Type Checking, Universe Polymorphism, and Typical Ambiguity in the Calculus of Constructions*

DRAFT

Robert Harper† Robert Pollack‡

Abstract

The Generalized Calculus of Constructions (CC^ω) of Coquand and Huet is a system for formalizing constructive mathematics. CC^ω includes a cumulative hierarchy of universes, with each universe closed under the type forming operations. Universe hierarchies are tedious to use in practice. Russell and Whitehead introduced a convention for dealing with stratification, called "typical ambiguity", in which universe levels are not explicitly mentioned, but it is tactily asserted that some correctly stratified level assignment exists. Using an "operational semantics" for type synthesis, we study type checking and typical ambiguity for CC^ω. We show type synthesis is effective in CC^ω. Even if explicit universe levels are erased from a term it is possible to compute a "schematic type" for that term, and a set of constraints, that characterize all types of all well-typed instances of the term. We also consider the extension with δ-reductions, which introduces a form of "universe polymorphism" induced by the failure of type unicity in CC^ω.

1 Introduction

The Calculus of Constructions (CC) was introduced by Coquand and Huet [CH85, Coq85] as a system for formalizing constructive mathematics. The system may be viewed as the λ-calculus associated with natural deduction proofs in an extension of Church's higher-order logic [Chu40]. It's proof-theoretic strength, measured by the set of number-theoretic functions representable in the theory, is enormous, encompassing at least higher-order arithmetic. The system has been proved both proof-theoretically [Coq85] and model-theoretically [Luo88a, Erh88, HP88] consistent, and the type checking problem has been proved decidable [Coq85, CH85].

In the course of formalizing a body of mathematics, it is often necessary to consider structures such as algebras (groups, rings, *etc.*), automata, and ordered sets. It is by now widely recognized [Mar82, Con86, Mac86, MH88] that the appropriate type-theoretic representation of mathematical structures is as elements of "strong sum" types[1] introduced by Martin-Löf [Mar73, Mar82, Mar84].

Strong sums are incompatible with impredicativity [Coq86, HH86, MH88]: one may not introduce a strong sum type at the level of propositions in CC. In response to this, Coquand introduced the "generalized" Calculus of Constructions [Coq86] (CC^ω) which includes a cumulative hierarchy

*Work done at the Laboratory for Foundations of Computer Science, University of Edinburgh; supported by the U.K. Science and Engineering Research Council, GR/D 64612 and GR/F 78487

†Dept. of Computer Science, Carnegie-Mellon University

‡Laboratory for Foundations of Computer Science, University of Edinburgh

[1] Also known in the literature as "dependent products" and "generalized sums." These are not to be confused with the "weak sums" (or "existential types") introduced in connection with data abstraction [MP85].

of *universes*. A universe is a type that is closed under the type-forming operations of the calculus, the formation of products indexed by any type, and the formation of strong sums indexed by a type of that universe level. Cumulative hierarchies of this kind are endemic to predicative systems; they arise in various guises in *Principia Mathematica* [Rus08, WR25] and in many contemporary type theories [Mar73, Mar82, Mar84, CZ82, CZ84, Con86].

Universe hierarchies are tedious to use in practice. Many workers have attempted to avoid the complications of such a hierarchy by assuming that there is a type of all types [Mar, BL84, MR86, Car86]. Whether or not this is advisable in the context of programming languages is the subject of current research [Car]. Such a choice is known, however, to be incompatible with the propositions-as-types principle [Mar73, MR86, How87], and hence is an unsuitable choice for CC^ω.

An alternative approach to dealing with stratification in formal systems was introduced by Russell and Whitehead in *Principia Mathematica*. They observed that in most situations it is not the exact universe level that matters, but only the relationships between the universe levels occurring within a given proof. They introduced an informal convention, called "typical ambiguity," in which universe levels are not explicitly mentioned, and in which it is tacitly asserted that there exists an assignment of levels such that the resulting proof is correct with respect to the predicativity requirements of the logic of *Principia Mathematica*.

From the modern perspective, typical ambiguity can be described as a way to achieve the flexibility of having a type of all types without sacrificing the logical consistency of the theory. At the level of the concrete syntax, the user can work without explicit mention of universe levels, leaving it to the proof checker to ensure that there is always a choice of levels that yields a type-correct term in the underlying calculus with explicitly stratified universes. That this can be done in CC^ω was suggested by Huet, and worked out independently in [Hue87].

The purpose of this paper is to study type checking and typical ambiguity in the context of CC^ω. In Section 2 we define the system CC^ω, and state some of its important properties. Following [HMT87, HMT88], we introduce in Section 3 an "operational semantics" for type checking. The operational semantics provides a normal form for typing derivations that is useful for establishing properties of typing in CC^ω. (Similar methods are used in [Mit84, GdR88]). In Section 4 we consider the type checking problem for CC^ω. The problem is reduced to the computation of a form of schematic type involving constraints on universe levels. The algorithm is presented in the style of the operational semantics, following [CDDK86]. In Section 5 we consider δ-reductions and LET. Since type unicity fails for CC^ω, definitions induce a form of polymorphism, called "universe polymorphism," into the system. The operational semantics and basic type synthesis algorithm are extended to account for this. In Section 6 we consider the problem of typical ambiguity. This involves defining two algorithms, one for conversion, the other for type synthesis, that keep track of the internal constraints between universe levels induced by the predicativity requirements of CC^ω. The problem is reduced to checking satisfiability of a finite set of inequalities over the natural numbers. An algorithm based on Chan's ARITH algorithm [Cha77] is sketched. Finally, the algorithm for type synthesis with typical ambiguity is extended to handle δ-reductions.

2 The Generalized Calculus of Constructions

The Generalized Calculus of Constructions [Coq86] (CC^ω) is obtained by extending the basic Calculus of Constructions with a full cumulative hierarchy of type universes. Let x, y, z range over some countable set of variables, and i, j, k range over the natural numbers. We use syntax given by the following grammar:

$$\begin{array}{lcll} \kappa & ::= & \mathsf{Prop} \mid \mathsf{Type}_i & \textit{kinds} \\ M & ::= & x \mid \kappa \mid [x{:}M]M \mid \{x{:}M\}M \mid M\,M & \textit{terms} \\ \Gamma & ::= & \langle\rangle \mid \Gamma[x{:}M] & \textit{contexts} \end{array}$$

The metavariables A, B, C, K, L, M, and N range over terms; κ and ι range over kinds. The terms Type_i are called *universes*. The pair $x{:}M$ in a context Γ is a *declaration*. We only consider contexts in

$$\langle\rangle \ \textit{valid} \qquad \frac{\Gamma \vdash A : \kappa \quad x \notin \mathrm{dom}(\Gamma)}{\Gamma[x{:}A] \ \textit{valid}}$$

$$\frac{\Gamma \ \textit{valid}}{\Gamma \vdash \mathsf{Prop} : \mathsf{Type}_0} \qquad \frac{\Gamma \ \textit{valid}}{\Gamma \vdash \mathsf{Type}_i : \mathsf{Type}_{i+1}} \qquad \frac{\Gamma \ \textit{valid} \quad x{:}A \in \Gamma}{\Gamma \vdash x : A}$$

$$\frac{\Gamma[x{:}A] \vdash B : \mathsf{Prop}}{\Gamma \vdash \{x{:}A\}B : \mathsf{Prop}}$$

$$\frac{\Gamma \vdash A : \mathsf{Prop} \quad \Gamma[x{:}A] \vdash B : \mathsf{Type}_i}{\Gamma \vdash \{x{:}A\}B : \mathsf{Type}_i} \qquad \frac{\Gamma \vdash A : \mathsf{Type}_i \quad \Gamma[x{:}A] \vdash B : \mathsf{Type}_i}{\Gamma \vdash \{x{:}A\}B : \mathsf{Type}_i}$$

$$\frac{\Gamma[x{:}A] \vdash M : B}{\Gamma \vdash [x{:}A]M : \{x{:}A\}B} \qquad \frac{\Gamma \vdash M : \{x : A\}B \quad \Gamma \vdash N : A}{\Gamma \vdash MN : [N/x]B}$$

$$\frac{\Gamma \vdash M : A \quad \Gamma \vdash B : \kappa \quad A \simeq B}{\Gamma \vdash M : B} \qquad \text{(B-CONV)}$$

$$\frac{\Gamma \vdash M : \mathsf{Type}_i}{\Gamma \vdash M : \mathsf{Type}_{i+1}} \qquad \text{(B-CUM)}$$

Table 1: Typing Rules for CC^ω

which no variable is declared more than once.

The relation $\rightarrow$ is one-step β-reduction, defined as usual. Reduction, the reflexive, transitive closure of $\rightarrow$, is denoted by $\twoheadrightarrow$, and conversion, the induced equivalence relation, is denoted by $\simeq$. One-step head and weak head reduction are denoted by $\xrightarrow{h}$ and $\xrightarrow{wh}$, and their reflexive, transitive closures are denoted by $\overset{h}{\twoheadrightarrow}$ and $\overset{wh}{\twoheadrightarrow}$. Reduction satisfies the Church-Rosser property [vD80, Coq85, Luo88b], so $M \simeq N$ iff $M \twoheadrightarrow P$ and $N \twoheadrightarrow P$ for some P. We do not consider η reduction in this paper[2].

CC^ω is a formal system for deriving assertions of the form $\Gamma \vdash M : A$. The axioms and rules of derivation for CC^ω are given in Table 1. We tend to write $\Gamma \vdash M : A$ to mean that the indicated assertion is derivable in the formal system.

All well-typed terms in CC^ω are strongly normalizing, and CC^ω has the subject reduction property. It is an immediate consequence of the normalization theorem and the Church-Rosser theorem that the conversion relation is decidable for well-typed terms.

3 Operational Semantics for CC^ω

The inference rules defining the typing relation for CC^ω are not completely syntax-directed: the structure of M does not determine when the rules B-CONV and B-CUM may be applied. In order to define a type checking algorithm, it is helpful to characterize exactly those points in a derivation at which these rules may be needed. We shall define an "operational semantics" for typing in CC^ω that

[2]For one thing, subject reduction fails for CC^ω with η, and with it our proof of soundness for the operational semantics in Section 3. Also, untyped terms fail to have the Church-Rosser property [vD80], and we don't know if well typed terms have this property, or the normalization property.

$$\mathsf{cum}(A,i) := \begin{cases} \mathsf{Type}_{j+i} & \textit{if } A \overset{wh}{\twoheadrightarrow} \mathsf{Type}_j \\ A & \textit{otherwise} \end{cases}$$

$$\kappa_1 \uparrow_i \kappa_2 := \begin{cases} \mathsf{Prop} & \textit{if } \kappa_2 = \mathsf{Prop} \\ \mathsf{Type}_{j+i} & \textit{if } \kappa_1 = \mathsf{Prop}, \kappa_2 = \mathsf{Type}_j \\ \mathsf{Type}_{\max(j,k)+i} & \textit{if } \kappa_1 = \mathsf{Type}_j, \kappa_2 = \mathsf{Type}_k \end{cases}$$

$$\Gamma \vdash \mathsf{Prop} \Rightarrow \mathsf{Type}_i \quad (i \geq 0) \tag{S-PROP}$$

$$\Gamma \vdash \mathsf{Type}_j \Rightarrow \mathsf{Type}_i \quad (i > j \tag{S-TYPE}$$

$$\Gamma \vdash x \Rightarrow \mathsf{cum}(A,i) \quad (x{:}A \in \Gamma,\ i \geq 0) \tag{S-VAR}$$

$$\frac{\Gamma \vdash A \Rightarrow K \quad K \overset{wh}{\twoheadrightarrow} \kappa_1 \quad \Gamma[x{:}A] \vdash B \Rightarrow L \quad L \overset{wh}{\twoheadrightarrow} \kappa_2 \quad i \geq 0}{\Gamma \vdash \{x{:}A\}B \Rightarrow \kappa_1 \uparrow_i \kappa_2} \ (x \notin \mathrm{dom}(\Gamma)) \tag{S-GEN}$$

$$\frac{\Gamma \vdash A \Rightarrow K \quad K \overset{wh}{\twoheadrightarrow} \kappa \quad \Gamma[x{:}A] \vdash M \Rightarrow B}{\Gamma \vdash [x{:}A]M \Rightarrow \{x{:}A\}B} \ (x \notin \mathrm{dom}(\Gamma)) \tag{S-ABS}$$

$$\frac{\Gamma \vdash M \Rightarrow A \quad A \overset{wh}{\twoheadrightarrow} \{x : A_1\}A_2 \quad \Gamma \vdash N \Rightarrow B \quad B \simeq A_1 \quad i \geq 0}{\Gamma \vdash MN \Rightarrow \mathsf{cum}([N/x]A_2, i)} \tag{S-APP}$$

Table 2: Operational Semantics for CC^ω

has the property that at most one rule applies to any given term. The operational semantics is a formal system for deriving assertions of the form $\Gamma \vdash M \Rightarrow A$. It is defined by the axioms and rules of derivation given in Table 2. The intended meaning of $\Gamma \vdash M \Rightarrow A$ is that A is a type for M in context Γ.

The rules S-VAR and S-APP use an auxiliary function, cum, to check whether or not the type ascribed in the conclusion is convertible to a universe, and if so, to account for any potential use of cumulativity at that point. To illustrate, let $\Gamma = [x{:}\{f{:}\mathsf{Type}_0 \to \mathsf{Type}_1\}(f\ \mathsf{Prop})]$, $M = [y{:}\mathsf{Type}_0]\mathsf{Type}_0$, and observe that $\Gamma \vdash x\,M : \mathsf{Type}_1$ since $M\ \mathsf{Prop} \simeq \mathsf{Type}_0$, and cumulativity applies. The rule S-GEN uses a similar auxiliary function $\uparrow$ to encode both the closure rules for products in CC$^\omega$ and possible uses of cumulativity.

Note that the operational semantics does not include rules for checking the validity of the context. It does, however, preserve context validity by ensuring that every extension to the context is by a well-formed binding. The operational semantics is closer than the basic system to a practical implementation since it avoids the redundancy associated with repeatedly verifying the validity of the context.

The relation between the operational semantics and CC$^\omega$ is made precise by the following theorem:

Theorem 3.1

Soundness *If* Γ *valid and* $\Gamma \vdash M \Rightarrow A$, *then* $\Gamma \vdash M : A$.

Completeness *If* $\Gamma \vdash M : A$, *then there exists* B *such that* $B \simeq A$ *and* $\Gamma \vdash M \Rightarrow B$. *Furthermore, if* $A \simeq \mathsf{Type}_i$, *then* $\Gamma \vdash M \Rightarrow \mathsf{Type}_j$, *for all* $j \geq i$.

The operational semantics is "syntax directed" in the sense that the structure of a derivation of $\Gamma \vdash M \Rightarrow A$ is determined by the structure of M. However, the relation $\Gamma \vdash M \Rightarrow A$ is not a partial function of Γ and M; for example $\Gamma \vdash [x{:}\mathsf{Prop}]\mathsf{Prop} \Rightarrow \{x{:}\mathsf{Prop}\}\mathsf{Type}_i$ is derivable for every $i \geq 0$. The only source of indeterminacy in the operational semantics is in the choice of the parameters i governing universe levels. Let us call the occurrences of these parameters in a derivation "springs" and call the value of such a parameter a "setting" of that spring. In a derivation certain spring settings are forced by context, while others are arbitrary. For example, in the term $([x{:}\mathsf{Type}_2]x)\ \mathsf{Prop}$, the spring setting corresponding to the occurrence of Prop is forced to be 2, whereas in the term Prop, the spring setting is arbitrary. Since the operational semantics is syntax-directed, the set of types derivable for Γ and M is determined entirely by the settings of the unforced springs.

The following lemma provides a useful characterization of the variability in the types of a term derivable in the operational semantics:

Lemma 3.2

1. *If* $\Gamma \vdash M \Rightarrow A$ *and* $\Gamma \vdash M \Rightarrow B$ *with* $A \neq B$, *then for some* $A_1, \ldots, A_n$ $(n \geq 0)$ *and some* $i \geq 0$,

$$\Gamma \vdash M \Rightarrow A \quad \textit{iff} \quad A = \{x_1{:}A_1\} \cdots \{x_n{:}A_n\}\mathsf{Type}_j\ (j \geq i).$$

2. *If* $\Gamma \vdash x \Rightarrow A$ *and* $\Gamma \vdash x \Rightarrow B$ *with* $A \neq B$, *then* $A = \mathsf{Type}_i$ *and* $B = \mathsf{Type}_j$.

Although there may be many unforced springs in a derivation (consider the term $[x{:}\mathsf{Prop}]\mathsf{Prop}$, which has two unforced springs), the lemma shows that the indeterminacy so induced is of a very limited kind. In fact, the result shows that at most one unforced spring matters; the rest may be safely ignored.

4 Type Checking

In this section we consider three problems related to type checking in CC$^\omega$. The *type checking problem* (TCP) is to decide, given a valid context Γ, and terms M and A, whether or not $\Gamma \vdash M : A$. The

well-typedness problem (WTP) is to decide, given a valid context Γ and term M, whether or not the set $\mathrm{Types}_\Gamma(M) := \{ A \mid \Gamma \vdash M : A \}$ is empty. The *type synthesis problem* (TSP) is to compute the characteristic function of $\mathrm{Types}_\Gamma(M)$. We state these problems for valid contexts since, in practice, we shall maintain the validity of the context and do not expect to check this property on each use. Since TCP is reducible to TSP, we restrict our attention to the latter two problems. The main result of this section is the decidability of TSP and WTP.

Our strategy for solving these problems is to work with the operational semantics; for by the soundness and completeness theorems, M is well-typed in a valid context Γ iff there exists an A such that $\Gamma \vdash M \Rightarrow A$, and A is a valid type for M in valid context Γ iff A is convertible to some A' such that $\Gamma \vdash M \Rightarrow A'$. We noted in the previous section that all possible derivations are structurally isomorphic, being determined by the syntax of M. To solve WTP, then, we need only decide whether there is an appropriate setting of the springs so that some such derivation may be constructed. To solve TSP, we must characterize all admissible spring settings. We adapt the idea of Damas and Milner [DM82] and introduce an analog of their type schemes to serve as a "local summary" of the variability in the settings of the springs, using a simple form of constrained matching to infer forced settings. Both for this, and later, purposes, we introduce the machinery of schematic terms.

Let α, β, and γ range over some set of *level variables*, and let λ and μ range over the *level expressions*, consisting of level variables and natural numbers. The *schematic terms*, ranged over by X, Y, and Z, are terms that may involve *universe schemes* of the form Type_α. The set $\mathrm{LV}(X)$ is the set of level variables occurring in X, and X is *level closed* if this set is empty. Universe schemes are regarded as kinds; we use κ to range over this extended set of kinds.

In general a schematic term stands for the set of all of its instances obtained by substituting natural numbers for level variables. A *level assignment* is a finite function mapping level variables to natural numbers; σ and τ range over level assignments. We write σX for the instance of X obtained by replacing all occurrences of α by $\sigma(\alpha)$. (The result need not be level closed.) Level assignments are explicitly indicated by writing $[i_1/\alpha_1, \ldots, i_k/\alpha_k]$.

Since reduction is defined without regard to typability, the reduction relations immediately extend to schematic terms. Moreover, the presence of level variables does not effect reduction. For example:

Lemma 4.1

1. *If* $Z \overset{wh}{\twoheadrightarrow} \{x : Z_1\}Z_2$, *then* $\sigma Z \overset{wh}{\twoheadrightarrow} \{x : \sigma Z_1\}\sigma Z_2$.

2. *If* $\sigma Z \overset{wh}{\twoheadrightarrow} \{x : A_1\}A_2$, *then* $Z \overset{wh}{\twoheadrightarrow} \{x : Z_1\}Z_2$, *with* $\sigma Z_1 = A_1$ *and* $\sigma Z_2 = A_2$.

Thus if a schematic term has *any* well-typed instance, it is normalizable.

Usually we are interested only in certain instances of a schematic term X. A *constraint set* is a finite set of inequalities of the form $\lambda \geq \mu$ or $\lambda > \mu$. The variables $\mathcal{C}$, $\mathcal{D}$, $\mathcal{E}$, and $\mathcal{F}$ range over constraint sets. The set $\mathrm{LV}(\mathcal{C})$ is the set of level variables occurring in $\mathcal{C}$. A level assignment σ *satisfies* a constraint set $\mathcal{C}$, written $\sigma \models \mathcal{C}$, iff $\mathrm{dom}(\sigma) \subseteq \mathrm{LV}(\mathcal{C})$ and each of the inequalities in $\mathcal{C}$ is true under the assignment σ. A *constrained term* is a pair $(X, \mathcal{C})$ where $\mathrm{LV}(X) \subseteq \mathrm{LV}(\mathcal{C})$. (We may always extend $\mathcal{C}$ by inequalities of the form $\alpha \geq 0$ to satisfy this condition.) A term A is an *instance* of $(X, \mathcal{C})$, written $(X, \mathcal{C}) \geq A$, iff there exists $\sigma \models \mathcal{C}$ such that $\sigma X = A$.

Returning to the type synthesis problem, note that by Lemma 3.2 the set of possible distinct types for a term in a context is quite restricted, and may be easily decscribed by a constrained term. We define a *pre-type scheme* to be a constrained term $(X, \mathcal{C})$ such that either

1. X is level closed and $\mathcal{C} = \emptyset$, or

2. $X = \{x_1{:}A_1\} \cdots \{x_n{:}A_n\}\mathsf{Type}_\alpha$, where α is a *level variable*, and $\mathcal{C} = \{ \alpha \geq i \}$ for some natural number i.

For pre-type schemes, the relation $(X, \mathcal{C}) \geq A$ is readily seen to be decidable. A pre-type scheme is a *type scheme* with respect to a context Γ iff whenever $(X, \mathcal{C}) \geq A$, $\Gamma \vdash A \Rightarrow \kappa$ for some κ. The relation

$$
\begin{array}{rcl}
\mathrm{lb}(\lambda,\mathcal{C}) & := & i \quad \textit{if} \lambda = i \textit{ or } \lambda = \alpha \textit{ and } \mathcal{C} = \{\alpha \geq i\} \\
\mathsf{CUM}(X,\mathcal{C}) & := & \begin{cases} (\mathsf{Type}_\alpha, \{\alpha \geq \mathrm{lb}(\lambda,\mathcal{C})\}) & \textit{if } X \overset{wh}{\twoheadrightarrow} \mathsf{Type}_\lambda \\ (X,\mathcal{C}) & \textit{otherwise} \end{cases} \\
(\kappa,\mathcal{C}) \Uparrow (\mathsf{Prop},\emptyset) & := & (\mathsf{Prop},\emptyset) \\
(\mathsf{Prop},\emptyset) \Uparrow (\mathsf{Type}_\lambda,\mathcal{C}) & := & (\mathsf{Type}_\alpha, \{\alpha \geq \mathrm{lb}(\lambda,\mathcal{C})\}) \\
(\mathsf{Type}_\lambda,\mathcal{C}) \Uparrow (\mathsf{Type}_\mu,\mathcal{C}) & := & (\mathsf{Type}_\alpha, \{\alpha \geq \max(\mathrm{lb}(\lambda,\mathcal{C}), \mathrm{lb}(\mu,\mathcal{C}))\})
\end{array}
$$

$$\Gamma \vdash \mathsf{Prop} \Rightarrow \mathsf{Type}_\alpha, \{\alpha \geq 0\} \qquad \text{(T-PROP)}$$

$$\Gamma \vdash \mathsf{Type}_i \Rightarrow \mathsf{Type}_\alpha, \{\alpha \geq i+1\} \qquad \text{(T-TYPE)}$$

$$\Gamma \vdash x \Rightarrow \mathsf{CUM}(A,\emptyset) \quad (x{:}A \in \Gamma) \qquad \text{(T-VAR)}$$

$$\frac{\Gamma \vdash A \Rightarrow X,\mathcal{C} \quad X \overset{wh}{\twoheadrightarrow} \kappa_1 \quad \Gamma[x{:}A] \vdash B \Rightarrow Y,\mathcal{D} \quad Y \overset{wh}{\twoheadrightarrow} \kappa_2}{\Gamma \vdash \{x{:}A\}B \Rightarrow (\kappa_1,\mathcal{C}) \Uparrow (\kappa_2,\mathcal{D})} \quad (x \notin \mathrm{dom}(\Gamma)) \qquad \text{(T-GEN)}$$

$$\frac{\Gamma \vdash A \Rightarrow X,\mathcal{C} \quad X \overset{wh}{\twoheadrightarrow} \kappa \quad \Gamma[x{:}A] \vdash M \Rightarrow Y,\mathcal{D}}{\Gamma \vdash [x{:}A]M \Rightarrow \{x{:}A\}Y,\mathcal{D}} \quad (x \notin \mathrm{dom}(\Gamma)) \qquad \text{(T-ABS)}$$

$$\frac{\Gamma \vdash M \Rightarrow X,\mathcal{C} \quad X \overset{wh}{\twoheadrightarrow} \{x : A_1\}X_2 \quad \Gamma \vdash N \Rightarrow Y,\mathcal{D} \quad (Y,\mathcal{D}) \gtrsim A_1}{\Gamma \vdash MN \Rightarrow \mathsf{CUM}([N/x]X_2,\mathcal{C})} \qquad \text{(T-APP)}$$

Table 3: Type Synthesis Algorithm

$(X,\mathcal{C}) \gtrsim A$ holds iff there exists B such that $(X,\mathcal{C}) \geq B$ and $B \simeq A$. This relation is decidable provided that $(X,\mathcal{C})$ is a type scheme, and A is a type, with respect to Γ. A pair $(X,\mathcal{C})$ is a *principal type scheme* (p.t.s.) for Γ and M iff $\Gamma \vdash M \Rightarrow A$ exactly when $(X,\mathcal{C}) \geq A$. Principal type schemes are unique up to choice of level variable names. If $(X,\mathcal{C})$ is a p.t.s. for Γ and M, then $\Gamma \vdash M : A$ iff $(X,\mathcal{C}) \gtrsim A$, and M is well-typed in Γ iff $\mathcal{C}$ is satisfiable. Our goal, then, is to give an algorithm to compute the principal type scheme of a term in a context, failing iff none exists.

Following [CDD+85, Des84, HMT87, HMT88], we present the type synthesis algorithm as a formal system for deriving assertions of the form $\Gamma \vdash M \Rightarrow (X,\mathcal{C})$. This form of presentation is advantageous because it makes clear the close relationship between the algorithm and the operational semantics. In particular, one can easily see that the type checking algorithm is essentially a "determinization" of the operational semantics. This system is defined by the axioms and derivations rules given in Table 3. This system makes use of two auxiliary functions CUM and $\Uparrow$ corresponding to cum and $\uparrow$ in the operational semantics.

The relationship between the inference system and the operational semantics is made precise by the following theorem:

Theorem 4.2

Soundness *If* $\Gamma \vdash M \Rightarrow (X,\mathcal{C})$, *and* $(X,\mathcal{C}) \geq A$, *then* $\Gamma \vdash M \Rightarrow A$.

Completeness *If* $\Gamma \vdash M \Rightarrow A$, *then there exists a pre-type scheme* $(X,\mathcal{C})$ *such that* $\Gamma \vdash M \Rightarrow (X,\mathcal{C})$ *and* $(X,\mathcal{C}) \geq A$.

It follows from the soundness of the operational semantics that if Γ *valid* and $\Gamma \vdash M \Rightarrow (X, C)$, then $(X, \mathcal{C})$ is a type scheme.

It is easy to see that $\Gamma \vdash M \Rightarrow (X, \mathcal{C})$ is a partial function of Γ and M. The following theorem establishes that this relation defines an algorithm for computing principal type schemes:

Theorem 4.3 *Given Γ and M, with Γ valid, it is decidable whether or not there exists $(X, \mathcal{C})$ such that $\Gamma \vdash M \Rightarrow (X, \mathcal{C})$.*

5 δ-Reductions

In practical applications it is useful to bind terms to identifiers so that a term may be referred to by name, rather than repeated at each occurrence. Definitions may be formalized using a simple form of *δ-reductions* [Bar84]. In order to avoid complicating the delicate proof-theory of the basic calculus, we extend only the operational semantics to support δ-reductions.

A *definition* is a pair of the form $x{=}M$; the variable x is *defined* by the definition. A *δ-context* is a finite sequence of declarations and definitions such that no variable is bound more than once, and such that if $\Delta = \Delta_x[x{=}M]\Delta^x$, then $\mathrm{FV}(M) \subseteq \mathrm{dom}(\Delta_x)$. The metavariable Δ ranges over δ-contexts.

Reduction is extended to account for defined identifiers by defining the relation $\Delta \vdash M \to N$ to be the compatible closure of the axiom schemes

$$\Delta \vdash M \to N \;\; \textit{if} \;\; M \to N \qquad \textit{and} \qquad \Delta \vdash x \to N \;\; \textit{if} \;\; x{=}N \in \Delta$$

The one-step head and weak-head reduction relations are defined similarly, and reduction and conversion are defined in terms of one-step reduction as before. The relation $\Delta \vdash M \to N$ is Church-Rosser [Bar84].

The type synthesis problem for the system with δ-reductions is to compute the characteristic function of the set $\mathrm{Types}_\Delta(M) = \{\, A \mid \Delta \vdash M \Rightarrow B \textit{ and } \Delta \vdash B \simeq A \,\}$, and the well-typedness problem is to decide whether or not this set is empty.

The operational semantics of Table 2 is modified to account for δ-reductions by replacing all uses of reduction and conversion with the corresponding relation relativized to Δ, and by adding the following rule of inference:

$$\frac{\Delta_x \vdash M \Rightarrow A}{\Delta \vdash x \Rightarrow A} \quad (\Delta = \Delta_x[x{=}M]\Delta^x) \qquad \text{(S-DEF)}$$

The rule S-DEF reflects the principle of eliminability of definitions: a defined variable has whatever types its definition has. Since types are not unique, this implies that distinct occurrences of a defined variable may be assigned distinct types. For example, it is easy to see that

$$[x{=}[y{:}\mathsf{Prop}]\mathsf{Type}_0] \vdash ([z{:}\mathsf{Prop} \to \mathsf{Type}_1][w{:}\mathsf{Prop} \to \mathsf{Type}_2]\mathsf{Prop})\; x\; x \Rightarrow \mathsf{Type}_0$$

where the occurrances of x take distinct types, namely $\mathsf{Prop} \to \mathsf{Type}_1$ and $\mathsf{Prop} \to \mathsf{Type}_2$.

Definitions do *not* add to the strength of the operational semantics. To express this, we define "expansion maps": $\Delta(M)$, which replaces all defined variables in M by their definitions, and $\overline{\Delta}$, which expands a δ-context to a context:

$$\begin{array}{rcl}
\Delta(\mathsf{Prop}) & = & \mathsf{Prop} \\
\Delta(\mathsf{Type}_i) & = & \mathsf{Type}_i \\
\Delta(x) & = & \begin{cases} \Delta(M) & \textit{if } x{=}M \in \Delta \\ x & \textit{otherwise} \end{cases} \\
\Delta([x:A]M) & = & [x:\Delta(A)]\Delta(M) \\
\Delta(\{x:A\}B) & = & \{x:\Delta(A)\}\Delta(B) \\
\Delta(MN) & = & \Delta(M)\Delta(N)
\end{array}
\qquad
\begin{array}{rcl}
\overline{\langle\rangle} & = & \langle\rangle \\
\overline{\Delta[x:A]} & = & \overline{\Delta}[x:\Delta(A)] \\
\overline{\Delta[x=M]} & = & \overline{\Delta}
\end{array}$$

Theorem 5.1 (Eliminability of definitions)

1. $\Delta \vdash M \simeq N$ *iff* $\Delta(M) \simeq \Delta(N)$.
2. *If* $\Delta \vdash M \Rightarrow A$, *then* $\overline{\Delta} \vdash \Delta(M) \Rightarrow \Delta(A)$.
3. *If* $\overline{\Delta} \vdash \Delta(M) \Rightarrow A$, *then* $\Delta \vdash M \Rightarrow B$ *for some* B *such that* $\Delta(B) = A$.

It follows that the type synthesis problem for the system with δ-reductions is decidable: to check $\Delta \vdash M \Rightarrow A$, check whether $\overline{\Delta} \vdash \Delta(M) \Rightarrow \Delta(A)$. This approach is not particularly practical since the elimination of definitions can result in an exponential (in the number of definitions) increase in the size of the term to be type checked. However, we can avoid re-computing the type of a defined variable on each use by storing its principal type scheme with the definition.

Lemma 3.2(1) may be proved for the operational semantics extended with δ-reductions. We therefore choose the same definition of type scheme as given in Section 4. A *generic δ-context* is a finite sequence of declarations and *generic definitions* of the form $x{=}M{:}(X,\mathcal{C})$. Let Φ range over generic δ-contexts, and define $|\Phi|$ to be the ordinary δ-context obtained from Φ by replacing each generic definition $x{=}M{:}(X,\mathcal{C})$ by the simple definition $x{=}M$.

The type synthesis algorithm for the language with δ-reductions is defined by replacing uses of reduction and conversion in Table 3 with their analogues for definitions, and by adding the rule:

$$\Phi \vdash x \Rightarrow (X,\mathcal{C}) \quad (\Phi = \Phi_x[x{=}M{:}(X,\mathcal{C})]\Phi^x) \qquad \text{(T-DEF)}$$

A generic δ-context Φ is *principal* iff whenever $\Phi = \Phi_x[x{=}M{:}(X,\mathcal{C})]\Phi^x$, then $(X,\mathcal{C})$ is a principal type scheme for M in $|\Phi_x|$. The soundness and completeness of the algorithm is expressed by:

Theorem 5.2 *Let Φ be a principal generic context.*

Soundness *If $\Phi \vdash M \Rightarrow (X,\mathcal{C})$ and $(X,\mathcal{C}) \geq A$, then $|\Phi| \vdash M \Rightarrow A$.*

Completeness *If $|\Phi| \vdash M \Rightarrow A$, then there exists $(X,\mathcal{C})$ such that $\Phi \vdash M \Rightarrow (X,\mathcal{C})$ and $(X,\mathcal{C}) \geq A$.*

Define $||\Phi||$ to be the ordinary context obtained from Φ by removing all definitions. Notice that if Φ is principal and $||\Phi||$ *valid*, then every definition in Φ is in fact well-typed.

Theorem 5.3 *If Φ is principal, and $||\Phi||$ valid, it is decidable whether or not there exists $(X,\mathcal{C})$ such that $\Phi \vdash M \Rightarrow (X,\mathcal{C})$.*

In short, given Φ and M, with Φ principal, then the type synthesis algorithm computes a principal type scheme for M, failing if none exists.

Local definitions may be introduced by extending the syntax to include terms of the form LET $x = M$ IN N. We may extend the algorithm and prove decidability and principal typing as above.

6 Typical Ambiguity

It is tedious in practice to assign specific levels to universes, particularly since it is usually only the relationship between the universe levels in a given term that matters, not the specific values. Therefore we want to extend the calculus to admit the "anonymous" universe Type, leaving it to the type checker to determine the range of possible universes that may appear in that position. For example, in the term $[x{:}\mathsf{Type}]x$, the anonymous universe Type may be replaced by any universe Type_i to obtain a well-typed term. But in the term $([x{:}\mathsf{Type}_1]x)\,\mathsf{Type}$, the anonymous universe may only be replaced by Type_0 to be typeable in the basic calculus. Even if absolute (natural number) universe levels do not occur, non-trivial constraints are needed to ensure that all instances are typeable. For example, the second instance of Type in $([x{:}\mathsf{Type}]x)\,\mathsf{Type}$ may only be assigned a level less than the first instance.

We see from these examples that we cannot simply extend the basic calculus with the anonymous universe Type, since we must somehow keep track of the relationships between occurrences of Type to ensure that all instances are sensible. Although the concrete syntax we want to use allows occurrences of Type, we formalize an extension of the calculus that admits universe schemes as part of the "input language" (in contrast to the situation considered above in which universe schemes occur only as part of the generated type scheme of a level closed term, i.e. in a metatheoretic description of the possible types of a basic term). Then anonymous universes are explained by implicitly replacing each occurrence of Type with a schematic universe, Type_α, where α is a fresh level variable unique to that occurrence[3]. Of course, absolute universe levels may still appear.

We first give an algorithm for type synthesis that doesn't handle δ−reductions, and then show how to implement eliminable δ−reductions analogous to Section 5.

6.1 Schematic Type Synthesis

A *schematic context* is a context containing *schematic declarations* of the form $x{:}(X,\mathcal{C})$, where $(X,\mathcal{C})$ is a constrained term. Let Θ range over schematic contexts. We will refer to the *global constraint set* of a schematic context, defined by $\mathcal{G}_\Theta := \bigcup\{\mathcal{C} \mid x{:}(X,\mathcal{C}) \in \Theta\}$, and write $\tau \models \Theta$ for $\tau \models \mathcal{G}_\Theta$. Also, define the application of an assignment to a schematic context by $\tau(\Theta[x{:}(X,\mathcal{C})]) := (\tau\Theta)[x{:}\tau X]$. Define Θ *valid* iff for all $\tau \models \Theta$, $\tau\Theta$ *valid*.

Given Θ and X such that Θ *valid*, the *schematic type synthesis problem* is the characteristic function of $\mathrm{Types}_\Theta(X) := \{A \mid \exists\sigma\,.\,\sigma \models \Theta \text{ and } \sigma\Theta \vdash \sigma X : A\}$. The *schematic well-typedness* problem is to decide whether or not $\mathrm{Types}_\Theta(X) = \emptyset$. In this section we present an algorithm that solves the schematic type synthesis problem and the schematic well-typedness problem. The idea of this algorithm is identical to that for the type synthesis algorithm described above: transform the operational semantics for the basic language (Table 2) into an algorithm by replacing explicit level numbers with schematic level variables and appropriate constraints. Now we must do this uniformly throughout the operational semantics, for all terms appearing in a derivation may be schematic. Nonetheless it is possible to generalize the definition of principal type scheme to this case, and there is an algorithm for computing principal type schemes that is sound and complete.

6.1.1 Schematic Conversion Algorithm

The schematic type synthesis algorithm makes use of an algorithm to decide schematic conversion: given X and Y, find $\mathcal{D}$ such that all and only those level assignments satisfying $\mathcal{D}$ make X and Y convertible. The relation $\vdash X \simeq Y, \mathcal{D}$ is defined by the rules of Table 4. It is easy to see that this relation is a partial function of X and Y. If X and Y each have a well typed instance, then X and Y are normalizable, and the relation is decideable. The interesting rule is CNV-TYPE, saying that any two schematic universes convert, under the constraint that their levels are equal. The essential properties of this algorithm are summarized in the following lemma:

Lemma 6.1

1. *Suppose* $\vdash X \simeq Y, \mathcal{D}$ *and* $\sigma \models \mathcal{D}$. *Then* $\sigma X \simeq \sigma Y$.
2. *Suppose* $\sigma X \simeq \sigma Y$. *Then there exists* $\mathcal{D}$ *such that* $\vdash X \simeq Y, \mathcal{D}$, *and* $\sigma \models \mathcal{D}$.

6.1.2 A Schematic Type Synthesis Algorithm

We introduce a notion of principal type scheme for the schematic language.

Definition 6.2 *A constrained term* $(Y,\mathcal{D})$ *is a* principal type scheme *for* X *in* Θ *iff*

[3]The assumption that each level variable occurrence is unique is never used in Section 6.1; in fact we solve the problem of type synthesis for the full language of schematic terms. In Section 6.2, on δ-reductions, we use this assumption to make sense of definitions with typical ambiguity.

$$\frac{X \overset{wh}{\twoheadrightarrow} \mathsf{Type}_\mu \quad Y \overset{wh}{\twoheadrightarrow} \mathsf{Type}_\lambda}{\vdash X \simeq Y, \{\lambda \geq \mu, \mu \geq \lambda\}} \qquad \text{(CNV-TYPE)}$$

$$\frac{X \overset{wh}{\twoheadrightarrow} \mathsf{Prop} \quad Y \overset{wh}{\twoheadrightarrow} \mathsf{Prop}}{\vdash X \simeq Y, \{\,\}} \qquad \frac{X \overset{wh}{\twoheadrightarrow} x \quad Y \overset{wh}{\twoheadrightarrow} x}{\vdash X \simeq Y, \{\,\}}$$

$$\frac{X \overset{wh}{\twoheadrightarrow} \{x : X_1\}X_2 \quad Y \overset{wh}{\twoheadrightarrow} \{y : Y_1\}Y_2 \quad \vdash X_1 \simeq Y_1, \mathcal{D} \quad \vdash X_2 \simeq Y_2, \mathcal{E}}{\vdash X \simeq Y, \mathcal{D} \cup \mathcal{E}}$$

$$\frac{X \overset{wh}{\twoheadrightarrow} [x : X_1]X_2 \quad Y \overset{wh}{\twoheadrightarrow} [y : Y_1]Y_2 \quad \vdash X_1 \simeq Y_1, \mathcal{D} \quad \vdash X_2 \simeq Y_2, \mathcal{E}}{\vdash X \simeq Y, \mathcal{D} \cup \mathcal{E}}$$

$$\frac{X \overset{wh}{\twoheadrightarrow} x\,X' \quad Y \overset{wh}{\twoheadrightarrow} y\,Y' \quad \vdash x \simeq y, \mathcal{D} \quad \vdash X' \simeq Y', \mathcal{E}}{\vdash X \simeq Y, \mathcal{D} \cup \mathcal{E}} \quad (x \text{ and } y \text{ variables})$$

Table 4: Schematic Conversion Algorithm

1. *for every* τ *such that* $\tau \models \mathcal{D}$ *and* $\mathrm{dom}(\tau) \supseteq \mathrm{LV}(\Theta) \cup LV(X)$, $\tau\Theta \vdash \tau X \Rightarrow \tau Y$
2. *for every* τ, A, *such that* $\mathrm{dom}(\tau) = LV(\Theta) \cup LV(X)$ *and* $\tau\Theta \vdash \tau X \Rightarrow A$, *there exists a* witness, τ_A, *such that: (i)* τ_A *extends* τ *and* $\tau_A \models \mathcal{D}$, *and (ii)* $\tau_A Y = A$ *(hence* $\tau_A\Theta \vdash \tau_A X \Rightarrow \tau_A Y$ *).*

Remark The requirements on $\mathrm{dom}(\tau)$ in Definition 6.2 are purely technical. In both parts of the definition we require $\mathrm{dom}(\tau) \supseteq \mathrm{LV}(\Theta) \cup \mathrm{LV}(X)$ so that $\tau\Theta$ and τX, which occur in assertions about the operational semantics, are level closed. In part 2 of the Definition, we require $\mathrm{dom}(\tau) \subseteq LV(\Theta) \cup LV(X)$ so that τ contains no "junk" which might prevent it being extended to τ_A.

The relation $\Theta \vdash X \Rightarrow Y, \mathcal{D}$ is defined by the rules of Table 5. It is easy to see that this relation is a partial function of Θ and X (up to the choice of level variables in $Y, \mathcal{D}$), hence can be viewed as an algorithm for the type synthesis problem. Notice that the algorithm doesn't check satisfiability of constraints, so sucessful termination does *not* depend on the actual level expressions or constraints in Θ and X. Also, if $\Theta \vdash X \Rightarrow Y, \mathcal{D}$, then $(Y, \mathcal{D})$ is a constrained term; In fact $\mathcal{D}$ contains all the constraints upon which any instance of the the schematic derivation actually depends. The soundness and completeness of the schematic type synthesis algorithm is established by:

Theorem 6.3

Soundness *If* $\Theta \vdash X \Rightarrow Y, \mathcal{C}$, $\mathrm{dom}(\tau) \supseteq LV(\Theta) \cup \mathrm{LV}(X)$, *and* $\tau \models \mathcal{C}$, *then* $\tau\Theta \vdash \tau X \Rightarrow \tau Y$.

Completeness *Given* Θ *and* X, *if there exist* σ, A *such that* $\sigma\Theta \vdash \sigma X \Rightarrow A$, *then the algorithm succeeds and returns a principal type scheme for* X *in* Θ.

Theorem 6.4 *Given* Θ *and* X, *with* Θ *valid, it is decidable whether or not there exists* $(Y, \mathcal{C})$ *such that* $\Theta \vdash X \Rightarrow (Y, \mathcal{C})$.

The schematic type synthesis and well-typedness problems are thus reduced to checking satisfiability of a constraint set: given Θ *valid*, and X, $\mathrm{Types}_\Theta(X)$ is non-empty iff $\Theta \vdash X \Rightarrow Y, \mathcal{D}$ (the algorithm succeeds) with $\mathcal{G}_\Theta \cup \mathcal{D}$ satisfiable. In particular $A \in \mathrm{Types}_\Theta(X)$ iff $\Theta \vdash X \Rightarrow Y, \mathcal{D}$ and $\vdash Y \simeq A, \mathcal{E}$ with $\mathcal{G}_\Theta \cup \mathcal{D} \cup \mathcal{E}$ satisfiable.

$$\mathsf{CUM}(X,\mathcal{C}) := \begin{cases} (\mathsf{Type}_\alpha, \mathcal{C} \cup \{\alpha \geq \lambda\}) & \textit{if } X \overset{wh}{\twoheadrightarrow} \mathsf{Type}_\lambda \quad (\alpha \textit{ new}) \\ (X,\mathcal{C}) & \textit{otherwise} \end{cases}$$

$$\kappa_1 \Uparrow_{\mathcal{C}} \kappa_2 := \begin{cases} (\mathsf{Prop}, \mathcal{C}) & \textit{if } \kappa_2 \overset{wh}{\twoheadrightarrow} \mathsf{Prop} \\ (\mathsf{Type}_\alpha, \mathcal{C} \cup \{\alpha \geq \lambda\}) & \textit{if } \kappa_1 \overset{wh}{\twoheadrightarrow} \mathsf{Prop}, \kappa_2 \overset{wh}{\twoheadrightarrow} \mathsf{Type}_\lambda \quad (\alpha \textit{ new}) \\ (\mathsf{Type}_\alpha, \mathcal{C} \cup \{\alpha \geq \lambda, \alpha \geq \mu\}) & \textit{if } \kappa_1 \overset{wh}{\twoheadrightarrow} \mathsf{Type}_\lambda, \kappa_2 \overset{wh}{\twoheadrightarrow} \mathsf{Type}_\mu \quad (\alpha \textit{ new}) \end{cases}$$

$$\Theta \vdash \mathsf{Prop} \Rightarrow \mathsf{Type}_\alpha, \{\alpha \geq 0\} \quad (\alpha \textit{ new}) \qquad \text{(I-PROP)}$$

$$\Theta \vdash \mathsf{Type}_\lambda \Rightarrow \mathsf{Type}_\alpha, \{\alpha > \lambda\} \quad (\alpha \textit{ new}) \qquad \text{(I-TYPE)}$$

$$\Theta \vdash x \Rightarrow \mathsf{CUM}(Y,\mathcal{C}) \quad (x{:}(Y,\mathcal{C}) \in \Theta) \qquad \text{(I-VAR)}$$

$$\frac{\Theta \vdash X \Rightarrow X', \mathcal{D} \quad X' \overset{wh}{\twoheadrightarrow} \kappa_1 \quad \Theta[x{:}(X,\mathcal{D})] \vdash Y \Rightarrow Y', \mathcal{E} \quad Y' \overset{wh}{\twoheadrightarrow} \kappa_2}{\Theta \vdash \{x{:}X\}Y \Rightarrow \kappa_1 \Uparrow_{\mathcal{D} \cup \mathcal{E}} \kappa_2} \; (x \notin \mathrm{dom}(\Theta)) \qquad \text{(I-GEN)}$$

$$\frac{\Theta \vdash X \Rightarrow X', \mathcal{D} \quad X' \overset{wh}{\twoheadrightarrow} \kappa \quad \Theta[x{:}(X,\mathcal{D})] \vdash Y \Rightarrow Y', \mathcal{E}}{\Theta \vdash [x{:}X]Y \Rightarrow \{x{:}X\}Y', \mathcal{D} \cup \mathcal{E}} \; (x \notin \mathrm{dom}(\Theta)) \qquad \text{(I-ABS)}$$

$$\frac{\Theta \vdash X \Rightarrow X', \mathcal{D} \quad X' \overset{wh}{\twoheadrightarrow} \{x{:}X_1'\}X_2' \quad \Theta \vdash Y \Rightarrow Y', \mathcal{E} \quad \vdash Y' \simeq X_1', \mathcal{F}}{\Theta \vdash X\,Y \Rightarrow \mathsf{CUM}([Y/x]X_2', \mathcal{D} \cup \mathcal{E} \cup \mathcal{F})} \qquad \text{(I-APP)}$$

Table 5: Schematic Type Synthesis Algorithm

Satisfiability of the constraint sets generated by the type synthesis algorithm can be checked in polynomial time using the methods developed by Chan [Cha77] to decide a larger theory of arithmetic inequalities. A constraint set $\mathcal{C}$ is represented by a weighted, directed graph with integer edge weights, and the graph is checked for positive-weight cycles. The graph associated to $\mathcal{C}$ is definded by first transforming all constraints into the form $\alpha \geq m$ or $\alpha \geq \beta + m$, where m is a (positive or negative) integer. (This transformation does not change the size of $\mathcal{C}$.) The transformed set of constraints determines a graph defined by taking as nodes the level variables occurring in $\mathcal{C}$, together with a distinguished node for 0, and as edges $\lambda \xrightarrow{m} \mu$, one per constraint of the form $\lambda \geq \mu + m$. Chan then proves that the original constraint set $\mathcal{C}$ is satisfiable iff the resulting graph has no positive-weight cycle, a condition that can be tested in time $\mathcal{O}(m, n^3)$, where m is the number of constraints and n is the number of level variables. It is worth remarking that the integer edge weights are necessary in order to express the fact that the natural numbers form a *discrete* linear order: a simple check for cyclicity would not account for the fact that the constraint set $\{\alpha \geq 2, \beta < 3, \beta > \alpha\}$ is unsatisfiable. Such constraint sets can arise in our situation because we admit both schematic and specific universes in the input language.

6.2 δ-Reductions with Typical Ambiguity

We would like to make sense of the the notion "definition" in the system with typical ambiguity. In analogy with Section 5, define a *schematic δ-context* to be a schematic context possibly containing *schematic definitions* of the form $x{=}X$, such that no variable is bound more than once, and such that if $\Delta = \Delta_x[x{=}M]\Delta^x$, then $\mathrm{FV}(M) \subseteq \mathrm{dom}(\Delta_x)$. Let Δ range over schematic δ-contexts. We could naively add the rule:

$$\frac{\Delta_x \vdash X \Rightarrow Y, \mathcal{C}}{\Delta \vdash x \Rightarrow Y, \mathcal{C}} \quad (\Delta = \Delta_x[x{=}X]\Delta^x) \qquad \text{(NAIVE)}$$

to the rules of Table 5 (suitably modified with Δ in place of Θ). This is sensible, but not exactly what we mean by "definition" in this system. Remember, the level variables in a term X input to the type synthesis algorithm are fresh, inserted by "pre-processing" a concrete term possibly containing instances of Type. Thus, thinking of definition as abbreviation for concrete syntax, we intend each instance of a definition to be expanded with fresh level variables. With this in mind, define *level variable renaming* to be an injective function from level variables to level variables, extended to schematic terms and contexts as usual. For Ψ ranging over finite sets of level variables, let ν_Ψ be a level variable renaming that assigns a fresh level variable to each level variable occuring in Ψ. Extend the system of Table 5 with a rules for definitions:

$$\frac{\Delta_x \vdash \nu_{\mathrm{LV}(X)}X \Rightarrow Y, \mathcal{C}}{\Delta \vdash x \Rightarrow Y, \mathcal{C}} \quad (\Delta = \Delta_x[x{=}X]\Delta^x) \qquad \text{(I-DEFN)}$$

Defining expansion maps as in Section 5, we see that definitions are eliminable from this system[4]. Notice that the system with I-DEFN succeeds on strictly more terms than the system with NAIVE. For example, with $\Delta = [id{=}[t : \mathsf{Type}_\alpha][x : t]x]$, $(id\ \mathsf{Type}_\beta\ id)$ is typable using I-DEFN, but not using NAIVE.

This extended system is an algorithm in the same sense as before, so is an implementation of definitions in the system with typical ambiguity. It is not, however, an efficient implementation in the sense the algorithm of Section 5 is efficient, typing the value of a definition only once, and looking its type up in the context when needed. In order to express such an efficient algorithm, define a *schematic generic δ-context* to be a finite sequence of schematic declarations and *schematic generic definitions* of the form $x{=}X{:}(Y,C)$. Let Φ range over schematic generic contexts. Extend the system of Table 5 (but without I-DEFN) with the rule (where $\setminus$ is set difference):

$$\Phi \vdash x \Rightarrow \nu_{\mathrm{LV}(C)\setminus\mathrm{LV}(\Phi_x)}(Y, C) \quad (\Phi = \Phi_x[x{=}X{:}(Y,C)]\Phi^x) \qquad \text{(I-DEFN')}$$

[4]Of course definitions would be eliminable with NAIVE as well; it's just that the expansion functions are different.

Let $|\Phi|$ be the schematic delta context obtained from Φ by replacing each schematic generic definition $x{=}X{:}(Y,\mathcal{C})$ by the schematic definition $x{=}X$, and call Φ *principal* iff whenever $\Phi = \Phi_x[x{=}X{:}(Y,\mathcal{C})]\Phi^x$ then $|\Phi_x| \vdash \nu_{\mathrm{LV}(X)}X \Rightarrow Y,\mathcal{C}$. Rule I-DEFN′ says that the level variables that should be considered generic in a schematic generic definition $x{=}X{:}(Y,\mathcal{C})$ are precisely those whose first occurrence in the context is in that definition. If Φ is principal, these are precisely the level variables of X plus those freshly generated by the algorithm in a derivation of $|\Phi_x| \vdash \nu_{\mathrm{LV}(X)}X \Rightarrow Y,\mathcal{C}$. We want to claim that every use of I-DEFN′ returns the same pair $(Y,\mathcal{C})$ as would a use of I-DEFN (up to the names of new level variables). But if Φ is principal, then $Y,\mathcal{C}$ is exactly what a use of I-DEFN (i.e $|\Phi| \vdash \nu_{\mathrm{LV}(X)}X \Rightarrow Y,\mathcal{C}$) did return (up to the names of new level variables). We have argued for:

Conjecture 6.5 *Suppose Φ is principal.*

Soundness *If $\Phi \vdash X \Rightarrow Y,\mathcal{C}$, there is a level variable renaming, ξ, that is identity on $\mathrm{LV}(\Phi) \cup \mathrm{LV}(X)$, such that $|\Phi| \vdash X \Rightarrow \xi(Y,\mathcal{C})$.*

Completeness *If $|\Phi| \vdash X \Rightarrow Y,\mathcal{C}$, there is a level variable renaming, ξ, that is identity on $\mathrm{LV}(\Phi) \cup \mathrm{LV}(X)$, such that $\Phi \vdash X \Rightarrow \xi(Y,\mathcal{C})$.*

6.3 An Implementation

The schematic type synthesis algorithm, extended for strong sums, is implemented as part of LEGO, a refinement style proof checker for the Calculus of Constructions [Pol88]. The algorithm we have described is very liberal. That self application of the polymorphic identity is typable was mentioned in Section 6.2. A related example (from [Hue87]) shows that this system is really not Type : Type: the algorithm returns an unsatisfiable set of constraints when applied to $\Theta = [U{=}\{t : \mathsf{Type}\}\{x : t\}t][u : U]$ and $X = (u\ U\ u)$. We have used the algorithm to typecheck Coquand's proof of Girard's Paradox [Coq86, Coq88]. The algorithm succeeds (showing the proof correct in a system with Type : Type) with an unsatisfiable set of constraints (showing the proof is *not* correct in CC^{ω}).

Acknowledgements We are grateful to Thierry Coquand, Gérard Huet, and Joëlle Despeyroux for their helpful comments on this paper. The second author especially thanks Gérard Huet, who has encouraged his interest in the Calculus of Constructions, and suggested that the problems addressed here could be solved by manipulating the integer arguments to the Type constant as symbolic expressions.

References

[Bar84] H. P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, Amsterdam, revised edition, 1984.

[BL84] R. Burstall and Butler Lampson. A kernel language for abstract data types and modules. In G. Kahn, D. MacQueen, and G. Plotkin, editors, *Semantics of Data Types*, volume 173 of *Lecture Notes in Computer Science*, pages 1–50. Springer-Verlag, 1984.

[Car] Luca Cardelli. Phase distinctions in type theory. unpublished manuscript.

[Car86] Luca Cardelli. A polymorphic λ-calculus with Type:Type. Technical report, DEC SRC, 1986.

[CDD+85] D. Clément, J. Despeyroux, T. Despeyroux, L. Hascoet, and G. Kahn. Natural semantics on the computer. Technical Report RR 416, INRIA, Sophia–Antipolis, France, June 1985.

[CDDK86] D. Clément, J. Despeyroux, T. Despeyroux, and G. Kahn. A simple applicative language: Mini-ML. In *Proceedings of the Conference on Lisp and Functional Programming*, 1986.

[CH85] Thierry Coquand and Gérard Huet. Constructions: a higher-order proof system for mechanizing mathematics. In B. Buchberger, editor, *EUROCAL '85: European Conference on Computer Algebra*, volume 203 of *Lecture Notes in Computer Science*, pages 151–184. Springer-Verlag, 1985.

[Cha77] Tat-Hung Chan. An algorithm for checking PL/CV arithmetical inferences. Technical Report 77–236, Computer Science Department, Cornell University, Ithaca, New York, 1977.

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

[Con86] Robert L. Constable, *et. al. Implementing Mathematics with the NuPRL Proof Development System.* Prentice–Hall, Englewood Cliffs, NJ, 1986.

[Coq85] Thierry Coquand. *Une théorie des constructions.* PhD thesis, Université Paris VII, January 1985.

[Coq86] Thierry Coquand. An analysis of Girard's paradox. In *Proc. of the Symposium on Logic in Computer Science*, pages 227–236, Boston, June 1986.

[Coq88] Thierry Coquand. Private communication.

[CZ82] Robert L. Constable and Daniel R. Zlatin. Report on the type theory (V3) of the programming logic PL/CV3. In *Logics of Programs*, volume 131 of *Lecture Notes in Computer Science*. Springer-Verlag, 1982.

[CZ84] Robert L. Constable and Daniel R. Zlatin. The type theory of PL/CV3. *ACM Transactions on Programming Languages and Systems*, 7(1):72–93, January 1984.

[Des84] T. Despeyroux. Executable specifications of static semantics. In G. Kahn, D. MacQueen, and G. Plotkin, editors, *Semantics of Data Types*, volume 173 of *Lecture Notes in Computer Science*. Springer-Verlag, June 1984.

[DM82] Luis Damas and Robin Milner. Principal type schemes for functional programs. In *Proceedings of the 9th ACM Symposium on the Principles of Programming Languages*, pages 207–212, 1982.

[Erh88] Thomas Erhard. A categorical semantics of Constructions. In *Proceedings of the Third Annual Symposium on Logic in Computer Science*, pages 264–273, Edinburgh, July 1988.

[GdR88] Paola Giannini and Simona Ronchi della Rocca. Characterization of typings in polymorphic type discipline. In *Proceedings of the Third Annual Symposium on Logic in Computer Science*, pages 61–71, July 1988.

[HH86] James G. Hook and Douglas Howe. Impredicative strong existential equivalent to Type:Type. Technical Report TR 86–760, Cornell University, Ithaca, New York, 1986.

[HMT87] Robert Harper, Robin Milner, and Mads Tofte. A type discipline for program modules. In *TAPSOFT '87*, volume 250 of *Lecture Notes in Computer Science*. Springer-Verlag, March 1987.

[HMT88] Robert Harper, Robin Milner, and Mads Tofte. The definition of Standard ML (version 2). Technical Report ECS–LFCS–88–62, Laboratory for the Foundations of Computer Science, Edinburgh University, August 1988.

[How87] Douglas Howe. The computational behavior of Girard's paradox. In *Proceedings of the Second Symposium on Logic in Computer Science*, pages 205–214, Ithaca, New York, June 1987.

[HP88] J. Martin E. Hyland and Andrew M. Pitts. The Theory of Constructions: categorical semantics and topos-theoretic models. In *Proceedings of the Boulder Conference on Categories in Computer Science*, 1988. To appear.

[Hue87] Gérard Huet. Extending the Calculus of Constructions with Type:Type. unpublished manuscript, April 1987.

[Luo88a] Zhaolui Luo. A higher-order calculus and theory abstraction. Technical Report ECS–LFCS–88–57, Laboratory for the Foundations of Computer Science, Edinburgh University, July 1988.

[Luo88b] Zhaolui Luo. $CC^{\infty}_{\subset}$ and its metatheory. Technical Report ECS–LFCS–88-58, Laboratory for the Foundations of Computer Science, Edinburgh University, July 1988.

[Mac86] David MacQueen. Using dependent types to express modular structure. In *Proceedings of the 13th ACM Symposium on the Principles of Programming Languages*, 1986.

[Mar] Per Martin-Löf. A theory of types. Unpublished manuscript.

[Mar73] Per Martin-Löf. An intuitionistic theory of types: predicative part. In H. E. Rose and J. C. Shepherdson, editors, *Logic Colloquium, '73*, pages 73–118, Amsterdam, 1973. North-Holland.

[Mar82] Per Martin-Löf. Constructive mathematics and computer programming. In *Sixth International Congress for Logic, Methodology, and Philosophy of Science*, pages 153–175, Amsterdam, 1982. North-Holland.

[Mar84] Per Martin-Löf. *Intuitionistic Type Theory*, volume 1 of *Studies in Proof Theory*. Bibliopolis, Naples, 1984.

[MH88] John Mitchell and Robert Harper. The essence of ML. In *Proceedings of the Fifteenth ACM Symposium on Principles of Programming Languages*, San Diego, California, January 1988.

[Mit84] John C. Mitchell. Type inference and type containment. In G. Kahn, D. MacQueen, and G. Plotkin, editors, *Semantics of Data Types*, volume 173 of *Lecture Notes in Computer Science*, pages 257–278. Springer-Verlag, 1984.

[MP85] John C. Mitchell and Gordon Plotkin. Abstract types have existential type. In *Proceedings of the 12th ACM Symposium on the Principles of Programming Languages*, 1985.

[MR86] Albert Meyer and Mark Reinhold. 'Type' is not a type: preliminary report. In *Proceedings of the 13th ACM Symposium on the Principles of Programming Languages*, 1986.

[Pol88] Robert Pollack. The theory of lego. Technical report, Laboratory for the Foundations of Computer Science, Edinburgh University, 1988. To appear.

[Rus08] Bertrand Russell. Mathematical logic as based on a theory of types. *American Journal of Mathematics*, 30:222–262, 1908.

[vD80] Diedrik T. van Daalen. *The Language Theory of AUTOMATH*. PhD thesis, Technical University of Eindhoven, Eindhoven, Netherlands, 1980.

[WR25] Alfred North Whitehead and Bertrand Russell. *Principia Mathematica, Volume 1*. Cambridge University Press, Cambridge, 1925.