Extension of ML type system with a sorted equation theory on types

Didier Rémy

UNITÉ DE RECHERCHE
INRIA-ROCQUENCOURT

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P. 105
78153 Le Chesnay Cedex
France
Tél.:(1)39 63 55 11

Rapports de Recherche

N°1766

Programme 2

Calcul symbolique, Programmation et Génie logiciel

EXTENSION OF ML TYPE SYSTEM WITH A SORTED EQUATIONAL THEORY ON TYPES

Didier Rémy

Octobre 92

Extension of ML type system with a sorted equational theory on types

Didier Rémy
INRIA-Rocquencourt*

*Author's address: INRIA, B.P. 105, F-78153 Le Chesnay Cedex. Email: Didier.Remy@inria.fr

Extension of ML type system with a sorted equational theory on types

Abstract

We extend the ML language by allowing a sorted regular equational theory on types for which unification is decidable and unitary. We prove that the extension keeps principal typings and subject reduction. A new set of typing rules is proposed so that type generalization is simpler and more efficient. We consider typing problems as general unification problems, which we solve with a formalism of unificands. Unificands naturally deal with sharing between types and lead to a more efficient type inference algorithm. The use of unificands also simplifies the proof of correctness of the algorithm by splitting it into more elementary steps.

Extension du système de type de ML par une théorie équationnelle avec sortes sur les types

Résumé

Le typage du langage ML est étendu en considérant les types modulo une théorie équationnelle régulière avec sortes pour laquelle l'unification est décidable. Cette extension conserve la propriété d'avoir un type principal ainsi que la conservation de la propriété d'être bien typé par β-réduction des programmes. Un nouvel ensemble de règles de typage est proposé de telle sorte que la généralisation de type est à la fois plus simple et plus efficace. Les problèmes de typage sont considérés comme des problèmes généraux d'unification que nous résolvons en utilisant le formalisme des unificandes. Les unificandes expriment naturellement le partage entre les types et conduit à un algorithme d'inférence de types plus efficace. L'utilisation des unificandes simplifie également les preuves de correction des algorithmes en les séparant des étapes élémentaires.

Introduction

The ML language introduced by R. Milner in 1978 has become very poplular, which is partly due to the possibility of inferring principal types for programs [DM82, Dam85]. This is a consequence of the restriction of arbitrary quantification as in the polymorphic lambda calculus to prenex-quantification in ML. However, viewing ML as a restriction of the polymorphic lambda calculus is sometimes misleading: ML terms are untyped terms (Church's view) while terms of the polymorphic lambda calculus are usually considered as typed terms (Curry's view). A consequence of the Church view of ML is that the order in which quantifiers appear in types is insignificant, and useless quantifications over non free variables can be removed, which leads to a simpler formulation of the typing rules, where type schemes are simply pairs of a set of bound variables and a simple type.

Some presentations of the ML typing rules, called "syntax directed", are such that all typing derivations of the same program have the same structure. They are usually used as the basis for finding type inference algorithms. Typechecking in ML can also be defined as typechecking with simple types after reduction of all LET redexes. This approach, which is closer to the conjunctive-type discipline than to the polymorphic lambda-calculus, simplifies the typing rules, but unfortunately, it leads to inefficient algorithms. All these variants of the typing rules have been used in the literature, but their equivalence with the Damas-Milner's formulation has not always been shown. Moreover, some of them have been taken as the basis for extensions of ML for which complete proofs of the main properties are often omitted.

The main goal of this article is to study one of these extensions: Types of ML are taken modulo an equational theory. The main result is theorem 4 that claims principal typings in such an extension. The proof is constructive and comes with the accompanying algorithm 3 for type inference. This extension nicely formalizes type abbreviations in ML. More importantly, there is a type system for extensible records in ML that uses an equational theory on types [Rém91].

The typing rules are those of the Damas Milner type system, and the most common variations from the original presentation are shown equivalent. Indeed this includes the core ML language as an instance. All classical results of the core ML language also hold for this extension.

Type inference is closely related to unification by considering typing problems as unificands. In this presentation, proofs of soundness and correctness of the type inference algorithm are simpler. The algorithm itself is described as a set of transformations on unificands. This approach is more general and control is flexible. The standard algorithm is obtained when priority is given to unification problems and typing problems are reduced in a leftmost outermost order. With fewer control, transformation rules can be easily interpreted as a logic program, similarly to the one given for Mini-ML [CDDK86]. But the benefit of treating typing problems as unificands is essentially the efficiency obtained by sharing as much as possible between types, and in all phases of type inference. This results from the possibility of viewing canonical system of multi-equations as terms, substitutions or graphs, interchangeably.

Another contribution of this work is to avoid a source of inefficiency that is present in the classical syntax-directed typing rules: Generalizable variables are defined as those that do not appear in the typing context. The cost of generalization may increase with the size of the context, and especially with the size of types bound in the context. By assigning ranks to variables as a measure of their freshness we obtain a new syntax-directed system where generalization only applies to freshest variables. Ranks can be computed incrementally by extending unification to ranked unification.

In the first section, we present our extension of ML and the typing rules with different formulations that are shown equivalent. In section 2, a more efficient syntax-directed system is

presented and proved equivalent to the previous formulations. Unification as transformations of unificands is recalled in section 3 and extended to ranked unification. In section 4, we study type inference as the simplification of typing problems; we give a set of transformation rules that reduce all typing problems to unification problems. A few extensions to the core language, and some instances with practical equational theories are described in the last section. A real implementation of the algorithm based on unificands and ranked unification is described in the appendix A.

Definitions are not delimited by theorem like structures, but are indicated by *emphasizing* the defining occurrences of mathematical words.

1 The original Damas-Milner's system and its many variants

In this section, we present the original Damas-Milner's, and some of its many variants. We prove the equivalence of all presentations.

The language of *expressions* of ML is defined by the following BNF grammar:

$M ::=$	Terms	M, N	
x	Variable	x, y	VAR
$\mid \lambda\, x.\, M$	Abstraction		FUN
$\mid M\ M$	Application		APP
$\mid$ *let* $x = M$ *in* M	Let binding		LET

The expressions of ML are considered modulo α-conversion. The substitution of x by N in M is noted $[N/x]M$. If necessary, it renames bound variables of M in order to avoid capture.

When adding axioms to types it is sometime useful to restrict types by sorts: they may restrict the equality on terms by leaving fewer instances of the axioms. An example is given in section 5. Thus we consider sorted types.

Let $\mathcal{K}$ be a set of atoms called sorts, containing at least one element ι. Letter κ ranges over sorts. Signatures are non empty sequences of sorts. We write κ for an atomic signature, and $\kappa_1 \otimes \ldots \kappa_p \Rightarrow \kappa_0$ for a longer sequence. We are also given a set of symbols $\mathcal{C}$ with a mapping Σ from $\mathcal{C}$ to signatures, also called the signature of $\mathcal{C}$. The arity of a symbol is the predecessor of the length of its signature. Letters f, g and h range over symbols. We assume that $\mathcal{C}$ contains at least a distinguished infix arrow symbol $\rightarrow$ of signature $\iota \otimes \iota \Rightarrow \iota$. Let $\mathcal{V}$ be an enumerable set of variables with infinitely many variables of every sort $\mathcal{V}^\kappa_{\kappa \in \mathcal{K}}$. We write $\mathcal{W}$ the set of subsets of $\mathcal{V}$. Types of ML are terms of the free sorted algebra $\mathcal{T}(\Sigma, \mathcal{V})$. We abbreviate $\mathcal{T}(\Sigma, \mathcal{V})$ as $\mathcal{T}$, and write for $\mathcal{T}(W)$ the set of terms whose variables are in W, and $\mathcal{T}^\kappa$ the set of terms of sort κ. Letters τ, σ and ρ range over arbitrary types. We write $\mathcal{V}(\tau)$ the set of (free) variables of τ and $\mathcal{V}^\kappa(\tau)$ those of sort κ.

The non sorted case is obviously an instance of the restricted sorted case where $\mathcal{K}$ is reduced to the sort ι, and signatures are reduced to arities. The signature of the arrow symbol and the restriction on the signature of typing judgements below make it possible to think in term of arities, identifying the sets $\mathcal{T}^\iota$ and $\mathcal{T}$. Places were the sorted case has to be considered more carefully are emphasized everywhere.

Substitutions are sort preserving mappings from a finite set of variables to terms. There are extended to mappings from terms to terms by compatibility with the structure of algebra. Letters μ, ν and ξ range over substitutions of $\mathcal{T}$; $\mathcal{D}(\mu)$ and $\mathcal{I}(\mu)$ are the domain and the codomain of μ, respectively. We also write $\mu : X \rightarrow Y$ for a substitution of domain X that ranges in Y. The one element substitution that sends α to τ is simply written $\alpha \mapsto \tau$. If W is a set of variables, $\mu \restriction W$ is the restriction of μ to W and $\mu \setminus W$ is the restriction of μ outside of W, i.e. $\mu \restriction (\mathcal{V} \setminus W)$.

We extend types of ML by taking them modulo an equational theory. A *regular* equational theory E is one such that two equal terms always have the same set of variables. We assume that a *regular* equational theory E on the algebra $\mathcal{T}$ is provided. We write $\mathcal{T}/E$ for the terms taken modulo the equations. We often consider the terms of $\mathcal{T}$ and manipulate their equality explicitly.

Since different type systems will use slightly different definition of type schemes, we parameterize the definitions by a set of type schemes Q. Type schemes are sorted just like types of $\mathcal{T}$. A $\mathcal{Q}$-context with respect to a set $\mathcal{Q}$ is a partial function from the set of term variables to $\mathcal{Q}$. If A is a Q-context, x is a variable and q is an element of $\mathcal{Q}$, we write $A[x : q]$ the function that is equal to A everywhere except on x to which it associates q. A one-element context $[x : q]$ is called an assertion. We write $\mathcal{C}(\mathcal{Q})$ for the set of Q-contexts.

Contexts need not have finite domains. For type inference though, there must exist an algorithm that given a term variable returns the type to which it is bound in the context, or fails if the variable is not bound in the the context, and a semi-algorithm that enumerates type variables that are not in the free variables ofa the context. A *typing relation* $(\_ \vdash \_ : \_)$ is a subset of $\mathcal{C}(\mathcal{Q}^\iota) \times \mathrm{ML} \times \mathcal{Q}^\iota$.

Typing relations are only defined for contexts and type-objects of the sort ι. An inference rule is an implication R, parameterized by a relation $\vdash$ in $\mathcal{C}(\mathcal{Q}^\iota) \times \mathrm{ML} \times \mathcal{Q}^\iota$, written $\frac{A}{B}(\mathrm{R})$ where A and B are typing relations. The notation $\vdash$ stands for an arbitrary relation: it does not make sense to ask whether a rule that depends on $\vdash$ is true or false until the parameter $\vdash$ is filled with a defined relation $\vdash_{\mathcal{R}}$. A typing relation $\vdash_{\mathcal{R}}$ is defined as the smallest relation that satisfies a set of inference rules $\mathcal{R}$. In typing rules, we use concrete syntax for ML terms. However the typing relation must be stable with respect to α-conversion of programs. This will be the case for all typing relations that will be defined here. In fact, terms could be represented using De Bruijn indices; contexts would then be just sequences of type schemes.

A derivation of a judgement $A \vdash_{\mathcal{R}} M : q$ is a constructive proof of this judgement where each step is an instance of an inference rules. Any valid judgement has at least one derivation. A proof of a judgement by structural induction is a proof by induction on the number of steps then by cases on the last rule used in a derivation of the judgement.

The set of Damas-Milner *type schemes* is the smallest set $\mathcal{S}$ that contains $\mathcal{T}$ and that is closed by

$$s \in \mathcal{S}, \alpha \in \mathcal{V} \Longrightarrow \forall\, \alpha \cdot s \in \mathcal{S}$$

The free variable function $\mathcal{V}$ is extended to type schemes by

$$\mathcal{V}(\forall\, \alpha \cdot s) = \mathcal{V}(s) \setminus \{\alpha\}$$

Substitutions are also extended to type schemes by

$$\mu(\forall\, \alpha \cdot s) = \forall\, \alpha \cdot (\mu \setminus \{\alpha\})(s)$$

and to contexts as pointwise substitution. The sort of a type scheme is the one of the type obtained by stripping all quantifiers.

The original Damas-Milner typing relation for ML is defined on $\mathcal{C}(\mathcal{S}^\iota) \times \mathrm{ML} \times \mathcal{S}^\iota$ by the set of the inference rules of figure 1.

Originally, type schemes are not equal modulo the re-ordering of quantifiers and renaming of bound variables. We define $\forall$-*equality* on type schemes as the smallest congruence that contains all pairs

$$\forall\, \alpha \cdot \forall\, \beta \cdot s =_\forall \forall\, \beta \cdot \forall\, \alpha \cdot s$$

$$\text{if } \beta \notin \mathcal{V}(s) \qquad \forall\, \beta \cdot s =_\forall s \qquad \text{and} \qquad \forall\, \alpha \cdot s =_\forall \forall\, \beta \cdot (\alpha \mapsto \beta)(s)$$

We naturally extend $\forall$-equality to contexts.

$$\frac{x : s \in A}{A \vdash_{\mathrm{DM}} x : s} \ (\textsc{Var}) \qquad \frac{A \vdash_{\mathrm{DM}} x : \forall\, \alpha \cdot s}{A \vdash_{\mathrm{DM}} x : (\alpha \mapsto \sigma)(s)} \ (\textsc{Inst})$$

$$\frac{A[x : \tau] \vdash_{\mathrm{DM}} M : \sigma \qquad \tau \in \mathcal{T}}{A \vdash_{\mathrm{DM}} \lambda\, x.\, M : \tau \to \sigma} \ (\textsc{Fun}) \qquad \frac{A \vdash_{\mathrm{DM}} M : \sigma \to \tau \qquad A \vdash_{\mathrm{DM}} N : \sigma}{A \vdash_{\mathrm{DM}} M\ \ N : \tau} \ (\textsc{App})$$

$$\frac{A \vdash_{\mathrm{DM}} x : s \qquad \alpha \notin \mathcal{V}(A)}{A \vdash_{\mathrm{DM}} x : \forall\, \alpha \cdot s} \ (\textsc{Gen}) \qquad \frac{A \vdash_{\mathrm{DM}} M : s \qquad A[x : s] \vdash_{\mathrm{DM}} N : \sigma}{A \vdash_{\mathrm{DM}} \mathit{let}\ x = M\ \mathit{in}\ N : \sigma} \ (\textsc{Let})$$

$$\frac{A \vdash_{\mathrm{DM}} M : \sigma \qquad \sigma =_E \tau}{A \vdash_{\mathrm{DM}} M : \tau} \ (\textsc{Equal})$$

Figure 1: Damas-Milner's Typing rules (DM)

It is immediate to prove

$$\frac{A \vdash_{\mathrm{DM}} M : s \qquad s =_\forall s'}{A \vdash_{\mathrm{DM}} M : s'} \qquad (\forall\text{-}\textsc{Equal})$$

Moreover, the property extends to contexts:

$$\frac{A \vdash_{\mathrm{DM}} M : s \qquad A =_\forall B}{B \vdash_{\mathrm{DM}} M : s} \qquad (\forall\text{-}\textsc{Context})$$

The proof is an easy structural induction on the derivation of $A \vdash_{\mathrm{DM}} M : s$. Therefore, type schemes can be taken modulo $\forall$-equality. We now consider substitutions modulo $\forall$-equality. We write $\forall\, \{\alpha\} \cup W \cdot \tau$ for $\forall\, \alpha \cdot \forall\, W \cdot \tau$ or also $\forall\, \alpha, W \cdot \tau$ when W does not contain α. Consistently, with $\forall$-equality, we take $\forall\, W \cdot \tau$ to be $\forall\, (\mathcal{V}(\tau) \cap W) \cdot \tau$ when W is infinite. We write $\forall\, \mathcal{W} \cdot \mathcal{T}$ the set of type schemes modulo $\forall$-equality.

We call DM' the set of rules DM modulo $\forall$-equality. It is defined on $\mathcal{C}(\forall\, \mathcal{W} \cdot \mathcal{T}^\iota) \times \mathrm{ML} \times \forall\, \mathcal{W} \cdot \mathcal{T}^\iota$ by the set of inference rules of figure 2. Using lemmas $\forall$-Equal and $\forall$-Context,

$$\frac{x : \forall\, W \cdot \tau}{A \vdash_{\mathrm{DM}'} x : \forall\, W \cdot \tau} \ (\textsc{Var}) \qquad \frac{A \vdash_{\mathrm{DM}'} x : \forall\, \alpha, W \cdot \tau}{A \vdash_{\mathrm{DM}'} x : \forall\, W \cdot (\alpha \mapsto \sigma)(\tau)} \ (\textsc{Inst})$$

$$\frac{A[x : \tau] \vdash_{\mathrm{DM}'} M : \sigma \qquad \tau \in \mathcal{T}}{A \vdash_{\mathrm{DM}'} \lambda\, x.\, M : \tau \to \sigma} \ (\textsc{Fun}) \qquad \frac{A \vdash_{\mathrm{DM}'} M : \sigma \to \tau \qquad A \vdash_{\mathrm{DM}'} N : \sigma}{A \vdash_{\mathrm{DM}'} M\ \ N : \tau} \ (\textsc{App})$$

$$\frac{A \vdash_{\mathrm{DM}'} x : \forall\, W \cdot \tau \qquad \alpha \in \mathcal{V}(\tau) \setminus W \setminus \mathcal{V}(A)}{A \vdash_{\mathrm{DM}'} x : \forall\, \alpha, W \cdot s} \ (\textsc{Gen})$$

$$\frac{A \vdash_{\mathrm{DM}'} M : s \qquad A[x : s] \vdash_{\mathrm{DM}'} N : \sigma}{A \vdash_{\mathrm{DM}'} \mathit{let}\ x = M\ \mathit{in}\ N : \sigma} \ (\textsc{Let})$$

$$\frac{A \vdash_{\mathrm{DM}'} M : \sigma \qquad \sigma =_E \tau}{A \vdash_{\mathrm{DM}'} M : \tau} \ (\textsc{Equal})$$

Figure 2: Variant of Damas Milner's typing rules (DM)

the relations $\vdash_{\mathrm{DM}}$ and $\vdash'_{\mathrm{DM}}$ are easily proved equal modulo $\forall$-equality. The system DM' has still unnecessary freedom on places where generalization and instantiation may occur.

We now consider typing relations in $\mathcal{C}(\forall\, \mathcal{W} \cdot \mathcal{T}^\iota) \times \mathrm{ML} \times \mathcal{T}^\iota$, where type schemes may appear only in contexts. The system S, composed of the rules FUN, APP and EQUAL of figure 2 plus the rules of figure 3, is such that at most one rule other than the equality rule can apply for each syntactic construct of the language; it is said syntax-directed.

$$\frac{x : \forall\, W \cdot \tau \in A \qquad \mu : W \to \mathcal{T}}{A \vdash_S x : \mu(\tau)} \qquad \text{(VAR-INST)}$$

$$\frac{A \vdash_S M : \tau \qquad A[x : \forall\, W \cdot \tau] \vdash_S N : \sigma \qquad W \cap \mathcal{V}(A) = \emptyset}{A \vdash_S \mathit{let}\ x = M\ \mathit{in}\ N : \sigma} \qquad \text{(GEN-LET)}$$

Figure 3: Syntax-directed rules for system (S)

Lemma 1 (Substitution lemma in S) *The relation $\vdash_S$ is stable by substitution.*

$$\frac{A \vdash_S M : \tau}{\mu(A) \vdash_S M : \mu(\tau)} \qquad \text{(SUB)}$$

A weaker instance of the lemma is often used:

$$\frac{A \vdash_S M : \tau \qquad \mathcal{D}(\mu) \cap \mathcal{V}(A) = \emptyset}{A \vdash_S M : \mu(\tau)} \qquad \text{(SUB')}$$

Proof: See section 2.1. ■

Lemma 2 *If W is disjoint from $\mathcal{V}(A)$, the judgements $A \vdash_{\mathrm{DM}'} M : \forall\, W \cdot \tau$ and $A \vdash_S M : \tau$ are equivalent.*

Proof: The rule VAR-INST can be proved for $\vdash_{\mathrm{DM}'}$, using the rules VAR and INST. The rule GEN-LET can be proved for $\vdash_{\mathrm{DM}'}$ using the rules GEN and LET. This shows that $\vdash_S$ is a subrelation of $\vdash_{\mathrm{DM}'}$. Conversely, we reason by structural induction on the judgement $A \vdash_{\mathrm{DM}'} M : \forall\, W \cdot \tau$.

Case VAR, FUN, APP, EQUAL: The judgement $A \vdash_S M : \tau$ follows from the induction hypothesis applied to all premises of the last rule in the derivation of $A \vdash_{\mathrm{DM}'} M : \tau$, and the corresponding rule in S.

Case GEN: The judgement $A \vdash_S M : \tau$ follows directly from the induction hypothesis applied to the premise.

Case INST: After renaming, the derivation in DM$'$ ends with

$$\frac{A \vdash_{\mathrm{DM}'} M : \forall\, \alpha, W \cdot \tau}{A \vdash_{\mathrm{DM}'} M : \forall\, W \cdot (\alpha \mapsto \sigma)(\tau)}$$

where W and α are disjoint from $\mathcal{V}(A)$. By the induction hypothesis, we know that $A \vdash_S M : \tau$ holds. We conclude by applying the SUB' rule with the substitution $\alpha \mapsto \sigma$

Case LET: The derivation in DM$'$ is of the form

$$\frac{A \vdash_{\mathrm{DM}'} M : \forall\, W \cdot \tau \qquad A[x : \forall\, W \cdot \tau] \vdash_{\mathrm{DM}'} N : \sigma}{A \vdash_{\mathrm{DM}'} \mathit{let}\ x = M\ \mathit{in}\ N : \sigma} \qquad \text{(LET)}$$

After renaming, we assume that W is disjoint from A. Applying the induction hypothesis to both premises produces:

$$A \vdash_S M : \tau \qquad \text{and} \qquad A[x : \forall\, W \cdot \tau] \vdash_S N : \sigma$$

Since W is disjoint from A, the conclusion follows by applying the GEN-LET rule. ■

2 A simple and efficient presentation of ML type system

The inference rules of system S are syntax directed, that is, there is only one rule other than the equality rule that can end the derivation of all typings of one construct of the language. There is an obvious type inference algorithm W obtained by reading the rules from bottom to top as rewriting rules. Directed by the syntax of the program, the algorithm finds the unique non equality rule that can be used last, instantiate it appropriately, and recursively calls the algorithm with smaller typing inference problems or unification problems.

Typing the program *let* $x = M$ *in* N in an environment A first requires typing M in A, which returns some type τ and a substitution μ, then typing N in $\mu(A)[x : \forall\, \mathcal{V}(\tau) \setminus \mathcal{V}(A) \cdot \tau]$. This requires the computation of the set of variables $\mathcal{V}(\tau) \setminus \mathcal{V}(A)$ that can be quantified in τ, which can be excessively expensive, since it does not only depend on the expression M but also on the size and the structure of the context A. The term M may contain a free variable x that is assigned a large monomorphic type σ. The part of τ that comes from σ only contains variables that are also in A and thus are inspected unnecessarily.

There is another, even simpler, presentation of the ML type system that leads to an algorithm where typing M in a context A only depends on the polymorphic parts of types assigned to the free variables of M.

Let ϕ be a mapping, called rank, of $\mathcal{V}$ into $\mathbb{N}$ such that there are infinitely many variables of every rank. We write $\mathcal{V}^n$ the set of variables of rank exactly n and $\mathcal{V}_n$ the set of variables of rank at most n. We extend rank to terms: the rank of a term is the maximum rank of its variables. We write $\mathcal{T}_n$ the set of terms of rank at most n, that is $\mathcal{T}(\mathcal{V}_n)$. Substitutions are restricted to rank decreasing ones, which are stable by composition.

Type schemes $\forall\, \mathcal{W} \cdot \mathcal{T}_n$ are taken modulo α-conversion in $\mathcal{T}$, without taking ranks of bound variables into account. We recall that $\forall\, \mathcal{V}^{n+1} \cdot \tau$ is $\forall\, (\mathcal{V}^{n+1}(\tau)) \cdot \tau$ by definition. It is in $\mathcal{T}_n$ provided τ is in $\mathcal{T}_{n+1}$. A type scheme of $\forall\, \mathcal{W} \cdot \mathcal{T}_n$ can always be written indefferently $\forall\, \mathcal{V}^{n+1} \cdot \tau$ with τ in $\mathcal{T}_{n+1}$, or $\forall\, W \cdot \tau$ with W and τ in $\mathcal{V}^n$.

We recursively define the typing relations $\vdash_n$ in $\mathcal{C}(\forall\, \mathcal{W} \cdot \mathcal{T}_n^\iota) \times \mathrm{ML} \times \mathcal{T}_n^\iota$ by the rules of figure 4:

$$\frac{x : \forall\, W \cdot \tau \in A \qquad \mu : W \to \mathcal{T}_n}{A \vdash_n x : \mu(\tau)} \ (\textsc{Var-Inst})$$

$$\frac{A[x : \tau] \vdash_n M : \sigma}{A \vdash_n \lambda\, x.\, M : \tau \to \sigma} \ (\textsc{Fun}) \qquad \frac{A \vdash_n M : \sigma \to \tau \qquad A \vdash_n N : \sigma}{A \vdash_n M\ \ N : \tau} \ (\textsc{App})$$

$$\frac{A \vdash_{n+1} M : \tau \qquad A[x : \forall\, \mathcal{V}^{n+1} \cdot \tau] \vdash_n N : \sigma}{A \vdash_n \mathit{let}\ x = M\ \mathit{in}\ N : \sigma} \ (\textsc{Gen-Let})$$

$$\frac{A \vdash_n M : \sigma \qquad \sigma =_E \tau}{A \vdash_n M : \tau} \ (\textsc{Equal})$$

Figure 4: Hierachical typing rules

Lemma 3 (Substitution lemma) *The relations* $\vdash_n$ *are stable under substitution. That is the rule*

$$\frac{A \vdash_n M : \tau}{\mu(A) \vdash_n M : \mu(\tau)} \ (\textsc{Sub})$$

is provable.

Proof: We prove "for all substitutions μ, for all integers n, if $A \vdash_n M : \tau$ is derivable by Δ, then $\mu(A) \vdash_n M : \mu(\tau)$ is derivable", first by induction on the structure of Δ, then by cases on the last rule of Δ.

Case EQUAL: Equality is stable by substitution.

Case VAR: We know from the hypothesis $A \vdash_n x : \tau$ that there exists $x : \forall W \cdot \sigma$ is in A and a substitution $\nu : W \to \mathcal{T}_n$ that sends σ to τ. We can choose W be $\mathcal{V}^{n+1}$. Let ξ be $(\mu \circ \nu) \restriction \mathcal{V}^{n+1}$. It sends W into $\mathcal{T}_n$. Since the domain of ν is disjoint from $\mathcal{V}_n$, the substitution $\xi \circ (\mu \restriction \mathcal{V}_n)$ is equal to $\mu \circ \nu$. Thus ξ sends $(\mu \setminus W)(\sigma)$ to $\mu(\tau)$. Therefore we can derive the conclusion $\mu(A) \vdash_n x : \mu(\tau)$.

Case APP, FUN: Apply the induction hypothesis to the premises, and conclude with the same rule.

Case LET: We know from the hypothesis that there exists σ in $\mathcal{V}_{n+1}$ such that $A \vdash_{n+1} M : \sigma$ and $A[x : \forall\, \mathcal{V}^{n+1}(\sigma) \cdot \sigma] \vdash_n N : \tau$ are derivable. The conclusion follows by applying the LET-rule to the substitution of both judgements by $\mu \restriction \mathcal{V}_n$. ■

The assertion $x : \forall\, \mathcal{V}^{n+1} \cdot \tau$ is said to be *smaller at rank n* than the assertion $x : \forall\, \mathcal{V}^{n+1} \cdot \sigma$ if τ can be taken in $\mathcal{T}_{n+1}$ such that there exists a substitution μ with domain $\mathcal{V}^{n+1}$ that maps τ to σ. A context A is said to be *smaller at rank n* than a context B if it is pointwise smaller at rank n. We write $A \subset_n B$.

Lemma 4 (Generalization of context) *The relation $\vdash_n$ is stable under generalization of context.*

$$\frac{A \vdash_n M : \tau \qquad B \subset_n A}{B \vdash_n M : \tau} \qquad \text{(C-GEN)}$$

Proof: The proof is by structural induction on the premise.

Case VAR: We know from the hypothesis $A \vdash x : \tau$ that there is an assertion $x : \forall W \cdot \sigma$ in A and a substitution μ with domain W that takes σ to τ. The set W can always be taken in $\mathcal{V}^{n+1}$. Since B is smaller than A at rank n, there is a substitution ν of domain $\mathcal{V}^{n+1}$ that takes ρ to σ such that $x : \forall\, \mathcal{V}^{n+1} \cdot \rho$ is in B. The substitution $\mu \circ \nu$ with domain included in $\mathcal{V}^{n+1}$ takes ρ to τ, therefore we can conclude $B \vdash_n M : \tau$.

Others cases: They are immediate. ■

Lemma 5 *The relations $\vdash_n$ are stable under E-equality.*

$$\frac{A \vdash_n M : \tau \qquad A =_E B}{B \vdash_n M : \tau} \qquad \text{(C-EQUAL)}$$

Proof: Intuitively, replace every VAR-INST rules in a derivation by a VAR-INST rule followed by an equality rule. Formally, the proof is by structural induction. ■

2.1 Equivalences

Lemma 6 *For any context A and any type τ both of rank at most n, the judgements $A \vdash_n M : \tau$ and $A \vdash_S M : \tau$ are equivalent.*

Proof: The typing relations $\vdash_n$ are clearly sub-relations of $\vdash_S$. Conversely, we reason by structural induction on the judgement $A \vdash_S M : \tau$.

Case LET: We could derive

$$\frac{A \vdash_S M : \tau \qquad A[x : \forall W \cdot \tau] \vdash_S N : \sigma}{A \vdash_S \mathit{let}\ x = M\ \mathit{in}\ N : \sigma}$$

where W is equal to $\mathcal{V}(\tau) \setminus \mathcal{V}(A)$. We cannot use the rule SUB in S here, since we want to use the equivalence that we are presently proving to deduce rule SUB in S from rule SUB in H. However, the derivations of S are stable by global α-conversion. Therefore the restricted case of SUB for renamings is obviously true in S, and we can rename variables of W so that they are of rank $n+1$. Then, both premises are well-formed and by the induction hypothesis we derive

$$A \vdash_{n+1} M : \tau \qquad \text{and} \qquad A[x : \forall W \cdot \tau] \vdash_n N : \sigma$$

in ML. Since W is exactly $\mathcal{V}^{n+1}(\tau)$, we conclude $A \vdash_n \mathit{let}\ x = M\ \mathit{in}\ N : \sigma$.

Case VAR, FUN, APP, EQUAL: All cases are easy or similar to the LET case. ∎

Using this equivalence, the lemmas C-GEN and SUB of H can easily be prooved in S. The substitution lemma for $\vdash_S$ (lemma 1) is:

$$\frac{A \vdash_S M : \tau}{\mu(A) \vdash_S M : \mu(\tau)} \qquad \text{(SUB)}$$

Proof: Assume that $A \vdash_S M : \tau$ is valid. We can always choose the function rank such that the free variables of A and τ and variables of both the domain and the range μ are of rank 0. Applying the substitution lemma in H proves the judgement $\mu(A) \vdash_0 M : \mu(\tau)$ which is equivalent to the judgement $\mu(A) \vdash_S M : \mu(\tau)$. ∎

The assertion $x : \forall W \cdot \tau$ is said to be *smaller* than the assertion of $x : \forall W' \cdot \tau'$ if there exists a substitution μ with domain W that takes τ to τ', and W' is included in the range of μ. A context A is said to be smaller that B if it is pointwise smaller.

Lemma 7 (Generalization of context) *The relation $\vdash_S$ is stable under generalization of context.*

$$\frac{A \vdash_S M : \tau \qquad B \subset A}{B \vdash_S M : \tau} \qquad \text{(C-GEN)}$$

Proof: Assume that $A \vdash_S M : \tau$ and $B \subset A$. We can always choose the function rank such that free variables of A, B and τ are of rank 0. Then $A \vdash_0 M : \tau$ and $B \subset_0 A$. By lemma C-GEN for H, we conclude $B \vdash_0 M : \tau$ and by the equivalence of H and S, we get $B \vdash_S M : \tau$. ∎

2.2 Typing and reduction

Lemma 8 (Context restriction) *If A and B are equal on free variables of M, the judgements $A \vdash_S M : \tau$ and $B \vdash_S M : \tau$ are equivalent.*

Proof: By symmetry, it is only necessary to show one direction. The proof is an easy structural induction on the premise. ∎

We define a relation on $\mathcal{C}(\forall \mathcal{W} \cdot \mathcal{T}^\iota) \times \mathrm{ML} \times \mathrm{ML}$ as

$$A \vdash M \subset M' \iff \forall \sigma \in \mathcal{T}^\iota,\ A \vdash_S M : \sigma \Longrightarrow A \vdash_S M' : \sigma$$

Theorem 1 (Term substitution)

$$\frac{A \vdash_S [M/x]N : \tau \qquad A \vdash M \subset M'}{A \vdash_S [M'/x]N : \tau}$$

Proof: We assume $A \vdash M \subset M'$ and we show

$$\frac{A' \vdash_S [M/x]N : \tau}{A' \vdash_S [M'/x]N : \tau}$$

for contexts A' that extend A, by induction on the structure of N. We may always assume that bound variables of N are not in the domain of A (and consequently distinct from free avriables of M and M').

Case N is x: The terms $[M/x]N$ and M are equal. Thus there exists a proof of $A' \vdash_S M : \tau$, and consequently there is also a derivation of $A' \vdash_S M' : \tau$, which is a derivation of $A \vdash_S [M'/x]N : \tau$.

Case N is y: The term $[M/x]N$ and $[M'/x]N$ are equal.

Case N is *let* $y = [M/x]N_1$ *in* $[M/x]N_2$: The first non equality rule that ends the derivation of $A' \vdash_S [M/x]N : \tau$ is necessarily a GEN-LET rule.

$$\frac{A' \vdash_S [M/x]N_1 : \tau_1 \qquad A'[y : \forall\, W \cdot \tau_1] \vdash_S [M/x]N_2 : \tau_2}{A \vdash_S \mathit{let}\ y = [M/x]N_1\ \mathit{in}\ [M/x]N_2 : \tau_2}$$

such that τ_2 is E-equal to τ. Since y is not free in M or M', we can apply the induction hypothesis to both premises, then rule GEN-LET, and conclude with an equality rule.

Case N is $N_1\ N_2$ or N is $\lambda\, y.\ N_1$: These cases are similar to case LET. ■

Corollary 9 *If W is disjoint from A, and x is not free in M, then*

$$\frac{A \vdash_S M : \sigma \qquad A[x : \forall\, W \cdot \sigma] \vdash_S N : \tau}{A \vdash_S [M/x]N : \tau}$$

Proof: Since x is not free in M, we have $A[x : \forall\, W \cdot \sigma] \vdash_S M : \sigma$ by context restriction. Rule SUB' shows that $A[x : \forall\, W \cdot \sigma] \vdash_S x \subset M$. Applying the term substitution theorem 1 with context $A[x : \forall\, W \cdot \sigma]$, and terms x for M and M for M', we get $A[x : \forall\, W \cdot \sigma] \vdash [M/x]N : \tau$ and the conclusion follows from the context-restriction lemma. ■

Corollary 10 (Subject reduction)

$$\frac{A \vdash_S (\lambda\, x.\ N)\ M : \tau}{A \vdash_S [M/x]N : \tau} \qquad \textit{and} \qquad \frac{A \vdash_S \mathit{let}\ x = M\ \mathit{in}\ N : \tau}{A \vdash_S [M/x]N : \tau}$$

Proof: In each case, the premise implies that there exist a type σ and a set of variables W taken outside of A (W is empty in the first rule) such that

$$A \vdash_S M : \sigma \qquad A[x : \forall\, W \cdot \sigma] \vdash_S N : \tau$$

We conclude by previous corollary. ■

The reverse of subject reduction does not hold, since if x does not occur in N, then M may not be typable while N is. A slightly weaker property is term expansion.

Lemma 11 (Term expansion)

$$\frac{A \vdash_S M : \sigma \qquad A \vdash_S [M/x]N : \tau}{A \vdash_S \mathit{let}\ x = M\ \mathit{in}\ N : \tau}$$

holds in S.

Proof: The proof assumes theorem 4 which is shown in section 4 independently of this lemma (there is also another but longer proof of the lemma that does not use theorem 4). We assume $A \vdash_S M : \sigma$. Theorem 4 ensures that there is a principal typing $A \vdash_S M : \sigma'$ for the typing problem $A \triangleright M : \beta$. We write W the set of variables of σ' that do not appear in A. We show

$$\frac{A' \vdash_S [M/x]N : \tau}{A'[x : \forall W \cdot \sigma'] \vdash_S N : \tau}$$

for all context A' that extend A, by induction on N. The lemma will follow immediately by a LET rule. We can always assume that all bound variables of N are outside the domain of A (consequently, they are disctinct from free variables of M).

Case N is x: Then $[M/x]N$ is M. Thus $A' \vdash M : \tau$, and by context-restriction, $A \vdash M : \tau$. Since $A \vdash M : \sigma'$ is a principal solution of the typing problem $A \triangleright M : \beta$, there exists a substitution μ of domain outside of A, that is in W, that maps σ' to τ. The conclusion follows by applying rule VAR-INST.

Other Cases: In all other cases, $[M/x]N$ and N have the same top structure. We apply the induction hypothesis to the premises and conclude with the same rule. This includes the case when N is y. ∎

Let $\vdash_X$ be the typing relation defined in $\mathcal{C}(\forall \mathcal{W} \cdot \mathcal{T}^\iota) \times \mathrm{ML} \times \mathcal{T}$ by the rules VAR-INST, FUN, APP and the rule

$$\frac{A \vdash_S M : \sigma \qquad A \vdash_S [M/x]N : \tau}{A \vdash_S \mathit{let}\ x = M\ \mathit{in}\ N : \tau} \qquad \text{(LET')}$$

Lemma 12 *The typing relation $\vdash_S$ restricted to $\mathcal{C}(\mathcal{T}^\iota) \times ML \times \mathcal{T}^\iota$ and the typing relation $\vdash_X$ are equivalent.*

Proof: The rule LET' can be proved for $\vdash_S$ by the subject reduction lemma. Conversely, the rule GEN-LET can be proved for $\vdash_X$ by the term-expansion lemma. ∎

The typing relation $\vdash_X$ may be used to prove syntactic results of type inference, but it should not be taken as the basis for a type inference algorithm. The expansion of LET-expressions during typechecking looses the underlying sharing of the LET-bound expression which may be typed several times.

3 Unificands

A type inference algorithm is obtained in the next section by considering typing problems as unificands. In this section we recall the generalization of term equations as unificands. The results of the first two parts are well known but the presentation is new. Unificands are treated more abstractly so that we can also consider typing problems as unificands in the next section. In the last part, we consider a special case of order-sorted unification, for which we have a simple fast unification algorithm. The results of this section were first introduced

in chapter 2 of [Rém90]. The formulation that we present here is much simpler. It uses existential unificands as in [KJ90].

In this section, we are given a term algebra $\mathcal{T}(\mathcal{S},\mathcal{V})$ and a regular equational theory on terms, whose unification is decidable and unitary unifying. Unitary unifying means that any unification problem that has a solution has a principal solution. In fact, it is not necessary that unification be unitary but only finitary unifying, then typing problems will have principal set of solutions instead of principal solutions. We will emphasize places where the difference matters. We study unification in $\mathcal{T}/E$ (read $\mathcal{T}$ modulo the E-equality). However we always work with terms of $\mathcal{T}$ and manipulate their E-equality explicitly. The equational theory E may be empty, of course, which includes free unification.

3.1 Set of unificands

The substitutions of $\mathcal{T}$ are functions from $\mathcal{V}$ to $\mathcal{T}$ equal to the identity almost everywhere. An *admissible set of substitutions* is a subset of substitutions $\mathcal{S}$ such that:

- $\mathcal{S}$ is stable by E-equality,
- $\mathcal{S}$ is stable by composition,
- $\mathcal{S}$ is stable by restrictions,
- For any variable α, there are infinitely many variables β such that both $\alpha \mapsto \beta$ and $\beta \mapsto \alpha$ are in $\mathcal{S}$

The last condition is technical and ensures that there are enough renamings. It follows that an admissible set of substitutions is stable by sums of substitutions.

Admissible sets of substitutions are more general that order-sorted substitutions. If α is a variable, $\mathcal{S}(\mathcal{T})$ does not necessarily contains all its subterms. In this part that an admissible set of substitutions is given.

Let $\mathcal{Q}$ be a set of objects, given with a function *subst* on $\mathcal{T}^{\mathcal{V}} \times \mathcal{Q}$ to $\mathcal{Q}$, a function *vars* from $\mathcal{Q}$ to $\mathcal{W}$, and a predicate *valid* on $\mathcal{Q}$ (i.e. a subset of $\mathcal{Q}$). We say that $\mathcal{Q}$ is a *set of unificands* if the following five properties hold for any unificand Q and any substitutions μ and ν:

1. $\mathcal{D}(\mu) \cap vars(Q) = \emptyset \Longrightarrow subst(\mu, Q) = Q$
2. $subst(\mu, subst(\nu, Q)) = subst(\mu \circ \nu, Q))$,
3. $vars(subst(\mu, Q)) = \mathcal{V}(\mu(vars(Q)))$,
4. $subst(\mu, valid) \subset valid$
5. $\mu =_E \nu \wedge subst(\mu, Q) \in valid \Longrightarrow subst(\nu, Q) \in valid$

Example 1 *Multi-equations* are multi-sets of terms, written $\tau_1 \doteq \ldots \tau_p$. The set of variables of a multi-equation e, written $\mathcal{V}(e)$ is the union of the variables of all its components. The substitution of a multi-equation is the multi-equation obtained by substituting all its components. The set of multi-equations form a set of unificands when valid multi-equations are those that have all their components E-equal.

Unificands defined on the same algebra can be combined. The *conjunction* (respectively the *disjunction*) of the set of unificands $(\mathcal{Q}', vars', subst', valid')$ with the set of unificands $(\mathcal{Q}'', vars'', subst'', valid'')$ is the set of unificands $(\mathcal{Q}, vars, subst, valid)$, written $\mathcal{Q}' \wedge \mathcal{Q}''$ (respectively $\mathcal{Q}' \vee \mathcal{Q}''$) defined by

1. $\mathcal{Q} = \mathcal{Q}' \times \mathcal{Q}''$. Elements of $\mathcal{Q}$ are written $Q' \wedge Q''$ (respectively $Q' \vee Q''$),
2. $vars(Q' \wedge Q'') = vars(Q') \cup vars(Q'')$,
3. $subst(\mu, Q' \wedge Q'') = subst(\mu, Q') \wedge subst(\mu, Q'')$,
4. $valid(Q' \wedge Q'') \iff valid'(Q')$ and $valid''(Q'')$,
   (respectively $valid(Q' \wedge Q'') \iff valid'(Q')$ or $valid''(Q'')$.)

The existential quantification of the set of unificands $(\mathcal{Q}', vars', subst', valid'')$ by a variable α, is the set of unificands $(\mathcal{Q}, vars, subst, valid)$, written $\exists\, \alpha \cdot \mathcal{Q}$, defined by

1. $\mathcal{Q} = \mathcal{Q}'$
2. $vars(\exists\, \alpha \cdot Q) = vars'(Q) \setminus \{\alpha\}$
3. $subst(\mu, \exists\, \alpha \cdot Q) = \exists\, \alpha \cdot subst'(\mu \setminus \{\alpha\}, Q)$
4. $valid(\exists\, \alpha \cdot Q) \iff \exists \tau, valid'((\alpha \mapsto \tau)(Q))$

A *solution* (also called a *unifier*) of a unificand U is a substitution that maps U to a unificand V such that $valid(V)$. A unificand is *solvable* if it has at least one solution. The set of solutions of a unificand Q is written $\mathcal{U}(Q)$. It is stable by substitution. A solution is *principal* if all other solutions are instances of it. In the case of finitary theories, we may consider sets of principal solutions, such that all other solutions are instances of one solution of the set. Two unificands are *equivalent* if they have the same set of solutions. This operation is defined even if the two unificands are taken in two different sets of unificands. We write $\bot$ for a unificand that is not solvable. The equivalence of unificands is indeed an equivalence relation.

Moreover, equivalences of unificands are compatible with the conjunctions, disjunction and existential quantification of unificands, that is, if Q' and Q'' are equivalent, then (we omit disjunction rules)

$$Q \wedge Q \equiv Q \wedge Q'' \qquad Q' \wedge Q \equiv Q'' \wedge Q \qquad \exists\, \alpha \cdot Q' \equiv \exists\, \alpha \cdot Q''$$

With respect to these equivalences, the conjunction of unificands (respectively the disjunction of unificands) is commutative and associative. Indeed, the existential quantifier acts as a binder for variables in unificands: existential unificands are equivalent by renaming of variables bound by $\exists$'s, exchange of consecutive $\exists$'s, and removal of vacuous $\exists$'s:

$$Q \wedge (Q' \wedge Q'') \equiv (Q \wedge Q') \wedge Q'' \qquad Q \wedge Q' \equiv Q' \wedge Q \qquad Q \wedge \bot \equiv \bot$$

$$\exists\, \alpha \cdot \exists\, \beta \cdot Q \equiv \exists\, \beta \cdot \exists\, \alpha \cdot Q \qquad \exists\, \alpha \cdot Q = \exists\, \beta \cdot (\alpha \mapsto \beta)(Q)$$

and, if $\alpha \notin vars(Q)$,

$$\exists\, \alpha \cdot Q \equiv Q \qquad Q \wedge (\exists\, \alpha \cdot R) \equiv \exists\, \alpha \cdot Q \wedge R$$

The above equivalences are called *structural*: they do not depend on the structure of unificands. We will consider unificand equality modulo structural equivalences.

Existential unificands capture the notion of garbage collection. During unification (or type inference) some variables may be introduced as auxiliary variables. However, they should be invisible at the end of the computation. In the literature, they are often called fresh variables, but rarely formalized. A possible formalization of fresh variables is to carry a finite set of variables that contains all variables that have been used and outside of which new variables

have to be taken. This is a global treatment. On the opposite, existential variables are treated locally. For instance in $Q \wedge R$, the unificand R may be replaced by a unificand R' introducing a variable α:

$$R \equiv \exists\, \alpha \cdot R'(\alpha)$$

Then, we automatically have:

$$Q \wedge R \equiv Q \wedge \exists\, \alpha \cdot R'(\alpha)$$

The existential quantifier can be pushed outside of Q if it does not contain α. After renaming α, we get:

$$Q \wedge R \equiv \exists\, \alpha' \cdot (Q \wedge R'(\alpha'))$$

Existential unificands permit the definition and reasoning on local rules, from which global rules can trivially be deduced and used in algorithms. "Fresh variable" annotations are replaced by the usual convention that bound variables are kept distinct from all others.

An arbitrary unificand can always be written $\exists\, W \cdot U$ where U does not contain any existential quantification, and U itself may always be written as a disjunction of conjunctions of basic unificands.

3.2 Term unification with systems of multi-equations

In this part all substitutions are admissible.

Arbitrary conjunctions of multi-equations are called *systems of multi-equations*. Systems of multi-equations are a good structure for describing unification on terms since they allow us to formalize sharing of terms. In case the theory is not unitary, we consider disjunction of systems of multi-equations.

We write $V(e)$ for the set of variable terms of e, and $T(e)$ the set of non variable terms of e. If $T(e)$ is a singleton τ, we write $\hat{e}$ for the substitution that maps $V(e)$ to τ. If all terms are variables, we write $\hat{e}$ for a substitution that maps $V(e)$ to an arbitrary element of V. If $T(e)$ has more than one term, then $\hat{e}$ is undefined. When it exists $\hat{e}$ is a trivial solution of e.

We describe a unification algorithm by rewriting rules $\overset{A}{\underset{B}{\rightsquigarrow}}$ that transforms unificands into equivalent unificands. Unification in the empty theory is mainly implemented by the following rules:

$$\frac{\alpha = e \wedge \alpha = e' \rightsquigarrow}{\alpha = e = e'} \qquad \text{(Fuse)}$$

$$\frac{e = f(\tau_1, \ldots \tau_p) = f(\sigma_1, \ldots \sigma_p) \rightsquigarrow}{\exists\, \alpha_1, \ldots \alpha_p \cdot \left(e = f(\alpha_1, \ldots \alpha_p) \wedge \bigwedge_{i \in [1,p]} (\alpha_i = \tau_i = \sigma_i)\right)} \qquad \text{(Decompose)}$$

if $f \neq g$,
$$\frac{f(\tau_1, \ldots \tau_p) = g(\sigma_1, \ldots \sigma_q) = e \rightsquigarrow}{\bot} \qquad \text{(Collision)}$$

A multi-equation is *completely decomposed* if it has at most one variable term per equation. A system is completely decomposed if it is fused and all multi-equations are completely decomposed. Applying them in any order the rules Fuse, Decompose and Collision always terminates (each step first decreasing the size of the system, then the number of multi-equations). In the empty theory, a system to which none of the rules apply is completely decomposed.

In a non-empty theory, the rule Decompose is only an equivalence for a subset of symbols called *decomposable symbols*. A *collision pair* is a pair of symbols for which the rule Collision

is an equivalence. The rules FUSE, DECOMPOSE and COLLISION are no longer sufficient to transform an arbitrary system into a completely decomposed system. We call MUTATE a valid rule that transforms a multi-equation into an equivalent system such that applying the four rules FUSE, DECOMPOSE, MUTATE and COLLISION always terminates with a completely decomposed system. Mutation does not necessarily exists. In particular, there is no possible mutation in a theory that is not finitary unifying. If the theory is unitary unifying, then the mutation need not to introduce any disjunction.

$$\frac{e}{mutate(e)} \qquad \text{(MUTATE)}$$

We say that a multi-equation e is *directly inner* the multi-equation e' if there is at least one variable term of e' that appears in a non variable term of e. We write $\lessdot$ the direct-inner relation, and $\not\lessdot$ its negation. We write

$$Q \lessdot R \qquad \text{for} \qquad \forall e' \in Q,\ \forall e \in R,\ e \lessdot e'$$

and

$$Q \not\lessdot R \qquad \text{for} \qquad \forall e \in Q,\ \forall e' \in R,\ e \not\lessdot e'$$

We call *inner* the transitive closure of the direct-inner relation on all multi-equations of Q, written as $\lessdot_Q^*$. The intuition of the inner relation is the following: if μ is a solution of Q and e and e' are two equations of Q such that $e \lessdot e'$, then μ maps all terms of the equation e to a common term τ and all terms of e' to τ', such that τ is a sub-term of τ' of $\mu(e')$.

A conjunction Q equal to $e_1 \wedge \ldots e_p$ is said to be in the *outer-first* order if

$$\forall i,\ \forall j \geq i,\ e_i \not\lessdot e_j$$

It is said to in *outer-last* order if the reverse sequence is outer-first order. A system is *free* if no multi-equation is in inner relationship with one another.

The inner relation of a solvable completely decomposed system is strict. Checking the strictness of the inner relation is called the *occur-check*. If occur check fails on a completely decomposed subsystem of Q, it surely fails on the whole system. A theory is *strict* if any solvable completely decomposed system has a strict inner relation. That is, in a strict theory, a term is never equal to one of its sub-terms. The occur check for a strict theory is the rule

$$\text{if } e \lessdot_Q^* e, \qquad \frac{Q}{\bot} \qquad \text{(OCCUR)}$$

It is completed by the replacement rule.

$$\text{if } e \wedge Q \not\lessdot e, \qquad \frac{e \wedge Q}{e \wedge \hat{e}(Q)} \qquad \text{(REPLACE)}$$

A principal unifier of a strict, free and completely decomposed system $e_1 \wedge \ldots e_p$, is the composition, in any order, of the trivial solutions of its equations: $\hat{e}_p \circ \ldots \hat{e}_1$.

A free system does not need to be computed to exhibit a principal solution. It can be directly read from a strict completely decomposed system: such a system can always be written in an outer-first order $e_1 \wedge \ldots e_p$. Successive replacements in outer-first order produce a free system in which $\hat{e}_p \circ \ldots \hat{e}_1$ is a principal unifier. It is noted $\hat{Q}$.

The principal solution $\hat{Q}\restriction W$ of a completely decomposed system $\exists\, W \cdot Q$ is defined modulo a renaming of variables in W.

Useless existential quantifiers can be eliminated at anytime:

$$\text{if } \alpha \notin \mathcal{V}(e \wedge Q), \qquad \frac{\exists\, \alpha \cdot (Q \wedge \alpha \doteq e)}{e \wedge Q} \qquad \text{(RESTRICT)}$$

The rule GENERALIZE could have been used initially to transform the system into a system of small terms (of depth at most one), for which other rewriting rules will work more efficiently by more sharing.

$$\text{if } \alpha \in e \setminus \mathcal{V}(\tau) \wedge \tau \notin \mathcal{V}, \qquad \frac{(\alpha \mapsto \tau)(e)}{\exists\, \alpha \cdot (e \wedge \alpha = \tau)} \qquad \text{(GENERALIZE)}$$

3.3 Hierarchical unification

We describe a very simple case of sorted unification where variables are given ranks, and all substitutions are rank decreasing, as in section 2. More precisely, we assume that a mapping ϕ of $\mathcal{V}$ into $\mathbb{N}$ is givent such that there are infinitely many variables of every rank. As in section 2, we write $\mathcal{T}^n$ the sets of terms of rank n and $\mathcal{T}_n$ the sets of terms of rank smaller or equal to n.

Rank decreasing substitutions are those for which there exists a decreasing function in $\mathbb{N} \to \mathbb{N}$ whose composition with ϕ is equal to $\phi \circ \mu$. They are stable by composition, and form an admissible set of substitutions for unificands. A renaming is a raw-renaming such that $\phi \circ \mu$ is equal to ϕ. Its inverse is also a renaming.

We refer to non admissible substitutions as raw substitutions. Any unificand that is raw-solvable is solvable, since the unificand can be mapped to $\mathcal{T}^0$ by a raw renaming into $\mathcal{V}^0$ which is rank decreasing, then it is solvable since unification in $\mathcal{T}^0$ is isomorphic to raw-unification in $\mathcal{T}$.

In the following we use the framework of unificands to show how to solve hierarchical unification problems. We add extra information to systems of multi-equations. Every multi-equation e is constrained by one integer p called its rank. We write $e \downarrow p$ the constrained multi-equation and $\downarrow(e)$ for p. The rank of a system is the maximum rank of its multi-equations.

A substitution is *solution of a hierarchical equation* if it is a solution of the raw equation. The rank of an equation is only a memory for efficiency purposes. A *constrained system* of multi-equations Q is one such that:

$$\forall \mu \in \mathcal{U}(Q),\ \forall e \in Q,\ \forall \tau \in e,\ \phi(\mu(\alpha)) \leq \downarrow(e) \leq \phi(\alpha)$$

The rank of a transformation $\frac{Q}{\rightsquigarrow Q'}$ is the rank of Q. The transformation is *constrained* if it is an equivalence such that if Q is constrained, then Q' is constrained as well. It is *compositional* if additionally $\frac{Q \wedge R}{\rightsquigarrow Q' \wedge R}$ is a constrained equivalence for all R. We associate a constrained transformation to each raw transformation.

$$\frac{\alpha = e \downarrow p \wedge \alpha = e' \downarrow p'}{\rightsquigarrow \alpha = e = e' \downarrow \min(p, p')} \qquad \text{(FUSE)}$$

$$\text{If } \alpha_i \in \mathcal{V}^p, \qquad \frac{e = f(\tau_1, \ldots \tau_p) = f(\sigma_1, \ldots \sigma_p) \downarrow p}{\rightsquigarrow \exists\, \alpha_1, \ldots \alpha_p \cdot \left(e = f(\alpha_1, \ldots \alpha_p) \wedge \textstyle\bigwedge_{i \in [1,p]} (\alpha_i = \tau_i = \sigma_i)\right) \downarrow p} \qquad \text{(DECOMPOSE)}$$

$$\text{If } f \neq g, \qquad \frac{f(\tau_1, \ldots \tau_p) = g(\sigma_1, \ldots \sigma_q) = e}{\rightsquigarrow \bot} \qquad \text{(COLLISION)}$$

We call *mutation at rank* p, a constrained mutation of a multi-equation $e \downarrow p$.

We add the compositional transformation

$$\text{If } e' \Leftarrow e, \downarrow(e) \leq q < p, \qquad \frac{e' \downarrow p \wedge e}{e' \downarrow q \wedge e} \rightsquigarrow \qquad \text{(PROPAGATE)}$$

$$\text{If } \forall \alpha \in T(e),\ (\phi(\alpha) \leq q < p) \vee (\exists e' \in Q, \alpha \in V(e') \wedge \downarrow(e') \leq q < p),$$

$$\frac{e \downarrow p \wedge Q}{e \downarrow q \wedge Q} \rightsquigarrow \qquad \text{(REALIZE)}$$

A constrained system Q is *canonical* if it is $\bot$ or completely decomposed, propagated and realized and if every multi-equation e composed of variables has a variable of rank $\uparrow(e)$. A completely decomposed, propagated and realized system can always be turned into a canonical system by using the following rule

$$\text{if } \begin{cases} \forall \beta \in V(e),\ \phi(\beta) > p, \\ \alpha \in \mathcal{V}^p, \end{cases} \qquad \frac{V(e) \downarrow p}{\exists\, \alpha \cdot V(e) = \alpha \downarrow p} \rightsquigarrow \qquad \text{(EXTEND)}$$

When e is reduced to variables, we always require that the arbitrary variable chosen to define $\hat{e}$ is of lower rank. This also applies to trivial solutions of strict, completely decomposed unificands.

Theorem 2 *If Q is strict, canonical and free, then $\hat{Q}$ is a principal solution of Q.*

Proof: When Q is free, $\hat{Q}$ is trivially a solution, since it is a raw-solution and rank-decreasing, and conversely, any solution μ is such that $\mu \circ \hat{Q} = \mu$. Therefore, it suffices to see that replacement in the outer-first order keeps the system canonical, because then Q is equivalent to a free canonical system, say R, obtained by replacement in an outer-first order, whose principal solution $\hat{R}$ is equal to $\hat{Q}$. ■

Thus hierarchical unification proceeds as simple unification leading to $\bot$ or a completely decomposed system, and then computes ranks of variables before a principal solution can be read. We show below that computation of ranks can be done by propagation and realization after the system has been completely decomposed. While maintaining a completely decomposed system can be done incrementally, the cost of propagation and realization dissuades from computing canonical systems incrementally. In the following, we introduce partial canonical systems, for which the cost of propagation and realization is done only on a subsystem.

A unificand $Q \wedge R$ is a *conjunction at rank n* if the maximum rank of Q is strictly lower than n, then we write $Q \wedge_n R$. A conjunction $Q \wedge_n R$ at rank n is a *separation at rank n*, if additionaly:

1. $\forall e \in R,\ n \leq \downarrow(e)$,
2. R is completely decomposed in $Q \wedge R$,
3. $R \not\Leftarrow Q$

A separation $Q \wedge_n R$ at rank n is *n-canonical* if no PROPAGATE, REALIZE or EXTEND rule may apply to a multi-equation of R.

Theorem 3 *A solvable n-canonical conjunction $\exists\, W \cdot Q \wedge_n R$ has a principal solution of the form $(\mu \circ \hat{R}) \restriction W$ where μ is a principal solution of $\exists\, W \cdot Q$.*

Proof: Any fusion, decomposition, propagation or realization of an n-canonical system $\exists W \cdot Q \wedge R$ only involves multi-equations of Q and leaves R in outer relation to the rest of the system. Thus there is an equivalent canonical system $\exists W, W' \cdot Q' \wedge R$ such that R is in outer relation to Q'. A principal solution of $\exists W \cdot Q \wedge R$ is $(\hat{Q}' \circ \hat{R}) \restriction W \cup W'$, that is $(\hat{Q}' \restriction W' \circ \hat{R}) \restriction W$. Indeed, $\hat{Q}' \restriction W'$ is a principal solution of $\exists W \cdot Q$. ■

Algorithm 1 *A n-canonical system can be obtained using the following steps:*

1. *Starting with R_0, apply decomposition, fusion and collision. Then apply the occur-check. This produces either $\bot$, or a strict, completely decomposed system $\exists W \cdot R_1$. Return $\bot$ in the first case, continue otherwise.*

2. *Apply propagation to R_1 in an outer-first order. It terminates in at most as many steps as there are multi-equations in R_1, with a unificand R_2.*

3. *Apply realization to R_2 in an inner-first order. It terminates in at most as many steps as there are multi-equations in R_2, with a unificand R_3.*

4. *Let Q the composition of all multi-equations of R_3 of rank lower than n, and R_4 of all others. Extend R_4 to $\exists W' \cdot R$.*

The conjunction $\exists W, W' \cdot Q \wedge_n R$ is n-canonical.

Proof: The unificand Q is propagated (respectively realized) in R if no propagation (respectively realization) of R involves a multi-equation of R. The correction of the algorithm easily follows from:

- Let $Q \wedge R$ be completely decomposed. If R is propagated in $Q \wedge R$ and R is outer to Q and if R' is a propagation of Q, then R' is propagated in $Q \wedge R'$.

- Let $Q \wedge R$ be completely decomposed. If Q is realized in $Q \wedge R$ and R is outer to Q and if R' is a realization of R, then Q is realized in $Q \wedge R'$.

- Realization of a completely decomposed and propagated system keeps the system propagated.

■

Lemma 13 *If $Q \wedge R$ is n-canonical, and all variables of Q' are of ranks at most n, then $(Q \wedge Q') \wedge R$ is n-canonical.*

Proof: It is a conjunction at rank n. Propagation of R into Q' cannot happen since the lower rank of R is greater than the higher rank of Q'. Realization of an equation of R by Q' cannot happen since no equation of Q' is not inner to R. Realization of Q by equations of Q' cannot indirectly fire the realization of an equation of R by Q. ■

Algorithm 2 *The algorithm 1 for computing a n-canonical form of R_0 can be improved by initially splitting R_0 into a conjunction at $Q' \wedge R'_0$ where Q' is the largest sub-unificand of Q_0 that has all its variables in $\mathcal{V}_n$. Running the algorithm 1 on R'_0 produces $Q \wedge R$. Then, $(Q' \wedge Q) \wedge R$ is n-canonical.*

The algorithm 2 is implemented in the appendix A.

4 Type inference

The substitution lemma allows us to consider triples $A \triangleright_n M : \tau$ as unificands for the admissible set of ranked substitutions and the following three definitions:

1. $\mathcal{V}(A \triangleright_n M : \tau)$ is $\mathcal{V}(A) \cup \mathcal{V}(\tau)$,
2. $\mu(A \triangleright_n M : \tau)$ is $\mu(A) \triangleright_n M : \mu(\tau)$,
3. $valid_n$ is the relation $\vdash_n$.

The closure of the disjoint union of these unificands and systems of multi-equations by conjunction and existential quantification are called typing problems.

The principal typing property is equivalent to having principal solutions to all solvable typing problems. We can formalize the algorithm that computes principal solutions as a set of simplifications of typing unificands that are equivalences.

Lemma 14 *Let A be a context of level n and α be a type variable of level at most n. The following rules are equivalences:*

1. *If $x : \forall W \cdot \tau$ is in A, τ in $\mathcal{T}_n$ and W a subset of $\mathcal{V}^n \setminus \{\alpha\}$, then:*

$$\frac{A \triangleright_n x : \alpha}{\exists W \cdot \alpha = \tau} \rightsquigarrow \qquad \text{(Var)}$$

   If x is not in A, then $A \triangleright_n x : \alpha$ is not solvable.

2. *If β is in $\mathcal{V}^n$, then*

$$\frac{A \triangleright_n M \;\; N : \alpha}{\exists \beta \cdot A \triangleright_n M : \beta \wedge A \triangleright_n N : \beta \to \alpha} \rightsquigarrow \qquad \text{(App)}$$

3. *If β and γ are in $\mathcal{V}_n$, then*

$$\frac{A \triangleright_n \lambda\, x.\, M : \alpha}{\exists \beta\gamma \cdot \; A[x : \beta] \triangleright_n M : \gamma \wedge \alpha = \beta \to \gamma \downarrow n} \rightsquigarrow \qquad \text{(Fun)}$$

4. *Let β be in $\mathcal{V}^{n+1} \setminus \mathcal{V}(A)$.*

   If $\exists W \cdot Q \wedge_n R$ is a solvable n-canonical unificand equivalent to $A \triangleright_{n+1} M : \beta$ (such that W is disjoint from $\mathcal{V}(A)$ and α), then

$$\frac{A \triangleright_n \mathit{let}\; x = M \;\mathit{in}\; N : \alpha}{\exists W \cdot Q \wedge A[x : \forall\, \mathcal{V}^{n+1} \cdot \hat{R}(\beta)] \triangleright_n N : \alpha} \rightsquigarrow \qquad \text{(Let)}$$

   If $A \triangleright_n M : \beta$ is not solvable, then neither is $A \triangleright_n \mathit{let}\; x = M \;\mathit{in}\; N : \alpha$.

Proof: Each case of the lemma is of the form $\frac{A}{B} \rightsquigarrow$ (R). We successively show that any solution μ of A is solution of B (R-Direct), and conversely that any solution μ of B is a solution of A (R-Inverse).

Case Var-Direct: We can choose W disjoint from $\mu(\alpha)$ and form μ. From the hypothesis $\mu(A) \vdash_n x : \mu(\alpha)$ we know that there exists a substitution ν with domain W that maps $\mu(\tau)$ to $\mu(\alpha)$ modulo E. The term $\mu(\alpha)$ is also equal to $(\nu \circ \mu)(\alpha)$. Therefore, $\nu \circ \mu$ is a solution of $\alpha \doteq \tau$, and so is $(\nu \circ \mu) \setminus W$.

Case VAR-INVERSE: We again choose W disjoint from $\mu(\alpha)$ and from μ. We know from the hypothesis that μ is a solution of $\exists\, W \cdot \alpha \doteq \tau$. That is, there exists a substitution ν of domain W such that $\nu \circ \mu$ is a solution of $\tau \doteq \alpha$. As for the VAR-DIRECT case, this means that ν maps $\mu(\tau)$ to $\mu(\alpha)$. Therefore, μ is a solution of $A \triangleright_n x : \alpha$.

Case APP, FUN: They are both immediate.

Case LET-DIRECT: We know from the hypothesis $\mu(A) \vdash_n \mathit{let}\ x = M\ \mathit{in}\ N : \mu(\alpha)$ that there exists τ such that $\mu(A) \vdash_{n+1} M : \tau$ and $\mu(A)[x : \forall\, \mathcal{V}^{n+1} \cdot \tau] \vdash_n N : \mu(\alpha)$. The substitution $(\beta \mapsto \tau) \circ \mu$ is a solution of $A \vdash_{n+1} M : \beta$ and thus, it is a solution also of $\exists\, W \cdot Q \wedge_n R$. Therefore, it can be extended to a solution ν of $Q \wedge_n R$ of domain W. By theorem 3, ν is of the form $\xi \circ \hat{R}$ where ξ is a solution of Q. Since $\hat{R}$ is of domain in $\mathcal{V}^{n+1}$, it leaves A and α unchanged. We have $\xi(A)[x : \forall\, \mathcal{V}^{n+1} \cdot \xi(\hat{R}(\beta))] \vdash_n N : \xi(\alpha)$ and by lemma C-GEN we get $\xi(A)[x : \forall\, \mathcal{V}^{n+1} \cdot (\xi \restriction \mathcal{V}_n)(\hat{R}(\beta))] \vdash_n N : \xi(\alpha)$, that is ξ is a solution of $A[x : \forall\, \mathcal{V}^{n+1} \cdot \hat{R}(\beta)] \triangleright_n N : \alpha$. Finally, ξ is a solution of the unificand $\exists\, W \cdot Q \wedge A[x : \forall\, \mathcal{V}^{n+1} \cdot \hat{R}(\beta)] \triangleright_n N : \alpha$ of rank n, and so is μ, which is equal to ξ on the set $\mathcal{V}_n$ (their restrictions to $\mathcal{V}_n$ are equal).

Case LET-INVERSE: We know that μ is a solution of $\exists\, W \cdot Q \wedge A[x : \forall\, \mathcal{V}^{n+1} \cdot \hat{R}(\beta)] \triangleright_n N : \alpha$. That is, μ can be extended to a solution ξ of domain W of both unificands Q and $A[x : \forall\, \mathcal{V}^{n+1} \cdot \hat{R}(\beta)] \triangleright_n N : \alpha$. The substitution $\xi \circ \hat{R}$, say ν, satisfies both Q and R. Therefore, $\nu \setminus W$, is a solution of $A \triangleright_n M : \beta$. So is ν since W is disjoint from both $\mathcal{V}(A)$ and β. The substitution ν also satisfies $\nu(A)[x : \forall\, \mathcal{V}^{n+1} \cdot \nu(\beta)] \triangleright_n N : \nu(\alpha)$. Using rule LET, we conclude that ν is a solution of the unificand $A \triangleright_n \mathit{let}\ x = M\ \mathit{in}\ N : \alpha$ at rank n. So are ξ and μ which are equal to ν on the set $\mathcal{V}_n$.

Case FAILURE: The two unsolvable cases are immediate by showing the inverse implications. ■

Theorem 4 *If the equational theory is unitary unifying, all solvable typing problems have principal solutions; there is an algorithm that given any typing problem, either returns a principal solution or returns failure, if the problem is unsolvable.*

Algorithm 3 *For any integer n, an n-canonical unificand equivalent to a typing problem $A \triangleright_n M : \tau$ is recursively computed by the two steps, starting with the unificand Q reduced to $A \triangleright_n M : \tau$:*

1. *Apply rules* VAR, APP, FUN *and* LET *to Q in any order. Rule* LET *recursively calls the algorithm on smaller typing problems. This terminates with a unification problem $\exists\, W \cdot R$.*

2. *Put R in n-canonical form using algorithm 2. This produces a conjunction $\exists\, W' \cdot Q' \wedge R'$.*

Then $\exists\, W, W' \cdot Q' \wedge R'$ is n-canonical and equivalent to $A \triangleright_n M : \tau$.

The DECOMPOSITION, FUSION, COLLISION and MUTATION of the last step may be merged with the first step. The soundness and correctness of the algorithm directly follows from lemma 14.

5 Extensions of the core language

In this section we describe a few extensions of the core language and examples of practical equational theories in ML. We first show that the language can be extended with constants. Type abbreviations in ML are nicely formalized by an equational theory on types. We briefly mention the extension of ML with extensible records and indicate other possible applications.

5.1 Term constants

For sake of simplicity, we have omitted term constants in the core language. They can be incorporated quite easily:

$$M ::= \ldots \mid c$$

Constants come with principal and closed type schemes, or equivalently, with a constant context B with the set of constants ranging into closed type schemes as.

The typing relation $\vdash_B$ for ML with term constants is defined with respect to the constant context B by all the rules of S plus the following:

$$\frac{c : \forall W \cdot \tau \in B \qquad \mu : W \to \mathcal{V},}{A \vdash_B c : \mu(\sigma)} \qquad \text{(Const-Inst)}$$

If the context A and the constant context B are disjoint, then the judgements $A \vdash_B M : \tau$ and $B \cup A \vdash_S M : \tau$ are equivalent. This also includes the case of an infinite set of constants, provided they are finitely representable.

5.2 Type abbreviations

Type abbreviations are often considered as syntactic sugar. This would be correct if parsers expanded them. However, it is more efficient to keep type abbreviations during typechecking, and expand them by need, since they are a more compact representation of large types. Type abbreviations can be nicely formalized with a very simple equational theory on types.

The set of type symbols is the union of original symbols $\mathcal{C}$ and an ordered set of abbreviation symbols $\mathcal{C}'$ composed of f'_0, f'_1, and so on. Type abbreviations are defined by a sequence of equalities:

$$f'_i(\alpha_1, \ldots \alpha_p) = \tau_i$$

where τ_i is only composed of symbols of indices lower that i and all of variables among α_1, $\ldots \alpha_p$. These equalities define the presentation of a theory E.

Types are the algebra $\mathcal{T}(\mathcal{V}, \Sigma_{\mathcal{C}\cup\mathcal{C}'})/E$. Type-abbreviation theories are acyclic [Tha92] and thus syntactic[1] [Kir85], but their presentation are not usually syntactic. Yet, there is a trivial mutation that can be deduced from the non syntactic presentation. All symbols are decomposable. All pairs of original symbols produce collision. Mutation is composed of the following rules:

$$\text{if } \mathrm{Top}\,(\tau) = \mathrm{Top}\,(\tau_i), \quad \frac{f'_j(\sigma_1, \ldots \sigma_p) \doteq \tau \doteq e}{\exists\, \alpha_1, \ldots \alpha_p \cdot \wedge \begin{cases} f_j(\alpha_1, \ldots \alpha_p) \doteq e \\ \tau \doteq \tau_j \\ \alpha_i \doteq \sigma_i, \qquad i \in [1, p] \end{cases}} \rightsquigarrow \qquad \text{(Mutate } (f))$$

where $\mathrm{Top}\,(\tau)$ is the symbol of τ at the empty occurrence.

The type-abbreviation theory is regular, and has a unitary unification algorithm, thus ML with abbreviations has principal typings and a type inference algorithm.

Type-abbreviation theories are very simple. Yet, they provide examples of theories that may involve the arrow symbol: for instance, the theory E reduced to the axiom:

$$f_0(\alpha) = \alpha \to \alpha$$

is a type abbreviation.

[1] Syntactic theories are equational theories such that any provable equality can be proved with at most one occurrence of an axiom at the root; they are theories for which there is a unification algorithm that extends the efficient unification algorithm for empty theories of [MM82].

5.3 Extensible records

An important application of the use of sorted types modulo an equational theory is type inference with extensible records [Rém91]. The equations are used to describe duplication of "row types". Sorts are used for two different purposes: they restrict the formation of terms and the power of the equations; they also introduce a fine control of polymorphism since generic variables range only over types of a certain sort.

A. Ohori also uses sorted types for record structure [Oho90] but his sorts are ordered and our results do not directly apply to his system. Types modulo equations have also been used in [Tha91] but type reconstruction is different: it uses coercions in programs.

5.4 Other applications

Treating typing problems as unificands has two main advantages. Soundness and completeness of the type-inference algorithm with respect to the typing rules can be proved independently for each transformation, and is thus easier. It also formalizes unification on graphs, that leads to efficient algorithms.

Ranks have been used to compute variables that can be generalized incrementally. It is not so critical for the usual Damas-Milner type system, but it is for some extensions, for example with subtypes where typing are constrained by a set of inequations. Simplification of inequations is usually expensive. The control of simplification is crucial. Too many simplifications are too expensive, but too few simplifications will unnecessarily duplicate large structures through generic polymorphism. Ranks keep track of the dependencies of polymorphic binding variables and give the right boundary for simplifications.

Conclusion

The extension of ML with a regular, decidable and unitary unifying equational theory on types preserve all usual syntactic properties of ML; the formulation itself is changed so little that the extension is quite natural. Types can be sorted, provided that judgements and arrow types are of the same toplevel sort.

The language of unificands is well adapted to type inference: properties are easily stated and proofs are usually simpler since they are separated in small independent parts. The algorithm is described by transformation of unificands, which leaves the control more flexible. Unificands allows one to treat type inference and unification in a uniform manner and formalize the sharing between types in typing problems and in unification problems naturally. Benefitting from the work on unification, the use of existential unificands nicely handles the introduction of variables during resolution, avoiding the informal fresh variables or the heavy formalization of garbage collection.

Generalization in ML only applies to variables that have been recently introduced. Hierarchical types keep track of the freshness of type variables during unification; they are very simple order-sorted types, and fit well eithin the same framework of unification. They make generalization a trivial step.

This work includes the original ML language and ML with type abbreviations as a simple instance. An important application is typechecking of extensible records.

A Type inference for the original ML language in Caml-Light

The aim of this appendix is to give a real implementation of the type inference algorithm based on unificands and ranked unification. The language that is considered is the one of section 1 extended with integer constants. Types are unsorted and taken in the empty theory. Type declarations and type abbreviations are not allowed. The language is a toy language, but the implementation is robust, and adding more features to the language is not difficult.

We separately describe all modules with their interfaces. We give the complete implementations of most important modules, but we leave to the user the front and back end module implementations. In fact, the implementation can be used for typechecking without its front and back ends by typing in the abstract syntax of programs instead of entering the programs themselves.

The implementation runs on Caml-Light, version 0.5[2] [LM92].

Module ml

This module defines the abstract syntax of ML programs and a parser of concrete programs.

```
type expr =
    Var of string
  | Fun of string * expr
  | App of expr * expr
  | Let of string * expr * expr
  | Int of int;;

value  parse_expr : char stream → expr;;
```

Module multi

This module defines the deep representation of types and unificands and provides a few functions to manipulate them.

Types and completely decomposed multi-equations are represented by the same multi structure. In order to perform in-place fusion (which corresponds to applying a partial solution of an unificand to the unificand itself in non destructive unification algorithms), all multi structures are mutable objects. The nod of a variable that has been substituted is a Linto nod and points to another multi structure, but eventually ends with a Nolink nod that contains the canonical element of the multi-equation. The term field of a variable or a multi-equation that has only variable terms is Var. Otherwise it is Term (symb, args) where symb and args are the top symbol and the list of immediate subterms of the canonical element. The field rank stores the rank of the multi-equation, and the mark field is used internally by algorithms to keep track of nods that have been visited at some stage of a recursive visit of the structure. They avoid looping since multi structures are graphs.

All multi-equations are kept completely decomposed at any time. A multi-equation $\alpha \doteq \beta \doteq \tau$ can be thought of as two variables α and β both substituted by the term τ.

```
type symbol = Arrow | Int;;

type multi = {mutable nod: nod}
and nod = Nolink  of desc | Linkto   of multi
and desc = {term: term; mutable rank: int; mutable mark : unit ref}
```

[2]This version is distributed by anonymous ftp at `nuri.inria.fr`

```
and term    = Var | Term of symbol * multi list;;

let rec repr u =
 match u.nod
  with Linkto v → let v = repr v in u.nod ← Linkto v;v
     | _ → u;;

let rec desc u =
  match u.nod
   with Nolink u_desc → u_desc
      | _ → desc (repr u);;

let mark() = ref();;
let no_mark = mark();;

let level = ref 0;;
let equations = ref ([||]: multi list vect);;

let reset_level() = level := 1; equations := [|[]; []|];;

let push_level() =
  if vect_length !equations < !level + 2  then
     begin
       let new_equations = make_vect (!level + 2) [] in
       for i = 0 to !level do new_equations.(i) ← (!equations).(i) done;
       equations := new_equations
     end;
  level := succ !level;;

let pop_level() =
  (!equations).(!level) ← []; level := pred !level;;

let new u =
  let u_desc = {term = u; rank = !level; mark = no_mark} in
  let u = {nod = Nolink u_desc} in
  (!equations).(!level) ← u::((!equations).(!level));
  u;;
```

The function repr computes canonical elements of multi-equations doing path compression. Its accompanying function nod return the nod of the canonical element.

Each creation of a variable or of a term is done at the current level and produces a new multi-equation that is sorted with respect to its rank and stored in the equations reference.

Module print

This module implements a printer of multi objects. It is left to the user.

```
#open "multi";;

value reset_namers : unit → unit and print_multi: multi → unit;;
```

Module generic

This module is the heart of the algorithm; it implements generalization and instantiation of types.

```
#open "multi";;

exception Cycle of multi;;

value generalize: multi → unit
  and instance: multi → multi;;
```

Generalization of a unificand at the highest rank first propagates and realizes the unificand at this rank. Then it turns variables of the highest rank into generic variables. Propagation, realization and marking of generic variables of a multi-equation are completed by the function propagate_and_generalize that recursively visits the unificand: propagation is executed before the recursive calls, which comes in outer-most order, and realization proceeds after returns, thus in inner-most order. Marking immediately follows the realization, since the sub-unificand cannot be affected by inner-most returns. The function generalize controls the process. It first marks fresh multi-equations and sort them according to their rank. Then it calls the propagate_and_realize function starting with multi-equations of lower ranks. Garbage unificands are not generalized but only checked for acyclicity.

Genericity is not a property of variables but of multi-equations. Generic multi-equations are represented by setting their rank field to a negative integer. The absolute value identifies the multi-equation locally to the unificand that is being generalized and it is used by the instance function that copies the generic part of the unificand as a graph.

```
#open "multi";;

let  occur_check visited =
  let visiting = mark() in occur_check
 where rec occur_check u =
  let u_desc = desc u in
  if u_desc.mark == visiting then raise (Cycle u) else
  if u_desc.mark != visited & u_desc.rank = !level then
  (match u_desc.term
    with Var → ()
       | Term(C,L) →
           u_desc.mark ← visiting; do_list occur_check L; u_desc.mark ← visited);;

let propagate_and_realize  n fresh visited =
 let generic_index = ref 0 and visiting = mark() in propagate_realize
 where rec propagate_realize k u =
  let u_desc = desc u in
  if u_desc.mark == visiting then raise (Cycle u);
  if u_desc.mark != visited then
   begin
       begin match u_desc.term
        with Var → u_desc.rank ← min u_desc.rank k
           | Term(C,L) →
               if u_desc.mark == fresh then
                 begin
                   u_desc.mark ← visiting;
                   let propagate_realize = propagate_realize (min u_desc.rank k) in
                   u_desc.rank ← it_list (fun k u → max k (propagate_realize u)) 1 L
                 end
       end;

       u_desc.mark ← visited;
       if u_desc.rank = n then (u_desc.rank ← (decr generic_index; !generic_index))
   end;
  if u_desc.rank < 0 then n else u_desc.rank;;


let generalize  u =

  let visited = mark() and fresh = mark() in
  let generalize = propagate_and_realize !level fresh visited
  and occur_check = occur_check visited in
  let sorted = make_vect (!level +1) [] in

  let sort u =
    match u.nod
     with Linkto _ → ()
```

```
        | Nolink u_desc →
            u_desc.mark ← fresh; sorted.(u_desc.rank) ← u::sorted.(u_desc.rank) in
  do_list sort !equations.(!level);
  for i = 0 to !level − 1 do do_list (generalize i) sorted.(i) done;
  generalize !level u;
  let update u =
    let u_desc = desc u in
    if u_desc.rank = !level then occur_check u else
    if u_desc.rank < 0 then () else
      (!equations).(u_desc.rank) ← u::(!equations).(u_desc.rank) in
  for i = 0 to !level do do_list update sorted.(i) done
;;


type table = None | Multi of multi;;
let instance u =
  let u = repr u in let u_desc = desc u in
  if u_desc.rank >= 0 then u else
  let table = make_vect (−u_desc.rank) None in
  instance u
 where rec instance u =
  let u = repr u in let u_desc = desc u in
  if u_desc.rank >= 0 then u else
  match table.(−u_desc.rank−1)
   with None →
        let v = new Var in
        table.(−u_desc.rank−1) ← Multi v;
        let u_desc =
          match u_desc.term
           with Var → Var
              | Term (C, L) → Term (C, map instance L) in
        v.nod ← Nolink {term = u_desc; rank = !level; mark = no_mark};
        v
      | Multi v → v;;
```

Module unify

This module implements free unification, following the three rules Fuse, Collision and Decomposition.

```
#open "multi";;
exception Collision of symbol * symbol;;

let  rec unify u v =
 let u = repr u in let v = repr v in
  if u == v then () else
  let U = desc u in let V = desc v in
  match U.term, V.term
   with Var, _ →
          u.nod ← Linkto v; V.rank ← min U.rank V.rank
      | _, Var →
          v.nod ← Linkto u; U.rank ← min U.rank V.rank
      | Term(C,L), Term(D,M) →
```

```
        if C ≠ D then raise (Collision (C, D));
        if U.rank < V.rank then v.nod ← Linkto u else u.nod ← Linkto v;
        do_list2 unify L M;;
```

The algorithm for syntactic unification would be similar, but the collision case would be preceeded by other successful decomposition cases, automatically deduced from a syntactic presentation of the theory.

Module typing

This module implements algorithm 3.

```
#open "multi";;
#open "print";;
#open "unify";;
#open "generic";;

exception Unbound of string;;

let rec expr A =  expr_A
 where rec expr_A =

 function ml_Var x →
            instance (try assoc x A with Not_found → raise (Unbound x))
        | ml_App (M, N) →
            let u = expr_A N in
            let v = new Var in
            unify (expr_A M) (new (Term (Arrow, [u;v])));
            v
        | ml_Fun (x,M) →
            let u = new Var in
            let v = expr ((x,u)::A) M in
            new (Term (Arrow, [u; v]))
        | ml_Let (x, M, N) →
            push_level ();
            let u =  expr_A M in
            generalize u;
            pop_level ();
            expr ((x,u)::A) N
        | ml_Int c →  new (Term (Int,[]))
;;

let type_expr B M =
   reset_level(); let u = expr B M in generalize u; u;;
```

Module step

The function `step` parses an expression, calls `type_expr` with an empty environment and this expression, and pretty prints the result type, handling and reporting errors that may have been raised.

```
value step: unit → unit;;
```

Module batch

The batch function make `step` loop for ever.

```
value batch: unit → unit;;
```

This provides a stand alone typechecker.

References

[CDDK86] Dominique Clément, Joëlle Despeyroux, Thierry Despeyroux, and Gilles Kahn. A simple applicative language: Mini-ML. Technical Report 529, INRIA-Rocquencourt, France, 1986.

[Dam85] Luis Damas. *Type assignment in programming languages.* PhD thesis, University of Edinburgh, 1985.

[DM82] Luis Damas and Robin Milner. Principal type-schemes for functional programs. In *Nineteenth Annual Symposium on Principles Of Programming Langages*, pages 207–212, 1982.

[Kir85] Claude Kirchner. *Méthodes et outils de conception systématique d'algorithmes d'unification dans les théories équationnelles.* Thèse de doctorat d'état en informatique, Université de Nancy 1, 1985.

[KJ90] Claude Kirchner and Jean-Pierre Jouannaud. Solving equations in abstract algebras: a rule-based survey of unification. Research Report 561, Université de Paris Sud, Orsay, France, April 1990.

[LM92] Xavier Leroy and Michel Mauny. The caml light system, version 0.5. documentation and users' guide. Logiciel 3, INRIA-Rocquencourt, BP 105, F-78 153 Le Chesnay Cedex, 1992.

[MM82] Alberto Martelli and Ugo Montanari. An efficient unification algorithm. *ACM Transactions on Programming Languages and Systems*, 4(2):258–282, 1982.

[Oho90] Atsushi Ohori. Extending ML polymorphism to record structure. Technical report, University of Glasgow, 1990.

[Rém90] Didier Rémy. *Algèbres Touffues. Application au Typage Polymorphe des Objects Enregistrements dans les Langages Fonctionnels.* Thèse de doctorat, Université de Paris 7, 1990.

[Rém91] Didier Rémy. Type inference for records in a natural extension of ML. Technical Report 1431, INRIA-Rocquencourt, BP 105, F-78 153 Le Chesnay Cedex, May 1991.

[Tha91] Satish Thatte. coercive type isomorphism. In *Functional Programming and Computer Architecture*, volume 523. Springer Verlag, 1991.

[Tha92] Satish Thatte. Finite acyclic theories are unitary. Personal communication, 1992.