark-Oliver Stehr José Meseguer

Computer Science Laboratory,
SRI International, Menlo Park, CA 94025, USA

**Abstract.** The logical and operational aspects of *rewriting logic* as a logical framework are 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 $\alpha$-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 in Maude [7] of a proof assistant for OCC, the *open calculus of constructions*, an equational extension of the calculus of constructions.

## 1 Introduction

This paper is a detailed case study on the ease and naturalness with which a family of higher-order formal systems, namely *pure type systems* (PTS) [4, 32], can be represented in the first-order logical framework of rewriting logic [25]. PTS systems generalize the $\lambda$-cube [1], which already contains important calculi like $\lambda\!\!\rightarrow$ [6], F [12, 29], F$\omega$ [12], a system $\lambda$P close to the logical framework LF [13], and their combination, the calculus of constructions CC [8]. PTS systems 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 [11] or via their use as a higher-order logical framework in the spirit of LF [13, 10] or Isabelle [26].

Exploiting the fact that rewriting logic and its membership equational sublogic [5] have initial and free models, we can define the representation of PTS systems 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 PTS logics. Furthermore, the representational versatility of rewriting logic, and of membership equational logic, are also exercised by considering four different representations of PTS systems at different levels of abstraction, from a more abstract textbook version in which terms are identified up to $\alpha$-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 PTS systems by executing the representation in the Maude language [7]. This more concrete version is the basis of a proof assistant for OCC, the *open calculus of constructions*, an equational extension of the calculus of constructions, that is under development.

This case study complements earlier work [20, 21, 22], 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 $\lambda$-calculus, and for binders and quantifiers have already been studied in [20], 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 theoremhood; 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 [10], and in the theory of general logics more liberal, namely *conservative* mappings of entailment systems [24], are studied. In this paper we further generalize conservative mappings to the notion of

a sound and complete total *correspondence of sentences* between two entailment systems. In particular, all the representations of PTS systems that we consider are correspondences of this kind. In fact, sound and complete total correspondences are systematically used not only to state the correctness of the representations of PTS systems 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 total correspondence $\mathcal{F} \rightsquigarrow \mathcal{G}$, representing $\mathcal{F}$ in $\mathcal{G}$. 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 pure type systems include the system $\lambda$P, 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 [10], but there is a conservative mapping of logics from linear logic to rewriting logic [20], 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.

# 2 Preliminaries

## 2.1 Entailment Systems

In the following sections we are concerned with a variety of different interrelated formal systems which can all be viewed as entailment systems, a notion defined in [24] 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. A *unary entailment system* $(Sen, \vdash)$ is a set of *sentences* $Sen$, together with a unary *entailment predicate* $\vdash \subseteq Sen$.

In [24] 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 $(Sen, \vdash)$, $(Sen', \vdash')$ be unary entailment systems. A *correspondence* between $(Sen, \vdash)$ and $(Sen', \vdash')$ is a relation $\rightsquigarrow \subseteq Sen \times Sen'$. Given such a correspondence $\rightsquigarrow$, we say that $\rightsquigarrow$ is *sound* iff for all $\phi \rightsquigarrow \phi'$, $\vdash \phi$ implies $\vdash' \phi'$. Similarly, we say that $\rightsquigarrow$ is *complete* iff for all $\phi \rightsquigarrow \phi'$, $\vdash' \phi'$ implies $\vdash \phi$. Moreover, $\rightsquigarrow$ is called *total* iff for each $\phi \in Sen$ there is a $\phi'$ such that $\phi \rightsquigarrow \phi'$. Correspondences compose in the obvious relational way, giving rise to a category **CEnt**. Often a correspondence of sentences $\rightsquigarrow \subseteq Sen \times Sen'$ takes the form of a function $\alpha : Sen \longrightarrow Sen'$, in which case we speak of a *map of sentences*. A *map of entailment systems* in the sense of [24] is a sound map of sentences, and a *conservative* map is precisely a sound and complete map.

## 2.2 Rewriting Logic and Membership Equational Logic

A rewrite theory is a triple $T = (\Sigma, E, R)$, with $\Sigma$ 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 $t'$ $\Sigma$-terms) which are applied *modulo* the equations $E$. *Rewriting logic* (RWL) then has rules of deduction to infer all possible rewrites provable in a given rewrite theory [25]. Since an equational theory $(\Sigma, E)$ 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) [5] 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 [25, 5]. 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 $(\Sigma, E)$. 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 $(\Sigma, E, R)$. 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 PTS systems.

Membership equational logic together with its categorical initial model semantics provides a very general scheme for *inductive definitions of equational theories* which is much more powerful than the free algebraic data types well-known from many functional programming languages. Beyond that, rewriting logic generalizes equational logic and gives us via its initial semantics a very general scheme for *inductive definitions of rewrite systems*.

# 3 Overview and Main Results

In Section 4 we show how the definition of PTS systems can easily be formalized in membership equational logic. The approach we use is not only less specialized than the one used in a higher-order logical framework like LF [13] or Isabelle [26], but it has also more explanatory power, since we explain higher-order calculi in terms of a first-order system with a straightforward semantics.

In order to make the specification of PTS systems more concrete, we introduce the notion of *uniform pure type systems* (UPTS) [31], that do not abstract from the treatment of names but use CINNI, a new and very simple first-order calculus of names and substitutions. UPTS systems solve the problem of closure under $\alpha$-conversion in a simple and elegant way. Again, a membership equational logic specification of UPTS systems can be given that directly formalizes the informal definition.

As an intermediate step we employ an optimized version of UPTS systems, namely *UPTS with valid contexts* (UPTS/VC). This system contains an explicit judgement for valid contexts, and can be seen as a refinement towards a more efficient implementation of type checking.

Last but not least, we describe how meta-operational aspects of an important class of UPTS/VC systems, like type checking and type inference, can be seen as rewrite systems and can likewise be formalized in rewriting logic. The result of this formalization is an executable specification of UPTS/VC systems that is correct w.r.t. the logical specification given before in a very obvious way. Let us abbreviate the rewriting based presentation of UPTS/VC by RUPTS/VC. A similar presentation is the basis of the proof assistant for the open calculus of constructions mentioned above.

Formally, these different presentations of PTS systems are families of unary entailment systems parameterized by PTS *specifications*. We use the notation $\mathrm{PTS}_S$, $\mathrm{UPTS}_S$, $\mathrm{UPTS/VC}_S$ and $\mathrm{RUPTS/VC}_S$ to denote the entailment systems associated with a PTS specification $S$.

For appropriate PTS specifications $S$ we obtain a chain of sound and complete correspondences

$$\mathrm{PTS}_S \rightsquigarrow \mathrm{UPTS}_S \rightsquigarrow \mathrm{UPTS/VC}_S \rightsquigarrow \mathrm{RUPTS/VC}_S.$$

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

$$\mathrm{PTS}_S \rightsquigarrow \mathrm{RUPTS/VC}_S$$

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

By internalizing derivability w.r.t. a rewrite theory $T$, rewriting logic can be seen as a unary entailment system $(\mathit{Sen}, \vdash)$ with sentences of the form $T \vdash \phi$, where $\phi$ is an equation, a membership or a rewrite. Then, $\vdash T \vdash \phi$ means that $\phi$ is derivable in the theory $T$. Membership equational logic can likewise be seen as the unary entailment system obtained by restricting $T$ to membership equational logic theories and $\phi$ to equations and memberships.

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

$$\mathrm{PTS}_S \rightsquigarrow \mathrm{MEL}, \qquad \mathrm{UPTS}_S \rightsquigarrow \mathrm{MEL}, \qquad \mathrm{UPTS/VC}_S \rightsquigarrow \mathrm{MEL}, \qquad \mathrm{RUPTS/VC}_S \rightsquigarrow \mathrm{RWL}.$$

In all cases the *representational distance* between each type system and its representation is practically zero, that is, both the syntax and the inference system of each type theory are very directly and faithfully represented in the framework logic.

The first correspondence is the representation of PTS systems in membership equational logic given in Section 4. Let $\overline{\text{PTS}_S}$ be the membership equational logic specification of $\text{PTS}_S$. Then, for all PTS judgements $\phi$ of $\text{PTS}_S$ and possible representations $\phi'$ of $\phi$ in membership equational logic, the sentence $\overline{\text{PTS}_S} \vdash \phi'$ is derivable in membership equational logic iff the judgement $\phi$ is derivable in $\text{PTS}_S$. This defines a total, sound and complete correspondence of the form $\text{PTS}_S \rightsquigarrow \text{MEL}$. We are concerned with a correspondence rather than a map of sentences, due to the fact that PTS systems abstract from names, but in the membership equational logic representation names are part of the description of terms, although by adding appropriate equations an equivalent abstraction can be achieved in membership equational logic.

In the remaining three systems $\text{UPTS}_S$, $\text{UPTS/VC}_S$, and $\text{RUPTS/VC}_S$ we do not abstract from names. Hence, the three associated representational correspondences actually take the form of conservative *maps* of entailment systems, i.e., with each judgement of the type system we can associate a unique sentence in membership equational logic or rewriting logic, respectively.

# 4 The Metalogical View of PTS

A PTS *specification* 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*. $S$ will range over PTS specifications.

In PTS systems there is no a priori distinction between terms and types. PTS *(pseudo-)terms* are defined by the following syntax with binders:

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

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 *$\alpha$-convertible* terms, i.e., terms that are equal up to renaming of bound variables, are identified.

A PTS *(pseudo-)context* is a finite list of *declarations*, each of the form $[X : A]$. The empty context is denoted by $[]$ and concatenation is written as juxtaposition. In the following, $\Gamma$ ranges over PTS contexts.

Given a PTS specification $S$, the set of derivable typing judgements of the form $\Gamma \vdash M : T$ is defined *inductively* by the following rules:

$$\frac{}{[] \vdash s_1 : s_2} \quad (s_1, s_2) \in \mathcal{A} \qquad \text{(AX)}$$

$$\frac{\Gamma \vdash A : s}{\Gamma[X : A] \vdash X : A} \quad X \notin \Gamma \qquad \text{(START)}$$

$$\frac{\Gamma \vdash M : A \quad \Gamma \vdash B : s}{\Gamma[X : B] \vdash M : A} \quad X \notin \Gamma \qquad \text{(WEAK)}$$

$$\frac{\Gamma \vdash A : s_1 \quad \Gamma[X : A] \vdash B : s_2}{\Gamma \vdash \{X : A\}B : s_3} \quad (s_1, s_2, s_3) \in \mathcal{R} \qquad \text{(PI)}$$

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

$$\frac{\Gamma \vdash M : \{X : A\}B \quad \Gamma \vdash N : A}{\Gamma \vdash (MN) : [X{:=}A]B} \qquad \text{(APP)}$$

$$\frac{\Gamma \vdash M : A \quad \Gamma \vdash B : s \quad \Gamma \vdash A \equiv B}{\Gamma \vdash M : B} \qquad \text{(CONV)}$$

$X \notin \Gamma$ means that there is no $[X : A] \in \Gamma$ for any $A$. $[X{:=}A]$ is the standard metatheoretic operator for capture-free substitution. In the last rule, $\equiv$ is the usual notion of $\beta$-convertibility which contains $\alpha$-convertibility (this is trivially satisfied in this presentation).

As an example, we can instantiate PTS systems by $\mathcal{S} = \{\mathsf{Prop}, \mathsf{Type}\}$, $\mathcal{A} = \{(\mathsf{Prop}, \mathsf{Type})\}$, and

$$\mathcal{R} = \{(\mathsf{Prop}, \mathsf{Prop}, \mathsf{Prop}), (\mathsf{Prop}, \mathsf{Type}, \mathsf{Type}), (\mathsf{Type}, \mathsf{Prop}, \mathsf{Prop}), (\mathsf{Type}, \mathsf{Type}, \mathsf{Type})\}$$

to obtain the calculus of constructions CC.

This presentation of PTS systems is rather abstract for two reasons: firstly, we are working modulo $\alpha$-conversion, i.e., we identify $\alpha$-equivalent terms, and secondly, we are concerned with an inductive definition of a *set* of derivable judgements, but *not* with an *algorithm* to type-check a particular term.

Mathematically the abstract presentation has an important benefit: It allows us to reason about pure type systems metalogically, without assuming anything about the concrete realization of names. This leads to very general results [1, 33] and frees proofs from unnecessary technical details.

## 4.1 PTS in Membership Equational Logic

In the following specifications, given in Maude syntax, we use the logical semantics of membership equational logic for representing PTS systems exactly as given above; a more operational version suited for use as an implementation is discussed in Section 5.2.

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

```
fth PTS-SPEC 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 specification of CC is given by the following functional module:

```
fmod CC-SPEC 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
```

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

```
fmod PTS[PAR :: PTS-SPEC] is
```

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

```
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 membership equational logic as demonstrated in [20, 22]. 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 [20, 22]. 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 $\alpha$-convertible (we refer to the induced equality as $\alpha$-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 $\beta$-convertibility, which is used in the CONV rule of PTS systems. The following (conditional) memberships, together with the initiality condition, define $\beta$-conversion as the smallest congruence (w.r.t. the term constructors) containing one step $\beta$-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 only judgements of PTS systems 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 [] : -> Context .
op [_:_] : Qid Trm -> Context .
op __ : Context Context -> Context [assoc id : []] .

var G : Context .

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

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

We are now ready to define the inference rules. Formally 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 ([] |- s1 : s2) : Derivable if (s1,s2) : Axioms .

cmb (G [X : A] |- X : A) : Derivable if
    (G |- A : s) : Derivable and not(X in G) .

cmb (G [X : B] |- M : A) : Derivable if
    (G |- M : A) : Derivable and
    (G |- B : s) : Derivable and not(X in G) .

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

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

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

cmb (G |- M : B) : Derivable if
    (G |- M : A) : Derivable and
    (G |- B : s) : Derivable and 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 PTS systems *without making any commitments*. In particular, there was no need to change the syntax or the rules of PTS systems to obtain a faithful representation.

## 4.2 Taking Names Seriously

Although the abstract treatment of names in PTS systems leads to a general metatheory that can be used as a high-level theoretical basis for quite different implementations of PTS systems, there is a price to be paid, namely in that an abstract view necessarily limits the expressivity of the theory. 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, for reasoning about a formal system a more concrete specification that is computationally meaningful is either necessary or useful, e.g., for formalizations in constructive type theories or logics with computational sublanguages.

However, as soon as we give up the identification of $\alpha$-convertible terms and take the inference rules literally, we encounter at least two problems first pointed out in [27].[1]

The *first problem* is that the set of derivable judgements is not closed under $\alpha$-conversion. For instance, adapting an example given for $\lambda\!\!\rightarrow$ in [27], we cannot derive a judgment of the form

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

say in CC, although an $\alpha$-equivalent version where all bound variables are distinct can be derived.

A *second difficulty* reported in [27] is that we want to derive

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

but we should *not* be able to derive

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

However, we cannot derive the first judgement, since the name $X$ in the conclusion of the $\mathsf{LDA}$ 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 $\lambda\!\!\rightarrow$. It uses a more liberal notion of context that allows multiple declarations of the same identifier, the most recent one being visible inside the judgement. Unfortunately, he did not pursue this direction further because of the second difficulty, which appears in the context of PTS systems with dependent types but is not present in $\lambda\!\!\rightarrow$. Concerning $\vdash_{lt}$, he remarks "I don't think we can do the same for PTS."

The solution finally discussed in [27] is the solution employed in the *constructive engine* [14] used in proof assistants such as LEGO [18] and COQ [15] and formalized rigorously in [23]. The 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, PTS systems based on such a hybrid naming scheme are a correct implementation of (abstract) PTS systems as described above. More precisely, PTS systems using the hybrid naming scheme can be seen as particular models of the membership logic specification of PTS systems 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 PTS systems would be much more satisfying and this is the direction we pursue in the following.

## 4.3 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 giving up identification by $\alpha$-conversion.

Consider for instance the formula

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

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

Hence, we are faced with the following problem: a calculus without $\alpha$-equality is not only less abstract, which is an unavoidable consequence of giving up identification by $\alpha$-conversion, but also, depending on the (accidental)

[1] The problem of $\alpha$-conversion also remains unsolved in [19], where a system with dependent types is presented that does not enjoy this property.

choice of names, visibility of (bound) variables may be restricted. It is important to emphazise that visibility is not restricted in the original calculus with $\alpha$-equality, since renaming can be performed *tacitly* at any time.

Clearly, this phenomenon of hiding that occurs in the example above is undesirable[2], because it is not present in the original calculus with $\alpha$-equality. It is merely an accident caused by giving up identification by $\alpha$-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 identifier $X$ 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 will 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

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

meaning that $X_1$ is bound by the outer $\forall$. To make the language a conservative extension of the traditional notation, we identify $X$ and $X_0$. This generalized syntax will be called CINNI syntax, where CINNI refers to *Calculus of Indexed Names and Named Indices* [31], a new and very simple calculus of explicit substitutions to be introduced in the next section.

It might appear that there is some similarity to a notation based on de Bruijn indices [9]. 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 relation 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.[3] In other words, if we restrict the available identifiers to a single one, we obtain de Bruijn's notation as a very special case. In this sense, the CINNI syntax can be formally seen as a proper generalization of de Bruijn's notation. Pragmatically, however, the relation 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 renamings[4] are avoided.

The pragmatic advantage of CINNI 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.

## 4.4 Explicit Substitutions

So far we have presented a simple first-order syntax for expressions 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 is capture-free substitution. Therefore, we now present the CINNI substitution calculus.

Strictly speaking, CINNI is a family of explicit substitution caluli, parameterized by the syntax (including information about binding) of the language we want to represent. Below we present the instantiation of this substitution calculus for the untyped $\lambda$-calculus with terms in CINNI syntax, i.e.

$$X_m \quad | \quad (M\ N) \quad | \quad [X]M$$

As a motivation for the substitution calculus given below, consider the following example of a $\beta$-reduction step in the traditional $\lambda$-calculus with distinct names $X$ and $Y$, again taking names literally, i.e., not presupposing identification by $\alpha$-conversion:

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

[2] Of course, in general hiding is important but it is not an issue of binding; it should be treated independently.
[3] With the slight difference that de Bruijn's indices start at 1 instead of 0.
[4] See the discussion on weird renaming in the next section.

Clearly, $Z$ must be an identifier different from $Y$ to avoid capturing. Unfortunately, there is no canonical choice if all identifiers 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 $\lambda$-calculus. It 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* substitutions $\uparrow_X$, substitutions can be *lifted* using $\Uparrow_X(S)$, where the variable $S$ ranges over substitutions.

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

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

We can instantiate the CINNI calculus to give a more concrete treatment of different 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 $\lambda$-calculus.

Now we can define *$\beta$-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 $\beta$-reduction:

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

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

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

where $\bullet$ is an arbitrary but fixed name. Using this rule every CINNI term can be reduced to a nameless *$\alpha$-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 CINNI syntax contains de Bruijn's indexed notation as a very special case, the instantiation of CINNI for the $\lambda$-calculus reduces to the calculus $\lambda\upsilon$ of explicit substitutions proposed by Pierre Lescanne [16, 17, 3], but only in the degenerate case where we only admit a single identifier. It is noteworthy that $\lambda\upsilon$ is the smallest known indexed substitution calculus enjoying good theoretical properties like confluence 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.

## 4.5 Uniform Pure Type Systems

The application of CINNI to PTS turns out to be surprisingly simple, and indeed it leads to a system which can be seen as Pollack's $\vdash_{lt}$ scaled up to PTS systems.

In contrast to the hybrid approach to PTS systems adopted in the constructive engine [14] and in the formalization [23], both distinguishing between global and local variables, we use indexed identifiers *uniformly*. This suggests defining *uniform pure type systems* (UPTS) by modifying PTS in three steps:

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

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

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

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

The third and final step is to define the derivable typing judgements. Since we do not want to identify $\alpha$-equivalent 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:

$$\frac{\Gamma \vdash A : s}{\Gamma[X : A] \vdash X_0\ : \uparrow_X A} \qquad \text{(START)}$$

$$\frac{\Gamma \vdash M : A \quad \Gamma \vdash B : s}{\Gamma[X : B] \vdash \uparrow_X M\ : \uparrow_X A} \qquad \text{(WEAK)}$$

It might appear that the UPTS systems we have defined above are a specialization of PTS systems, 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 PTS systems where names may occur multiple times in the same 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 UPTS systems. We would also like to point out, that, in particular, we have not touched the LDA rule: the only place where $\alpha$-conversion comes into play is the CONV rule, where $\equiv$ subsumes $\alpha$- and $\beta$- conversion, just as in the original PTS systems.

Finally, we describe how these changes are reflected in the membership equational logic specification.

First, instead of using identifiers as variables we use indexed identifiers. 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 $\alpha$-conversion, 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 and
    (G |- B : s) : Derivable .
```

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

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

$$\frac{\Gamma \vdash M : A}{\Gamma \vdash_1 M : A} \quad \text{if no variable is declared in } \Gamma \text{ more than once.} \qquad (\mathsf{CTXTRESTR})$$

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

Using the terminology introduced in Section 2.1 for entailment systems, each of the following two propositions establishes a total, sound and complete correspondence of the form $\mathrm{PTS}_S \rightsquigarrow \mathrm{UPTS}_S$, where $S$ is an arbitrary PTS specification.

**Proposition 4.1 (Soundness and Completeness of UPTS I)** For all PTS terms $M$,$A$ and PTS contexts $\Gamma$, if the PTS judgement $\Gamma \vdash_1 M : A$ is derivable in $\mathrm{UPTS}_S$ then $\Gamma \vdash M : A$ is derivable in $\mathrm{PTS}_S$ and vice versa.[5]

This proposition implies that UPTS systems are conservative over PTS systems. A slightly weaker but more comprehensive correspondence between PTS and UPTS can be given modulo renaming of variables. For this purpose one can extend the renaming equivalence $\equiv_\alpha$ 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 renaming of declared *and* bound variables. Then we have the following

**Proposition 4.2 (Soundness and Completeness of UPTS II)** For all UPTS terms $M$,$A$, PTS terms $M'$,$A'$, UPTS contexts $\Gamma$ and PTS contexts $\Gamma'$ with $\Gamma \vdash M : A \equiv_\alpha \Gamma' \vdash M' : A'$, if the UPTS judgement $\Gamma \vdash M : A$ is derivable in $\mathrm{UPTS}_S$ then $\Gamma' \vdash M' : A'$ is derivable in $\mathrm{PTS}_S$ and vice versa.

The last proposition implies that, concerning judgements of the form $\Gamma \vdash M : A$, PTS and UPTS are equivalent modulo $\alpha$-equivalence. Hence all (metatheoretic) results about PTS apply to UPTS after apropriate renaming.

[5] Here we make use of the convention introduced in Section 4.3 that ordinary terms (here PTS terms) can be seen as CINNI terms (here UPTS terms).

The proposition also implies that the new form of judgement $\Gamma \vdash_1 M : A$ is not necessary to ensure soundness and could therefore be dropped. Sometimes, however, judgements of the form $\Gamma \vdash_1 M : A$ instead of $\Gamma \vdash M : A$ are more convenient, e.g., to formulate the thinning lemma, since contexts without multiple declarations of the same name can be treated as sets. Hence, both kinds of judgements are useful for metatheoretic reasoning.

## 4.6 A Conservative Optimization

The presentations of pure type systems (PTS and UPTS) 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 $\Gamma$ is a well-typed context. Since in practice checking contexts is as important as checking types, we switch to a conservative extension of UPTS systems 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 valid contexts can be seen as an optimization which avoids unnecessary rechecking of contexts in each subderivation. We will refer to this optimized type system as UPTS *with valid contexts* (UPTS/VC). The only modifications we need are described below. We use judgements of the form $\Gamma \vdash$ (valid context), $\Gamma \vdash M : A$ (weak typing) and $\Gamma \Vdash M : A$ (strong typing) and we add the following rules:

$$\frac{}{[] \vdash} \qquad \text{(CEMPTY)}$$

$$\frac{\Gamma \vdash \quad \Gamma \vdash A : s}{\Gamma[X : A] \vdash} \qquad \text{(CEXT)}$$

$$\frac{}{\Gamma \vdash X_m : \mathrm{lookup}(\Gamma, \mathrm{X_m})} \ \text{if } \mathrm{lookup}(\mathrm{X_m}, \Gamma) \neq \bot \qquad \text{(LOOKUP)}$$

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

where $\bot$ denotes a failure and $\mathrm{lookup}(\Gamma, \mathrm{X_m})$ is defined by

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

Then we replace AX and CTXTRESTR by

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

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

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 [34], but here we are concerned with UPTS systems instead of PTS systems and as a minor difference we make use of an explicit lookup function. Also all freshness side conditions are eliminated thanks to CINNI.

Again, the representation in membership equational logic is quite direct. It nicely illustrates the mixed specification style using equations and memberships:

```
sort Trm? .
subsort Trm < Trm? .
op  undefTrm : -> Trm? .

op  lookup : Context Var -> Trm? .
eq  lookup([], X{m}) = undefTrm .
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  ([] |-) : Derivable .

cmb (G [X : A] |-) : Derivable if
    (G |-) : Derivable and (G |- A : s) : Derivable .

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

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

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

UPTS/VC are equivalent to UPTS, i.e. there is total, sound and complete correspondence of the kind $\text{UPTS}_S \rightsquigarrow \text{UPTS/VC}_S$ for arbitrary PTS specifications $S$, in the following sense:

**Proposition 4.3 (Soundness and Completeness of UPTS/VC)** Let $M$,$A$ be UPTS terms and $\Gamma$ a UPTS context. If the judgement $\Gamma \Vdash M : A$ ($\Gamma \Vdash_1 M : A$) is derivable in UPTS/VC then $\Gamma \vdash M : A$ ($\Gamma \vdash_1 M : A$) is derivable in UPTS and vice versa.

This proposition is similar to Lemma 23 in [34], but here we are considering UPTS instead of PTS systems.

# 5 The Meta-Operational View of PTS

PTS systems can not only be equipped with a logical semantics, e.g., via the proposition-as-types interpretation[6], but, more fundamentally, PTS systems are usually equipped with an operational semantics, defined by an internal notion of functional computation, like $\beta$-reduction. The operational view of PTS systems 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 PTS systems 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 Maude.

We introduce below several classes of PTS specifications giving rise to corresponding PTS systems that are practically interesting and enjoy particulary good properties.

**Definition 5.1** A PTS specification $S$ 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 of a PTS specification is a reasonable requirement to ensure that type inference and type checking do not become undecidable because of a too complex specification $S$.

**Definition 5.2** A PTS specification $S$ 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 specifications, the relations $\mathcal{A}$ and $\mathcal{R}$ can be viewed as functions $\mathcal{A} : \mathcal{S} \to \mathcal{S}?$ and $\mathcal{R} : \mathcal{S} \times \mathcal{S} \to \mathcal{S}?$ where $\mathcal{S}? := \mathcal{S} \cup \{\bot\}$. Functionality ensures that every term has a unique type (up to conversion). The class of functional PTS systems[7] includes, for example, all systems of the $\lambda$-cube.

[6] Of course, we must be careful, since many PTS systems are inconsistent under the propositions-as-types interpretation.

[7] The attributes for PTS specifications are naturally lifted to the corresponding entailment systems.

**Definition 5.3** A PTS specification $S$ 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 specification $S$ 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 PTS systems allow us to form $\{X : A\}B$ types 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 system.

**Definition 5.4** Given a PTS specification $S$, 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 $\mathcal{S}_{\text{top}}$. $S$ is *topless* iff $\mathcal{S}_{\text{top}}$ is empty.

Topless PTS disallow top sorts, which introduce some kind of non-uniformity in the set of sorts. Just as in full PTS specifications $\mathcal{R}$ can be seen as a function $\mathcal{R} : \mathcal{S} \times \mathcal{S} \to \mathcal{S}$, in functional, topless PTS specifications $\mathcal{A}$ can be viewed as a function $\mathcal{A} : \mathcal{S} \to \mathcal{S}$.

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

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

The premises together with the side conditions in LDA' imply that $\{X : A\}B$ is a well-formed type (cf. rule PI). Indeed, as explained in [34] in the context of PTS systems, replacing LDA by LDA' does not change the set of derivable judgements in semi-full UPTS systems.

For full and topless UPTS systems we can eliminate the side conditions in the rule LDA', and we obtain LDA" without changing the set of derivable judgements:

$$\frac{\Gamma \vdash A : s \quad \Gamma[X : A] \vdash M : B}{\Gamma \vdash [X : A]M : \{X : A\}B} \qquad \text{(LDA")}$$

The calculus of constructions has Type as a top sort and therefore is not topless. However, it is straightforward to extend CC by an infinite universe hierarchy yielding a topless PTS.

Together with the introduction of UPTS in the previous section, we have now presented three families of inference systems which only differ in the choice of the rule LDA. For a full and topless PTS specification $S$ all of them define the same unary entailment system, which is denoted by $\text{UPTS}_S$.

In the remainder of this paper we will present a type checking algorithm for a class of UPTS using rewriting logic as a formal specification language. Type checking for PTS is not trivial, but in spite of some unsolved theoretical questions such as the expansion postponement problem, efficient algorithms for the important classes of functional PTS and semi-full PTS (satisfying appropriate decidability and normalization properties) have been presented in [34]. 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 UPTS that are decidable, normalizing[8], functional, full and topless. The class of UPTS systems 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 UPTS instead of PTS is motivated by our desire to obtain a formal representation that takes names seriously and makes type checking more uniform. This is different from [34] that uses names informally for presentation purposes but actually assumes identification by $\alpha$-conversion as justified by the formalization [23] which abstracts from local names by representing them using de Bruijn indices.

## 5.1 UPTS in Membership Equational Logic

The standard way to implement type checking 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 membership equational logic or in any other functional programming language, but the use of membership equational logic is attractive, since it allows us to formulate the logical and operational versions of PTS systems in a single uniform language with an extremely

[8] w.r.t. $\beta$-reduction

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 membership equational logic 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 UPTS systems with decidable type checking can be written in membership equational logic 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 typable term and yields `undefTrm` otherwise. The function can be defined in a way similar to the one given in [30], 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 the constructive engine with its hybrid treatment of names. As an additional advantage, multiple declarations of the same identifier are naturally admitted in contexts (if we use judgement $\Gamma \Vdash M : A$). However, it is also easy to disallow these more general contexts if desired (by implementing the more conventional judgement $\Gamma \Vdash_1 M : A$).

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 to more complex type theories (e.g., extensions of UPTS systems) in a more satisfactory way than the purely functional and equational approaches to type checking.

## 5.2 UPTS in Rewriting Logic

As shown by an extensive collection of examples in [20, 21, 22], 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 clear 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 UPTS systems.

The rewriting logic specification represents RUPTS/VC systems and is able to perform type checking, i.e., to decide derivability of judgements of the form $\Gamma \vdash M : A$ and $\Gamma \vdash$, for the class of decidable, normalizing, functional, full and topless UPTS/VC systems discussed before. As in PTS systems, type checking reduces to type inference, that is, to solving incomplete queries of the form $\Gamma \vdash M \rightarrow : ?T$.

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

First, we exploit our assumption that the PTS specification is decidable, functional, full and topless, which means that the relations $\mathcal{A}$ and $\mathcal{R}$ can be specified by equationally-defined functions `Axioms` and `Rules`:

```
fth PTS-SPEC is
sort Sorts .
op Axioms : Sorts -> Sorts .
op Rules : Sorts Sorts -> Sorts .
endfth
```

As in the syntax-directed approach, we "invert" the inference rules in order to obtain a goal-directed algorithm from the generating inductive definition. In contrast to a purely equational and functional approach, the rewriting logic specification we aim at has rewrite transition systems as models, and can therefore be seen as an operational generalization of the equational and functional paradigms. In contrast to [34], the type-checking algorithm itself receives a direct formal status, which is a prerequisite for reasoning formally about its correctness.

The inductive definition of UPTS systems, e.g., the one in membership equational logic, can also be seen as a static description of a set of judgements that we would like to equip with a dynamic interpretation. More precisely, a *(static) logical implication*

$$A_1 \wedge \cdots \wedge A_n \Rightarrow B$$

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

$$B \to A_1 \wedge \cdots \wedge A_n$$

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 (given by the static description) 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 membership equational logic fragment. It is this mix of different paradigms in a uniform framework that allows us to express the type-checking algorithm in a way that is very close to the logical specification.

In the refined specification we make use of a number of auxiliary judgements:

| Judgement | Meaning |
|---|---|
| $A$ Sort | there is an $s \in \mathcal{S}$ such that $A \equiv s$ |
| $(A, B, s)$ Rule | there are $s_1, s_2 \in \mathcal{S}$ such that $A \equiv s_1$, $B \equiv s_2$ and $(s_1, s_2, s) \in \mathcal{R}$ |
| $A = B$ | $A = B$ literally |
| $A \leftrightarrow B$ | $A \equiv B$ (for $A$ and $B$ normalizing) |
| $\Gamma \vdash M \to\!: A$ | there is an $A'$ with $A \equiv A'$ such that $\Gamma \vdash M : A'$ |
| $\Gamma \vdash ((M \to\!: A)(N \to\!: B)) \to\!: C$ | $\Gamma \vdash M \to\!: A$, $\Gamma \vdash N \to\!: B$ and $\Gamma \vdash (MN) \to\!: C$ |

We discuss below the rewriting logic specification of the UPTS type checker in some detail. Instead of a (purely) functional module, introduced by `fmod`, the specification takes the form of a system module, introduced by `mod`, that has a rewrite system as its initial semantics:

```
mod PTS[PAR :: PTS-SPEC] is
```

We reuse most components of the functional module defined before, but we add the 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 .
```

In order to express intermediate goals or queries, like $\Gamma \vdash M \rightarrow: ?T$, 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 ?T : MetaVar .
```

The use of weak head normal form, calculated by the following function `whnf`, is an efficient way to check whether 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. For sake of brevity we omit the straightforward definitions in membership equational logic.

```
sort WhNf WhReducible .
subsort WhNf < Trm .
subsort WhReducible < Trm .
op whnf : Trm -> Trm? .
```

A configuration is a conjunctive set of judgements that have to be solved or verified by the type checker:

```
sort JudgementSet .

op emptyJudgementSet : -> JudgementSet .
subsort Judgement < JudgementSet .
op __ : JudgementSet JudgementSet -> JudgementSet
        [assoc comm id: emptyJudgementSet] .

var JS : JudgementSet .

sort Configuration .

op {{_}} : JudgementSet -> 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 JudgementSet -> JudgementSet .
```

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

```
rl {{ (?T = A) JS }} => {{ < ?T := 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 [20, 22]. Instead of detecting an inconsistency, the goal is to eliminate all constraints. In addition to simplification of constraints by general rewrite rules, simplification by equational rewriting also plays a major role in our approach.

For example, the judgement of convertibility between normalizing terms can be checked as follows. In order to avoid redundant reductions we reduce the general problem to checking convertibility between weak head normal forms (which are treated by the last three rules below). In the case of binders we perform renaming to equalize names.

```
rl  (T <-> T) => emptyJudgementSet .
crl (M <-> N) => (whnf(M) <-> N) if M : WhReducible .
crl (M <-> N) => (M <-> whnf(N)) if N : WhReducible .
crl (M N) <-> (M' N') => (M <-> M') (N <-> N') if (M N) : WhNf and (M' N') : WhNf .
rl  ({X : A} T <-> {Y : A'} T') => (A <-> A') (T <-> [Y := X{0}] [shift X] T') .
rl  ([X : A] M <-> [Y : A'] M') => (A <-> A') (M <-> [Y := X{0}] [shift X] M') .
```

We use two auxiliary judgements to implement side conditions:

```
rl  (s Sort) => emptyJudgementSet .

rl  ((s1,s2,?T) Rule) => (?T = Rules(s1,s2)) .
```

Each inference rule of UPTS/VC systems gives rise to a rewrite rule obtained by reversing the direction of inference:

```
rl  (G |- s ->: ?T) => (?T = Axioms(s)) .

crl (G |- X{m} ->: ?T) => (?T = lookup(G,X{m})) if lookup(G,X{m}) =/= undefTrm .

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

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

rl  (G |- (M N) ->: ?T) =>
    (G |- (M ->: ?(NEW1))(N ->: ?(NEW2)) ->: ?T)
    (G |- M ->: ?(NEW1)) (G |- N ->: ?(NEW2)) .

rl  (G |- (M ->: {X : A} B)(N ->: A') ->: ?T) =>
    (A <-> A') (?T = [X := N] B) .
```

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 identifiers 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 UPTS systems, but belong instead to the metalevel.[9]

According to the explanations given before, the new judgements have certain conversion closure properties. The following partial normalization rules allow us to work with normalized judgements in the above rules:

```
crl (T Sort) => (whnf(T) Sort) if T : WhReducible .

crl ((A,B,T) Rule) => ((whnf(A),B,T) Rule) if A : WhReducible .

crl ((A,B,T) Rule) => ((A,whnf(B),T) Rule) if B : WhReducible .

crl (G |- (M ->: A)(N ->: B) ->: T) => (G |- (M ->: whnf(A))(N ->: B) ->: T) if A : WhReducible .
```

This completes the definition of the type-inference system for judgements of the form $\Gamma \vdash M \rightarrow: A$. Since our goal was to define the operational counterpart of $\Gamma \vdash M : A$, i.e., to give a type-checking algorithm, we reduce type checking to type inference in the standard way using the conditional rules:

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

```
crl (G |- M : A) => (G |- M ->: ?(NEW2)) (?(NEW2) <-> A)
    if {{ A Sort }} => {{ emptyJudgementSet }} .

crl (G |- M : A) => (G |- M ->: ?(NEW2)) (?(NEW2) <-> A)
    if {{ (G |- A ->: ?(NEW1)) }} => {{ emptyJudgementSet }} .
```

Actually these two rules constitute an operational formulation of Lemma 3 (Characterization of PTS) proved in [28] for PTS. Finally, we add rules in reversed form to check valid contexts and the strong typing judgement:

```
rl  ([] |-) => emptyJudgementSet .

crl (G [X : A] |-) =>
    (G |- A ->: ?(NEW)) (?(NEW) Sort)
    if {{ (G |-) }} => {{ emptyJudgementSet }} .

crl (G ||- M : A) => (G |- M : A)
    if {{ (G |-) }} => {{ emptyJudgementSet }} .

endm
```

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

To verify a judgement $J$ we start with an initial configuration $\{\{J\}\}$. Either this configuration can be reduced to {{emptyJudgementSet}}, meaning that the judgement has been proved, or the final configuration contains unsolved constraints that can be seen an informative indication of a type-checking error.

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

The most important property of a type checker is soundness. The soundness of each of the rewrite rules above is obvious and can be verified by inspection (even by a user of the algorithm who would like to obtain confidence in its correctness) without resorting to difficult metatheoretical proofs.

Let $S$ range over decidable, normalizing, functional, full and topless PTS specifications. RUPTS/VC denotes the rewrite based version of UPTS/VC that has been presented above in terms of rewriting logic. Then the next proposition gives a sound and complete correspondence $\text{UPTS/VC}_S \rightsquigarrow \text{RUPTS/VC}_S$.

**Proposition 5.5 (Soundness and Completeness of RUPTS/VC)** Let $M$,$A$ be UPTS terms, let $\Gamma$ 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 sentence $\{\{J\}\} \longrightarrow \{\{\texttt{emptyJudgementSet}\}\}$ is derivable in $\text{RUPTS/VC}_S$, then $J$ is derivable in $\text{UPTS/VC}_S$ and vice versa.

Completeness as stated above does not immediately imply completeness of the implementation, since the rewrite theory is usually executed using a strategy that restricts the rewrites to those that are actually chosen. Ideally, and this is the case in our specification, there is no additional restriction on the strategy beyond the freshness requirement for metavariables mentioned before.

# 6 Conclusions

In this paper we give presentations of PTS systems at different *levels of abstraction*. Moreover we have discussed very natural *representations* of these systems in membership equational logic or rewriting logic. Both, abstractions and representations are uniformly captured by the notion of a *correspondence* between entailment systems. Apart from this more general contribution that demonstrates how pure type systems can be formally specified using rewriting logic as a logical framework, there are more technical contributions, namely CINNI, a

simple and general *calculus of explicit substitutions*, and UPTS, a *new variant of pure type systems* that can be seen as a new approach to the problems with closure under $\alpha$-conversion in systems with dependent types.

Furthermore, we would like to point out that the techniques presented in this paper are currently being applied in the design and implementation of a proof assistant for OCC, the *open calculus of constructions*, an extension of the calculus of constructions that incorporates equational logic as a computational sublanguage. Similar to membership equational logic, OCC supports conditional equations and conditional assertions together with an operational semantics based on conditional rewriting modulo equations. Using the Maude rewriting engine and its reflective capabilities, we have developed with a modest amount of effort an experimental version of a proof assistant for OCC of acceptable performance that is based on the ideas on CINNI and UPTS presented here.

We conclude with the remark that we have emphasized the representational aspects in this paper, since the choice of the right formal representation is important in its own right and should preceed attempts to give *formal* metatheoretical proofs. There are many interesting properties that should not require complex proofs. For example, soundness is a property that can often be made easy to verify using specification techniques like those employed above. On the other hand, membership equational logic and rewriting logic together with their initial model semantics provide very general notions of *equational inductive definitions*, a fact that has been exploited for representing (inductively defined) formal systems in this paper. The remaining problem of carrying out metatheoretical proofs about such closed formal systems – completeness proofs are one example – requires the development of useful induction principles on the basis of possibly different but related presentations of the formal system. Once appropriate induction principles are found, they can be formulated using either *higher-order logic*, e.g., simply by using a formal system such as OCC as a metalogic, or using *reflective techniques* (cf. the approach to reflective metalogical frameworks presented in [2]).

# 7 Acknowledgements

Support for this work by DARPA and NASA (Contract NAS2-98073), by Office of Naval Research (Contract N00014-96-C-0114), by National Science Foundation Grant (CCR-9633363), and by a DAAD grant in the scope of HSP-III is gratefully acknowledged. We also would like to thank Francisco Durán and Steven Eker for their help concerning theory and practice of Maude and, furthermore, Manuel Clavel, Narciso Martí-Oliet and the anonymous referees for their constructive criticism.

# 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*. Oxford: Clarendon Press, 1992.

[2] D. Basin, M. Clavel, and J. Meseguer. Reflective metalogical frameworks. To appear in *Proc. of LFM'99: Workshop on Logical Frameworks and Meta-languages, September 28, 1999, Paris, France.*

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

[4] 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.

[5] 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 Int. Joint Conf. CAAP/FASE, Lille, France, April 1997, Proceedings*, volume 1214 of *LNCS*. Springer-Verlag, 1997.

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

[7] 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`.

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

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

[10] P. Gardner. *Representing Logics in Type Theory.* PhD thesis, University of Edinburgh, 1992.

[11] H. Geuvers. *Logics and Type Systems*. PhD thesis, University of Nijmegen, 1993.

[12] J. Y. Girard. *Interpretation fonctionelle et elimination des coupures dans l'arithmetique d'ordre superieur*. PhD thesis, Université Paris VII, 1972.

[13] R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. In *Second Annual Symposium on Logic in Computer Science*, pages 193 – 204, Ithaca, New York, 22–25 June 1987. IEEE Computer Society.

[14] G. Huet. The constructive engine. In R. Narasimhan, editor, *A Perspective in Theoretical Computer Science*. World Scientific Publishing, 1989.

[15] G. Huet, C. Paulin-Mohring, et al. The Coq Proof Assistent Reference Manual, Version 6.2.4, Coq Project. Technical report, INRIA, 1999. `http://pauillac.inria.fr/coq/`.

[16] P. Lescanne. From $\lambda\sigma$ to $\lambda v$, a journey through calculi of explicit substitutions. In Hans Boehm, editor, *Proc. 21st Annual ACM Synposium on Principles of Programming Languages, Portland, USA*, pages 60–69. ACM, 1994.

[17] P. Lescanne and J. Rouyer-Degli. The calculus of explicit substitutions $\lambda v$. Technical Report RR-2222, INRIA-Lorraine, January 1994.

[18] Z. Luo and R. Pollack. Lego proof development system: User's manual. LFCS Technical Report ECS-LFCS-92-211, University of Edinburgh, 1992.

[19] 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, Dept. of Computer Science, 1994.

[20] N. Martí-Oliet and J. Meseguer. Rewriting logic as a logical and semantic framework. Technical Report SRI-CSL-93-05, SRI International, Computer Science Laboratory, August 1993. To appear in D. Gabbay, ed., *Handbook of Philosophical Logic*, Kluwer Academic Publishers.

[21] 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.

[22] N. Martí-Oliet and J. Meseguer. Rewriting logic as a logical and semantic framework. In J. Meseguer, editor, *Proc. First Intl. Workshop on Rewriting Logic and its Applications*, volume 4 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 1996. `http://www.elsevier.nl/cas/tree/store/tcs/free/noncas/pc/volume4.htm`.

[23] J. McKinna and R. Pollack. Pure type systems formalized. In M. Bezem and J.F. Groote, editors, *International Conference on Typed Lambda Calculi and Applications, TLCA '93, Utrecht, The Netherlands, March 16-18, 1993*, volume 664 of *LNCS*, 1993.

[24] J. Meseguer. General logics. In H.-D. Ebbinghaus et al., editors, *Proceedings, Logic Colloquium, 1987*, pages 275–329. North-Holland, 1989.

[25] J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.

[26] L. C. Paulson. *Isabelle*, volume 828 of *LNCS*. Springer Verlag, 1994.

[27] R. Pollack. Closure under alpha-conversion. In Henk Barendregt and Tobias Nipkow, editors, *Types for Proofs and Programs: International Workshop TYPES'93, Nijmegen, May 1993, Selected Papers*, volume 806 of *LNCS*, pages 313–332. Springer-Verlag, 1993.

[28] 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 *LNCS*. Springer-Verlag, 1995.

[29] J. Reynolds. Towards a theory of type structure. In *Programming Symposium, Paris*, volume 19 of *LNCS*. Springer-Verlag, 1974.

[30] P.G. Severi. *Normalization in Lambda Calculus and its relation to Type Inference*. PhD thesis, Eindhoven University of Technology, 1996.

[31] 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.

[32] J. Terlouw. Een nadere bewijstheoretische analyse van GSTTs. Manuscript, University of Nijmegen, The Netherlands, 1989.

[33] B. van Benthem Jutting. Typing in pure type systems. *Information and Computation*, 105:30–41, 1993.

[34] B. van Benthem Jutting, J. McKinna, and R. Pollack. Checking algorithms for pure type systems. In Henk Barendregt and Tobias Nipkow, editors, *Types for Proofs and Programs: International Workshop TYPES'93, Nijmegen, May 1993, Selected Papers.*, volume 806 of *LNCS*, pages 313–332. Springer-Verlag, 1993.