Ramified Higher-Order Unification (Revised Report)

Jean Goubault-Larrecq *
G.I.E. Dyade, Inria bâtiment 3, Domaine de Voluceau, F-78153 Le Chesnay Cedex
Tel: +33 1 39 63 56 55 Fax: +33 1 39 63 58 66 Jean.Goubault@inria.fr

December 10, 1996

Abstract

While unification in the simple theory of types (a.k.a. higher-order logic) is undecidable, we show that unification in the pure ramified theory of types with integer levels is decidable. Since pure ramified type theory is not very expressive, we examine the impure case, which has an undecidable unification problem even at order 2. However, the decidability result for the pure subsystem indicates that unification terminates more often than general higher-order unification. We present an application to ACA_0 and other expressive subsystems of second-order Peano arithmetic.

This is a revised version of the report 1996-31, University of Karlruhe; a few bugs are corrected and a "related works" section has been added.

1 Introduction

Higher-order logic is one of the most expressive formalisms in which we can express and prove theorems. Bertrand Russell proposed two ways of formalizing it. In *ramified type theory* [WR27], expressions are stratified in a double hierarchy of *types* (individuals, sets, sets of sets, etc.) and of predication *levels* (corresponding to times of definition, and called orders by them); but the resulting logical system is too weak to found mathematics, and so-called reducibility axioms are called for. In *simple type theory*, levels are dispensed with; the resulting language, as further simplified by Church, is the simply-typed λ-calculus with additional constants representing logical and non-logical symbols [Chu40]. (See also [And86].)

A central problem in automated theorem proving and logic programming is that of unification, i.e. deciding whether two terms have a common instance. Unfortunately, unification in the simple theory of types is undecidable even at order 2 [Gol81]. Our original idea was to refine the study of higher-order unification by reintroducing ramification. This seemed an interesting idea for two complementary reasons: first, we shall see that ramified higher-order unification in its pure form is decidable; second, we can, in theory, get back the full power of simple type theory by introducing Whitehead and Russell's axioms of reducibility. Controlling the use of these axioms would then provide a natural way of controlling the search for unifiers in simple type theory. It also seemed interesting even if we didn't allow for these axioms. Indeed, ramified type theory has natural restrictions that allow us to formalize weak subsystems of arithmetic like ACA_0 or ATR_0, which are still strong enough to formalize most of mathematics [Sim85]. However, this is only valid in some impure form of ramified type theory, for which unification is undecidable again (see Section 8.2). But the decidability result for the pure subsystem indicates that unification should fail to terminate less often than general higher-order unification (in the simple theory of types), and so impure ramified type theory should be of practical value in implementing automated proof methods for the subsystems of arithmetic mentioned above.

We may sum up the contributions of this paper as follows. First, we formalize ramified type theory in both a simple and general way, which does not seem to have been done before. Then, we prove that unification in this pure system is decidable (Corollary 30). We also provide two negative results: first, that this system is much too weak to formalize any useful fragment of logic; and second, that the most natural extension that cures this problem, ramified type theory with operators, is undecidable even at order 2 —

*Research funded by the HCM grant 7532.7-06 from the European Union, done at the Institut für Logik, Komplexität und Deduktionssysteme, Universität Karlsruhe, D-76128 Karlsruhe, Germany, while on leave from Bull Corporate Research Center, rue Jean Jaurès, F-78340 Les Clayes-sous-Bois.

Goldfarb's encoding still works (Theorem 31). Finally, we argue that the latter theory should be of practical value for automating proof search in ACA_0 and related systems.

The reader willing to learn more about ramification is directed to [LN95] for a modern analysis of what Whitehead and Russell may have meant precisely by ramification and predicativity. Modern accounts of these topics are somewhat different, and we shall base our study on Hazen's point of view [Haz83]; see also [Cop71, Chu76].

The plan of the paper is as follows. We first investigate related works in Section 2. We formalize a variant of ramified type theory as a particular type system for the λ-calculus with equality defined by the β and η rules. This will take Section 3. We then recall Gallier and Snyder's formalization of Huet's higher-order unification procedure, and give the rough ideas for adapting it to ramified type theory in Section 4. The grunt work starts in Section 5, where we establish all needed basic theorems for our ramified calculus. We show that inferring levels of expressions is decidable in Section 6; we need this to check that a unifier in the simple theory of types is in fact one in the ramified theory. Then, in Section 7, we replay the ideas expounded in Section 4 in a more formal way. We also deal with the β-case (without η) in Section 7.2. Finally, we try to put these ideas to work in Section 8, and come up with our negative results: the logic is too weak, and adding operators (see Section 8.2) to cure the problem makes the unification problem undecidable again. Section 9 shows how ACA_0 and related subsystems of second-order Peano arithmetic are naturally encoded in the latter impure ramified system. We conclude in Section 10.

2 Related Works

Ramification has seldom been studied in computer science (apart from work by Leivant and others on ramified recursion, which is not relevant here). Laan and Nederpelt disected Whitehead and Russell's attempts at defining ramified, or predicative, systems of logics in [LN95, Laa95, Laa96], producing a remarkably legible and formal account of Whitehead and Russell's ideas. Kamareddine and Laan [KL96a] then used this formalization to show that the use of hierarchy of universes in Martin-Löf's theory of types and other logical frameworks was basically an instance of ramification at the level of types. Another recent investigation by the same authors [KL96b] shows that the use of levels without types themselves naturally leads to a logical system by Kripke [Kri75] that is able to express its own notion of truth without being inconsistent.

Modern views on ramification and predicativity differ somewhat, not only from Russell's, but also from each other. The term "predicative" tends to be used whenever systems of logic are built in progressions, starting from fairly basic systems, and building more and more powerful systems by induction on ordinals [Fef75, Fef68b, Fef64]; or building the progression inside the system itself [Fef68a, Fef78]. Applications have led to the identification of Γ_0 as the least impredicative ordinal [Gal91], or to the discovery of weak subsystems of second-order Peano arithmetic which are still strong enough to formalize most of mathematics [Sim85].

Returning to what is indended nowadays as ramified higher-order logic, Church [Chu76] presents a system of ramified second-order logic that enables him to show that the stratification of formulas in levels is a syntactic analogue of the Tarskian distinction between language and meta-language, iterated. Church's system is faithful to the spirit of ramification, but introduces a few *ad hoc* features; notably, substitution of terms for variables is only defined in a few restricted cases. In this paper, we adopt Hazen's view [Haz83] on ramified logics, because it is general and formal enough (see notably pages 345–346). We elaborate on it, however, because we wish to reuse Huet's higher-order unification procedure as basis, and therefore we have to cast ramified systems as particular typed λ-calculi. This is close to [Laa95], Sections 5 and 9, except that we take up Hazen's point of view instead of Russell's.

For an alternative presentation of ramification, see [Cop71]. Further references are given in Hazen (op.cit.) and Copi (op.cit.)

On the other hand, works on higher-order unification have been many since Huet presented the first successful procedure for higher-order preunification in [Hue75]. (See this reference for previous work by Guard, Gould, and Pietrzykowski and Jensen notably.) Numerous extensions have been studied, notably to logics built from richer type systems: higher-order logic with base types organized in an order-sorted hierarchy, and with type variables [Nip90]; dependent type theories like that of LF [Ell89]; or even logics where the types of formulae are taken from a suitable fragment of linear logic [CP96], to name a few.

A few restrictions have also been proposed to make the resulting unification problem decidable: the most well-known is Miller's higher-order patterns [Mil91]. Prehofer [Pre94] shows that unification of linear pattern/second-order term pairs, and of linear second-order term/second-order term pairs, where the terms in the pairs share no variable, are decidable as well. Finally, Farmer [Far88] shows that *monadic* second-order unification is decidable, and Schmidt-Schauß [SS94] shows that *stratified* second-order unification is decidable, by reductions to the solvability of word equations (i.e., Makanin's algorithm).

Ramified unification falls in none of the above categories, and conversely: $\lambda x \cdot x(Xx)$, where X is free, is a pattern but is not ramified (see Theorem 8), whereas $\lambda x \cdot X(\lambda y \cdot yxx)$ is ramified (it has the type e.g., $\tau_1 \xrightarrow{0} \tau_2/3$ if $X : ((\tau_1 \xrightarrow{0} \tau_1 \xrightarrow{0} \tau_3) \xrightarrow{1} \tau_3) \xrightarrow{2} \tau_2/3)$. Moreover, ramified terms are not restricted to second-order terms. They are not necessarily monadic either (replace y by some binary function in the last example). They are not necessarily stratified (the second-order prefixes of positions of given variables X in a term may differ [SS94], as in $\lambda x \cdot xX(\lambda y \cdot yX)$). Conversely, second-order monadic terms need not be ramified: by Theorem 8 again, the monadic term $\lambda x \cdot X(Xx)$ (a variant of Church's integer 2) is not ramified. It is not clear whether there are non-ramified stratified second-order terms; anyway, the unification problem that Schmidt-Schauß considers differs in that his second-order variables can only be instantiated to terms of smaller arity.

3 Ramified Type Theory

In this section, we propose a formalization of ramified type theory in a notation that we hope will be familiar to computer scientists, namely a type system for Church's λ-calculus. We also discuss how well it fits with usual views on ramification, but bear in mind that our main objective is to find decidable restrictions of higher-order unification through ramification.

Let ST be the algebra of *simple types*: we have a non-empty set B of *base types* (among which, typically, the types ι of individuals and o of propositions), and simple types ϵ are either base types, or function types $\epsilon_1 \to \epsilon_2$.

We define the algebra RT of *ramified types* by adding *levels* ℓ taken from an initial segment L of the ordinals, i.e. L is an ordinal. In the sequel, we shall assume that L is a *limit ordinal*, i.e. a non-zero ordinal that is not a successor, so that for every level ℓ there is a greater one in L. We shall really be interested in integer levels, i.e. $L = \omega$, as in the original works by Whitehead and Russell. It should be noted that there is nothing wrong in taking for L some segment of the computable ordinals. The idea originates with Gödel and Wang, and culminated in Feferman's works on autonomous progressions of predicative systems, where $L = \Gamma_0$ [Haz83].

Definition 1 (Ramified Types) *We define both the set RT of* ramified types τ *and their* levels $l(\tau)$ *by simultaneous induction as follows.*

The ramified types *are either base types b — and we let $l(b) = 0$ — or function types $\tau_1 \xrightarrow{\ell} \tau_2$, where τ_1 and τ_2 are ramified types and ℓ is a level such that $\ell \geq l(\tau_1)$ — then we let $l(\tau_1 \xrightarrow{\ell} \tau_2) = \max(\ell + 1, l(\tau_2))$.*

We assume that $\to$ and $\xrightarrow{\ell}$ associate to the right, so that $\epsilon_1 \to \epsilon_2 \to \epsilon_3$ denotes $\epsilon_1 \to (\epsilon_2 \to \epsilon_3)$, for example.

The idea is that, while the simple type $\epsilon_1 \to \epsilon_2$ is the type of all total functions from ϵ_1 to ϵ_2, $\tau_1 \xrightarrow{\ell} \tau_2$ is the type of all those functions that act on concepts defined at or after time ℓ. (Not just at time ℓ: contrarily to Whitehead and Russell, but following Hazen, we consider the hierarchy of levels to be cumulative, i.e. everything at level ℓ is also at all higher levels.)

There is a simple connection between simple and ramified types: let the *erasing map* be defined on ramified types as follows: $E(b) = b$ for all base types b, and $E(\tau_1 \xrightarrow{\ell} \tau_2) = E(\tau_1) \to E(\tau_2)$.

The *order* $r(\tau)$ of a ramified type τ is defined inductively as follows. For any base type b, $r(b) = 0$; and $r(\tau_1 \xrightarrow{\ell} \tau_2) = \max(r(\tau_1) + 1, r(\tau_2))$. Levels do not play any role here, and orders of simple types are defined similarly.

We now consider a variant of the λ-calculus [Bar84] as the basic language for building terms and formulas. The λ-terms s, t, u, v, ... are *variables* x, y, z, ..., *constants* c, *applications* uv, and *abstractions* $\lambda x \cdot u$. We assume that application associates to the left, i.e. uvw denotes $(uv)w$, and that abstractions extend as much right as possible. We also denote by $\mathrm{FVC}(s)$ the set of free variables or constants of s. (Think of constants as ordinary variables of the λ-calculus, with the difference that we cannot substitute any term for them.) To avoid variable capture problems, we also adopt Barendregt's convention that no free variables occurs bound, and that no two occurrences of λ bind the same variables (this involves some renaming of bound variables). Substitutions σ are finite maps from variables $x_1, \ldots, x_n$ to λ-terms $v_1, \ldots, v_n$, and are written $[v_1/x_1, \ldots, v_n/x_n]$. The application $u\sigma$ of the substitution σ to the term u is then defined straightforwardly as textual substitution; the composition $\sigma\sigma'$ is the only substitution such that $u(\sigma\sigma') = (u\sigma)\sigma'$ for all terms u; and the *domain* $\mathrm{dom}\, \sigma$ of σ is the set of variables x such that $x \neq x\sigma$.

We consider that the variables and constants have a uniquely determined ramified type τ and level ℓ such that $\ell \geq l(\tau)$. To emphasize it, we shall write x_τ^ℓ instead of x to state that x is given type τ and level ℓ (resp. c_τ^ℓ instead of c). When context permits, we shall leave these annotations implicit.

We consider that two terms that differ only by a change of bound variables of the same *simple* type are equal (α-renaming), i.e. $\lambda x_\tau^\ell \cdot u = \lambda y_{\tau'}^{\ell'} \cdot u[y_{\tau'}^{\ell'}/x_\tau^\ell]$ provided that $E(\tau) = E(\tau')$. We shall explain why in Section 3.1.

The calculus is then endowed with the following two standard reduction relations:

$$(\beta) \quad (\lambda x \cdot u)v \to u[v/x]$$
$$(\eta) \quad \lambda x \cdot ux \to u \quad \text{(provided } x \text{ is not free in } u\text{)}$$

We write $\longrightarrow_\beta$ the smallest relation on terms containing (β) and stable by context application. More formally, call a *context* any term with exactly one hole $[]$: the contexts $\mathcal{C}$ are described by the grammar:

$$\mathcal{C} ::= [] \mid \mathcal{C}T \mid T\mathcal{C} \mid \lambda X \cdot \mathcal{C}$$

where T is the set of terms and X the set of variables. We note $\mathcal{C}[t]$ the result of replacing the hole in $\mathcal{C}$ with the term (or context) t. The $\longrightarrow_\beta$ relation is then defined as $\mathcal{C}[s] \longrightarrow_\beta \mathcal{C}[t]$ for every context $\mathcal{C}$, and every β-redex s ($= (\lambda x \cdot u)v$) whose contractum ($= u[v/x]$) is t. We write $\xrightarrow{*}_\beta$ its reflexive transitive closure, $\xrightarrow{+}_\beta$ its transitive closure, and $=_\beta$ its reflexive symmetric transitive closure. Similarly with $\longrightarrow_{\beta\eta}$, $\xrightarrow{*}_{\beta\eta}$, $\xrightarrow{+}_{\beta\eta}$ and $=_{\beta\eta}$, etc.

Because of our α-renaming convention, we shall write abstractions as $\lambda x_\epsilon \cdot u$, where ϵ is the erasure of any ramified type that decorates the bound variable x, and we forget about the level labelling x. Although this seems to defeat the purpose of ramification, this is not so. We shall see, for example, that higher-order ramified unification with integer levels is decidable, although it is not so in the simply-typed case.

Finally, we consider the typing rules shown in Figure 1. A decorated type τ/ℓ is just a pair of a type-to-be τ and a level ℓ. These rules derive a decorated type τ/ℓ for each expression u as a judgement $u : \tau/\ell$. We shall see (Lemma 2) that, whenever we can derive $u : \tau/\ell$, then τ will really be a ramified type and $\ell \geq l(\tau)$ will hold. The latter is a condition of ramification, saying that u cannot be defined earlier than its constituents, and is analogous to the constraint that we have imposed on function types in Definition 1. Furthermore, rule (Abs) includes explictly the α-conversion rule. We might have left it implicit by stating (Abs) as:

$$\frac{u : \tau_2/\ell_2}{\lambda x_{\tau_1}^{\ell_1} \cdot u : \tau_1 \xrightarrow{\ell_1} \tau_2/\max(\ell_1+1, \ell_2)}$$

and stating that if $u : \tau/\ell$ is derivable, then any α-variant of u also has type τ and level ℓ, but we feel that rule (Abs) is more readable. Notice also that, in rule (App), the level ℓ_1 on the arrow type of u is exactly the level of v, and that the level ℓ_2 of uv is exactly that of u itself.

Type and level annotations on variables and constants are used in rule (Var). Recall that, for every variable x_τ^ℓ, τ is assumed to be a ramified type and ℓ is assumed to be at least $l(\tau)$. Observe that this implicitly means that τ_1 is a ramified type and $\ell_1 \geq l(\tau_1)$ in rule (Abs). In all other cases, we do not require the objects τ, τ_1, τ_2, τ_1', τ_2' to be types, unless explicitly required (in rule (Rfl)). We shall however see (Lemma 2) that, in any derivation, they will be ramified types.

Although it is not assumed in usual texts on ramified type theory [Haz83], we also assume a subtype relation $\sqsubseteq$ to express the cumulativity of levels. (We need this in establishing the basic theorems of Section 5.) That the levels are cumulative means that if an object is typed at some level ℓ, then it is also typed at all higher levels. Now, intuitively, every object of type $\tau_1 \xrightarrow{\ell} \tau_2$ is a function that can be applied to objects of level ℓ, or by cumulativity to objects of levels at most ℓ. So it is consistent to require every object of type $\tau_1 \xrightarrow{\ell} \tau_2$ to be of type $\tau_1 \xrightarrow{\ell'} \tau_2$ also, with $\ell' \leq \ell$: this is the gist of rules (Cml) and (Sub).

Our system is slightly richer than usual ramified systems, in that everything at some level is also at all higher levels. Usually [Haz83], it is just assumed that we may promote v to a higher level when using it as an argument of some function symbol f or of some other term u. In particular, first-order variables (variables of base types) are usually taken to be at level 0, where 0 is the least level; we allow them to be at any level we wish. This is naturally not essential.

We finally define:

Definition 2 *For every ordinals $\alpha \leq \omega$ and β, we let λ_α^β be the set of λ-terms that are typable in the ramified type theory, and such that:*

- *all types of subterms have order $< \alpha$;*
- *the set L of levels is the set of all ordinals $< \beta$.*

Therefore, the theory of Whitehead and Russell with cumulative levels would be λ_ω^ω, while Feferman's systems could be expressed in $\lambda_2^{\Gamma_0}$.

$$\frac{}{x^\ell_\tau : \tau/\ell}\,(Var)$$
(x variable or constant)

$$\frac{u : \tau/\ell \quad \tau \sqsubseteq \tau' \quad \ell \leq \ell'}{u : \tau'/\ell'}\,(Cml)$$

$$\frac{u : \tau_1 \xrightarrow{\ell_1} \tau_2/\ell_2 \quad v : \tau_1/\ell_1}{uv : \tau_2/\ell_2}\,(App)$$

$$\frac{u[y^{\ell_1}_{\tau_1}/x] : \tau_2/\ell_2 \quad \epsilon = E(\tau_1) \quad y \notin \mathrm{FVC}(u)}{\lambda x_\epsilon \cdot u : \tau_1 \xrightarrow{\ell_1} \tau_2/\max(\ell_1 + 1, \ell_2)}\,(Abs)$$

$$\frac{b \text{ base type}}{b \sqsubseteq b}\,(Rfl)$$

$$\frac{\tau_1' \sqsubseteq \tau_1 \quad \tau_2 \sqsubseteq \tau_2' \quad l(\tau_1') \leq \ell' \leq \ell}{\tau_1 \xrightarrow{\ell} \tau_2 \sqsubseteq \tau_1' \xrightarrow{\ell'} \tau_2'}\,(Sub)$$

Figure 1: Typing rules

3.1 α-Renaming and Comprehension Schemes

It would seem more natural to use a more restricted α-renaming rule, whereby $\lambda x^\ell_\tau \cdot u = \lambda y^{\ell'}_{\tau'} \cdot u[y^{\ell'}_{\tau'}/x^\ell_\tau]$ provided that $\tau = \tau'$ and $\ell = \ell'$. However, in the presence of η, this is not enough to ensure that the resulting calculus is confluent: if x is not free in u, then $\lambda x^\ell_\tau \cdot (\lambda y^{\ell'}_{\tau'} \cdot u)x$ reduces by β to $\lambda x^\ell_\tau \cdot u[x/y]$, or by η to $\lambda y^{\ell'}_{\tau'} \cdot u$. (This is Nederpelt's counterexample.) If the first term is typable, then intuitively τ and τ' will have the same erasures, and this is why we consider the less restricted α-equivalence to be able to consider them equal.

Let us analyze the paradox in intuitive terms. The λ-notation is short for expressing the comprehension axioms of ramified type theory, i.e. (omitting a few details) for every term u there is a term v such that $\forall x^\ell_\tau \cdot (vx \equiv u)$, where $\equiv$ is a suitable notion of equality. (Well, in general we have to require more, namely there is a term v such that $\forall x_1, \ldots, x_n \cdot (vx_1 \ldots x_n \equiv u)$, where $x_1, \ldots, x_n$ are the free variables of u.) If we take $\equiv$ to mean coextensionality (as in [Chu76], for example), then $v_1 \equiv v_2$ if and only if $\forall x_1, \ldots, x_n \cdot v_1 x_1 \ldots x_n \Leftrightarrow v_2 x_1 \ldots x_n$, where $x_1, \ldots, x_n$ are variable containing all those of v_1 and v_2, and the usual type and level provisos are obeyed. We have decided to write $\lambda x^\ell_\tau \cdot u$ for such a term v. Now, assuming that $\equiv$ is a congruence (in particular, passes to context), it follows that η is valid, i.e. transforms a term into an $\equiv$-equal term. The real problem is that $\equiv$ may not be a congruence: indeed, the rule that $u \equiv v$ implies $fu \equiv fv$ is not deductible in general.

On the other hand, if we choose for $\equiv$ a form of Leibniz equality, namely $v_1 \equiv v_2$ if and only if for every property P of suitable type and level, Pv_1 holds if and only if Pv_2 holds, then there is no reason why η-equality should hold. But there is no reason why the rule ξ: $u \equiv v$ implies $\lambda x \cdot u \equiv \lambda x \cdot v$, should hold either. I.e., if we take the purely intensional route to ramified logics (as described and discussed in [Cop71]), the λ-notation itself is at stake. This is discussed further in [Haz83], Section 2.

All this boils down to the fact that we have just made a choice of a particular ramified logic, with rather strong extensionality conditions expressed by the η rule and a liberalized α-conversion rule. This is not a departure from usual predicative logics, while it certainly simplifies the presentation of the framework. We shall examine briefly what happens if we drop the η rule in Section 7.2.

For more about the λ-notation and its relations with the comprehension scheme, see [Dow95].

4 Unification

Say that a substitution σ is *well-typed* if and only if it binds every variable x^ℓ_τ to a term $x\sigma$ of type τ/ℓ; equivalently, to a term $x\sigma$ of type τ'/ℓ' such that $\tau' \sqsubseteq \tau$ and $\ell' \leq \ell$. Similarly, we say that a substitution σ is *well-simply-typed* if and only if it binds every variable x^ℓ_τ to a term $x\sigma$ of simple type $E(\tau)$.

We write $\{\!|a, b, c, \ldots|\!\}$ the multiset consisting of a, b, c, ... and $\uplus$ for multiset union. The notation $\langle a, b\rangle$ denotes an unordered pair, which is the same as $\langle b, a\rangle$.

The unifiability problem in any language λ^β_α is the following:

INPUT: a finite multiset M of unordered pairs of simply-typed terms in β-normal form $\{\!|\langle u_i, v_i\rangle \mid 1 \leq i \leq n|\!\}$ in λ^β_α.

QUESTION: is there a well-typed substitution σ such that $u_i\sigma =_{\beta\eta} v_i$ for every i, $1 \leq i \leq n$?

This problem is so close to unifiability in the simple theory of types (just ignore the levels) that a variant of Huet's procedure [Hue75], augmented with level checks, will solve the problem. The idea is simple: if σ is a ramified unifier of M, then $u_i\sigma =_{\beta\eta} v_i$ for every i, $1 \leq i \leq n$, so σ is also a unifier of M is simple type theory. To find such σ's, therefore, apply Huet's procedure, and at specified times check that the current partial unifier has some instances that are erasures of well-typed ramified substitutions σ. The only difficulty lies in the latter: level checks need to be carefully designed.

4.1 Type Restrictions

With unification in the simple theory of types, the input M could be restricted so that u_i and v_i had the same type for each i, $1 \leq i \leq n$; otherwise, there could not be any unifier for M. The reason is that:

(R1) if u has simple type ϵ, then for any well-simply-typed substitution σ, $u\sigma$ has type ϵ, and

(R2) if for some well-simply-typed substitution σ, $u\sigma$ has simple type ϵ, then u has type ϵ.

(R3) if u has simple type ϵ and $u \xrightarrow{*}_{\beta\eta} v$, then v has simple type ϵ (subject reduction), and

(R4) if u has a simple type and v has simple type ϵ and $u \xrightarrow{*}_{\beta\eta} v$, then u has simple type ϵ.

(R3) in fact follows from (R1), and (R4) follows from (R3) and the fact that each term has a unique simple type.

In the ramified theory of types, (R1) still holds if we read decorated type instead of type: this is Theorem 10, to be proved in Section 5. However, (R2) does not hold. For instance, let u be x^6_τ and σ be $[a^2_\tau/x]$, where a is some constant. Then the types of u are all those that are at least $\tau/6$ in the $\sqsubseteq \times \leq$ ordering on decorated types, while those of $u\sigma$ are those of a^2_τ, namely all types that are at least $\tau/2$. In particular, $u\sigma$ has types $\tau/2$, $\tau/3$, $\tau/4$, $\tau/5$, which are not types of u.

So, we cannot restrict ourselves to pairs $\langle u_i, v_i\rangle$ having the same decorated types. However, let's say that two decorated types τ/ℓ and τ'/ℓ' are *compatible* if and only if $E(\tau) = E(\tau')$. Then we can assume that the input to the unification problem consists of pairs $\langle u_i, v_i\rangle$ of typed terms with *compatible* types. Otherwise, the unification problem trivially has no solution.

4.2 Huet's Procedure

We recall how Huet's procedure works. There is nothing new in this section, as it is but a summary of [SG89].

Assume that all the terms that we use are simply typed. Huet's procedure not only tests for unifiability, but returns a complete set of preunifiers, in a sense which we shall recall shortly.

First, we only use terms in β-normal form. This is possible because all terms are terminating. We shall write $u\downarrow$ for the unique β-normal form of the term u, and similarly for multisets of pairs of terms, etc.

These normal forms are of the form $\lambda x_1 \cdot \ldots \cdot \lambda x_m \cdot a u_1 \ldots u_n$, where $m \geq 0$, $n \geq 0$, a is a variable or a constant called the *head* of the term, and $u_1, \ldots, u_n$ are terms in β-normal form. For brevity, we write this as $\lambda\overline{x_m} \cdot a\overline{u_n}$. If a is a constant or a bound variable x_i, $1 \leq i \leq m$, then we say that the term is *rigid*; it is *flexible* otherwise.

The *η-expanded form* $\eta[u]$ of a term u is defined as follows: if $u = \lambda\overline{x_m} \cdot a\overline{u_n}$ and u has simple type $\epsilon_1 \to \ldots \to \epsilon_m \to \epsilon_{m+1} \to \ldots \to \epsilon_{m+k} \to b$, where $k \geq 0$ and b is a base type, then $\eta[u]$ equals:

$$\lambda x_1 \cdot \ldots \cdot \lambda x_m \cdot \lambda x_{m+1\,\epsilon_{m+1}} \cdot \ldots \cdot \lambda x_{m+k\,\epsilon_{m+k}} \cdot a(\eta[u_1]) \ldots (\eta[u_n])(\eta[x_{m+1}]) \ldots (\eta[x_{m+k}])$$

Then, $u_i\sigma =_{\beta\eta} v_i\sigma$ if and only if $\eta[u_i]\eta[\sigma] =_\beta \eta[v_i]\eta[\sigma]$, where $\eta[\sigma]$ is the substitution mapping each $x \in \mathrm{dom}\,\sigma$ to $\eta[x\sigma]$; i.e., we only need compare terms by β-equality. Moreover, we may look for unifiers σ that are *idempotent*, i.e. such that $\mathrm{FVC}(x\sigma) \cap \mathrm{dom}\,\sigma = \emptyset$ for every $x \in \mathrm{dom}\,\sigma$, and *normalized*, i.e. $x\sigma$ is an η-expanded form for every $x \in \mathrm{dom}\,\sigma$. This is Section 2 of [SG89].

The unification procedure works by stepwise transformation of systems S, i.e. of multisets of unordered pairs of terms with the same simple types. A pair $\langle\eta[x], \eta[s]\rangle$ is *in solved form in* S if and only if the only occurrence of x in S is in the $\eta[x]$ component of the given pair. S is itself *solved* if and only if all its pairs are solved in S. As usual, a solved system $\{\!|\langle\eta[x_1], \eta[s_1]\rangle, \ldots, \langle\eta[x_k], \eta[s_k]\rangle|\!\}$ defines an idempotent and normalized substitution $[\eta[s_1]/x_1, \ldots, \eta[s_k]/x_k]$.

To approximate individual bindings, we need the following notion of partial bindings (see [SG89], Definition 4.8):

Definition 3 *Let ϵ be a simple type, which we write $\epsilon_1 \to \ldots \epsilon_n \to b$ with b a base type, and a be a variable or constant of simple type $\epsilon'_1 \to \ldots \epsilon'_m \to b$.*

For each $1 \leq i \leq m$, let's write ϵ'_i as ${\epsilon''}^i_1 \to \ldots {\epsilon''}^i_{p_i} \to b_i$, where b_i is a base type.

The set $B(\epsilon, a)$ of partial bindings *of type ϵ and head a is the set of terms of the form:*

$$\lambda y_{1\,\epsilon_1} \cdot \ldots \cdot \lambda y_{n\,\epsilon_n} \cdot a(\lambda\overline{z^1_{p_1}} \cdot H_1\overline{y_n}\,\overline{z^1_{p_1}}) \ldots (\lambda\overline{z^m_{p_m}} \cdot H_m\overline{y_n}\,\overline{z^m_{p_m}})$$

where:

1. *for every* $1 \leq i \leq m$, $1 \leq j \leq p_i$, z^i_j *is a variable* $(z^i_j)_{\epsilon''^i_j}$;

2. *for every* $1 \leq i \leq m$, H_i *is a variable* $(H_i)_{\epsilon_1 \to \ldots \epsilon_n \to \epsilon''^i_1 \to \ldots \epsilon''^i_{p_i} \to b_i}$;

If u has type ϵ, the set $B(u, a)$ of partial bindings of head a appropriate to u *is* $B(\epsilon, a)$.

A partial binding is an imitation binding *for a if a is either a function symbol or a free variable; it is a* projection binding *if a is one of the bound variables y_i, $1 \leq i \leq n$; we let $PB_i(\epsilon)$ and $PB_i(u)$ denote $B(\epsilon, y_i)$ and $B(u, y_i)$ respectively.*

Notice that any element of $B(\epsilon, a)$ has type ϵ. We say that a partial binding has *fresh head variables* if and only if $H_1, \ldots, H_m$ are pairwise distinct and do not occur free in the context.

Finding solved forms is not required in most applications, and the search for them involves a highly non-deterministic procedure [SG89]. Huet's solution was to modify the notion of solved form so that flexible-flexible pairs (i.e., pairs $\langle \lambda \overline{x_k} \cdot F \overline{u_m}, \lambda \overline{x_k} \cdot G \overline{v_n} \rangle$, with F and G free variables) would be considered as solved. We say that a pair in S is *presolved in S* if and only if it is either solved or consists of two flexible terms. A system S is in *presolved form* if and only if it consists in presolved pairs.

The trick is that any flexible-flexible pair is always unifiable: for each simple type ϵ, which we write $\epsilon_1 \to \ldots \epsilon_n \to b$ with b a base type, let $\hat{e}_\epsilon$ be the term $\lambda x_{1\,\epsilon_1} \cdot \ldots \cdot \lambda x_{n\,\epsilon_n} \cdot v_b$, where v_b is a distinguished variable that will never be used in any other term. Letting ζ be the substitution mapping each variable of type ϵ to $\hat{e}_\epsilon$, and letting σ be the substitution corresponding to the non-flexible-flexible pairs in S, then $\sigma\zeta$ is always a unifier of S, hence of M. Such a σ is called the *preunifier* associated with S.

Gallier and Snyder's rules for preunification are those of Figure 2. These rules are sound (in the sense that all presolved forms denote preunifiers) and complete (in that every unifier is an instance of some preunifier derived from the rules). Moreover, Gallier and Snyder show that **(Delete)**, **(Decomp)** and **(Bind)** can be applied eagerly (in particular, right after **(Imitate)** or **(Project)**). And, as far as the flexible-rigid pair on which to apply **(Imitate)** or **(Project)** is concerned, there is no non-determinism associated to its choice, and we can commit ourselves to any pair we please. The restriction "F free in S" in **(Bind)** is ours, as well as "F unsolved" in **(Imitate)** and **(Project)**; this preserves soundness and completeness, and is intended to prevent this rule from being applied indefinitely in the ramified case.

4.3 Introducing Ramification

What do we have to do to adapt this procedure to ramified type theory? Basically, we have to check all the constraints between levels that we have left out by reasoning in the simple theory of types.

The first thing is, given a system S in solved form describing a unifier σ, to check whether there is a normalized instance $(\sigma\sigma') \downarrow$ of σ that is a well-typed *ramified* substitution, at least when restricted to $\mathrm{FVC}(S)$ and with proper levels annotating types and all new free variables.

We shall attack this problem by showing that it reduces to a simpler subproblem (the reduction is the content of the technical Lemma 13 to come). The simpler problem is: can we put back levels on type arrows and free variables of a term, or of several terms, so that it becomes well-typed in the ramified sense? It will turn out that this is a rather simple problem, which we shall solve by graph-theoretic techniques in Section 6.

Once we have overcome these small hurdles, it will be clear that ramified higher-order unification is not much more complicated that unification in the simple theory of types. In fact, in terms of algorithmic complexity, is is much simpler, at least in the case where all levels are integer: indeed, the main claim of this paper is that it is *decidable*.

The intuitive reason why is the following. Assume that σ is a unifier. In rules **(Imitate)** and **(Project)**, the partial binding t is of the general form $\lambda \overline{y_n} \cdot a(\overline{\lambda \overline{z_{p_m}} \cdot H_m \overline{y_n}\ \overline{z_{p_m}}})$, to take a condensed notation due to Gallier and Snyder. Now, modulo σ, the level of F should be that of t, which by (Abs) should be at least the level ℓ of $a(\overline{\lambda \overline{z_{p_m}} \cdot H_m \overline{y_n}\ \overline{z_{p_m}}})$. The level of a itself should then be ℓ by rule (App), but it can only apply to objects of strictly lower levels ℓ_i, $1 \leq i \leq n$ (the type of a is of the form $\tau_1 \overset{\ell_1}{\to} \ldots \tau_m \overset{\ell_m}{\to} \tau/\ell$, with $\ell \geq \max(\ell_1 + 1, \ldots, \ell_m + 1, l(\tau))$). So the level of F is strictly more than that of $\lambda \overline{z_{p_i}} \cdot H_i \overline{y_m}\ \overline{z_{p_i}}$, in particular than that of H_i, for each i.

Now in any derivation $M = S_0 \to S_1 \to \ldots \to S_m \to \ldots$ following the rules of Figure 2, the only rules that create new variables are the above two, and they must create variables of lower and lower ordinal levels. Therefore, if we keep track of levels of free variables as above, these rules can be invoked only finitely many times in such a derivation. And since **(Delete)**, **(Decomp)** and **(Bind)** terminate, any de-

(Delete)	$S' \uplus \{\|\langle u, u\rangle\|\}$	$\longrightarrow S'$
(Decomp)	$S' \uplus \{\|\langle \lambda\overline{x_k} \cdot a\overline{u_n}, \lambda\overline{x_k} \cdot a\overline{v_n}\|\}$	$\longrightarrow S' \uplus \{\|\langle \lambda\overline{x_k} \cdot u_1, \lambda\overline{x_k} \cdot v_1\rangle, \ldots, \langle \lambda\overline{x_k} \cdot u_n, \lambda\overline{x_k} \cdot v_n\rangle\|\}$
	where a is a constant or a bound variable x_i, $1 \leq i \leq k$	
(Bind)	$S' \uplus \{\|\langle \lambda\overline{x_k} \cdot F\overline{x_k}, \lambda\overline{x_k} \cdot v\rangle\|\}$	$\longrightarrow (S'[\lambda\overline{x_k} \cdot v/F]) \downarrow \uplus \{\|\langle \lambda\overline{x_k} \cdot F\overline{x_k}, \lambda\overline{x_k} \cdot v\rangle\|\}$
	with F variable not free in $\lambda\overline{x_k} \cdot v$, $F \notin \{x_1, \ldots, x_k\}$, F free in S'	
(Imitate)	$S' \uplus \{\|\langle \lambda\overline{x_k} \cdot F\overline{u_n}, \lambda\overline{x_k} \cdot a\overline{v_m}\rangle\|\}$	$\longrightarrow S' \uplus \{\|\langle \eta[F], t\rangle, \langle \lambda\overline{x_k} \cdot F\overline{u_n}, \lambda\overline{x_k} \cdot a\overline{v_m}\rangle\|\}$
	with F unsolved variable other than $x_1, \ldots, x_k$, a constant, and $t \in B(F, a)$ with fresh head variables	
(Project)	$S' \uplus \{\|\langle \lambda\overline{x_k} \cdot F\overline{u_n}, \lambda\overline{x_k} \cdot a\overline{v_m}\rangle\|\}$	$\longrightarrow S' \uplus \{\|\langle \eta[F], t\rangle, \langle \lambda\overline{x_k} \cdot F\overline{u_n}, \lambda\overline{x_k} \cdot a\overline{v_m}\rangle\|\}$
	with F unsolved variable other than $x_1, \ldots, x_k$, a constant or bound variable, $t \in PB_i(F)$ with fresh head variables and such that the head of u_i is a	

Figure 2: Rules for higher-order preunification

rivation from M must terminate.

As it stands, this informal argument is not entirely correct, but it gives the rough idea. We shall formalize it in Section 7.

When the set of levels is ω or smaller, it follows by König's Lemma that the whole modified procedure terminates; for the language λ^ω_ω, for instance, whenever we know that the level of F should not exceed ℓ, we know that those of $H_1, \ldots, H_m$ should not exceed $\ell - 1$. Then, the set of all derivations as above forms a finitely-branching tree whose branches are all finite, so by König's Lemma, this tree must be finite, yielding a terminating algorithm for unifiability in λ^ω_ω.

However, for larger sets of levels, this is not so easy; indeed, when keeping track of levels as above, whenever ℓ is a limit ordinal, we cannot compute $\ell - 1$, and we have to guess a new bound $\ell_i < \ell$ on the level of H_i. There are always infinitely many such possible bounds, because the cofinality of any limit ordinal is at least ω. Then, although all branches of the search-tree are finite, the rules that need to produce a partial binding must be replicated in an infinite-branching non-deterministic choice, which invalidates the above argument. We conjecture that unification in λ^β_α, for α and β large enough, is undecidable. More precisely, we conjecture that unification in $\lambda^{\omega+1}_2$ is undecidable.

5 Basic Theorems

We first study the $\sqsubseteq$ relationship, which is entirely defined by the only rules (Rfl) and (Sub):

Lemma 1 *$\sqsubseteq$ is an order relation on ramified types. Moreover, if $\tau \sqsubseteq \tau'$, then $l(\tau) \geq l(\tau')$.*

(The $\geq$ is not a typo: type levels behave contravariantly.) **Proof:** Reflexivity follows by a straightforward induction on the type derivation.

Symmetry means that if $\tau \sqsubseteq \tau'$ and $\tau' \sqsubseteq \tau$, then $\tau = \tau'$. We show this by simultaneous induction on derivations of these two inequalities. If the last rule of at least one of the derivations is (Rfl), then this is trivial. Otherwise, the last rule is (Sub) in each case, so that $\tau = \tau_1 \xrightarrow{\ell} \tau_2$, $\tau' = \tau'_1 \xrightarrow{\ell'} \tau'_2$ with $\tau'_1 \sqsubseteq \tau_1$, $\tau_2 \sqsubseteq \tau'_2$ and $\ell' \leq \ell$ on one hand, and $\tau_1 \sqsubseteq \tau'_1$, $\tau'_2 \sqsubseteq \tau_2$ and $\ell \leq \ell'$ on the other hand. By induction hypothesis we have $\tau_1 = \tau'_1$, $\tau_2 = \tau'_2$, and by symmetry of $\leq$ we have $\ell = \ell'$. So $\tau = \tau'$.

We go on to transitivity. Assume that we have derivations of $\tau \sqsubseteq \tau'$ and of $\tau' \sqsubseteq \tau''$: we show that $\tau \sqsubseteq \tau''$ by simultaneous induction on the former derivations. This is clear if one of those derivations ends in (Rfl). Otherwise we have $\tau = \tau_1 \xrightarrow{\ell} \tau_2$, $\tau' = \tau'_1 \xrightarrow{\ell'} \tau'_2$, $\tau'' = \tau''_1 \xrightarrow{\ell''} \tau''_2$, with $\tau'_1 \sqsubseteq \tau_1$, $\tau_2 \sqsubseteq \tau'_2$ and $\ell' \leq \ell$ on the one hand, and $\tau''_1 \sqsubseteq \tau'_1$, $\tau'_2 \sqsubseteq \tau''_2$ and $\ell'' \leq \ell'$ on the other hand. By induction hypothesis it follows that $\tau''_1 \sqsubseteq \tau_1$, $\tau_2 \sqsubseteq \tau''_2$, and by transitivity of $\leq$ we have $\ell'' \leq \ell$, so by one application of (Sub) we get $\tau \sqsubseteq \tau''$.

Finally, we show that $\tau \sqsubseteq \tau'$ implies $l(\tau) \geq l(\tau')$ by induction on the derivation of the former. If the last rule is (Rfl), this is clear. Otherwise we have $\tau = \tau_1 \xrightarrow{\ell} \tau_2$ with $\tau'_1 \sqsubseteq \tau_1$, $\tau_2 \sqsubseteq \tau'_2$ and $\ell' \leq \ell$. By induction hypothesis, $l(\tau_2) \geq l(\tau'_2)$, so that $l(\tau) = \max(\ell + 1, l(\tau_2)) \geq \max(\ell' + 1, l(\tau'_2)) = l(\tau')$. □

It follows from this Lemma that we can normalize derivations by merging every sequence of instances of (Cml) into just one instance. Since nothing prevents us from taking $\tau' = \tau$ and $\ell' = \ell$ in this rule, we may also consider that exactly one instance of (Cml) is used just after every application of (Var), (App) or (Abs).

Lemma 2 *If the inequality $\tau \sqsubseteq \tau'$ is derivable, then τ and τ' are ramified types.*

Proof: By structural induction on the derivation. If it ends in (Rfl), then this is clear because then $\tau = \tau' = b$ for some base type b. If it ends in (Sub), then $\tau = \tau_1 \xrightarrow{\ell} \tau_2$, $\tau' = \tau_1' \xrightarrow{\ell'} \tau_2'$. By induction hypothesis, τ_1, τ_2, τ_1', τ_2' are ramified types. Moreover, $\ell' \geq l(\tau_1')$, so τ' is a ramified type. And since $\tau_1' \sqsubseteq \tau_1$, we have $l(\tau_1') \geq l(\tau_1)$ by Lemma 1: since $\ell \geq \ell' \geq l(\tau_1')$, it follows that $\ell \geq l(\tau_1)$, so that τ is also a ramified type. $\square$

We then check that all the rules preserve the ramification constraints:

Lemma 3 *For every term u, if $u : \tau/\ell$ is derivable, then τ is a ramified type and $\ell \geq l(\tau)$.*

Proof: By induction on the derivation of $u : \tau/\ell$. If the last rule is (Var), then this follows from our assumption that variables and constants are decorated with ramified type τ and levels ℓ such that $\ell \geq l(\tau)$.

If the last rule is (Abs), then let it be as in Figure 1. Then τ, i.e. $\tau_1 \xrightarrow{\ell_1} \tau_2$, is a ramified type since τ_1 and τ_2 are (by assumption on the variable $y^{\ell_1}_{\tau_1}$ and by induction hypothesis respectively) and $\ell_1 \geq l(\tau_1)$ (by assumption on the variable x). Moreover $\ell = \max(\ell_1 + 1, \ell_2)$. By induction hypothesis $\ell_2 \geq l(\tau_2)$. Therefore $\ell \geq \max(\ell_1 + 1, l(\tau_2)) = l(\tau_1 \xrightarrow{\ell_1} \tau_2) = l(\tau)$.

If the last rule is (App), then let it be as in Figure 1. Then by induction hypothesis on the left premise, $\tau_1 \xrightarrow{\ell_1} \tau_2$ is ramified, hence τ_2 is ramified; and $\ell = \ell_2 \geq l(\tau_1 \xrightarrow{\ell_1} \tau_2) = \max(\ell_1 + 1, l(\tau_2)) \geq l(\tau_2) = l(\tau)$.

If the last rule is (Cml), then by Lemma 1 $l(\tau) \geq l(\tau')$ and by induction hypothesis $\ell \geq l(\tau)$. Since $\ell' \geq \ell$, it follows by transitivity that $\ell' \geq l(\tau')$. Moreover by Lemma 2 τ' is a ramified type, hence the claim. $\square$

5.1 Normal Derivations

This type system may be simplified as follows:

Definition 4 *Let (Var') be the following rule:*

$$\frac{\tau \sqsubseteq \tau' \quad \ell \leq \ell'}{x^{\ell}_{\tau} : \tau'/\ell'}$$

Call a normal type derivation *any derivation of a type judgement using only rules (Var'), (App), (Abs), (Rfl) and (Sub).*

Lemma 4 *Let π be a normal derivation of $u : \tau_2/\ell_2$. For any ramified type τ_1' and level ℓ_1' such that $\tau_1' \sqsubseteq \tau_1$, $\ell_1' \leq \ell_1$ and $\ell_1' \geq l(\tau_1')$, there is a normal derivation of $u[y^{\ell_1'}_{\tau_1'}/x^{\ell_1}_{\tau_1}] : \tau_2/\ell_2$, where y is not free in u.*

Proof: By structural induction on u. If u is $x^{\ell_1}_{\tau_1}$, then π is:

$$\frac{\tau_1 \sqsubseteq \tau_2 \quad \ell_1 \leq \ell_2}{x : \tau_2/\ell_2}\ (Var')$$

By the transitivity of $\sqsubseteq$ (Lemma 1) and of $\leq$ we have $\tau_1' \sqsubseteq \tau_2$ and $\ell_1' \leq \ell_2$, so that $y^{\ell_1'}_{\tau_1'} : \tau_2/\ell_2$ is derivable by (Var') as well.

If u is another variable, then $u[y/x] = u$ and the claim is trivial.

If u is an application vw, then the last rule of π was (App), and π was of the form:

$$\frac{\begin{matrix}\vdots \\ v : \tau_3 \xrightarrow{\ell_3} \tau_2/\ell_2\end{matrix} \quad \begin{matrix}\vdots \\ w : \tau_3/\ell_3\end{matrix}}{vw : \tau_2/\ell_2}\ (App)$$

By induction hypothesis $v[y^{\ell_1'}_{\tau_1'}/x^{\ell_1}_{\tau_1}] : \tau_3 \xrightarrow{\ell_3} \tau_2/\ell_2$ and $w[y^{\ell_1'}_{\tau_1'}/x^{\ell_1}_{\tau_1}] : \tau_3/\ell_3$ have normal derivations. Adding an instance of (App) to the latter two yields a normal derivation of $(vw)[y^{\ell_1'}_{\tau_1'}/x^{\ell_1}_{\tau_1}] : \tau_2/\ell_2$.

If u is an abstraction $\lambda z_\epsilon \cdot v$, then π is of the form:

$$\frac{\begin{matrix}\vdots \\ v[z'^{\ell_3}_{\tau_3}/z] : \tau_4/\ell_4\end{matrix} \quad \epsilon = E(\tau_3) \quad z' \notin \mathrm{FVC}(v)}{\lambda z_\epsilon \cdot v : \tau_3 \xrightarrow{\ell_3} \tau_4/\max(\ell_3 + 1, \ell_4)}\ (Abs)$$

where $\tau_2 = \tau_3 \xrightarrow{\ell_3} \tau_4$ and $\ell_2 = \max(\ell_3 + 1, \ell_4)$. By induction hypothesis $v[z'^{\ell_3}_{\tau_3}/z][y^{\ell_1'}_{\tau_1'}/x^{\ell_1}_{\tau_1}] : \tau_4/\ell_4$ has a normal derivation. By the variable naming convention, $x \neq z$, $y \neq z$; and we can always transform the normal derivation above so as to ensure that z' is not equal to either x or y. Then the term above is also $v[y^{\ell_1'}_{\tau_1'}/x^{\ell_1}_{\tau_1}][z'^{\ell_3}_{\tau_3}/z]$, and we can prolong the latter normal derivation by appending (Abs) to get a normal derivation of $(\lambda z_\epsilon \cdot v)[y^{\ell_1'}_{\tau_1'}/x^{\ell_1}_{\tau_1}] : \tau_3 \xrightarrow{\ell_3} \tau_4/\max(\ell_3+1, \ell_4)$. $\square$

Lemma 5 *Let π be a normal derivation of $s : \tau/\ell$. For any τ' and ℓ' with $\tau \sqsubseteq \tau'$ and $\ell \leq \ell'$, there is a normal derivation of $s : \tau'/\ell'$.*

Proof: By induction on π (or s). If π ends in (Var'), then this follows from the transitivity of $\sqsubseteq$ and $\leq$. If π ends in (App), then this follows by induction hypothesis on the left premise.

If π ends in (Abs), then let it be as in Figure 1: (1) $\tau = \tau_1 \xrightarrow{\ell_1} \tau_2$ and (2) $\ell = \max(\ell_1 + 1, \ell_2)$. Since $\tau \sqsubseteq \tau'$, it must be the case that (3) $\tau' = \tau_1' \xrightarrow{\ell_1'} \tau_2'$, with (4) $\tau_1' \sqsubseteq \tau_1$, (5) $\tau_2 \sqsubseteq \tau_2'$, (6) $\ell_1' \leq \ell_1$ and (7) $l(\tau_1') \leq \ell_1'$. Since $\ell' \geq \ell$ and by (2), we must have (8) $\ell' \geq \ell_2$. By induction hypothesis on the premise using (5) and (8), there is a normal derivation of $u[y_{\tau_1}^{\ell_1}/x] : \tau_2'/\ell'$. By Lemma 4 using the fact that τ_1' is a ramified type, and using (4), (6) and (7), there is a normal derivation of $u[y_{\tau_1}^{\ell_1}/x][y'^{\ell_1'}_{\tau_1'}/y] : \tau_2'/\ell'$ for any y' not free in $u[y_{\tau_1}^{\ell_1}/x]$. Choose y' not free in u and different from y. As y is not free in u, it follows that we have a normal derivation of $u[y'^{\ell_1'}_{\tau_1'}/x] : \tau_2'/\ell'$. Then, by (4) $E(\tau_1) = E(\tau_1') = \epsilon$, so we can apply (Abs) and get a normal derivation of $\lambda x_\epsilon \cdot u : \tau_1' \xrightarrow{\ell_1'} \tau_2' / \max(\ell_1'+1, \ell')$. Since $\ell' \geq \ell$ and by (2), we have $\ell' \geq \ell_1 + 1$, whence $\ell' \geq \ell_1' + 1$ by (6), so $\max(\ell_1'+1, \ell') = \ell'$. Moreover $\tau_1' \xrightarrow{\ell_1'} \tau_2'$ is just τ' by (3), so that we have a normal derivation of $\lambda x_\epsilon \cdot u : \tau'/\ell'$. □

Theorem 6 (Normalization) *A typing judgement $u : \tau/\ell$ has a derivation in the system of Figure 1 if and only if it has a normal derivation.*

Proof: If: any normal derivation can be trivially rewritten into a derivation in the system of Figure 1 by replacing each instance of (Var') into the appropriate instances of (Var) and (Cml).

Only if: this is proved by structural induction on a derivation π. If π ends in (Var) and concludes $x_\tau^\ell : \tau/\ell$, then we replace it by:

$$\frac{\tau \sqsubseteq \tau \quad \ell \leq \ell}{x_\tau^\ell : \tau/\ell}\,(Var')$$

Indeed, because $\Gamma, x : \tau/\ell$ is a context, τ is a ramified type, and therefore $\tau \sqsubseteq \tau$ is derivable by Lemma 1.

If π ends in (App) or in (Abs) then by induction hypothesis we can transform the subderivations whose conclusions are the premises of the rules into normal derivations. This transforms π into a normal derivation.

Finally, if π ends in (Cml), then it has the form:

$$\frac{\begin{array}{c}\vdots\,\pi' \\ u : \tau/\ell\end{array} \quad \tau \sqsubseteq \tau' \quad \ell \leq \ell'}{u : \tau'/\ell'}$$

By induction hypothesis, there is a normal proof π'' of $u : \tau/\ell$. By Lemma 2, τ' is a ramified type, so by Lemma 5, there is a normal derivation of $u : \tau'/\ell'$. □

One property that we have in the simple theory of types and which fails here is the unicity of types. Namely, every typable term has many different types, because of the subtyping rule (Sub). Because of this same rule, the set of types of a given typable term is an *upper $\sqsubseteq$-ideal.* This will complicate a bit our analysis of unification.

5.2 The Vicious Circle Principle

One of the arguments that Russell proposed in favor of ramification was that it disallowed so-called vicious circles ("whatever involves all of a collection must not be one of the collection"). Theorem 8 below gives the precise sense in which our system implements Russell's vicious circle principle.

Lemma 7 *Let x be a variable or a constant. Let s be a term where x_τ^ℓ occurs free, and assume that $s : \tau'/\ell'$ is derivable in the system of Figure 1. Then $\ell' \geq \ell$.*

Proof: By induction on the derivation. If the last rule is (Var), this is obvious. If it is (Cml), this follows directly from the induction hypothesis.

If the last rule is (App), then $s = uv$ and $\ell' = \ell_2$. If x is free in u, then by the induction hypothesis on the left premise, $\ell' \geq \ell$; otherwise, x must be free in v, so that by induction hypothesis in the right premise, $\ell_1 \geq \ell$. But by Lemma 3 on the left premise, $\ell_2 \geq l(\tau_1 \xrightarrow{\ell_1} \tau_2) = \max(\ell_1 + 1, l(\tau_2)) \geq \ell_1 + 1 > \ell_1 \geq \ell$, hence $\ell' > \ell$.

If the last rule is (Abs), then $s = \lambda x_\epsilon \cdot u$ and $\ell' = \max(\ell_1 + 1, \ell_2)$, where by induction hypothesis $\ell_2 \geq \ell$. So $\ell' \geq \ell_2$. □

Theorem 8 *Let $u_1, \ldots, u_n$ be terms, $n \geq 1$, and assume that the variable or constant x occurs free in some u_i, $1 \leq i \leq n$. Then $xu_1 \ldots u_n$ is not typable in the system of Figure 1.*

Proof: Assume that $xu_1 \ldots u_n$ is typable. By Theorem 6 it has a normal derivation. Furthermore x must be of the form $x^\ell_{\tau_1 \xrightarrow{\ell_1} \ldots \tau_n \xrightarrow{\ell_n} \tau}$, with in particular (1) $\ell \geq \max(\ell_1 + 1, \ldots, \ell_n + 1, l(\tau))$.

Without loss of generality, we may consider that x is free in u_n. Consider a normal derivation of a decorated type for $xu_1 \ldots u_n$:

$$\frac{\dfrac{\dfrac{}{x : \tau_1' \xrightarrow{\ell_1'} \ldots \tau_n' \xrightarrow{\ell_n'} \tau'/\ell'}\,(Var') \qquad u_1 : \tau_2'/\ell_2'}{xu_1 : \tau_2' \xrightarrow{\ell_2'} \ldots \tau_n' \xrightarrow{\ell_n'} \tau'/\ell'}\,(App)}{\begin{array}{c}\vdots\end{array}}$$

$$\frac{xu_1 \ldots u_{n-1} : \tau_n' \xrightarrow{\ell_n'} \tau'/\ell' \qquad u_n : \tau_n'/\ell_n'}{xu_1 \ldots u_n : \tau'/\ell'}\,(App)$$

where in particular (2) $\ell_i' \leq \ell_i$ for every $1 \leq i \leq n$. By Lemma 7 and since x is free in u_n, $\ell_n' \geq \ell$. By (2) with $i = n$, it follows that $\ell_n \geq \ell$. By (1) $\ell \geq \ell_n + 1$, hence $\ell \geq \ell + 1$, which is impossible. $\Box$

So, if x is a n-ary predicate variable, we cannot express the fact that it must hold of some n-tuple of individuals whose definitions involve x itself. This is how ramification prohibits vicious circles.

5.3 Reduction

Let's now return to more mundane, basic properties, and in particular let's examine how term typings evolve through reduction. First, ramification behaves well under substitution:

Lemma 9 *If $u : \tau'/\ell'$ and $v : \tau/\ell$ are derivable, then so is $u[v/x^\ell_\tau] : \tau'/\ell'$.*

Proof: By induction on the derivation of $u : \tau'/\ell'$. If $u = x$, then this is obvious. Otherwise, the last rule in the derivation is (App), (Abs), or (Cml), and the claim follows straightforwardly from the induction hypothesis (and the fact that, thanks to the variable naming convention in the case of (Abs), we have $(\lambda y \cdot u)[v/x] = \lambda y \cdot u[v/x]$). $\Box$

Theorem 10 (Subject Reduction) *If $s : \tau/\ell$ is derivable and $s \xrightarrow{*}_\beta t$ (resp. $s \xrightarrow{*}_{\beta\eta} t$), then $t : \tau/\ell$ is derivable.*

Proof: By induction on the length of a normal derivation. It is enough to prove the claim for the two cases $s \longrightarrow_\star t$, where $\star$ is either β or η. Take $s = \mathcal{C}[s']$, $t = \mathcal{C}[t']$, where s' is the redex and t' is its contractum. We prove the claim by induction on $\mathcal{C}$. The induction cases ($\mathcal{C}$ of the form $\mathcal{C}'w$, $w\mathcal{C}'$ or $\lambda x \cdot \mathcal{C}'$) are trivial, and it remains to prove the base case, namely when s is itself the redex.

If s is a β-redex $(\lambda x_\epsilon \cdot u)v$, then $t = u[v/x]$. Look at the normal derivation of $s : \tau/\ell$:

$$\dfrac{\dfrac{\begin{matrix}\vdots \\ u[y^{\ell_1}_{\tau_1}/x] : \tau/\ell\end{matrix}}{\lambda x_\epsilon \cdot u : \tau_1 \xrightarrow{\ell_1} \tau/\ell}\,(Abs) \qquad \begin{matrix}\vdots \\ v : \tau_1/\ell_1\end{matrix}}{(\lambda x_\epsilon \cdot u)v : \tau/\ell}$$

where $\epsilon = E(\tau_1)$. By Lemma 9, $u[y^{\ell_1}_{\tau_1}/x][v/y^{\ell_1}_{\tau_1}] : \tau/\ell$ is derivable. Since y is not free in u, the latter term is $u[v/x]$, whence the claim.

If s is an η-redex $\lambda x_\epsilon \cdot ux$, with x not free in u, then necessarily the derivation of $s : \tau/\ell$ ends in:

$$\dfrac{\dfrac{\begin{matrix}\vdots\,\pi \\ u : \tau'_1 \xrightarrow{\ell'_1} \tau_2/\ell_2\end{matrix} \qquad \dfrac{}{y^{\ell_1}_{\tau_1} : \tau'_1/\ell'_1}\,(Var')}{uy : \tau_2/\ell_2}\,(App)}{\lambda x_\epsilon \cdot ux : \tau_1 \xrightarrow{\ell_1} \tau_2/\max(\ell_1 + 1, \ell_2)}\,(Abs)$$

where (1) $\tau_1 \sqsubseteq \tau'_1$, (2) $\ell_1 \leq \ell'_1$ (rule (Var')), (3) $\epsilon = E(\tau_1)$ (rule (Abs)) and (4) $\tau = \tau_1 \xrightarrow{\ell_1} \tau_2$, (5) $\ell = \max(\ell_1 + 1, \ell_2)$. Moreover (6) $\ell_1 \geq l(\tau_1)$ because of the assumption on the decoration of the variable y. By (1), (2) and (6) it follows that $\tau'_1 \xrightarrow{\ell'_1} \tau_2 \sqsubseteq \tau_1 \xrightarrow{\ell_1} \tau_2$; since $\ell_2 \leq \max(\ell_1 + 1, \ell_2)$ as well, we can build the derivation:

$$\dfrac{\begin{matrix}\vdots\,\pi \\ u : \tau'_1 \xrightarrow{\ell'_1} \tau_2/\ell_2\end{matrix}}{u : \tau_1 \xrightarrow{\ell_1} \tau_2/\max(\ell_1 + 1, \ell_2)}\,(Cml)$$

which proves the claim. $\Box$

Subject reduction is important, for without it β-reduction or $\beta\eta$-reduction would be a meaningless concept within the ramified typed universe. In particular, the Church-Rosser property of the untyped λ-calculus transfers to the ramified typed λ-calculus just because subject reduction works (and because of α-conversion). Also, it will be important in unification, because applying a substitution to a term is then followed by a normalization phase, which by Theorem 10 will preserve well-typedness.

Equally important is the fact that, in some sense, the ramified type theory is a subtheory of the simple type theory, a fact that we have used in the informal presentation of Sections 4.2 and 4.3:

Lemma 11 *If $u : \tau/\ell$ is derivable in the system of Figure 1, then $u : E(\tau)$ is derivable in the simple theory of types.*

Proof: To be precise, by the simple theory of types we mean the set of rules:

$$\dfrac{}{x^\ell_\tau : E(\tau)}\,(Var^*)$$

$$\dfrac{u : \epsilon_1 \to \epsilon_2 \quad v : \epsilon_1}{uv : \epsilon_2}\,(App^*)$$

$$\dfrac{u : \epsilon_2}{\lambda x_{\epsilon_1} \cdot u : \epsilon_1 \to \epsilon_2}\,(Abs^*)$$

i.e., we keep levels on variables, although they are useless.

The claim is then a trivial induction on the derivation: (Var), (App) and (Abs) rules translate to (Var^*), (App^*) and (Abs^*) respectively, while (Cml) steps are erased. Indeed, whenever $\tau \sqsubseteq \tau'$, we have $E(\tau) = E(\tau')$: this is an easy induction on the derivation of the inequality. □

Corollary 12 *The ramified theory of types is convergent: every β (resp. $\beta\eta$) reduction terminates to a unique normal form.*

Proof: Every derivation in the ramified calculus is also a derivation in the simply-typed calculus. (In particular, any α-conversion step in the former is an α-conversion step in the latter.) So the ramified calculus is terminating.

Conversely, let u be a term of decorated type τ/ℓ that reduces to v and w along different reduction paths. Because the pure λ-calculus is confluent, the simply-typed calculus is [Bar84]; now v and w have types $E(\tau)$ in the simple theory of types by Lemma 11 applied to u and Theorem 10. So v and w have a common reduct s in the simple theory of types. But the reductions from v to s and from w to s are also reductions in the ramified calculus: this establishes the fact that the ramified calculus is confluent, hence Church-Rosser.

Finally, any rewrite system that is both terminating and Church-Rosser is convergent. □

Recall that we write $u\downarrow_\beta$, or simply $u\downarrow$, the unique β-normal form of u.

5.4 Can a Term be Instantiated to a Well-Typed One?

There is a kind of converse of Theorem 10, or a kind of analogue of remark (R4) at the beginning of Section 4.1. The goal is to establish Theorem 15, which states a necessary and sufficient condition for a term to have well-typed ramified instances.

Lemma 13 *For every simple type ϵ, writen uniquely as $\epsilon_1 \to \dots \epsilon_n \to b$ with b a base type, let $\hat{e}_\epsilon$ be the term $\lambda x_{1\epsilon_1} \cdot \dots \cdot \lambda x_{n\epsilon_n} \cdot v_b^0$, where v_b^0 is a distinguished variable of type b and level 0 that will never be used in any other term.*

We let ζ be the substitution mapping each variable of type τ (and arbitrary level) to $\hat{e}_{E(\tau)}$. Then:

1. *ζ is a well-typed substitution;*

2. *for every β-normal term t, the β-normal form $(t\zeta)\downarrow$ exists;*

3. *for every β-normal simply-typed term t, for every well-simply-typed substitution σ, if $t\sigma$ β-reduces to a ramified well-typed term of some decorated type τ/ℓ, then $(t\zeta)\downarrow : \tau/\ell$ is derivable.*

Proof: 1. Let x_τ^ℓ be any variable, with (1) $\ell \geq l(\tau)$, (2) $\tau = \tau_1 \xrightarrow{\ell_1} \dots \tau_n \xrightarrow{\ell_n} b$, with b a base type, and $\epsilon = E(\tau)$, so that $x\zeta = \hat{e}_\epsilon$. By n-fold application of rule (Abs), $\hat{e}_\epsilon : \tau_1 \xrightarrow{\ell_1} \dots \tau_n \xrightarrow{\ell_n} b/\max(\ell_1+1, \dots, \ell_n+1, 0)$ is derivable. By (2), $\max(\ell_1+1, \dots, \ell_n+1, 0) = l(\tau)$, so by (1) it is at most ℓ: apply (Cml) to get a derivation of $\hat{e}_\epsilon : \tau/\ell$.

2. Let W be FVC(t), and let σ_0 be $\zeta|_W$, so that σ_0 is a regular substitution with $u\sigma_0 = u\zeta$. Recall that β-normal forms may be characterized as those terms defined inductively as $\lambda\overline{y_k} \cdot a\overline{u_m}$, where $k, m \geq 0$, a is a variable or a constant, and $u_1, \dots, u_m$ are normal forms. (We call this the *hnf-decomposition* of the term.) The (unique) normal form $(t\zeta)\downarrow$ (i.e., $(t\sigma_0)\downarrow$) may then be defined by recursion on its hnf-decomposition by:

- if $a \in W$, then $a\sigma_0$ is some $\hat{e}_\epsilon$, with $\epsilon = \epsilon_1 \to \dots \epsilon_n \to b$, $n \geq m$, and $(t\sigma_0)\downarrow = \lambda y_1 \cdot \dots \cdot \lambda y_k \cdot \lambda x_{m+1\,\epsilon_{m+1}} \cdot \dots \cdot \lambda x_{n\,\epsilon_n} \cdot v_b^0$;
- otherwise, $(t\sigma_0)\downarrow = \lambda\overline{y_k} \cdot a\overline{(u_m\sigma_0)\downarrow}$.

3. Let W, σ_0 be as above. We prove the claim by induction on the hnf-decomposition of t, with t as in 2. Let y_i be $y_{i\,\epsilon'_i}$, for every $1 \leq i \leq k$, and assume that $t\sigma \xrightarrow{*}_\beta s$ for some typable s. Then:

- if $a \in W$, then look at s. Whenever $s : \tau/\ell$ is derivable, s has type $E(\tau)$ in the simple theory of types by Lemma 11. So $t\sigma$ has type $E(\tau)$ in the simple theory of types (see Remark (R4) at the beginning of Section 4.1). But the simple type of a is some ϵ of the form $\epsilon_1 \to \dots \epsilon_n \to b$, $n \geq m$, so the simple type of $t\sigma$ must be $E(\tau) = \epsilon'_1 \to \dots \to \epsilon'_k \to \epsilon_{m+1} \to \dots \to \epsilon_n \to b$. For this to be the case, τ must be of the form (1) $\tau'_1 \xrightarrow{\ell'_1} \dots \xrightarrow{\ell'_{k-1}} \tau'_k \xrightarrow{\ell'_k} \tau_{m+1} \xrightarrow{\ell_{m+1}} \dots \xrightarrow{\ell_{n-1}} \tau_n \xrightarrow{\ell_n} b$, with (2) $E(\tau'_i) = \epsilon'_i$, $1 \leq i \leq k$ and (3) $E(\tau_j) = \epsilon_j$, $1 \leq j \leq n$. Moreover by Lemma 3 we have (4) $\ell \geq l(\tau)$.

  On the other hand, $(t\sigma_0)\downarrow = \lambda y_{1\,\epsilon'_1} \cdot \dots \cdot \lambda y_{k\,\epsilon'_k} \cdot \lambda x_{m+1\,\epsilon_{m+1}} \cdot \dots \cdot \lambda x_{n\,\epsilon_n} \cdot v_b^0$. Applying rule (Abs) k times using (2) and $n-m$ times using (3), it follows that $(t\sigma_0)\downarrow : \tau'_1 \xrightarrow{\ell'_1} \dots \xrightarrow{\ell'_{k-1}} \tau'_k \xrightarrow{\ell'_k} \tau_{m+1} \xrightarrow{\ell_{m+1}} \dots \xrightarrow{\ell_{n-1}} \tau_n \xrightarrow{\ell_n} b/\max(\ell'_1+1, \dots, \ell'_k+1, \ell_{m+1}+1, \dots, \ell_n+1)$ is derivable, i.e. $(t\sigma_0)\downarrow : \tau/l(\tau)$ using (1). By (Cml) and (4), it follows that $(t\sigma_0)\downarrow : \tau/\ell$ is derivable.

- If $a \notin W$, then $t\sigma = \lambda\overline{y_k} \cdot a\overline{u_m\sigma}$, and a is a constant or a variable, so s must be of the form $\lambda\overline{y_k} \cdot a\overline{v_m}$ with $u_i\sigma \xrightarrow{*}_\beta v_i$ for every $1 \leq i \leq m$. Moreover, since s is (ramified) typable, so is each v_i (with $y_1, \ldots, y_k$ replaced by suitable fresh variables of the right types and levels). By induction hypothesis, $(u_i\sigma_0)\downarrow$ can be assigned the same decorated type as v_i for each $1 \leq i \leq m$. It follows that $\lambda\overline{y_k} \cdot a\overline{(u_m\sigma_0)\downarrow}$ can be given the same decorated type as s; but this is just $(t\sigma_0)\downarrow$ by construction.

□

The case of the η-rule is more trivial:

Lemma 14 *If $u \xrightarrow{*}_\eta v$, u is simply-typed and $v : \tau/\ell$ is derivable, then $u : \tau/\ell$.*

Proof: By induction on the length of the reduction from u to v, it is enough to prove the claim when $u \longrightarrow_\eta v$. In turn, this is proved by structural induction on $\mathcal{C}$, where $u = \mathcal{C}[\lambda x_{\epsilon_1} \cdot tx]$ and $v = \mathcal{C}[t]$. The only non trivial case is the base case, where $\mathcal{C}$ is the empty context. Then, assume $t : \tau/\ell$ derivable. Because u is simply-typed and by Lemma 11, we must have $E(\tau) = \epsilon_1 \to \epsilon_2$ for some simple type ϵ_2. So τ must be of the form $\tau_1 \xrightarrow{\ell_1} \tau_2$, with $\epsilon_1 = E(\tau_1)$, $\epsilon_2 = E(\tau_2)$, $\ell_1 \geq l(\tau_1)$ and: (1) $\ell \geq \max(\ell_1 + 1, l(\tau_2))$. Since x is not free in t by assumption, we can produce the following derivation:

$$\dfrac{\dfrac{\vdots}{t y^{\ell_1}_{\tau_1} : \tau_2/\ell}}{\lambda x_{\epsilon_1} \cdot tx : \tau_1 \xrightarrow{\ell_1} \tau_2 / \max(\ell_1 + 1, \ell)} \ (Abs)$$

where $\tau_1 \xrightarrow{\ell_1} \tau_2$ is just τ, and $\max(\ell_1 + 1, \ell) = \ell$ by (1). □

Theorem 15 *Let t be a β-normal simply-typed term. Call a* well-simply-typed instance *of t any term $t\sigma$, where σ is a well-simply-typed substitution.*

Then t has a well-simply-typed instance that is $\beta\eta$-equivalent to a ramified well-typed term if and only if $(t\zeta)\downarrow$ has a ramified type.

Proof: If the term t has a well-simply-typed instance $t\sigma$ that is $\beta\eta$-equivalent to a well-typed ramified term u, then by confluence $t\sigma \xrightarrow{*}_{\beta\eta} v$ and $u \xrightarrow{*}_{\beta\eta}$ for some term v. By Theorem 10 applied to u, it follows that t has a well-simply-typed instance $t\sigma$ that *$\beta\eta$-reduces* to some ramified well-typed term. Conversely, if $t\sigma$ $\beta\eta$-reduces to some ramified well-typed term, it is $\beta\eta$-equivalent to it. To prove the theorem, it is therefore enough to prove the following claim: t has a well-simply-typed instance that $\beta\eta$-reduces to a ramified well-typed term if and only if $(t\zeta)\downarrow$ has a ramified type.

By Lemma 13 1. and 2., $(t\zeta)\downarrow$ is always well-defined and that $t\zeta$ is a well-typed instance of t. Recall also that, in any $\beta\eta$-reduction to normal form, we may first take the β-normal form, then the η-normal form of the latter (this is postponement of the η-rule [Bar84]).

If: if $(t\zeta)\downarrow$ has a ramified type, then let $t\zeta$ be the desired instance: it β-reduces to a term with a ramified type, namely $(t\zeta)\downarrow$, which then η-normalizes to another term with the same ramified type, by Theorem 10.

Only if: assume that there is a substitution σ such that $t\sigma \xrightarrow{*}_{\beta\eta} v$, with v having the ramified decorated type τ/ℓ. By postponement of η, there is a term u such that $t\sigma \xrightarrow{*}_\beta u \xrightarrow{*}_\eta v$. By subject reduction in the simply-typed case, u is simply typed. By Lemma 14, $u : \tau/\ell$ is therefore derivable. By Lemma 13, $(t\zeta)\downarrow : \tau/\ell$ is then derivable. □

Therefore, the problem of finding instances of a simply-typed term modulo $\beta\eta$ that is well-typed in the ramified sense reduces to the type-checking problem in the ramified system of Figure 1.

6 Retrieving Levels

We wish to show that type-checking ramified terms is decidable, and in fact computationally easy. To type-check a term, we wish to find a normal derivation of a decorated type for it, or to prove that none exists. We do this classically by constructing such a derivation bottom-up, being guided by the structure of the term. The only problem lies in rule (Abs), because there we need to guess a type τ_1 and a level ℓ_1 for the fresh variable y. Since we already know the shape of τ_1 ($E(\tau_1)$ must equal ϵ), this is just a question of guessing the right levels to annotate the function arrows in τ_1.

So first, we replace every level annotating variables or type arrows by fresh *level variables*, and express the typing constraints as *systems of level constraints* in Section 6.1. We shall then show that we can infer the general form of types and levels of terms in polynomial time in Section 6.2. This is the analogue of [Hin69] or of the ML type system [Mil78] without lets. We give a few examples in Section 6.3, and propose a few improvements in Section 6.4.

6.1 Level Variables and Constraints

Introducing level variables demands that we change our representation of types and terms to accomodate variables:

Definition 5 *Let V be a coutably infinite set of so-called* level variables *μ, ν, etc.*

The set $RT(V)$ of ramified pre-types *with variables as levels is the smallest containing all base types b, and all expressions $\tau_1 \xrightarrow{\mu} \tau_2$, where τ_1 and τ_2 are ramified pre-types and μ is in $V \cup L$.*

The set $\lambda(V)$ of ramified pre-terms *is the smallest containing variables x_τ^μ where $\tau \in RT(V)$ and $\mu \in V \cup L$, applications uv with $u \in \lambda(V)$ and $v \in \lambda(V)$, and abstractions $\lambda x_\epsilon \cdot u$ with $u \in \lambda(V)$ and $\epsilon \in ST$.*

Notice that a pre-term with no level variables is just an ordinary term. These new objects are to be understood under *assignments* ρ mapping level variables to actual levels in L. To represent sets of assignments, we use the following notion of constraints.

Definition 6 *A* system of level constraints *K is a finite set of* constraints *of the form $\mu \geq \nu + n$ where μ and ν are level variables and $n \in \mathbb{N}$, or $\mu \geq \ell$ or $\ell \geq \mu$ where μ is a level variable and ℓ is a constant level in L.*

The domain dom K *of a system K is the set of level variables appearing in it. A level assignment ρ is said to* satisfy *K, and we write $\rho \models K$, if and only if all the inequalities gotten from constraints in K by replacing variables μ by $\rho(\mu)$ are valid.*

For any sentence about $RT(V)$ or $\lambda(V)$, we say that it holds under *K provided that for any ρ satisfying K, the same sentence holds with all level variables replaced by their values under ρ.*

Systems of level constraints are interesting because they express all the constraints that we shall need, and most problems on them are solvable in polynomial time. But before we can speak of polynomial time, we must define our data representations, and define measures of size. We define the *size* $|o|$ of an object o as usual, by $|uv| = |u|+|v|+1$, $|\lambda x_\epsilon \cdot u| = |\epsilon|+|u|+1$, $|x_\epsilon| = |\epsilon| + 1$, $|x_\tau^\mu| = |\tau| + 1$, $|b| = 1$, $|\epsilon_1 \to \epsilon_2| = |\epsilon_1| + |\epsilon_2| + 1$, $|\tau_1 \xrightarrow{\mu_1} \tau_2| = |\tau_1| + |\tau_2| + 1$.

Then, we represent K as the following graph $G(K)$: the vertices are all variables in dom K and all level constants appearing in K, and the edges are $\mu \xrightarrow{n} \nu$ for all constraints $\mu \geq \nu + n$, $\mu \xrightarrow{0} \ell$ for all constraints $\mu \geq \ell$ and $\ell \xrightarrow{0} \mu$ for all constraints $\ell \geq \mu$ in K. The labels on arrows are called *weights*. It will always be assumed that $G(K)$ is in fact the way that K is really represented in memory, so that we don't have to translate back and forth between K and $G(K)$ in practice. Furthermore, we assume that $G(K)$ is represented in memory as an adjacency list [McH90], i.e. a list of vertices, as pointers to records containing the description of the vertex (an ordinal, or a special tag denoting a variable vertex), and a list of successor/weight pairs. (If there are two edges from v_1 to v_2, with respective weights n_1 and n_2, we only represent one with weight $\max(n_1, n_2)$; this does not change the semantics of K.) The size $|K|$ of K is then the sum of sizes of edges (equated to the size $|n| = \max(1, \lceil \log_2(n + 1) \rceil)$ of the weights labelling them, represented in binary), plus the sum of sizes of vertices in dom K, where a variable vertex has size 1.

To define the size of constant vertices, we need to make precise our system of ordinal notations. It must allow us to compute the sum of an ordinal and an integer, to compare by $=$ or $\leq$ any two ordinals in polynomial time. If we use only integers as levels, say in binary, this is trivial (the size of ℓ is then $|\ell| = \max(1, \lceil \log_2 \ell \rceil)$). This is also certainly possible up to Γ_0 [Gal91], by using Schütte's ψ function and natural ordinal sums: this yields ordinal notations where $=$ is just structural comparison and $<$ is the lexicographic path ordering. The size of an ordinal notation, there, is the number of signs needed to write it on paper.

The essence of our algorithms will be the computation of the set of *strongly connected components* of $G(K)$ [McH90]. Recall that a subgraph is strongly connected if and only if every vertex in the subgraph is reachable from any other vertex in the subgraph, and that the strongly connected components of a graph of the maximal strongly connected subgraphs. Leaving labels aside, the *condensation graph* $\overline{G}$ of a graph G is the graph whose vertices are the strongly connected components of G, and where there is an edge from C_1 to C_2 if and only if there is an edge $v_1 \longrightarrow v_2$ in G, with v_1 a vertex of C_1 and v_2 a vertex of C_2. We let $\overline{v}$ be the strongly connected component of v. Observe that $\overline{G}$ is then a directed *acyclic* graph, or intuitively a "tree with shared subtrees".

Say that the *value* of a vertex under ρ is ℓ if the vertex is a level constant ℓ, and $\rho(\mu)$ if it is a level variable μ. We write (abusively) $\rho(v)$ the value of v under ρ.

Lemma 16 *If ρ satisfies K, then for every strongly connected component C of $G(K)$, there is a unique level ℓ such that for every vertex v of C, $\rho(v) = \ell$. We write this level $\rho(C)$.*

Proof: Notice that for any edge $v_1 \xrightarrow{n} v_2$, we must have $\rho(v_1) \geq \rho(v_2) + n$, with $n \geq 0$, and therefore $\rho(v_1) \geq \rho(v_2)$. Now if there is a path from v_1

to v_2, then by induction on the length of the path, $\rho(v_1) \geq \rho(v_2)$. In a strongly connected component C, for any two distinct vertices v_1 and v_2, by definition v_2 is reachable from v_1, so $\rho(v_1) \geq \rho(v_2)$, and v_1 is reachable from v_2, so $\rho(v_2) \geq \rho(v_1)$. It follows that $\rho(v_1) = \rho(v_2)$. □

Lemma 17 *If ρ satisfies K, then in every strongly connected component C of $G(K)$, all the weights labeling edges inside C are* 0.

Proof: Let $v_1 \xrightarrow{n} v_2$ be an edge in C. By Lemma 16, $\rho(v_1) = \rho(v_2)$. But since ρ satisfies K, $\rho(v_1) \geq \rho(v_2) + n$, hence $n = 0$. □

We say that a strongly connected component is *consistent* if and only if all its edges have weights equal to 0, and it contains at most one constant level vertex. If it contains one such constant vertex ℓ, then we say that it is *fixed*, otherwise it is *variable*.

Lemma 18 *There is a polynomial-time algorithm which, given any system K of constraints, decides whether it is satisfiable, and if so, returns the least assignment ρ on* dom K *satisfying K with respect to the pointwise ordering on ordinals.*

Proof: Build the condensation graph $\overline{G}$ of $G(K)$, and label each edge $C_1 \longrightarrow C_2$ in $\overline{G}$ by the least upper bound n of all weights m of edges $v_1 \xrightarrow{m} v_2$, with v_1 a vertex of C_1 and v_2 a vertex of C_2. This can be done in polynomial time by a slight variant of Tarjan's algorithm for finding strongly connected components [McH90].

Now decorate each vertex of $\overline{G}$ in reverse topological order (i.e., bottom-up, since this graph is acyclic) as follows. At vertex C, with immediate successors $C_1, \ldots, C_k$ reached respectively through edges of weights $n_1, \ldots, n_k$ (at this step $C_1, \ldots, C_k$ have already been decorated, say with respective levels $\ell_1, \ldots, \ell_k$):

1. Check whether C is consistent; if not, then fail: K is unsatisfiable. Otherwise, do:

2. If C is fixed, then let ℓ be the unique constant vertex in C; if $\ell < \ell_i + n_i$ for some i, $1 \leq i \leq k$, then fail: K is unsatisfiable. Otherwise, decorate C with ℓ.

3. If C is variable, then decorate C with $\max(\ell_1 + n_1, \ldots, \ell_k + n_k)$.

This algorithm clearly runs in polynomial time in the size of the input K. If it succeeds, let ρ_0 be the assignment that maps each level variable μ to the decoration of $\overline{\mu}$. We now show that the algorithm answers the question.

($\Longrightarrow$) Assume that K is satisfiable. We claim that the algorithm does not fail, and that every assignment ρ satisfying K is (pointwise) greater than or equal to ρ_0. We prove the claim by well-founded induction on the directed acyclic graph $\overline{G}$, showing at each step (where C is the currently examined vertex of $\overline{G}$) that ρ is pointwise at least ρ_0 on all vertices of $G(K)$ reachable from vertices in C.

So consider the current vertex C in $\overline{G}$, $C \xrightarrow{n_i} C_i$, $1 \leq i \leq k$, be the outgoing edges. By Lemma 16, C contains at most one level constant, and by Lemma 17 all the edges in G between elements of C have weight 0, so C is consistent: the algorithm therefore does not fail at step 1.

If C is fixed, then necessarily $\rho(C) = \ell$ where ℓ is the unique constant in C. Consider a fixed arbitrary C_i, $1 \leq i \leq k$. For every edge $v_1 \xrightarrow{n} v_2$ in G going from a vertex v_1 in C to a vertex v_2 in C_i, we have $\rho(v_1) \geq \rho(v_2) + n$. Because n_i is the least upper bound of all such n, $\rho(C) \geq \rho(C_i) + n_i$; this is in turn at least $\rho_0(C_i) + n_i$ by induction hypothesis, that is, we have $\ell \geq \ell_i + n_i$ for every i, $1 \leq i \leq k$, where $\ell_i = \rho_0(C_i)$. Therefore, the algorithm does not fail at step 2 either. Moreover for every vertex v in C, $\rho(v) = \ell = \rho_0(v)$, and for every vertex v reachable from some C_i, $1 \leq i \leq k$, $\rho(v) \geq \rho_0(v)$ by induction hypothesis. So ρ is pointwise at least ρ_0 on vertices reachable from C.

If C is variable, then the algorithm cannot fail. Moreover, by a similar argument $\rho(C) \geq \rho(C_i) + n_i$ for every i, $1 \leq i \leq k$. By induction hypothesis $\rho(C_i) \geq \rho_0(C_i) = \ell_i$, so $\rho(C) \geq \max(\ell_1 + n_1, \ldots, \ell_k + n_k) = \rho_0(C)$. The claim is then proved.

($\Longleftarrow$) Assume that the algorithm does not fail. We claim that ρ_0 satisfies K. Again we prove it by well-founded induction on $\overline{G}$, showing at each step (where the current vertex in $\overline{G}$ is C) that ρ_0 satisfies all the constraints described by edges of $G(K)$ that are reachable from vertices in C.

So let C be the current vertex in $\overline{G}$, $C \xrightarrow{n_i} C_i$, $1 \leq i \leq k$, be the outgoing edges. By induction hypothesis, ρ_0 satisfies all the constraints in K represented by edges in $C_1, \ldots, C_k$ or below. We show that it also satisfies all constraints represented by edges of $G(K)$ inside C, and by edges of $G(K)$ going from vertices inside C to vertices inside C_i.

Since the algorithm does not fail at step 1., C is consistent. Let then be $v_1 \xrightarrow{0} v_2$ any edge from vertex v_1 in C to vertex v_2 in C. This corresponds to the constraint $\rho_0(v_1) \geq \rho_0(v_2)$, which is trivially satisfied since $\rho_0(v_1) = \rho_0(v_2) = \rho_0(C)$.

Moreover, ρ_0 is such that $\rho_0(C) \geq \max(\rho_0(C_1) + n_1, \ldots, \rho_0(C_k) + n_k)$, whether C is fixed or variable. So let $v_1 \xrightarrow{n} v_2$ be any edge in $G(K)$ from a vertex v_1 in C to a vertex v_2 in some C_i, $1 \leq i \leq k$. Then $\rho_0(v_1) = \rho_0(C)$ (by definition) $\geq \max(\rho_0(C_1) + n_1, \ldots, \rho_0(C_k) + n_k)$ (by the remark above) $\geq \rho_0(C_i) + n_i = \rho_0(v_2) + n_i$ (by definition of ρ_0) $\geq \rho_0(v_2) + n$ (by definition of n_i). So the constraint represented by this edge is again satisfied. $\Box$

It would be interesting to transform this algorithm into an incremental algorithm, i.e. an algorithm where new constraints are progressively added and satisfiability is checked at each step. We leave this as an improvement to be done.

6.2 Level Inference

We can then translate conditions for being well-typed, for two types to be less or equal, and so on, as systems of constraints. We let $\mu = \nu$ denote the set of two constraints $\mu \geq \nu$, $\nu \geq \mu$.

Lemma 19 *Given any ramified pre-types τ, τ', and level variable μ, we can build in polynomial time systems of constraints $K(l(\tau) \leq \mu)$, $K(\tau$ ramified$)$, $K(\tau \sqsubseteq \tau')$ and $K(\tau = \tau')$ such that the assignments that satisfy them are precisely those under which $l(\tau) \leq \mu$, τ is a ramified type, $\tau \sqsubseteq \tau'$ and $\tau = \tau'$ respectively.*

Proof: In the following, let b stand for base types, τ for any ramified type (possibly primed or indexed). Let $\top$ (true) denote the empty set $\emptyset$ of constraints, and $\perp$ (false) denote any unsatisfiable set of constraints, like $\{0 \geq \mu, \mu \geq 1\}$ for some variable μ.

$$
\begin{aligned}
K(l(b) \leq \mu) &= \top \\
K(l(\tau_1 \xrightarrow{\nu} \tau_2) \leq \mu) &= \{\mu \geq \nu + 1\} \cup K(l(\tau_2) \leq \mu)
\end{aligned}
$$

$$
\begin{aligned}
K(b \text{ ramified}) &= \top \\
K(\tau_1 \xrightarrow{\nu} \tau_2 \text{ ramified}) &= K(\tau_1 \text{ ramified}) \\
&\quad \cup K(\tau_2 \text{ ramified}) \\
&\quad \cup K(l(\tau_1) \leq \nu)
\end{aligned}
$$

$$
\begin{aligned}
K(b \sqsubseteq b) &= \top \\
K(\tau_1 \xrightarrow{\mu} \tau_2 \sqsubseteq \tau_1' \xrightarrow{\mu'} \tau_2') &= K(\tau_1' \sqsubseteq \tau_1) \cup K(\tau_2 \sqsubseteq \tau_2') \\
&\quad \cup \{\mu \geq \mu'\} \cup K(l(\tau_1') \leq \mu') \\
K(\tau \sqsubseteq \tau') &= \perp \text{ otherwise}
\end{aligned}
$$

In the latter clause, it will be interesting in practice to just fail, aborting the whole computation, and returning $\perp$.

$$
\begin{aligned}
K(b = b) &= \top \\
K(\tau_1 \xrightarrow{\mu} \tau_2 = \tau_1' \xrightarrow{\mu'} \tau_2') &= K(\tau_1 = \tau_1') \cup K(\tau_2 = \tau_2') \\
&\quad \cup \{\mu = \mu'\} \\
K(\tau \sqsubseteq \tau') &= \perp \text{ otherwise}
\end{aligned}
$$

Similarly, in the latter clause, it will be interesting in practice to just fail. $\Box$

Let $\mathrm{FLV}(s)$ be the set of *free level variables* in the pre-term s, defined as $\mathrm{FLV}(\lambda x_\epsilon \cdot u) = \mathrm{FLV}(u)$, $\mathrm{FLV}(uv) = \mathrm{FLV}(u) \cup \mathrm{FLV}(v)$, $\mathrm{FLV}(x_\tau^\mu) = \{\mu\} \cup \mathrm{FLV}(\tau)$, and where $\mathrm{FLV}(\tau)$ is defined as $\mathrm{FLV}(b) = \emptyset$ if b is a base type, and $\mathrm{FLV}(\tau_1 \xrightarrow{\mu} \tau_2) = \mathrm{FLV}(\tau_1) \cup \mathrm{FLV}(\tau_2) \cup \{\mu\}$.

We wish to find a system of level constraints that would express exactly when a given pre-term s has a ramified type. This system of level constraints should therefore have $\mathrm{FLV}(s)$ as domain. But a predicate denoting typability of s will be of the form $\exists \mu_1, \ldots, \mu_k \cdot K$, where K is a system of constraints expressing all the ordering constraints between level variables that occur in a derivation, and $\mu_1, \ldots, \mu_k$ are unknowns that must be introduced to represent unknown levels in (Var') and (Abs). We might show that we can represent such predicates as systems of constraints (to represent $\exists \mu \cdot K$, the rough idea is to add edges $v \xrightarrow{m+n} v'$ to $G(K)$ whenever we have edges $v \xrightarrow{m} \mu$ and $\mu \xrightarrow{n} v'$, then to eliminate all edges incident on μ; the real procedure is a bit more complicated).

But it will be easier to leave the existentially quantified variables explicit, and instead to consider a refined notion of satisfaction. For any set S of level variables, we say that a level assignment ρ satisfies a system of constraints K *up to* S if and only if there is a level assignment ρ' satisfying K such that $\rho(\mu) = \rho'(\mu)$ for every $\mu \in S$. (That is, we quantify on all variables in $\mathrm{dom}\, K \setminus S$.)

Theorem 20 (Level Reconstruction) *Typability of ramified pre-terms is decidable in polynomial time.*

More precisely, there is a polynomial-time algorithm which, given a ramified pre-term s, either fails if s is not typable, or returns a ramified pre-type τ_s, a level variable μ_s and a system of constraints $K(s : \tau_s/\mu_s)$ such that s is well-typed and has decorated type τ_s/μ_s under assignment ρ if and only if ρ satisfies $K(s : \tau_s/\mu_s)$ up to $\mathrm{FLV}(s)$.

Proof: This is a more or less direct translation of the rules used in normal derivations (but we might as well do this directly on the rules of Figure 1, although this would be less efficient). We first define a suitable notion of *occurrence* p in a pre-term or pre-type. Occurrences will be words on the alphabet $\{0, 1\}$. We define the set of occurrences of u and the subterm $u|_p$ of u at occurrence p inductively as follows. The empty word ε is an occurrence in every term u, and $u|_\varepsilon = u$; if p is an occurrence in u, and $u|_p$ is an abstraction $\lambda x_\epsilon \cdot v$, then $p0$ is an occurrence in u and

$u|_{p0} = v$; if $u|_p$ is an application vw, then $p0$ and $p1$ are occurrences in u and $u|_{p0} = v$, $u|_{p1} = w$. Similarly, in simple types, if p is an occurrence in ϵ and $\epsilon|_p = \epsilon_1 \to \epsilon_2$, then $p0$ and $p1$ are occurrences in ϵ and $\epsilon|_{p0} = \epsilon_1$, $\epsilon|_{p1} = \epsilon_2$.

First, for every subterm occurrence p in s, let μ_p be a level variable (this will denote the level of $s|_p$). For every occurrence of an abstraction $\lambda x_\epsilon \cdot t$ in s, let ν_x and ν_x^p be level variables, for every occurrence p of a functional subtype $\epsilon_1 \to \epsilon_2$ in ϵ; we let θ_x be the ramified pre-type obtained by recursively decorating each functional subtype at occurrence p in ϵ as above by ν_x^p (intuitively, θ_x/ν_x denotes the decorated type of the variable y to guess in rule (Abs)). Finally, for every occurrence p of a constant or a variable (free or bound) x_ϵ (resp. x_τ^ℓ), for every occurrence q of a functional subtype $\epsilon_1 \to \epsilon_2$ in ϵ (resp. $E(\tau)$), let ξ_p^q be a level variable; similarly, we define θ_p as the ramified pre-type obtained by recursively decorating each functional subtype at occurrence q in ϵ as above by ξ_p^q (intuitively, θ_p/μ_p will be the decorated type that we assign to x by rule (Var')). We assume that all these level variables are pairwise distinct and distinct from the level variables in $\mathrm{FLV}(s)$.

We arrange the set of occurrences in s as a finite tree by sharing occurrence prefixes (this tree is the *skeleton* of s). The algorithm is then defined by structural recursion on this tree, returning a pre-type $\tau_p(\sigma)$ and a set of constraints $K_p(\sigma)$ at each occurrence p, where σ is a substitution mapping all variables x_ϵ bound in s but possibly free in $s|_p$ to variables of the form y_τ^ℓ. (The purpose of σ is to effect all substitutions needed in rule (Abs) in a lazy way.)

Algorithm 21 *For each occurrence p in s:*

(Variable case) If p is an occurrence of a constant or a variable, equal to or mapped by σ to x_τ^μ, then let $\tau_p(\sigma)$ be θ_p, and $K_p(\sigma)$ be $K(\tau \sqsubseteq \theta_p) \cup \{\mu_p \geq \mu\} \cup K(l(\tau) \leq \mu)$ (rule (Var')).

(Application case) If p is an occurrence of an application uv, then check that $\tau_{p0}(\sigma)$ (intuitively, the type of u) is of the form $\tau_1 \xrightarrow{\xi} \tau_2$, with $E(\tau_1) = E(\tau_{p1}(\sigma))$; otherwise, fail ($s$ is not typable). Let then $\tau_p(\sigma)$ be τ_2, and $K_p(\sigma)$ be $K_{p0}(\sigma) \cup K_{p1}(\sigma) \cup K(\tau_1 = \tau_{p1}) \cup \{\xi = \mu_{p1}, \mu_p = \mu_{p0}\}$ (rule (App)).

(Abstraction case) If p is an occurrence of an abstraction $\lambda x_\epsilon \cdot v$, then let $y_{\theta_x}^{\nu_x}$ be a variable not free in v, $\tau_p(\sigma) = \theta_x \xrightarrow{\nu_x} \tau_{p0}(\sigma[y_{\theta_x}^{\nu_x}/x])$, and $K_p(\sigma)$ be $K_{p0}(\sigma[y_{\theta_x}^{\nu_x}/x]) \cup K(\theta_x$ ramified$) \cup K(l(\theta_x) \leq \nu_x\} \cup \{\mu_p \geq \nu_x + 1, \mu_p \geq \mu_{p0}\}$ (rule (Abs); notice that we don't encode $\mu_p = \max(\nu_x + 1, \mu_{p0})$, but $\mu_p \geq \max(\nu_x + 1, \mu_{p0})$, but this is all right, as we shall see).

Finally, we let τ_s be $\tau_\varepsilon([])$, μ_s be μ_ε, and $K(s : \tau_s/\mu_s)$ be $K_\varepsilon([])$.

Proof of Correctness: We now prove that the algorithm is correct. We claim that for each occurrence p, (1) if $\tau_p(\sigma)$ is undefined, then $u|_p\sigma$ is not typable, and (2) if $\tau_p(\sigma)$ is defined, then the assignments ρ under which $u|_p\sigma$ is typable are exactly those which satisfy $K_p(\sigma)$ up to $\mathrm{FLV}(u|_p\sigma)$, and that its type is then τ_p/μ_p. We prove the claim by structural induction on the tree of occurrences.

If p is an occurrence of a variable, then $u|_p\sigma$ is some x_τ^μ. (1) is trivial since $\tau_p(\sigma)$ is always defined. As for (2), if x is typable of type τ'/μ_p under ρ, then we must have $\mu \geq l(\tau)$, and the type has been obtained by rule (Var'), so $\tau \sqsubseteq \tau'$ and $\mu \leq \mu_p$. From $\tau \sqsubseteq \tau'$ it follows that $E(\tau) = E(\tau')$, hence that the general form of τ' is described by θ_p. So $\tau_p(\sigma)$ must indeed be θ_p with $K_p(\sigma)$ satisfied. Conversely, if $K_p(\sigma)$ is satisfied by ρ, then under ρ we have $\tau \sqsubseteq \theta_p$, hence by Lemma 2 τ is a ramified type; moreover $l(\tau) \leq \mu$, $\tau \sqsubseteq \theta_p$ and $\mu \leq \mu_p$ under ρ, so by rule (Var') $u|_p\sigma$ has indeed type θ_p/μ_p.

Assume that p is an occurrence of an application uv. If the algorithm fails, then either it failed in u or in v, in which case $(uv)\sigma$ is not typable by induction hypothesis, claim (1); or it fails because $\tau_{p0}(\sigma)$ is not an functional type or $E(\tau_1) \neq E(\tau_{p1}(\sigma))$: in each case, rule (App) cannot be applied, so $(uv)\sigma$ is not typable (this is by Theorem 6). This proves (1). As for (2), if $(uv)\sigma$ is typable, then u and v are, too, the type τ_1 of u's argument must equal the type τ_{p1} of v, the level μ_1 of u's argument must equal the level μ_{p1} of v, and the level μ_p of $(uv)\sigma$ must equal that of $u\sigma$, namely μ_{p0}: this yields all the constraints in $K_p(\sigma)$. Moreover, the type returned must be the type τ_2 of the results of $u\sigma$. Conversely, it is clear that if $K_p(\sigma)$ is satisfied by ρ, then by induction hypothesis we have type derivations of $u\sigma : \tau_1 \xrightarrow{\mu_1} \tau_2/\mu_{p0}$ and $v\sigma : \tau_1/\mu_1$, so that we can apply rule (App) to get $(uv)\sigma : \tau_2/\mu_p$.

Finally, assume that p is an occurrence of an abstraction $\lambda x_\epsilon \cdot v$. Claim (1) is clear from the induction hypothesis. As for (2), if $(\lambda x_\epsilon \cdot v)\sigma$ is typable under ρ, then by Theorem 6 the last rule in the derivation can be assumed to be (Abs), so there is a variable $y_{\theta_x}^{\nu_x}$ not free in v such that $E(\theta_x) = \epsilon$ and $v\sigma[y_{\theta_x}^{\nu_x}/x]$ is typable. By induction hypothesis, $K_{p0}(\sigma[y_{\theta_x}^{\nu_x}/x])$ must therefore be satisfied by ρ. And θ_x must be a ramified type, with $\nu_x \geq l(\theta_x)$ so $K(\theta_x$ ramified$) \cup K(l(\theta_x) \leq \nu_x)$ must also be satisfied by ρ. Moreover, we must have $\mu_p = \max(\nu_x + 1, \mu_{p0})$, so in particular $\mu_p \geq \nu_x + 1, \mu_p \geq \mu_{p0}$; hence, $K_p(\sigma)$ is satisfied by ρ. And the type of $(\lambda x_\epsilon \cdot v)\sigma$ must then be $\theta_x \xrightarrow{\nu_x} \tau_{p0}(\sigma[y_{\theta_x}^{\nu_x}/x])$, i.e. $\tau_p(\sigma)$. Conversely, if

$K_p(\sigma)$ is satisfied, then it is clear by induction hypothesis that under any ρ satisfying it, we can infer $(\lambda x_\epsilon \cdot v)\sigma : \theta_x \xrightarrow{\nu_x} \tau_{p0}(\sigma[y^{\nu_x}_{\theta_x}/x])/\max(\nu_x+1, \mu_{p0})$, i.e. $(\lambda x_\epsilon \cdot v)\sigma : \tau_p(\sigma)/\max(\nu_x+1, \mu_{p0})$, where θ_x is ramified and $\nu_x \geq l(\theta_x)$. To infer $(\lambda x_\epsilon \cdot v)\sigma : \tau_p(\sigma)/\mu_p$, since $\mu_p \geq \max(\nu_x+1, \mu_{p0})$, we just apply rule (Cml) (and apply Lemma 5 if we insist on getting normal derivations).

Running Time: Finally, Algorithm 21 runs in polynomial time. We assume that σ is represented by a balanced tree or any data structure where adding an element and retrieving an element is fast — typically in time logarithmic in the cardinality of σ, i.e. at most $O(\log |s|)$. Furthermore, we assume that unions of sets of constraints are done by adding one edge at a time to a global graph; for example, the behavior on applications is just adding the edges resulting from $\tau_1 = \tau_{p1}$, $\mu_1 = \mu_{p1}$ and $\mu_p = \mu_{p0}$ to the global graph.

We then claim that computing $\tau_\varepsilon([])$ and $K_\varepsilon([])$ takes $O(|s|^2 \log |s|)$ time, for any $\alpha > 1$ (more precisely, time bounded by $k|s|^2 \log(|s|+1)$). To show this, we prove by structural induction on the occurrence tree that computing $\tau_p(\sigma)$ and $K_p(\sigma)$ takes $k|u|^2 \log(|s|+1)$ time, where $u = s|_p$, and that $|\tau_p(\sigma)| \leq O(|u|)$.

Each variable step, for a variable equal to or mapped by σ to x^μ_τ adds at most $O(|\tau|)$ new edges to the global graph, and therefore takes at most $O(|x^\mu_\tau| \times \log |s|) \leq k|x^\mu_\tau|^2 \log(|s|+1)$. Moreover, the returned type has the same size as τ, i.e. less than $O(|x^\mu_\tau|)$.

Each application step (where $s|_p = uv$) needs time $O(|\tau_{p1}(\sigma)|)$ to test for failure (u having non-functional type or $E(\tau_1) \neq E(\tau_{p1}(\sigma))$), and adds $O(|\tau_{p1}(\sigma)|)$ new edges to the global graph. This needs at most $O(|\tau_{p1}(\sigma)|) = O(|v|)$ time by induction hypothesis. Now typing u took time at most $k|u|^2 \log(|s|+1)$, typing v took $k|v|^2 \log(|s|+1)$, so all in all we need time $k|u|^2 \log(|s|+1) + k|v|^2 \log(|s|+1) + k'|v| + o(|v|)$ to check uv. But since $|uv| = |u| + |v| + 1$, $k|uv|^2 \log(|s|+1) \geq k|u|^2 \log(|s|+1) + k|v|^2 \log(|s|+1) + 2k|u||v| \log(|s|+1)$. So, provided that k is high enough that $2k|u| \log(|s|+1) \geq k'$ (i.e. for example $k \geq k'/(2 \log 2)$), the time taken is bounded above by $k|uv|^2 \log(|s|+1)$. Moreover, we have $|\tau_p(\sigma)| \leq O(|u|) \leq O(|uv|)$.

Each abstraction step (where $s|_p = \lambda x_\epsilon \cdot u$) needs $O(|\epsilon|)$ time to build θ_x, $O(\log |s|)$ time to add the binding $y^{\nu_x}_{\theta_x}/x$ to σ, $k|u|^2 \log(|s|+1)$ time to type u, and adds $O(|\epsilon|)$ edges to the global graph. Therefore it takes at most $k|u|^2 \log(|s|+1) + O(\log |s|) + O(|\epsilon|)$ time, which is (much) less that $k|\lambda x_\epsilon \cdot u|^2 \log(|s|+1)$, since $|\lambda x_\epsilon \cdot u| = |\epsilon| + |u| + 1$. And the returned type $\tau_p(\sigma)$ is $\theta_x \xrightarrow{\nu_x} \tau_{p0}(\sigma[y^{\nu_x}_{\theta_x}/x])$, where by induction hypothesis $|\tau_{p0}(\sigma[y^{\nu_x}_{\theta_x}/x])| \leq O(|u|)$, and by construction $|\theta_x| \leq |\epsilon|$, so the returned type has size $O(|\lambda x_\epsilon \cdot u|)$.

The claim is proved. The total time is then $O(|s|^2 \log |s|)$: this is polynomial in $|s|$. □

Corollary 22 *Testing whether a given term is typable is decidable in polynomial time.*

Proof: Apply Algorithm 21, and then apply the algorithm of Lemma 18 on the graph $K(s : \tau_s/\mu_s)$, of size $O(|s|^2)$. □

6.3 Example

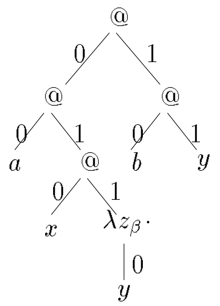

Figure 3: Term example

For example, consider the term $a(x(\lambda z_\beta \cdot y))(by)$, where β is a base type, $a = a^{\ell_3}_{\beta \xrightarrow{\ell_1} \beta \xrightarrow{\ell_2} \beta}$ and $b = b^{\ell_5}_{\beta \xrightarrow{\ell_4} \beta}$ are constants, and $x = x^{\ell_8}_{(\beta \xrightarrow{\ell_6} \beta) \xrightarrow{\ell_7} \beta}$ and $y = y^{\ell_9}_\beta$ are variables. The free level variables are $\ell_1, \ldots, \ell_9$. This term is shown as a tree in Figure 3. At marks (@) denote application nodes, and 0's and 1's denote letters forming paths. We create new variables μ_ε, μ_0, μ_{00}, μ_{01}, μ_{010}, μ_{011}, μ_{0110}, μ_1, μ_{10} and μ_{11} (levels of subterms); ν_z (and $\theta_z = \beta$); ξ^ε_{00}, ξ^1_{00}, (for a,) ξ^ε_{010}, ξ^0_{010}, (for x,) ξ^ε_{10} (for b). We have $\theta_{00} = \beta \xrightarrow{\xi^\varepsilon_{00}} \beta \xrightarrow{\xi^1_{00}} \beta$, $\theta_{010} = (\beta \xrightarrow{\xi^0_{010}} \beta) \xrightarrow{\xi^\varepsilon_{010}} \beta$, $\theta_{10} = \beta \xrightarrow{\xi^\varepsilon_{10}} \beta$, while $\theta_{0110} = \theta_{11} = \beta$.

Consider first the occurrence 00 of a. Since there is no λ-header above this occurrence, σ will be the empty substitution. Since a is a constant, we are in the first case of the Algorithm 21. Then $\tau_{00}(\sigma) = \theta_{00} = \beta \xrightarrow{\xi^\varepsilon_{00}} \beta \xrightarrow{\xi^1_{00}} \beta$. The constant a is of the form a^μ_τ, where $\tau = \beta \xrightarrow{\ell_1} \beta \xrightarrow{\ell_2} \beta$ and $\mu = \ell_3$. So $K_{00}(\sigma) = \{\ell_1 \geq \xi^\varepsilon_{00}, \ell_2 \geq \xi^1_{00}\} \cup \{\mu_{00} \geq \ell_3\} \cup \{\ell_3 \geq \ell_1+1, \ell_3 \geq \ell_2+1\}$.

Then, look at the occurrence 010 of x. Again, σ is the empty substitution, and we are in the first case

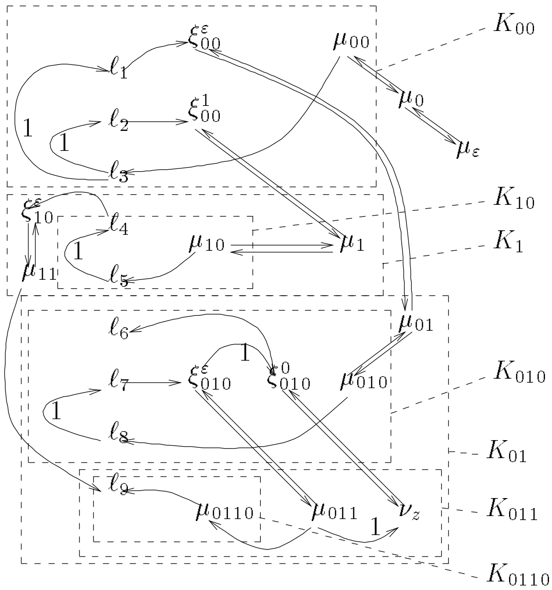

Figure 4: Corresponding graph

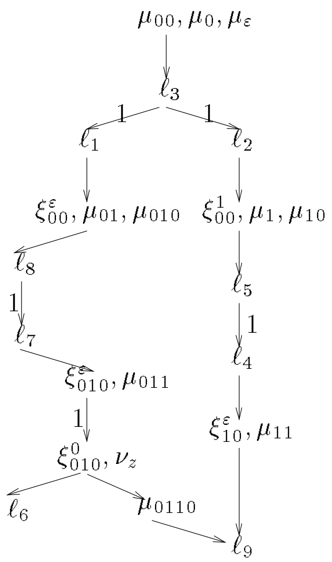

Figure 5: Condensation graph

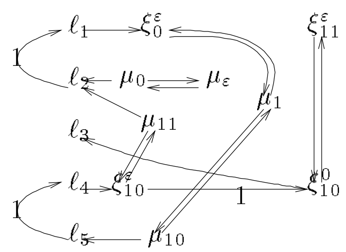

Figure 6: An unsatisfiable graph

of Algorithm 21. $\tau_{010}(\sigma) = \theta_{010} = (\beta \xrightarrow{\xi^0_{010}} \beta) \xrightarrow{\xi^\varepsilon_{010}} \beta$, $\tau = (\beta \xrightarrow{\ell_6} \beta) \xrightarrow{\ell_7} \beta$, $\mu = \ell_8$; so $K(\tau \sqsubseteq \theta_{010}) = \{\xi^0_{010} \geq \ell_6, \ell_7 \geq \xi^\varepsilon_{010}, \xi^\varepsilon_{010} \geq \xi^0_{010} + 1\}$, and $K_{010}(\sigma) = \{\xi^0_{010} \geq \ell_6, \ell_7 \geq \xi^\varepsilon_{010}, \xi^\varepsilon_{010} \geq \xi^0_{010} + 1, \mu_{010} \geq \ell_8, \ell_8 \geq \ell_7 + 1\}$.

The only abstraction in this term is at occurrence 011. We create a new variable $y^{\nu_z}_{\theta_z}$, where $\theta_z = \beta$, and map it to z in σ. Then $K_{011}(\sigma) = K_{0110}(\sigma[y^{\nu_z}_{\theta_z}/z]) \cup \emptyset \cup \emptyset \cup \{\mu_{011} \geq \nu_z + 1, \mu_{011} \geq \mu_{0110}\}$. And the K_{0110} set is just $\{\mu_{0110} \geq \ell_9\}$.

We find an application for example at occurrence 01. The type of the function is θ_{010}, i.e. $(\beta \xrightarrow{\xi^0_{010}} \beta) \xrightarrow{\xi^\varepsilon_{010}} \beta$, and the type of the argument is the type of the abstraction above, namely $\theta_z \xrightarrow{\nu_z} \beta = \beta \xrightarrow{\nu_z} \beta$. The erasure of the typed expected by the function is $\beta \rightarrow \beta$, as is the erasure of the type of the argument, so Algorithm 21 does not fail (the term is simply-typed). Then $K_{01}(\sigma) = K_{010}(\sigma) \cup K_{011}(\sigma) \cup \{\xi^0_{010} = \nu_z\} \cup \{\xi^\varepsilon_{010} = \mu_{011}, \mu_{01} = \mu_{010}\}$ (each of these equalities being really two inequalities, i.e. a length 2 cycle in the graph), and the returned type is β.

Continuing this process, we get as most general type β, with level μ_ε submitted to the constraints represented in Figure 4. We have omitted all 0-weights on edges, so as to make the graph more readable. The condensation graph is shown in Figure 5; each $\overline{C}$ is consistent, so there are level assignments satisfying this set of constraints. The smallest assigns level 3 to the whole expression.

Let's take another example, and consider the term

$x(yx)$, where $x = x^{\ell_2}_{\beta \xrightarrow{\ell_1} \beta}$ and $y = y^{\ell_5}_{(\beta \xrightarrow{\ell_3} \beta) \xrightarrow{\ell_4} \beta}$. By Theorem 8, this term cannot be typable, so let's check it. The computed graph is shown in Figure 6, and has four connected components: $\{\ell_3\}$, $\{\xi^0_{10}, \xi^\varepsilon_{11}\}$, $\{\xi^\varepsilon_{10}, \mu_{11}, \ell_2, \ell_1, \xi^\varepsilon_0, \mu_1, \mu_{10}, \ell_5, \ell_4\}$, and $\{\mu_0, \mu_\varepsilon\}$. But the third is inconsistent, as it contains two edges of non-zero weight (alternatively, it contains a cycle of weight 2).

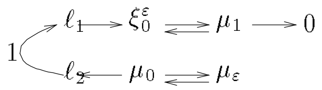

Figure 7: Checking for satisfiable instances

Naturally, although $x(yx)$ is untypable, it has typable instances (modulo $\beta\eta$). This comes from Lemma 13, noticing that $(x(yx)\zeta) \downarrow\ = \hat{e}_{\beta \to \beta}$ is clearly typable. If x is a constant and y is a variable, $x(yx)$ still has typable instances as $(x(yx)\zeta) \downarrow\ = x\hat{e}_{(\beta\to\beta)\to\beta}$ is typable. Indeed, this term is $x^{\ell_2}_{\beta \xrightarrow{\ell_1} \beta} v^0_\beta$, and its graph is shown in Figure 7.

6.4 Improvements

We can improve Algorithm 21 a great deal. First, we are not forced to generate all level variables in advance, and we may create them as we need them.

Then, we may define K_p for variable or constant occurrences p as just $K(\tau \sqsubseteq \theta_p) \cup \{\mu_p \geq \mu\}$ so as to reduce the number of constraints that we add; a preprocessing step will then add the constraints $K(l(\tau) \leq \mu)$ for all free variables and constants x^μ_τ. (Notice that we don't need to do this for bound variables, because the constraints are those in $K(l(\theta_x) \leq \nu_x)$ introduced in the abstraction case.) This avoids rebuilding these same constraints over and over at each occurrence of the same variable. We can also avoid adding the constraints in $K(\theta_x$ ramified) in the abstraction case as soon as x occurs free in the abstraction, since then each occurrence of x will produce constraints of the form $K(\tau \sqsubseteq \theta_p)$, where $\tau = \theta_x$, and this forces θ_x to be ramified by Lemma 2. Still on the chapter of variables, we can use Theorem 8 to fail right away when trying to type applications of the form xu, where x is a variable or a constant that occurs free in u.

In the application step, we add constraints of the form $K(\tau_1 = \tau^1_p)$, $\mu_1 = \mu_{p1}$, $\mu_p = \mu_{p0}$, which are equalities between level variables. Instead of representing an equality $\mu = \nu$ as two edges $\mu \xrightarrow{0} \nu$ and $\nu \xrightarrow{0} \mu$, it is more efficient to simply merge the nodes of μ and ν. This not only decreases the size of the graph $G(K_p(\sigma))$, but also and therefore accelerates the algorithm of Lemma 18 applied on $G(K_\varepsilon([]))$. (In particular, observe that this correctly identifies all connected components of the first example of Section 6.3 right away.)

In the abstraction step, we can also dispense with the creation of a variable $y^{\nu_x}_{\theta_x}$. Observe that what we really need is just θ_x and ν_x, not y. This is only a minor point.

A more important point is that although we examine applications as unary applications, it is more profitable to deal with n-ary applications in one fell swoop. That is, observe that any term can be written uniquely as $\lambda x_{1\epsilon_1} \cdot \ldots \cdot \lambda x_{m\epsilon_m} \cdot hu_1 \ldots u_n$, where $m \geq 0$, $n \geq 0$ and h is not an application. Then we type it (at occurrence p) by extending σ to $\sigma[y^{\nu_{x_1}}_{\theta_{x_1}}/x_1, \ldots, y^{\nu_{x_m}}_{\theta_{x_m}}/x_m]$, typing h, u_1, $\ldots$, u_n, failing if the type of h is not of the form $\tau_1 \xrightarrow{\mu_1} \ldots \tau_n \xrightarrow{\mu_n} \tau$, with τ_i having the same erasure as the type τ'_i of u_i for each i, $1 \leq i \leq n$, and then producing $K(\theta_{x_j}$ ramified) and $K(l(\theta_{x_j}) \leq \nu_{x_j})$ for every $1 \leq j \leq m$, $K(\tau_i = \tau'_i)$, $\mu_i = \mu'_i$ (where μ'_i is the level of u_i) for every $1 \leq i \leq n$, plus $\mu_p = \mu$ if $m = 0$ or $\mu_p \geq \nu_{x_1} + 1$, $\ldots$, $\mu_p \geq \nu_{x_m} + 1$, $\mu_p \geq \mu$ if $m > 0$, where μ is the level of h; and we return $\theta_{x_1} \xrightarrow{\nu_{x_1}} \ldots \theta_{x_m} \xrightarrow{\nu_{x_m}} \tau$ as $\tau_p(\sigma)$. This cuts drastically on the number of auxiliary level variables that we need, at least when we have many abstractions, constants or variables taking several arguments; indeed, we don't need any variable for $(\lambda x_{2\epsilon_2} \cdot \ldots \cdot \lambda x_{m\epsilon_m} \cdot hu_1 \ldots u_n)$, $\ldots$, $(\lambda x_{m\epsilon_m} \cdot hu_1 \ldots u_n)$, $(hu_1 \ldots u_m)$, $(hu_1 \ldots u_{m-1})$, $\ldots$, or hu_1 any longer.

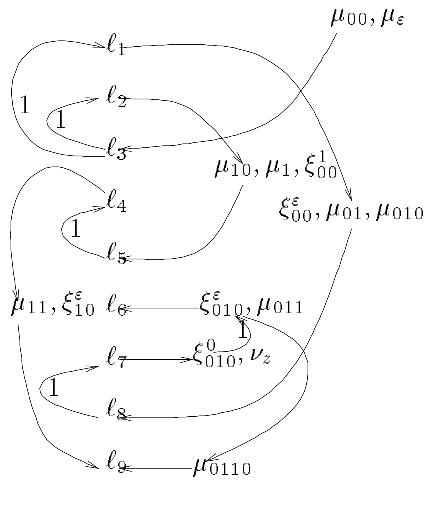

Figure 8: Simplified graph

Returning to the example $a(x(\lambda z_\beta \cdot y))(by)$ of Section 6.3, and applying the tricks above, we get the graph of Figure 8. The main gain comes from identifying nodes that are equated in the application step. (Observe also that this made us merge the two $\xrightarrow{1}$ arrows from ξ^{ε}_{010} to ξ^{0}_{010}, and from μ_{011} to ν_z.) The trick where we decompose terms as m-ary abstractions of n-ary applications $\lambda x_{1\epsilon_1} \cdot \ldots \cdot \lambda x_{m\epsilon_m} \cdot h u_1 \ldots u_n$ only makes the variable μ_0 disappear, which is not much: this is because there is only one binary application and no n-ary abstraction, $n > 1$, in the example.

We can also improve Algorithm 21 in common cases by doing a small amount of preprocessing, where we type-check the term in the *simple* theory of types first, and at the same time check that in each component $\lambda x_{1\epsilon_1} \cdot \ldots \cdot \lambda x_{m\epsilon_m} \cdot h u_1 \ldots u_n$ where h is a constant or a variable, h does not occur free in u_1, ..., or u_n (the vicious circle principle). Otherwise, by Theorem 8, type-checking fails. If the preprocessing phase does not fail, then we apply Algorithm 21 (with the improvements described above). The latter, now, cannot fail, and merely builds a constraint graph, which we then solve by Lemma 18.

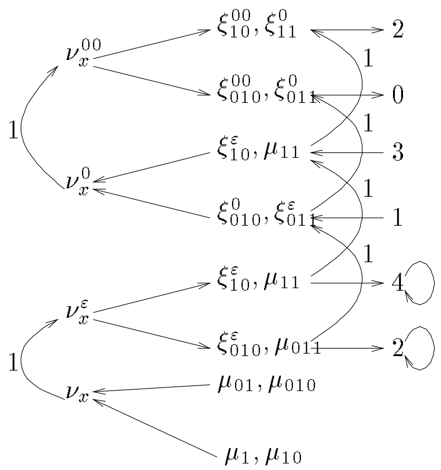

Figure 9: Another untypable term

This preprocessing will catch most untypable instances. However, notice that it won't catch all of them: there are well-simply-typed terms in normal form that obey the vicious circle principle but cannot be well-typed in the ramified sense. For instance, consider $\lambda x_{((\beta\to\beta)\to\beta)\to\beta} \cdot \lambda y_{\beta\to\beta} \cdot y(x a^2_{(\beta \xrightarrow{0} \beta)\xrightarrow{1}\beta})(x b^4_{(\beta\xrightarrow{2}\beta)\xrightarrow{3}\beta})$. This term is well-simply-typed of type β, is $\beta\eta$-normal, obeys the vicious circle principle but is not typable. Indeed, the part of the graph that the Algorithm 21 builds corresponding to the subterms xa and xb is shown in Figure 9: check that the path $1 \xrightarrow{0} \xi^{0}_{010}, \xi^{\varepsilon}_{011} \xrightarrow{0} \nu^{0}_{x} \xrightarrow{1} \nu^{00}_{x} \xrightarrow{0} \xi^{00}_{10}, \xi^{0}_{11} \xrightarrow{0} 2$ is indeed insatisfiable. The reason why this example is not typable is that the types of a and b have no common $\sqsubseteq$-lower bound, so that the same variable x cannot be applied to both a and b.

7 Decidability of Unification

7.1 Integer Levels, $\beta\eta$-Equality

Building on Sections 5 and 6, we prove that ramified higher-order ramification with integer levels ($L = \omega$) is decidable.

First, we make precise the informal argument of decreasing levels discussed in Section 4.3. We have already said that this informal argument was not quite correct. The main reason why is that it is not so much the levels of free variables that count as the levels that we can get by instantiating these free variables by a unifier and reducing. This justifies the following definition; recall that a substitution is normalized when it maps variables to terms in β-normal η-expanded form.

Definition 7 *For every β-normal term u of the form $\lambda y_{1\epsilon_1} \cdot \ldots \cdot \lambda y_{n\epsilon_n} \cdot t$, where t is not an abstraction, let the* min-level *$l(u)$ of u be the least ℓ such that $t[x^{\ell_1}_{\tau_1}/y_1, \ldots, x^{\ell_n}_{\tau_n}/y_n]$ is typable of type τ/ℓ for some type τ, where $E(\tau_1) = \epsilon_1, \ldots, E(\tau_n) = \epsilon_n$.*

Lemma 23 *For any well-typed normalized substitution σ, for every variable F^{ℓ}_{τ}, $l(F\sigma) \leq \ell$.*

Proof: Since σ is well-typed, $F\sigma$ is typable of some type τ'/ℓ' with $\tau' \sqsubseteq \tau$ and (1) $\ell' \leq \ell$. Let's write τ' as $\tau_1 \xrightarrow{\ell_1} \ldots \tau_n \xrightarrow{\ell_n} b$, where b is a base type. Since σ is normalized, $F\sigma$ must be of the form $\lambda y_{1\epsilon_1} \cdot \ldots \cdot \lambda y_{n\epsilon_n} \cdot t$; and by rule (Abs), $E(\tau_1) = \epsilon_1, \ldots, E(\tau_n) = \epsilon_n$, and $t[x^{\ell_1}_{\tau_1}/y_1, \ldots, x^{\ell_n}_{\tau_n}/y_n]$ is typable of type b/ℓ'', with $\ell' = \max(\ell_1 + 1, \ldots, \ell_n + 1, \ell'')$. In particular, (2) $\ell'' \leq \ell'$. By Definition 7, (3) $l(F\sigma) \leq \ell''$. By (1), (2) and (3) it follows that $l(F\sigma) \leq \ell$. □

Let's say that a substitution is *weakly well-typed* if and only if it maps every variable x^{ℓ}_{τ} to some term t of type τ'/ℓ' with $E(\tau) = E(\tau')$. A well-typed substitution imposes moreover $\tau = \tau'$ and $\ell = \ell'$.

Lemma 24 *Let σ be an arbitrary weakly well-typed normalized substitution, F a variable, $t = \lambda\overline{y_n} \cdot a(\overline{\lambda \overline{z_{p_m}} \cdot H_m \overline{y_n}\ \overline{z_{p_m}}})$ a partial binding appropriate to F, where a is a constant or a bound variable.*

If σ unifies F and t, then for every $1 \leq i \leq m$, $l(H_i\sigma) < l(F\sigma)$.

Proof: $F\sigma$ is a β-normal η-expanded term, say $\lambda y_{1\epsilon_1} \cdot \ldots \cdot \lambda y_{n\epsilon_n} \cdot s$. (The abstraction header is the same as that of t, because both those terms are η-expanded terms and of the same simple type.)

Since σ is normalized, for each $1 \leq i \leq m$, $H_i\sigma$ is β-normal and η-expanded, so it is $\lambda y'_1 \cdot \ldots \cdot \lambda y'_n \cdot \lambda z'_1 \cdot \ldots \cdot \lambda z'_{p_i} \cdot t_i$ for some β-normal term t_i of type b_i. Without loss of generality, we may drop the primes on the bound variables in the header of $H_i\sigma$, so that $(H_i\sigma)\overline{y_n}\ \overline{z_{p_i}}$ is just t_i.

Since σ unifies F with t, $F\sigma =_{\beta\eta} (t\sigma)\downarrow$, and because both sides of the equality are β-normal and η-expanded, $F\sigma = (t\sigma)\downarrow = \lambda\overline{y_n} \cdot a(\lambda\overline{z_{p_1}} \cdot t_1) \ldots (\lambda\overline{z_{p_m}} \cdot t_m)$. So $s = a(\lambda\overline{z_{p_1}} \cdot t_1) \ldots (\lambda\overline{z_{p_m}} \cdot t_m)$. Letting $x_1{}^{\ell_1}_{\tau_1}$, ..., $x_n{}^{\ell_n}_{\tau_n}$ be fresh variables with $E(\tau_1) = \epsilon_1$, ..., $E(\tau_n) = \epsilon_n$, we have:

$$\begin{aligned} & s[x_1/y_1, \ldots, x_n/y_n] = \\ & \quad a'(\lambda\overline{z_{p_1}} \cdot t_1[x_1/y_1, \ldots, x_n/y_n]) \\ & \quad\quad \ldots (\lambda\overline{z_{p_m}} \cdot t_m[x_1/y_1, \ldots, x_n/y_n]) \end{aligned}$$

where $a' = a[x_1/y_1, \ldots, x_n/y_n]$. (So $a' = a$ for imitation bindings, and $a' = x_j$ for some j in the case of projection bindings.) In particular, $l(F\sigma)$ is the least ℓ such that $a'(\lambda\overline{z_{p_1}} \cdot t_1[x_1/y_1, \ldots, x_n/y_n]) \ldots (\lambda\overline{z_{p_m}} \cdot t_m[x_1/y_1, \ldots, x_n/y_n])$ is typable of type τ/ℓ for some τ.

Any normalized derivation of the latter must end in m instances of (App), following one instance of (Var') to type a'. Now a' is a variable or constant, which is assigned some ramified type $\tau'_1 \xrightarrow{\ell'_1} \ldots \tau'_m \xrightarrow{\ell'_m} b$ and some level ℓ' in the derivation by rule (Var'), with (1) $\ell' = \ell$ (by rule (App)); since this type is ramified by Lemma 3, (2) $\ell' \geq \max(\ell'_1 + 1, \ldots, \ell'_m + 1)$. On the other hand, for each $1 \leq i \leq m$, $\lambda\overline{z_{p_i}} \cdot t_i[x_1/y_1, \ldots, x_n/y_n]$ must have received the decorated type τ'_i/ℓ'_i, by rule (App). To type the latter, we must have used rule (Abs) p_i times, and $t_i[x_1/y_1, \ldots, x_n/y_n, x'_1{}^{l_1}_{\theta_1}/z_1, \ldots, x'_{p_i}{}^{l_{p_i}}_{\theta_{p_i}}/z_{p_i}]$, where $x'_j{}^{l_j}_{\theta_j}$, $1 \leq j \leq p_i$, are fresh variables, must have been given a (base) type b_i and a level l'_i; so $\tau'_i = \theta_1 \xrightarrow{l_1} \ldots \theta_{p_i} \xrightarrow{l_{p_i}} b_i$ and (3) $\ell'_i = \max(l_1 + 1, \ldots, l_{p_i} + 1, l'_i)$ (by (Abs)). And by Definition 7, (4) $l'_i \geq l(H_i\sigma)$.

By (1) and (2) $\ell = \ell' \geq \ell'_i + 1$. But by (3) $\ell'_i \geq l'_i$, so $\ell \geq l'_i + 1$, and by (4) $\ell \geq l(H_i\sigma) + 1$. Since $l(F\sigma)$ is the lowest such ℓ, it follows that $l(F\sigma) \geq l(H_i\sigma) + 1$. □

We shall also need the following:

Definition 8 *For any simple type ϵ, let $\tau(\epsilon)$ be the ramified type defined by $\tau(b) = b$ for any base type b and $\tau(\epsilon_1 \to \epsilon_2) = \tau(\epsilon_1) \xrightarrow{l(\tau(\epsilon_1))} \tau(\epsilon_2)$.*

Lemma 25 *For any simple type ϵ, $\tau(\epsilon)$ is a ramified type τ of minimal level such that $E(\tau) = \epsilon$.*

Proof: That $\tau(\epsilon)$ is a ramified type follows from the definition. By an easy structural induction on ϵ, $E(\tau(\epsilon)) = \epsilon$. Finally, let τ be another type such that $E(\tau) = \epsilon$, then we claim that $l(\tau) \geq l(\tau(\epsilon))$. This is proved by structural induction on ϵ. If ϵ is a base type, then $l(\tau) = 0 = l(\tau(\epsilon))$. And if ϵ is of the form $\epsilon_1 \to \epsilon_2$, then τ is of the form $\tau_1 \xrightarrow{\ell_1} \tau_2$, where by induction hypothesis (1) $l(\tau_1) \geq l(\tau(\epsilon_1))$, (2) $l(\tau_2) \geq l(\tau(\epsilon_2))$, and because τ is ramified (3) $\ell_1 \geq l(\tau_1)$. By (1) and (3), it follows (4) $\ell_1 \geq l(\tau(\epsilon_1))$. And $l(\tau) = \max(\ell_1 + 1, l(\tau_2)) \geq \max(l(\tau(\epsilon_1)) + 1, l(\tau_2))$ (by (4)) $\geq \max(l(\tau(\epsilon_1)) + 1, l(\tau(\epsilon_2)))$ (by (2)) $= l(\tau(\epsilon))$. □

We are now able to formulate our algorithm for ramified higher-order unification with integer levels. Recall that the input is a multiset M of unordered pairs of simply-typed terms in β-normal form. For each free variable x, we estimate an upper bound ℓ on the possible $l(x\sigma)$ for any unifier σ. To map each variable to this upper bound, we use the following trick: we store ℓ as the level of the variable, i.e., we write x as x^ℓ_τ, for some τ. This is certainly consistent with the usual meaning of levels, by Lemma 23. Lemma 25 will allow us to find τ. The only difficulty is that a unifier σ need not assign a term of level at most ℓ to these new variables, hence the notion of weakly well-typed substitutions that we have introduced just before Lemma 24.

We use Lemma 24 and Lemma 25 to refine Definition 3:

Definition 9 *Let ϵ be a simple type, which we write $\epsilon_1 \to \ldots \epsilon_n \to b$ with b a base type, ℓ be an integer level, and a be a variable or constant of simple type $\epsilon'_1 \to \ldots \epsilon'_m \to b$.*

For each $1 \leq i \leq m$, let's write ϵ'_i as ${\epsilon''}^i_1 \to \ldots {\epsilon''}^i_{p_i} \to b_i$, where b_i is a base type. Assume that:

$$(A) \quad \ell > l(\tau(\epsilon_1 \to \ldots \epsilon_n \to {\epsilon'}^i))$$

for each $1 \leq i \leq m$.

The set $B(\epsilon, a, \ell)$ of partial bindings *of type ϵ, head a, and level ℓ, is the set of terms of the form:*

$$\begin{aligned} & \lambda y_{1\epsilon_1} \cdot \ldots \cdot \lambda y_{n\epsilon_n} \cdot \\ & \quad a(\lambda\overline{z^1_{p_1}} \cdot H_1\overline{y_n}\overline{z^1_{p_1}}) \ldots (\lambda\overline{z^m_{p_m}} \cdot H_m\overline{y_n}\overline{z^m_{p_m}}) \end{aligned}$$

where:

1. *for every $1 \leq i \leq m$, $1 \leq j \leq p_i$, z^i_j is a variable $(z^i_j)_{{\epsilon''}^i_j}$;*

2. *for every* $1 \leq i \leq m$, H_i *is a variable* $(H_i)^{\ell-1}_{\tau(\epsilon_1 \to \dots \epsilon_n \to \epsilon''^i_1 \to \dots \epsilon''^i_{p_i} \to b_i)}$;

The set $B(F, a)$ *of partial bindings head* a appropriate to F^ℓ_τ *is* $B(E(\tau), a, \ell)$.

We let projection and imitation bindings be as in Definition 3, and keep the same notation, so that we can use the rules of Figure 2 for ramified pre-unification. Notice that an essential difference is that projection bindings only exist if condition (A) above is satisfied. Indeed, without condition (A), there could be no variable H_i of ramified type of erasure $\epsilon_1 \to \dots \epsilon_n \to \epsilon''^i_1 \to \dots \epsilon''^i_{p_i} \to b_i$ and level at most $\ell - 1$, by Lemma 25.

Algorithm 26 *Let* N *be an auxiliary function defined as follows: for any such multiset* S, $N(S)$ *applies* **(Delete)**, **(Decomp)**, **(Bind)** *on* S *until this is no longer possible, and returns the resulting multiset.*

1. *Initialize* S *to* $N(M)$.
2. *While* S *is not in solved form, do:*
   (a) *pick any flexible-rigid pair* $\langle \lambda \overline{x_k} \cdot F^\ell_\tau \overline{u_n}, \lambda \overline{x_k} \cdot a \overline{v_m} \rangle$ *in* S, *with* ℓ *satisfying condition* (A), *namely* $\ell > l(\tau(\epsilon_i)) + 1$ *for each* $x_{i\epsilon_i}$, $1 \leq i \leq k$, *and* $\ell > l(\tau(\epsilon'))$ *where* ϵ' *is the simple type of* a;
   (b) *if there is none, then fail;*
   (c) *otherwise, apply* **(Imitate)** *or* **(Project)** *non-deterministically on it, getting a new multiset* S_a;
   (d) *let* S *be* $N(S_a)$, *and loop.*
3. *Let* σ *the substitution represented by* S, *restricted to the free variables of* M. *For every variable* x^ℓ_τ *in* dom σ, *check that* $(x\sigma\zeta)\downarrow$ *is typable of type* τ/ℓ. *If so, return* σ, *otherwise fail.*

Step 3 is accomplished by using Algorithm 21 and Lemma 18, and testing for each $x\tau^\ell \in \mathrm{dom}\ \sigma$ whether $K(s : \tau_s/\mu_s) \cup K(\tau = \tau_s) \cup \{\mu_s = \ell\}$ is satisfiable, where $s = (x\sigma\zeta)\downarrow$.

Furthermore, the meaning of "pick" above denotes an arbitrary choice: picking another flexible-rigid pair does not affect soundness or completeness — although it may affect the efficiency of the algorithm. On the contrary, the rules to apply to the given flexible-rigid pair are applied non-deterministically, i.e. by backtracking for example.

Theorem 27 (Termination) *Algorithm 26 terminates.*

Proof: We first claim that N is well-defined, i.e. that any sequence of applications of **(Delete)**, **(Decomp)**, **(Bind)** on S must terminate. Let $s(S)$ be defined as $\sum_{c \in S} s(c)$, where $s\langle u_i, v_i \rangle = s(u_i) + s(v_i)$, $s(x) = 1$ for any variable or constant x, $s(uv) = s(u) + s(v) + 1$ and $s(\lambda x_\epsilon \cdot u) = s(u)$. Let $\#(S)$ be defined as the number of solved pairs $\langle \eta[x], \eta[s] \rangle$ in S (or equivalently, as the number of solved variables). Then **(Delete)** does not increase $\#(S)$ and decreases $s(S)$ by $2s(u)$. **(Decomp)** leaves $\#(S)$ constant and decreases $s(S)$ by (see Figure 2 for notations) $(s(a) + \sum_{i=0}^n (s(u_i) + 1) + \sum_{i=0}^n (s(v_i) + 1)) - (\sum_{i=0}^n u_i + \sum_{i=0}^n v_i) = 2n + 1 > 0$. And in the case of **(Bind)** (see Figure 2 for notations), since F is assumed to be free in S', F is not solved before applying the rule; but F is solved after applying it. Moreover, any other variable G that occurred in a solved pair $\langle \eta[G], \eta[t] \rangle$ before occurs only in the pair $\langle \eta[G], (\eta[t][\lambda \overline{x_k} \cdot v/F])\downarrow \rangle$ afterwards, which is solved because G does not occur in v. So $\#(S)$ decreases strictly in this case.

We now claim that the loop in step 2 of Algorithm 26 can only be traversed finitely many times. Let $\mathcal{L}(S)$ be defined as the set of unsolved variables in $N(S)$. We order such sets A by letting $\alpha(A)$ denote $\sum_{x^\ell_\tau \in A} \omega^\ell$, where the sum is the natural sum of ordinals (i.e. the summands are first sorted in decreasing order); this is akin to a multiset extension of the ordering on levels of variables [Der87].

Then, any use of **(Imitate)** or **(Project)** on S, followed by a call to the procedure N, will apply **(Bind)** on F and **(Decomp)** on the pair under consideration. That is, step 2.(b) transforms S into a new multiset S_a where F is solved, whereas F was not solved in S, by the side-conditions on **(Imitate)**, resp. **(Project)**. Moreover, all the solved free variables in S remain solved in S_a. So $\mathcal{L}(S_a)$ is obtained from $\mathcal{L}(S)$ by replacing the unsolved variable F at level ℓ by finitely many unsolved variables H_i, $1 \leq i \leq m$, with levels $\ell - 1$, and possibly erasing some other unsolved variables. Therefore $\alpha(\mathcal{L}(S_a)) < \alpha(\mathcal{L}(S))$. Since the ordering on ordinals is well-founded, step 2 can only be applied finitely many times.

Finally, step 3 terminates because Algorithm 21 and the algorithm of Lemma 18 terminate. □

Theorem 28 (Soundness) *For any* σ *returned by Algorithm 26,* $(\sigma\zeta)\downarrow$ *is a ramified unifier of* M.

Proof: By the soundness of simply-typed unification, the substitution ς represented by the multiset S of step 3 is a well-simply-typed pre-unifier of M. That is, $(\varsigma\zeta)\downarrow$ is a unifier of M. Letting σ be $\varsigma|_{\mathrm{FVC}(M)}$, therefore, $(\sigma\zeta)\downarrow$ is also a unifier of M. Moreover, it is well-typed by step 3 and the correctness of Algorithm 21 and of the algorithm of Lemma 18. □

Theorem 29 (Completeness) *For any ramified unifier ς of M, there is a computation branch of Algorithm 26 that returns a substitution σ such that ς is an instance of σ — i.e., there is a substitution σ' such that $\varsigma =_{\beta\eta} \sigma\sigma'$.*

Proof: Since ς is a ramified unifier of M, it is also a well-simply-typed unifier of M. By the completeness of Huet's algorithm, with the strategy applying **(Bind)**, **(Decomp)** and **(Delete)** eagerly, ς is an instance (modulo $\beta\eta$) of some substitution found by applying steps 1 and 2 of Algorithm 26, with the exception that we don't check condition (A). More formally, there is a finite sequence of unification problems $S_0, S_1, \ldots, S_p$, $p \geq 0$, such that $S_0 = N(M)$ (step 1), and for every $1 \leq j \leq p$, $S_i = N(S_{a\,i-1})$, where $S_{a\,i-1}$ is obtained from S_{i-1} by applying **(Imitate)** or **(Project)** on some arbitrary flexible-rigid pair of S_{i-1}. Moreover, ς unifies every S_j, $0 \leq j \leq p$, in the simply-typed sense. In particular, ς is weakly well-typed in the ramified sense.

We claim that for every variable x^ℓ_τ free in any S_j, $0 \leq j \leq p$, $l(x\varsigma) \leq \ell$. We prove the claim by induction on j. This is indeed true for all variables free in S_0, since ς is a well-typed (ramified) unifier of M and by Lemma 23. Then, for all variables free in S_j but not in S_{j-1}, such variables are variables H_i, $1 \leq i \leq m$, coming from a partial binding appropriate to some variable F free in S_{j-1}; so by Lemma 24, $l(H_i\varsigma) < l(F\varsigma)$. By induction hypothesis, and letting ℓ be the level annotating F, we have $l(F\varsigma) \leq \ell$, hence $l(H_i\varsigma) < \ell$. It follows that $l(H_i\varsigma) \leq \ell - 1$, where $\ell - 1$ is precisely the level decorating H_i, for each $1 \leq i \leq m$.

Therefore the finite sequence of unification problems verifies condition (A) at each turn through step 2, by Lemma 25. Therefore, the step 2 loop terminates successfully. Finally, by Lemma 13 and the correctness of Algorithm 21 and the algorithm of Lemma 18, step 3 also terminates successfully. Since ς unifies S_p, i.e. the S that we find in step 3, it is by construction an instance of the σ returned by the algorithm. □

Corollary 30
Ramified higher-order unification with integer levels is decidable.

Proof: By Theorems 27, 28 and 29. □

Algorithm 26 is not incremental as it stands, in that we cannot unify $\{\!|\langle u_1, v_1\rangle, \langle u_2, v_2\rangle|\!\}$ by first running it on the pair $\langle u_1, v_1\rangle$, choosing one of the answers σ, and then unifying $\langle u_2\sigma, v_2\sigma\rangle$. Indeed, $u_2\sigma$ and $v_2\sigma$ may contain variables H_i invented by the **(Imitate)** and **(Project)** steps, and the levels of these variables are mere codings of upper bounds on the min-level $l(H_i\sigma)$, not on the levels of terms to subtitute for these variables.

Incremental unification algorithms are useful in automated theorem proving and elsewhere, and the following modification to Algorithm 26 makes it incremental: we separate the set of free variables in two disjoint sets. Variables x^ℓ_τ from the first set can only be instantiated by terms of level at most ℓ, while variables y^ℓ_τ from the second set are used as fresh variables in **(Imitate)** and **(Project)**, and can only be instantiated by terms of *min-level* at most ℓ. Algorithm 26 is left unchanged.

7.2 β-Equality

It is also interesting to consider unification modulo β, i.e. without the η rule. This is in particular important in the case of ramified type theory, which is so intensional in nature [Cop71]. We sketch here why these cases are still decidable.

Just dropping the η-rule entails that we cannot use the rules of Figure 2 any longer. Instead, we have to use Huet's method for β-unification as underlying simply-typed unification procedure [Hue75]. This procedure is a bit more complicated, because it cannot use η-expanded forms any longer. In any flexible-rigid pair $\langle \lambda\overline{x_k} \cdot F^\ell_\tau \overline{u_n}, \lambda\overline{x_{k'}} \cdot a\overline{v_m}\rangle$, k is not necessarily equal to k' as before, and we have to adjust arities before imitating or projecting; that is, we must have $k \leq k'$, and F must be mapped to some term of the form $\lambda x_{k+1} \cdot \ldots \cdot \lambda x_j \cdot u$, where u is a suitable partial binding and $k \leq j \leq k'$. Such partial bindings are of the form $\lambda\overline{y_n} \cdot a'(H_1\overline{y_n}) \ldots (H_m\overline{y_n})$, where $a' = a$ or a' is a bound variable.

Another variant is to drop the η-rule *and* choose a weaker notion of α-equivalence; recall that we were forced to choose such a lax notion of α-equivalence to be consistent with η-equality. Consider therefore the language consisting of variables and constants x^ℓ_τ, applications uv, and abstractions $\lambda x^\ell_\tau \cdot u$, with conversion rules:

$$\begin{array}{ll} (\alpha_w) & (\lambda x^\ell_\tau \cdot u) = (\lambda y^\ell_\tau \cdot u[y/x]) \\ (\beta) & (\lambda x \cdot u)v \to u[v/x] \end{array}$$

where y is not free or bound in u in the α_w rule.

The typing rules are unchanged but for abstractions:

$$\frac{u : \tau_2/\ell_2}{\lambda x^{\ell_1}_{\tau_1} \cdot u : \tau_1 \xrightarrow{\ell_1} \tau_2/\max(\ell_1 + 1, \ell_2)}$$

Almost all results of this paper are unchanged, then. Theorem 6 on normal type derivations still holds, and is in fact simpler to prove than before. Theorem 8 (the vicious circle principle) is unchanged. Subject reduction (Theorem 10) also holds in this case, and is also a bit simpler to prove. Lemma 11 on erasing levels trivially holds. And, provided that we replace "$\beta\eta$-equivalent" by "β-equivalent", Theorem 15 on how to test whether a term has a ramified well-typed instance also holds.

The results of Section 6 had nothing to do with λ-conversion and therefore still apply. The only real things that change are the notions of Section 7.1. The *min-level* $l(u)$ of a β-normal term u of the form $\lambda {y_1}_{\tau_1}^{\ell_1} \cdot \ldots \cdot \lambda {y_n}_{\tau_n}^{\ell_n} \cdot t$ is now defined as the level of t, i.e. the least ℓ such that $t : \tau/\ell$ is derivable for some τ. The analogues of Lemmas 23 and 24 then hold, provided that by normalized substitution we understand substitution mapping variables to β-normal terms, not necessarily η-expanded. It follows that Algorithm 26, with the appropriate notions of partial bindings, is a terminating, sound and complete algorithm for β-unification with weak α-conversion.

8 Logic

Unification is a basic component of automated theorem proving. But in which system of logic? The answer to this question is not so easy as it may seem, and we discuss several possible approaches.

8.1 Setting Up A Deduction System

Ramified type theory gives rise to a system of logic that we call *ramified higher-order logic*. Due to the choices that we have made, this logic will have cumulative levels and be weakly extensional (the η rule), but we may as well choose more intensional logics (see Section 7.2. The language of the logic enables us to build formulas, for instance by including operators for negation $\neg$, conjunction $\wedge$ and universal quantification $\forall$. Universal quantifications must exist at all types τ and at all levels ℓ: $\forall x_\tau^\ell \cdot F$ means that F holds of all objects of type τ and level ℓ (or lower).

We adopt for instance the Gentzen-style deduction system of Figure 10. A sequent is any expression of the form $\Gamma \triangleright \Delta$, where Γ, Δ are finite sets of formulae F, G, etc. The quantifier rules $(\forall L)$ and $(\forall R)$ are special compared to the corresponding rules in simple type theory in that they not only enforce that t (in $(\forall L)$) or y (in $(\forall R)$) have the correct type, but also all the correct levels.

To get a tableau system from the latter, we interpret all these rules bottom-up [Fit90]. The $(\forall L)$ rule needs to guess a term t: we represent this term t by a meta-variable x_τ^ℓ (i.e., a free variable). The real value of t will be found by instantiating x when we try to close a path, i.e. to conclude that the current sequent is an instance of (Ax): this involves finding one formula on the left and one formula on the right of later sequent that can be unified. This much is the rationale behind our definition of ramified unification, and in particular of well-typed substitutions, in Section 4. The $(\forall L)$ rule needs to introduce a fresh variable y_τ^ℓ that should never be instantiated, i.e. that should be treated as a constant: to represent this, we may do as Kohlhase in simple type theory [Koh95], and manage a database of instantiable variables, of non-instantiable variables, and of dependencies between them. (We may also dispense with the λ-headers in Gallier and Snyder's unification rules by introducing the third class of bound variables, as does Kohlhase.)

The most natural way to represent logical operators in λ_α^β is to make them constants of the language, i.e. we create negations $\neg^\ell : o \xrightarrow{\ell} o/\ell+1$, conjunctions $\wedge^\ell : o \xrightarrow{\ell} o \xrightarrow{\ell} o/\ell+1$ at all levels $\ell \in L$, and universal quantifiers $\Pi^{\ell'+1}_{(\tau \xrightarrow{\ell} o) \xrightarrow{\ell'} o}$ for all types τ and all levels $\ell, \ell' \in L$ with $\ell' \geq \ell + 1$. Then, we omit type and level annotations when they are not strictly necessary. Moreover, we write $F \wedge G$ instead of $\wedge F G$, we define $F \vee G$ as $\neg(\neg F \wedge \neg G)$, $F \Rightarrow G$ as $\neg F \vee G$. Finally, we write $\forall x_\tau^\ell \cdot F$ for $\Pi^{\ell'+1}_{(\tau \xrightarrow{\ell} o) \xrightarrow{\ell'} o} (\lambda y_\epsilon \cdot F)$, where $\epsilon = E(\tau)$ and ℓ' is such that $F[x_\tau^\ell / y] : o/\ell'$ is derivable.

But we face a serious problem, here, as this encoding cannot represent enough formulas. This is Theorem 8 on the vicious circle principle: in a formula $F \wedge G$, neither F nor G can contain any conjunctions, for example. We therefore need to relax the typing conditions on constant operators. We shall see what other constraints ramification puts on expressible formulas in Section 8.3.

8.2 Adding Operators

The most natural way to do this is to enrich the λ_α^β languages with *operators*, aking to the function symbols of first-order logic. Intuitively, an operator f would be such that $f(u_1, \ldots, u_m)$ would be at level $\max(\ell_1, \ldots, \ell_m)$ when u_i is at level ℓ_i for each $1 \leq i \leq m$. We can then encode the logical connectives and quantifiers as operators, and the problem above disappears.

More formally, we enrich the languages λ_α^β with operators f, each given with a unique *signature* $\tau_1 \times$

$$\frac{}{\Gamma, F \rhd F, \Delta}\,(Ax)$$

$$\frac{\Gamma \rhd F, \Delta}{\Gamma, \neg F \rhd \Delta}\,(\neg L) \qquad \frac{\Gamma, F \rhd \Delta}{\Gamma \rhd \neg F, \Delta}\,(\neg R)$$

$$\frac{\Gamma, F, G \rhd \Delta}{\Gamma, F \wedge G \rhd \Delta}\,(\wedge L) \qquad \frac{\Gamma \rhd F, \Delta \quad \Gamma \rhd G, \Delta}{\Gamma \rhd F \wedge G, \Delta}\,(\wedge R)$$

$$\frac{\Gamma, (F[t/x]) \downarrow_{\beta\eta} \rhd \Delta \quad t : \tau/\ell}{\Gamma, \forall x_\tau^\ell \cdot F \rhd \Delta}\,(\forall L) \qquad \frac{\Gamma \rhd F[y_\tau^\ell/x], \Delta \quad y \notin \mathrm{FVC}(\Gamma, \Delta)}{\Gamma \rhd \forall x_\tau^\ell \cdot F, \Delta}$$

Figure 10: Ramified deduction rules

$\ldots \tau_m \Rightarrow \tau$. The terms are either variables x_τ^ℓ, applications uv, abstractions $\lambda x_\epsilon \cdot u$ or *algebraic terms* $f(u_1, \ldots, u_m)$, where f is an operator. The typing rules are as in Figure 1, plus the additional rule:

$$\frac{u_1 : \tau_1/\ell_1 \quad \ldots \quad u_m : \tau_m/\ell_m \qquad f \text{ of signature } \tau_1 \times \ldots \times \tau_m \Rightarrow \tau}{f(u_1, \ldots, u_m) : \tau/\max(l(\tau), \ell_1, \ldots, \ell_m)}\,(Alg)$$

The adaptation of Huet's procedure to operators is straightforward: just consider an algebraic term $f(u_1, \ldots, u_m)$ as a simply-typed term $fu_1 \ldots u_m$, where f is now considered a constant. The theory then goes through, until Lemma 24, where we cannot guarantee any longer that partial bindings must produce fresh variables of strictly lower min-levels. And indeed:

Theorem 31 *Unification in the ramified theory of types with operators is undecidable. This holds even at the second order, with integer levels, with only one binary operator g of signature $\tau \times \tau \Rightarrow \tau$ and two constants a, b of type $\tau/0$.*

Proof: This is just Goldfarb's proof [Gol81]. Let τ be a base type, a and b be of type $\tau/0$. The operator g is used as a pairing operator; define $\overline{0}t = t$, $\overline{n+1}t = g(a, \overline{n}t)$ for every integer n, then the game is to reencode Hilbert's tenth problem as a unification problem. For this, we only need to produce unification problems of the form (0) $\langle \overline{1}(Fa), F(\overline{1}a)\rangle$, where F is a variable of type $\tau \stackrel{0}{\to} \tau/1$, (1) $\langle F_1(F_2a), F_3a\rangle$, where F_1, F_2, F_3 are variables of type $\tau \stackrel{0}{\to} \tau/1$, (2) $\langle d_1, e_1\rangle, \langle d_2, e_2\rangle$ where:

$$\begin{aligned} d_1 &= Gab(g(g(F_3a, F_2b), a)) \\ e_1 &= g(g(a, b), G(F_1a)(\overline{1}b)a) \\ d_2 &= Gba(g(g(F_3b, F_2a), a)) \\ e_2 &= g(g(b, a), G(F_1b)(\overline{1}a)a) \end{aligned}$$

F_1, F_2, F_3 are variables of type $\tau \stackrel{0}{\to} \tau/1$, and G is a variable of type $\tau \stackrel{0}{\to} \tau \stackrel{0}{\to} \tau \stackrel{0}{\to} \tau/1$.

Problems of type (0) encode the type of integers, in that any well-simply-typed unifier σ of such problems must map F to $\lambda x_\tau \cdot \overline{n}x$, for some $n \in \mathbb{N}$. But we have:

$$\frac{\begin{array}{c} y_\tau^0 : \tau/0 \\ \vdots \\ \overline{n}y_\tau^0 : \tau/0 \end{array}}{\lambda x_\tau \cdot \overline{n}x : \tau \stackrel{0}{\to} \tau/1}$$

so $\sigma|_{\{F\}}$ is well-typed (in the ramified sense).

Problems of type (1) encode addition, in that any unifier σ mapping F_i to $\lambda x_\tau \cdot \overline{n_i}x$, $1 \leq i \leq 3$, must be such that $n_1 + n_2 = n_3$. Because Goldfarb's encoding adds problems of type (0) for each of F_1, F_2, F_3, $\sigma|_{\{F_1, F_2, F_3\}}$ is well-typed.

And problems of type (2) encode multiplication, in that any unifier mapping F_i to $\lambda x_\tau \cdot \overline{n_i}x$, $1 \leq i \leq 3$, must be such that $n_1.n_2 = n_3$. Such unifiers then map G to $\lambda x_\tau \cdot \lambda y_\tau \cdot \lambda z_\tau \cdot g(t_0xy, g(t_1xy, \ldots, g(t_{n-1}xy, z)\ldots))$, where $t_ixy = g(\overline{i.n_1}x, \overline{i}y)$. This term has ramified type $\tau \stackrel{0}{\to} \tau \stackrel{0}{\to} \tau \stackrel{0}{\to} \tau/1$, and because Goldfarb's encoding adds problems of type (0) for each of F_1, F_2, F_3, $\sigma|_{\{F_1, F_2, F_3, G\}}$ is well-typed.

In conclusion, any simply-typed unifier of Goldfarb's problem is also a ramified well-typed unifier of the same problem. The converse is trivial. Therefore Goldfarb's problem also encodes Hilbert's tenth problem as a unification problem in ramified type theory with the operator g. $\Box$

Note that the role of operator g in Theorem 31 is played in logic, for instance, by conjunction $\wedge$ (of signature $o \times o \Rightarrow o$), so undecidability seems unescapable.

Of course, we still have a pre-unification procedure for this system: this is mostly the same algorithm as before, except that imitating on a term of the form $\lambda\overline{u_k} \cdot f(v_1, \ldots, v_i)v_{i+1} \ldots v_m$ produces fresh variables with identical, not lower min-levels.

8.3 Restrictions on Ramified Logics

The vicious circle principle (Theorem 8) is still valid in the impure case, i.e. with operators thrown in. The only thing that we have gained in impure ramified logics is that we are now allowed to nest operators, like in $f(f(x))$, but we still cannot apply a variable or a constant x to something where x occurs free.

This puts a very strong constraint on well-typed ramified λ-terms. On the one hand, this provides us yet another easy to check constraint for pruning branches in the search tree of our pre-unification algorithm: in rule **(Bind)**, whenever the computation of $(S[\lambda\overline{x_k} \cdot v/F]) \downarrow$ places a term where the constant or bound variable x occurs free as argument of x, we can stop right away on failure. Moreover, in rule **(Project)**, we can build more precise projection bindings by excluding the imitated variable y_i from the list of the arguments of the new free variables H_j, $1 \leq j \leq m$ (see Definition 3).

But the vicious circle principle also has drawbacks. For example, the only Church integers $\lambda f_{\epsilon\to\epsilon} \cdot \lambda x_\epsilon \cdot \underbrace{f(\ldots(f(}_{n \text{ times}} x)\ldots)$ that are ramified are 0 and 1, although all of them are simply-typed.

To correct this, the simplest solution is to define integers as an inductive datatype. I.e., we take a constant $0^0_{\mathbb{N}}$ (zero), and a successor operator $S : \mathbb{N} \Rightarrow \mathbb{N}$. Primitive recursions (at every higher order, thus representing the computational contents of the induction scheme of first-order Peano arithmetic, by Gödel's Dialectica interpretation [GLT89]) are encoded by means of a *recursor* R, which is an operator:

$$R : (\mathbb{N} \xrightarrow{0} \mathbb{N} \xrightarrow{0} \mathbb{N}) \times \mathbb{N} \times \mathbb{N} \Rightarrow \mathbb{N}$$

satisfying the equations:

$$\begin{array}{lll} R(f, z, 0) & = z & (1) \\ R(f, z, S(n)) & = f(S(n))(R(f, z, n)) & (2) \end{array}$$

for every f of the right type. Typing R with level 0 on the arrows is not a restriction, because by cumulativity every function $f : \mathbb{N} \xrightarrow{\ell_1} \mathbb{N} \xrightarrow{\ell_2} \mathbb{N}/\ell$ is also of type $\mathbb{N} \xrightarrow{0} \mathbb{N} \xrightarrow{0} \mathbb{N}/\ell$. The problem, here, is that by Theorem 8 again, the right-hand side $f(S(n))(R(f, z, n))$ of equation (2) cannot be typed in the ramified discipline. There is in fact no way that we can express the recursor R in its full generality. (The problem occurs already in the more mundane example of the function $\min : (\mathbb{N} \xrightarrow{0} o) \Rightarrow o$ returning the least element of some set in argument: we cannot express the fact that, provided X is a non-empty set, $\min(X)$ is in X, i.e. $X(\min(X))$.)

But we can still express recursion! Namely, we write the axiom:

$$\begin{array}{c} \forall f^2_{\mathbb{N} \xrightarrow{0} \mathbb{N} \xrightarrow{1} \mathbb{N}} \cdot \forall z^1_{\mathbb{N}} \cdot \exists g^1_{\mathbb{N} \xrightarrow{0} \mathbb{N}} \cdot g0 = z \wedge \\ \forall n^0_{\mathbb{N}} \cdot g(S(n)) = f(S(n))(gn) \end{array}$$

(Exercise: do the same with the min function.)

The levels that we have put in here are not the most general possible. We have chosen z of level 1 so that $g0$ and z have exactly the same types. But note that $g(Sn) : \mathbb{N}/1$ and $f(S(n))(gn) : \mathbb{N}/2$, and that in fact it is impossible to give both sides of the last equality the same sets of types. This is not a problem: $=$ should include the $=_{\beta\eta}$ relation, and we have already seen that two $\beta\eta$-convertible terms may have different sets of ramified types.

What this example reveals is the fact that *skolemisation* cannot be expressed in ramified logic. In the example above, the recursor R is (up to a few changes of levels) a Skolem constant for the existentially quantified variable g in the latter axiom. In general [Hue73], by skolemization we understand the process of replacing existential statements $\exists y \cdot F$ at the top-level of formulas by $F[g(x_1, \ldots, x_m)/y]$, where g is a new operator and $x_1, \ldots, x_m$ are the variables free in F other than y ($g(x_1, \ldots, x_m)$ is an abstraction of $\varepsilon(\lambda y \cdot F)$, where ε is Hilbert's choice function). The problem occurs in its full generality whenever some other variable x_j, $1 \leq j \leq m$, occurs free and in function position inside F, with some argument having y free. Namely, $F = \mathcal{C}[x_j u_1 \ldots u_n]$, where $\mathcal{C}$ is some context (a term with one hole, represented as pairs of square brackets), and y is free in some u_i, $1 \leq i \leq n$. Then skolemizing on y yields $\mathcal{C}[x_j \ldots u_i[g(x_1, \ldots, x_m)/y] \ldots]$, which cannot be typed by the vicious circle principle. In automated deduction, this is no great deal: skolemization is already unsound when not done at toplevel in higher-order logic, and the recommended technique is to manage a database of instantiable variables, of non-instantiable variables, and of dependencies between them [Koh95].

The usual solution [WR27] in ramified type theories is to use function-free formulations of logic. That is, we only have predicate constants and variables, no function constants, and no function variables. This is enough in theory to represent the whole of logic, because we can always replace n-ary functions by $n+1$-ary predicates plus an axiom saying that the predicate is functional (i.e., there is a unique y such that $P(x_1, \ldots, x_n, y)$, for every $x_1, \ldots, x_n$). Although this solves the problem, it does so at the expense of one of the most useful features of logic, namely non-variable terms.

Finally, we can get back the full power of ordinary higher-order logic by adding axioms of reducibility, of the form:

$$\forall x^{\ell}_{\tau} \cdot \exists y^{l(\theta)}_{\theta} \cdot x \equiv y$$

where τ is of the form $\tau_1 \xrightarrow{\ell_1} \ldots \tau_n \xrightarrow{\ell_n} \tau$, for each $1 \leq i \leq n$, $\theta_i = \tau(E(\tau_i))$ is the type τ_i redecorated with minimal levels, and similarly for $\theta = \tau(E(\tau))$; and $x \equiv y$ denotes $\forall {x_1}^{l(\theta_1)}_{\theta_1} \cdot \ldots \cdot \forall {x_n}^{l(\theta_n)}_{\theta_n} \cdot x x_1 \ldots x_n \Leftrightarrow y x_1 \ldots y_n$.

These axioms are however quite awkward to invoke in an automated theorem prover. If we wish to reason in full higher-order logics, it is actually simpler to just take the simply-typed calculus as foundation. (But see [Chu76] for an argument in favor of ramification even in the case of full higher-order logic.) We may also wish to reason in weaker systems, and this is where ramification comes into play, as we now argue.

9 Encoding Subsystems of Arithmetic

The Reverse Mathematics programme [Sim85] aimed at finding the weakest natural systems of logic that allow us to prove several important theorems of mathematics. It turned out that, although first-order arithmetic is not always enough to prove even combinatorial theorems, we usually do not need the full power of even second-order arithmetic to prove more involved theorems like the Bolzano-Weierstrass theorem or Kruskal's Lemma.

Let's consider ACA_0 first. This is a subsystem of second-order arithmetic constructed as follows. The language is built on a set of first-order variables x, y, z, … denoting naturals, and set variables X, Y, Z, … denoting sets of naturals. Terms $s, t, \ldots$ are built from first-order variables using the function symbols $+$ (addition, binary), $*$ (multiplication, binary), s (successor, unary), and the constant 0. Atomic formulas are of the form $s < t$, $s = t$, and $s \in X$, where X is a set variable. Formulas are built from atomic formulas using negation $\neg$, conjunction $\wedge$, first-order quantification $\forall_1 x \cdot$ and second-order quantification $\forall_2 X \cdot$. We use $\neq$, $\leq$, $\vee$, $\Rightarrow$, $\exists$, and so on, as derived notations. The semantics is defined in the obvious way. For the deduction system, we take for example the one of Figure 10.

The *basic arithmetical axioms* BAA are the universally quantified closures of:

$$\begin{array}{l} s(x) \neq 0 \\ s(x) = s(y) \Rightarrow x = y \\ x + 0 = x \\ x + s(y) = s(x+y) \\ x * 0 = 0 \\ x * s(y) = x * y + y \\ \neg x < 0 \\ x < s(y) \Rightarrow x < y \vee x = y \end{array}$$

and the *restricted induction axiom* RIA_0:

$$\begin{array}{l} \forall_2 X \cdot 0 \in X \wedge \\ \quad (\forall_1 x \cdot x \in X \Rightarrow s(x) \in X) \\ \Rightarrow \forall_1 x \cdot x \in X \end{array}$$

ACA_0 is the theory axiomatized by BAA, RIA_0, plus the *arithmetical comprehension scheme*, expressing that for every arithmetical formula F (i.e., formula without any second-order quantifier, but possibly with free first-order and set variables) and for every first-order variable x:

$$\exists_2 X \cdot \forall_1 x \cdot (x \in X \Leftrightarrow F)$$

holds.

We encode this in ramified type theory with operators by letting the operators, with their signatures, be:

$$\begin{array}{ll} s : \iota \Rightarrow \iota & \neg : o \Rightarrow o \\ + : \iota \times \iota \Rightarrow \iota & \wedge : o \times o \Rightarrow o \\ * : \iota \times \iota \Rightarrow \iota & \forall_1 : (\iota \xrightarrow{0} o) \Rightarrow o \\ = : \iota \times \iota \Rightarrow o & \\ < : \iota \times \iota \Rightarrow o & \end{array}$$

The constants are:

$$\begin{array}{l} 0 : \iota/0 \\ \forall^0_2 : ((\iota \xrightarrow{0} o) \xrightarrow{1} o) \xrightarrow{2} o/3 \end{array}$$

The variables are x^0_ι, y^0_ι, z^0_ι, … (all at level 0), $X^1_{\iota \xrightarrow{0} o}$, … (set variables, all at level 1).

The atomic formulas $s \in X$ are encoded as applications $X^1_{\iota \xrightarrow{0} o}\ s$, of type $o/1$. The arithmetical comprehension axiom is simply encoded as the use of λ-abstraction. Indeed, any arithmetical formula F has type $o/0$ or $o/1$, hence $\lambda x \cdot F$ has type $\iota \xrightarrow{0} o/1$. We can then deduce $\exists^0_2 X \cdot \forall_1 x \cdot X x \Leftrightarrow F$, because $[\lambda x \cdot F/X]$ is a well-typed substitution; conversely, since no set variable has level > 1, no instance of the comprehension axiom is provable in the ramified system unless F is arithmetical. BAA is coded in the obvious way, and RIA_0 as:

$$\forall^0_2 X_{\iota \to o} \cdot X 0 \wedge (\forall_1 x_\iota \cdot X x \Rightarrow X(s(x))) \Rightarrow \forall_1 x_\iota \cdot X x$$

ACA_0 is already strong enough to prove several non-trivial theorems, but we may long for stronger systems. Consider for example Σ^1_1-AC_0, which is ACA_0 plus the so-called *countable choice scheme for* Σ^1_1 *formulas* Σ^1_1-CCS, which is the universal closure of:

$$(\forall_1 n \cdot \exists_2 X \cdot \varphi(n, X)) \Rightarrow \exists_2 Y \cdot \forall_1 n \cdot \varphi(n, (Y)_n)$$

for every Σ^1_1-formula $\varphi(n, X)$ (i.e. a formula of the form $\exists_2 Z \cdot F$, where F is arithmetical), and where $(Y)_n$ denotes the definable operation of taking the nth element in Y viewed as a sequence of sets. (We use standard encodings of pairs of integers as integers, and of sequences as sets of pairs. Sequences of sets $(Y)_n$ are defined as $\{m \mid (m, n) \in Y\}$.) A more natural way to code this in ramified type theory is to replace the set variable Y by a function variable mapping integers to sets of integers. That is, we introduce variables of $\iota \xrightarrow{0} \iota \xrightarrow{0} o/1$, and write Σ^1_1-CCS as:

$$(\forall_1 n_\iota \cdot \exists^0_2 X_{\iota\to o} \cdot \varphi(n, X)) \Rightarrow \exists^0_2 Y_{\iota\to\iota\to o} \cdot \forall_1 n_\iota \cdot \varphi(n, Yn)$$

To get a stronger system still, we may consider Friedman's system ATR_0, which is ACA_0 plus a principle of definition by induction of arithmetical formulas along any well-founded ordering, called the *arithmetical transfinite recursion scheme* Δ^1_0-TRS. To define it, we let classically $LO(\prec)$ be a formula stating that $\prec$ is a linear ordering (again represented as a set of integers coding pairs of integers):

$$\begin{array}{l} \forall_1 i, j, k \cdot i \prec j \wedge j \prec k \Rightarrow i \prec k \\ \wedge \forall_1 i \cdot \neg i \prec i \\ \wedge \forall_1 i, j \cdot i = j \vee i \prec j \vee j \prec i \end{array}$$

where $i \prec j$ is an abbreviation for $(i, j) \in \prec$. Then $\prec$ is a well-founded ordering (written $WO(\prec)$) if and only if it is a linear ordering with finite descending chains, or equivalently on which well-founded induction is valid. That is, $WO(\prec)$ is defined as:

$$\begin{array}{l} LO(\prec) \wedge \\ \forall_2 Y \cdot (\forall_1 j \cdot (\forall_1 i \cdot i \prec j \Rightarrow i \in Y) \Rightarrow j \in Y) \\ \quad \Rightarrow \forall_1 j \cdot j \in Y \end{array}$$

(It is a Π^1_1 formula.) Then Δ^1_0-TRS is:

$$WO(\prec) \Rightarrow \exists_2 Y \cdot \forall_1 j, k \cdot j \in (Y)_k \Leftrightarrow \varphi(j, (Y)^k_\prec)$$

for every arithmetic formula φ, where Y is intended to denote a sequence of sets, and where $(Y)^k_\prec$ describes the subsequence of all $(Y)_i$, $i \prec k$. We can recode this scheme in ramified type theory as follows:

$$\begin{array}{l} LO(\prec^0_{\iota \xrightarrow{0} \iota \xrightarrow{0} o}) = \left\{ \begin{array}{l} \forall_1 i_\iota, j_\iota, k_\iota \cdot \prec ij \wedge \prec jk \Rightarrow \prec ik \\ \wedge \forall_1 i_\iota \cdot \neg(\prec ii) \\ \wedge \forall_1 i_\iota, j_\iota \cdot i = j \vee \prec ij \vee \prec ji \end{array} \right. \\ WO(\prec^0_{\iota \xrightarrow{0} \iota \xrightarrow{0} o}) = LO(\prec) \wedge \\ \quad \forall^0_2 Y_{\iota\to o} \cdot (\forall_1 j_\iota \cdot (\forall_1 i_\iota \cdot \prec ij \Rightarrow Yi) \Rightarrow Yj) \\ \quad\quad \Rightarrow \forall_1 j_\iota \cdot Yj \\ \Delta^1_0\text{-TRS}(\varphi) = \forall^0_2 \prec_{\iota\to\iota\to o} \cdot WO(\prec) \Rightarrow \\ \quad \exists^0_2 Y_{\iota\to\iota\to o} \cdot \forall_1 j_\iota, k_\iota \cdot Ykj \Leftrightarrow \\ \quad\quad \varphi(j, \lambda i_\iota \cdot \lambda n_\iota \cdot \prec ij \wedge Yin) \end{array}$$

ATR_0 is strong enough to prove the Bolzano-Weierstrass theorem and is equivalent in strength to Kruskal's Lemma. It is also equivalent to the system of iterated inductive definitions $\widehat{\mathrm{ID}}_{<\omega}$, although the former has much shorter proofs [Avi96].

It is tempting to abbreviate the rule schemes Σ^1_1-CCS and Δ^1_0-TRS as third-order axioms. This way, we only have to write down finitely many axioms. Σ^1_1-CCS is implemented as:

$$\begin{array}{l} \forall^0_3 \Phi_{\iota\to(\iota\to o)\to o} \cdot \\ \quad (\forall_1 n_\iota \cdot \exists^0_2 X_{\iota\to o} \cdot \Phi n X) \Rightarrow \\ \quad\quad \exists^0_2 Y_{\iota\to\iota\to o} \cdot \forall_1 n_\iota \cdot \Phi n (Yn) \end{array}$$

where the arithmetical formula $\varphi(n, X)$ has been subsumed by the application of a third-order variable Φ to n and X, quantified universally. This requires that we create a new third-order quantifier:

$$\forall^0_3 : ((\iota \xrightarrow{0} (\iota \xrightarrow{0} o) \xrightarrow{1} o) \xrightarrow{2} o) \xrightarrow{3} o/4$$

and corresponding third-order variables $\Phi^2_{\iota \xrightarrow{0} (\iota \xrightarrow{0} o) \xrightarrow{1} o}$. Similarly, we can implement Δ^1_0-TRS by:

$$\begin{array}{l} \forall^0_3 \Phi_{\iota\to(\iota\to\iota\to o)\to o} \cdot \\ \quad \forall^0_2 \prec_{\iota\to\iota\to o} \cdot WO(\prec) \Rightarrow \\ \quad \exists^0_2 Y_{\iota\to\iota\to o} \cdot \forall_1 j_\iota, k_\iota \cdot Ykj \Leftrightarrow \\ \quad\quad \Phi j (\lambda i_\iota \cdot \lambda n_\iota \cdot \prec ij \wedge Yin) \end{array}$$

where we create the new quantifier:

$$\forall^0_3 : ((\iota \xrightarrow{0} (\iota \xrightarrow{0} \iota \xrightarrow{0} o) \xrightarrow{1} o) \xrightarrow{2} o) \xrightarrow{3} o/4$$

and third-order variables $\Phi^2_{\iota \xrightarrow{0} (\iota \xrightarrow{0} \iota \xrightarrow{0} o) \xrightarrow{1} o}$. In this case, we can only instantiate Φ by terms whose normal forms are $\lambda j_\iota \cdot \lambda Z_{\iota\to\iota\to o} \cdot F(j, Z)$, where $F(j, Z)$ has type at most $o/2$ under the assumptions that $j : o/0$ and $Z : \iota \xrightarrow{0} \iota \xrightarrow{0} o/1$. It follows that $F(j, Z)$ is arithmetical again, because it would be at level at least 3 if it contained any second-order or third-order quantifier ... except that it might contain free third-order variables, i.e. a third-order theory is in general (much) more expressive than a second-order one. (The same thing happens with our third-order formulation of Σ^1_1-CCS.) It is in fact likely that the third-order formulations of Σ^1_1-CCS (hence of Σ^1_1-AC$_0$) and of Δ^1_0-TRS (hence of ATR$_0$) are strictly more expressive than their second-order (usual) formulations.

It is also tempting to try and use ramification to formalize subsystems of second-order arithmetic more powerful than ATR$_0$ still, like Π^1_1-CA$_0$ and in general Π^1_k-CA$_0$, $k \geq 1$. These systems are just like ACA$_0$, except that their comprehension axiom is now limited to Π^1_k formulas, i.e. formulas with all second-order quantifiers in front, where the number of alternations between $\exists_2$ and $\forall_2$ is at most k. Because all these languages can encode pairs of integers as integers, any finite sequence of $\forall_2$ can be replaced by just one, so we can restrict Π^1_k formulas to be of the form:

$$\forall_2 X_1 \cdot \exists_2 X_2 \cdot \ldots \cdot Q_k X_k \cdot F$$

where F is arithmetical and Q_k is $\forall_2$ if k is odd and $\exists_2$ otherwise (quantifiers alternate). Alternatively, they are of the form:

$$\forall_2 X_1 \cdot \neg\forall_2 X_2 \cdot \ldots \cdot \neg\forall^0_2 X_k \cdot F$$

One way of encoding this is to create quantifiers:

$$\forall^j_2 : ((\iota \xrightarrow{0} o) \xrightarrow{1} o) \xrightarrow{j+2} o / j+3$$

for every $0 \leq j \leq k$, and to encode Π^1_k formulas as:

$$\forall^{k-1}_2 X_1 \cdot \neg\forall^{k-2}_2 X_2 \cdot \ldots \cdot \neg\forall^0_2 X_k \cdot F$$

This is the best we can do (i.e., the lowest levels we can get), since the typing derivation is:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{F : o/1}{\lambda X_k \cdot F : (\iota \xrightarrow{0} o) \xrightarrow{1} o/2}}{\forall^0_2 X_k \cdot F : o/3}}{\neg\forall^0_2 X_k \cdot F : o/3}}{\lambda X_{k-1} \cdot \neg\forall^0_2 X_k \cdot F : (\iota \xrightarrow{0} o) \xrightarrow{1} o/3}}{\forall^1_2 X_{k-1} \cdot \neg\forall^0_2 X_k \cdot F : o/4}}{\neg\forall^1_2 X_{k-1} \cdot \neg\forall^0_2 X_k \cdot F : o/4}}{\vdots}}{\forall^{k-1}_2 X_1 \cdot \neg\forall^{k-2}_2 X_2 \cdot \ldots \cdot \neg\forall^0_2 X_k \cdot F : o/k+2}$$

We then need to allow for formulas of level up to $k+2$ in the restricted induction axiom, i.e. we replace RIA$_0$ by the following formula RIA$_k$:

$$\forall^k_2 X_{\iota \to o} \cdot X0 \wedge (\forall_1 x_\iota \cdot Xx \Rightarrow X(s(x))) \Rightarrow \forall_1 x_\iota \cdot Xx$$

The problem lies in the comprehension axiom. If we express it as the existence (at the meta-level) of an object X such that $Xx \Leftrightarrow G$ for every Π^1_k formula G and every first-order variable x, the comprehension axiom is trivially true: just take $\lambda x_\iota \cdot G$, as before. But we cannot prove in the logic just defined that $\exists_2 X \cdot \forall_1 x \cdot Xx \Leftrightarrow G$; indeed, this would require to instantiate X by $\lambda x_\iota \cdot G$, which has level $k+2$. But all our second-order variables have type $\iota \xrightarrow{0} o$ and level 1, and therefore no second-order variable can be instantiated to $\lambda x_\iota \cdot G$. The solution consisting in adding second-order variables of higher levels is not viable, since our Π^1_k-formulas G may contain any second-order variable as a free variable: if we add variables at level $k+2$, then F may be at level as high as $k+2$ instead of 1, hence the Π^1_k formulas may have levels as high as $2k+2$, so we need variables at level $2k+2$ to prove the comprehension axiom, etc. Adding second-order variables at all levels then annuls the restriction of the comprehension axiom to Π^1_k formulas, and we end up having full second-order Peano arithmetic under a new disguise. On the other hand, not adding variables at higher levels produces a look-alike of Π^1_k-CA$_0$, which might be interesting in its own right.

Mechanising proof search in systems like ACA$_0$, Σ^1_1-AC$_0$ or ATR$_0$ is not the purpose of this paper. We only mention the main difficulties in doing so. The first difficult point is the failure of the subformula property: this is already a problem in the simple theory of types, but it already plagues usual formalisations of first-order Peano arithmetic. In fact, to do any serious mathematics probably involves tackling this difficult problem.

The second difficulty is the fact that, although Π^1_1-CA$_0$ is strong enough to prove most theorems of everyday mathematics, it does so in quite contrived ways: we must encode pairs of integers as integers, sequences of integers as integers, even inner models of the theory inside the theory. This difficulty is probably only apparent. Nothing prevents us indeed from formalising a richer theory, with an explicit pairing operator:

$$(\_, \_) : \iota \times \iota \Rightarrow \iota$$

In fact, most any inductive datatype that crops up in computer science can be endowed with a theory *à la* ACA$_0$ or Π^1_k-CA$_0$. For example, to encode lists of integers, we add the following operator:

$$cons : \iota \times \iota\ \mathsf{list} \Rightarrow \iota\ \mathsf{list}$$

and the constant $nil : \iota\ \mathsf{list}/0$, where ι list is a new base type. We also add an induction axiom on lists, restricted in the same way that RIA_k was restricted:

$$\begin{array}{l}\forall_2^k X^{k+1}_{\iota\ \mathsf{list} \xrightarrow{k} o} \cdot X\ nil \wedge \\ \quad (\forall_1^k x^k_{\iota\ \mathsf{list}}, y^k_\iota \cdot X x \Rightarrow X(cons(y, x))) \\ \Rightarrow \forall_1^k x^k_{\iota\ \mathsf{list}} \cdot X x\end{array}$$

The shape of the induction axioms is entirely determined from the definition of the datatype, here $\iota\ \mathsf{list} ::= nil \mid cons(\iota, \iota\ \mathsf{list})$: see [GLT89]. Moreover, considering $\iota\ \mathsf{list}$ not as a new base type but as the application of a type operator list to the base type ι, thus allowing other list types and a limited form of polymorphism, is a benign extension to the type system.

We therefore believe that ramified type theory with operators is a sensible starting point for formalizing powerful enough systems of mathematics, in an automated deduction perspective.

10 Conclusion

We have proposed a formalization of the pure ramified theory of types through a typed λ-calculus that is simple, rigorous, and arguedly in the spirit of usual ramified theories with cumulative levels. We have shown that unification, or rather pre-unification, in such pure theories with integer levels was decidable. However, the logical systems for which they can provide foundations are too weak to express any non-trivial logical facts. Extending the frameworks with operators makes the unification problem undecidable, already at order 2, but the decidability result above should be taken as an indication that ramified type theory with operators is a computationally more sensible basis for automated deduction than simple type theory. It is all the more sensible as most theorems of everyday mathematics can be proved in theories that are formalizable in such ramified systems.

Acknowledgements

Thanks to Jaco van de Pol, Twan Laan, Bob Riemenschneider, Michael Zeleny, Alain Deutsch, Gilles Dowek, and to Peter H. Schmitt and all the other people at the Universität Karlsruhe for offering me a pleasant work environment, as well as their friendship.

References

[And86] Peter B. Andrews. *An Introduction to Mathematical Logic and Type Theory: To Truth through Proof.* Computer Science and Applied Mathematics. Academic Press, 1986.

[Avi96] Jeremy Avigad. On the relationship between ATR_0 and $\widehat{\mathrm{ID}}_{<\omega}$. *Journal of Symbolic Logic*, 61(3):768–779, September 1996.

[Bar84] Henk Barendregt. *The Lambda Calculus, Its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundations of Mathematics.* North-Holland Publishing Company, Amsterdam, 1984.

[Chu40] Alonzo Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.

[Chu76] Alonzo Church. Comparison of Russell's resolution of the semantical antinomies with that of Tarski. *Journal of Symbolic Logic*, 41(4):747–760, December 1976.

[Cop71] Irving M. Copi. *The Theory of Logical Types.* Monographs in modern logic series. London, Routledge and Kegan Paul, 1971.

[CP96] Iliano Cervesato and Frank Pfenning. Linear higher-order preunification. In *CADE-13 Workshop on Proof Search in Type-Theoretic Languages*, 1996. Available at `http://foxnet.cs.cmu.edu/people/fp/papers/`.

[Der87] Nachum Dershowitz. Termination of rewriting. *Journal of Symbolic Computation*, 3:69–116, 1987.

[Dow95] Gilles Dowek. Lambda-calculus, combinators and the comprehension scheme. In *Typed Lambda Calculi and Applications (TLCA'95)*, pages 154–170, 1995. Full version in Rapport de Recherche 2565, Inria.

[Ell89] Conal M. Elliott. Higher-order unification with dependent function types. In N. Dershowitz, editor, *Proceedings of the 3rd International Conference on Rewriting Techniques and Applications*, pages 121–134. Springer Verlag LNCS 355, 1989.

[Far88] W.M. Farmer. A unification algorithm for second-order monadic terms. *Annals of Pure and Applied Logics*, 39:131–174, 1988.

[Fef64] Solomon Feferman. Systems of predicative analysis. *Journal of Symbolic Logic*, 29:1–30, 1964. Reprinted in J. Hintikka (ed.), The Philosophy of Mathematics, Oxford University Press, London, 1969, pp. 95–127.

[Fef68a] Solomon Feferman. Autonomous transfinite progressions and the extent of predicative mathematics. In B. van Rootselaar and J.F. Staal, editors, *Logic, Methodology and Philosophy of Science*, volume III, Amsterdam, 1968. North Holland.

[Fef68b] Solomon Feferman. Systems of predicative analysis II: Representation of ordinals. *Journal of Symbolic Logic*, 33(2):193–220, June 1968.

[Fef75] Solomon Feferman. Systems of predicative analysis. In *Algebra and Logic*, pages 87–139, Berlin, 1975. Springer LNM 450.

[Fef78] Solomon Feferman. A more perspicuous formal system for predicativity. In K. Lorenz, editor, *Konstruktionen versus Positionen*, pages 68–93, Berlin, 1978. Walter de Gruyter.

[Fit90] Melvin C. Fitting. *First-Order Logic and Automated Theorem Proving.* Springer, 1990. Second edition, 1996.

[Gal91] Jean Gallier. What's so special about Kruskal's Theorem and the ordinal Γ_0. A survey of some results in proof theory. *Annals of Pure and Applied Logic*, 53(3):199–260, September 1991.

[GLT89] Jean-Yves Girard, Yves Lafont, and Paul Taylor. *Proofs and Types*, volume 7 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1989.

[Gol81] Warren D. Goldfarb. The undecidability of the second-order unification problem. *Theoretical Computer Science*, 13:225–230, 1981.

[Haz83] Allen Hazen. Predicative logics. In D. Gabbay and F. Guenther, editors, *Handbook of Philosophical Logic I: Elements of Classical Logic*, chapter I.5, pages 331–407. D. Reidel Publishing Company, Dordrecht, The Netherlands, 1983. (Synthese library Volume 164).

[Hin69] J. R. Hindley. The principal type scheme of an object in combinatory logic. *Transations of the American Mathematical Society*, 146:29–60, 1969.

[Hue73] Gérard P. Huet. A mechanization of type theory. In *Proceedings of the 3rd International Joint Conference on Artificial Intelligence*, pages 139–146, Stanford University, Stanford, California, August 1973.

[Hue75] Gérard P. Huet. A unification algorithm for typed λ-calculus. *Theoretical Computer Science*, 1:27–57, 1975.

[KL96a] Fairouz Kamareddine and Twan Laan. A correspondence between Nuprl and the ramified theory of types. Technical report, Eindhoven Univerity of Technology, 1996. 12 pages. Available at `http://www.win.tue.nl/pub/techreports/laan/csn96-yy.ps.gz`.

[KL96b] Fairouz Kamareddine and Twan Laan. A reflection on Russell's ramified types and Kripke's hierarchy of truths. *Journal of the Interest Group in Pure and Applied Logic*, 4(2), 1996. Available at `http://www.win.tue.nl/pub/techreports/laan/csn95-18.ps.gz`.

[Koh95] Michael Kohlhase. Higher-order tableaux. In *Workshop on Theorem Proving with Analytic Tableaux and Related Methods*, 1995.

[Kri75] Saül Kripke. Outline of a theory of truth. *Journal of Philosophy*, 72:690–716, 1975.

[Laa95] Twan Laan. A formalization of the ramified type theory. Technical Report 94/33, Eindhoven Univerity of Technology, 1995. 40 pages. Available at `http://www.win.tue.nl/pub/techreports/laan/csn94-33.ps.gz`.

[Laa96] Twan Laan. A modern elaboration of the Ramified Theory of Types. *Studia Logica*, 1996. To appear.

[LN95] Twan Laan and Rob Nederpelt. A formalization of the ramified type theory. 26 pages. Available from the author at `laan@win.tue.ml`, 1995.

[McH90] James A. McHugh. *Algorithmic Graph Theory.* Prentice-Hall International, 1990.

[Mil78] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

[Mil91] Dale Miller. A logic programming language with lambda-abstraction, function variables, and simple unification. *Journal of Logic and Computation*, 1(4):497–536, 1991.

[Nip90] Tobias Nipkow. Higher-order unification, polymorphism, and subsorts (extended abstract). In S. Kaplan and M. Okada, editors, *Proceedings of the 2nd International Workshop on Conditional and Typed Rewriting Systems*, pages 436–447. Springer Verlag LNCS 516, 1990.

[Pre94] Christian Prehofer. Decidable higher-order unification problems. In *Proceedings of the 12th International Conference on Automated Deduction*, pages 635–649. Springer Verlag LNAI 814, 1994.

[SG89] W. Snyder and J. Gallier. Higher order unification revisited: Complete sets of tranformations. *Journal of Symbolic Computation*, 8(1 & 2):101–140, 1989. Special issue on unification. Part two.

[Sim85] Stephen G. Simpson. Reverse mathematics. In A. Nerode and R. A. Shore, editors, *Recursion Theory*, pages 461–471. American Mathematical Society, 1985. Proceedings of Symposia in Pure Mathematics, vol. 42.

[SS94] Manfred Schmidt-Schauß. Unification of stratified second-order terms. Interner Bericht 12/94, J.W. Goethe Universität Frankfurt, Fachbereich Informatik, 1994. Available on `http://kiste-J.ki.informatik.uni-frankfurt.de/papers/D-uni-SO-9-95.ps`.

[WR27] Alfred North Whitehead and Bertrand Russell. *Principia Mathematica*. Cambridge University Press, 1910, 1927.