Decidability of Type-checking in the Calculus of Algebraic Constructions with Size Annotations

Frédéric Blanqui

Laboratoire Lorrain de Recherche en Informatique et Automatique (LORIA)
Institut National de Recherche en Informatique et Automatique (INRIA)
615 rue du Jardin Botanique, BP 101, 54602 Villers-lès-Nancy, France
blanqui@loria.fr

Abstract. Since Val Tannen's pioneering work on the combination of simply-typed λ-calculus and first-order rewriting [11], many authors have contributed to this subject by extending it to richer typed λ-calculi and rewriting paradigms, culminating in the Calculus of Algebraic Constructions. These works provide theoretical foundations for type-theoretic proof assistants where functions and predicates are defined by oriented higher-order equations. This kind of definitions subsumes usual inductive definitions, is easier to write and provides more automation.

On the other hand, checking that such user-defined rewrite rules, when combined with β-reduction, are strongly normalizing and confluent, and preserve the decidability of type-checking, is more difficult. Most termination criteria rely on the term structure. In a previous work, we extended to dependent types and higher-order rewriting, the notion of "sized types" studied by several authors in the simpler framework of ML-like languages, and proved that it preserves strong normalization.

The main contribution of the present paper is twofold. First, we prove that, in the Calculus of Algebraic Constructions with size annotations, the problems of type inference and type-checking are decidable, provided that the sets of constraints generated by size annotations are satisfiable and admit most general solutions. Second, we prove the latter properties for a size algebra rich enough for capturing usual induction-based definitions and much more.

1 Introduction

The notion of "sized type" was first introduced in [21] and further studied by several authors [20,3,1,31] as a tool for proving the termination of ML-like function definitions. It is based on the semantics of inductive types as fixpoints of monotone operators, reachable by transfinite iteration. For instance, natural numbers are the limit of $(S_i)_{i<\omega}$, where S_i is the set of natural numbers smaller than i (inductive types with constructors having functional arguments require ordinals bigger than ω). The idea is then to reflect this in the syntax by adding size annotations on types indicating in which subset S_i a term is. For instance, subtraction on natural numbers can be assigned the type $- : nat^\alpha \Rightarrow nat^\beta \Rightarrow nat^\alpha$, where α and β are implicitly universally quantified, meaning that the size of its

output is not bigger than the size of its first argument. Then, one can ensure termination by restricting recursive calls to arguments whose size – by typing – is smaller. For instance, the following ML-like definition of $\lceil \frac{x}{y+1} \rceil$:

```
letrec div x y = match x with
  | 0 -> 0
  | S x' -> S (div (x' - y) y)
```

is terminating since, if x is of size at most α and y is of size at most β, then x' is of size at most $\alpha - 1$ and $(x' - y)$ is of size at most $\alpha - 1 < \alpha$.

The Calculus of Constructions (CC) [17] is a powerful type system with polymorphic and dependent types, allowing to encode higher-order logic. The Calculus of Algebraic Constructions (CAC) [8] is an extension of CC where functions are defined by higher-order rewrite rules. As shown in [10], it subsumes the Calculus of Inductive Constructions (CIC) [18] implemented in the Coq proof assistant [15], where functions are defined by induction. Using rule-based definitions has numerous advantages over induction-based definitions: definitions are easier (*e.g.* Ackermann's function), more propositions can be proved equivalent automatically, one can add simplification rules like associativity or using rewriting modulo AC [6], etc. For proving that user-defined rules terminate when combined with β-reduction, [8] essentially checks that recursive calls are made on structurally smaller arguments.

In [7], we extended the notion of sized type to CAC, giving the Calculus of Algebraic Constructions with Size Annotations (CACSA). We proved that, when combined with β-reduction, user-defined rules terminate essentially if recursive calls are made on arguments whose size – by typing – is strictly smaller, by possibly using lexicographic and multiset comparisons. Hence, the following rule-based definition of $\lceil \frac{x}{y+1} \rceil$:

$$\begin{array}{rcl} 0 \ / \ y & \to & 0 \\ (s\ x) \ / \ y & \to & s\ ((x\ -\ y)\ /\ y) \end{array}$$

is terminating since, in the last rule, if x is of size at most α and y is of size at most β, then $(s\ x)$ is of size at most $\alpha + 1$ and $(x - y)$ is of size at most $\alpha < \alpha + 1$. Note that this rewrite system cannot be proved terminating by criteria only based on the term structure, like RPO or its extensions to higher-order terms [22,29]. Note also that, if a term t is structurally smaller than a term u, then the size of t is smaller than the size of u. Therefore, CACSA proves the termination of any induction-based definition like CIC/Coq, but also definitions like the previous one. To our knowledge, this is the most powerful termination criterion for functions with polymorphic and dependent types like in Coq. The reader can find other convincing examples in [7].

However, [7] left an important question open. For the termination criterion to work, we need to make sure that size annotations assigned to function symbols are valid. For instance, if subtraction is assigned the type $- : nat^{\alpha} \Rightarrow nat^{\beta} \Rightarrow nat^{\alpha}$, then we must make sure that the definition of $-$ indeed outputs a term whose size is not greater than the size of its first argument. This amounts to

check that, for every rule in the definition of $-$, the size of the right hand-side is not greater than the size of the left hand-side. This can be easily verified by hand if, for instance, the definition of $-$ is as follows:

$$\begin{array}{rcl} 0 \ - \ x & \rightarrow & 0 \\ x \ - \ 0 & \rightarrow & x \\ (s\ x) \ - \ (s\ y) & \rightarrow & x \ - \ y \end{array}$$

The purpose of the present work is to prove that this can be done automatically, by inferring the size of both the left and right hand-sides, and checking that the former is smaller than the latter.

Fig. 1. Insertion sort on polymorphic and dependent lists

$$\begin{array}{r} nil : (A : \star) list^{\alpha}\ A\ 0 \\ cons : (A : \star) A \Rightarrow (n : nat) list^{\alpha}\ A\ n \Rightarrow list^{s\alpha}\ A\ (sn) \\ if\_in\_then\_else : bool \Rightarrow (A : \star) A \Rightarrow A \Rightarrow A \\ insert : (A : \star)(\leq : A \Rightarrow A \Rightarrow bool) A \Rightarrow (n : nat) list^{\alpha}\ A\ n \Rightarrow list^{s\alpha}\ A\ (sn) \\ sort : (A : \star)(\leq : A \Rightarrow A \Rightarrow bool)(n : nat) list^{\alpha}\ A\ n \Rightarrow list^{\alpha}\ A\ n \end{array}$$

$$\begin{array}{rcl} if\ true\ in\ A\ then\ u\ else\ v & \rightarrow & u \\ if\ false\ in\ A\ then\ u\ else\ v & \rightarrow & v \\ insert\ A\ \leq\ x\ \_\ (nil\ \_) & \rightarrow & cons\ A\ x\ 0\ (nil\ A) \\ insert\ A\ \leq\ x\ \_\ (cons\ \_\ y\ n\ l) & \rightarrow & if\ x \leq y\ in\ list\ A\ (s\ (s\ n)) \\ & & then\ cons\ A\ x\ (s\ n)\ (cons\ A\ y\ n\ l) \\ & & else\ cons\ A\ y\ (s\ n)\ (insert\ A\ \leq\ x\ n\ l) \\ sort\ A\ \leq\ \_\ (nil\ \_) & \rightarrow & nil\ A \\ sort\ A\ \leq\ \_\ (cons\ \_\ x\ n\ l) & \rightarrow & insert\ A\ \leq\ x\ n\ (sort\ A\ \leq\ n\ l) \end{array}$$

We now give an example with dependent and polymorphic types. Let $\star$ be the sort of types and $list : \star \Rightarrow nat \Rightarrow \star$ be the type of polymorphic lists of fixed length whose constructors are *nil* and *cons*. Without ambiguity, s is used for the successor function both on terms and on size expressions. The functions *insert* and *sort* defined in Figure 1 have size annotations satisfying our termination criterion. The point is that *sort* preserves the size of its list argument and thus can be safely used in recursive calls. Checking this automatically is the goal of this work.

An important point is that the ordering naturally associated with size annotations implies some subtyping relation on types. The combination of subtyping and dependent types (without rewriting) is a difficult subject which has been studied by Chen [12]. We reused many ideas and techniques of his work for designing CACSA and proving important properties like β-subject reduction (preservation of typing under β-reduction) [5].

Another important point is related to the meaning of type inference. In ML, type inference means computing a type of a term in which the types of free and bound variables, and function symbols (`letrec`'s in ML), are unknown. In other words, it consists in finding a simple type for a pure λ-term. Here, type inference

means computing a CACSA type, hence dependent and polymorphic (CACSA contains Girard's system F), of a term in which the types and size annotations of free and bound variables, and function symbols, are known. In dependent type theories, this kind of type inference is necessary for type-checking [16]. In other words, we do not try to infer relations between the sizes of the arguments of a function and the size of its output like in [13,4]. We try to check that, with the annotated types declared by the user for its function symbols, rules satisfy the termination criterion described in [7].

Moreover, in ML, type inference amounts to solve equality constraints in the type algebra. Here, type inference amounts to solve equality and ordering constraints in the size algebra. The point is that the ordering on size expressions is not anti-symmetric: it is a quasi-ordering. Thus, we have a combination of unification and symbolic quasi-ordering constraint solving.

Finally, because of the combination of subtyping and dependent typing, the decidability of type-checking requires the existence of minimal types [12]. Thus, we must also prove that a satisfiable set of equality and ordering constraints has a smallest solution, which is not the case in general. This is in contrast with non-dependently typed frameworks.

Outline. In Section 2, we define terms and types, and study some properties of the size ordering. In Section 3, we give a general type inference algorithm and prove its correctness and completeness under general assumptions on constraint solving. Finally, in Section 4, we prove that these assumptions are fulfilled for the size algebra introduced in [3] which, although simple, is rich enough for capturing usual inductive definitions and much more, as shown by the first example above. Missing proofs are given in [9].

2 Terms and types

Size algebra. Inductive types are annotated by *size expressions* from the following algebra $\mathcal{A}$:

$$a ::= \alpha \mid sa \mid \infty$$

where $\alpha \in \mathcal{Z}$ is a *size variable*. The set $\mathcal{A}$ is equipped with the quasi-ordering $\leq_{\mathcal{A}}$ defined in Figure 2. Let $\simeq_{\mathcal{A}} = \leq_{\mathcal{A}} \cap \geq_{\mathcal{A}}$ be its associated equivalence.

Let $\varphi, \psi, \rho, \ldots$ denote size substitutions, *i.e.* functions from $\mathcal{Z}$ to $\mathcal{A}$. One can easily check that $\leq_{\mathcal{A}}$ is stable by substitution: if $a \leq_{\mathcal{A}} b$ then $a\varphi \leq_{\mathcal{A}} b\varphi$. We extend $\leq_{\mathcal{A}}$ to substitutions: $\varphi \leq_{\mathcal{A}} \psi$ iff, for all $\alpha \in \mathcal{Z}$, $\alpha\varphi \leq_{\mathcal{A}} \alpha\psi$.

We also extend the notion of "more general substitution" from unification theory as follows: φ is *more general than* ψ, written $\varphi \sqsubseteq \psi$, iff there is φ' such that $\varphi\varphi' \leq_{\mathcal{A}} \psi$.

Terms. We assume the reader familiar with typed λ-calculi [2] and rewriting [19]. Details on CAC(SA) can be found in [8,7]. We assume given a set $\mathcal{S} = \{\star, \Box\}$ of *sorts* ($\star$ is the sort of types and propositions; $\Box$ is the sort of predicate types), a set $\mathcal{F}$ of function or predicate *symbols*, a set $\mathcal{CF}^{\Box} \subseteq \mathcal{F}$ of *constant predicate symbols*, and an infinite set $\mathcal{X}$ of *term variables*. The set $\mathcal{T}$ of terms is:

Fig. 2. Ordering on size expressions

$$\text{(refl)}\ \ a \leq_{\mathcal{A}} a \qquad \text{(trans)}\ \ \frac{a \leq_{\mathcal{A}} b \quad b \leq_{\mathcal{A}} c}{a \leq_{\mathcal{A}} c}$$

$$\text{(mon)}\ \ \frac{a \leq_{\mathcal{A}} b}{sa \leq_{\mathcal{A}} sb} \qquad \text{(succ)}\ \ \frac{a \leq_{\mathcal{A}} b}{a \leq_{\mathcal{A}} sb} \qquad \text{(infty)}\ \ a \leq_{\mathcal{A}} \infty$$

$$t ::= \mathtt{s} \mid x \mid C^a \mid f \mid [x : t]t \mid (x : t)t \mid tt$$

where $\mathtt{s} \in \mathcal{S}$, $x \in \mathcal{X}$, $C \in \mathcal{CF}^{\square}$, $a \in \mathcal{A}$ and $f \in \mathcal{F} \setminus \mathcal{CF}^{\square}$. A term $[x : t]u$ is an *abstraction*. A term $(x : T)U$ is a *dependent product*, simply written $T \Rightarrow U$ when x does not occur in U. Let $\boldsymbol{t}$ denote a sequence of terms $t_1, \ldots, t_n$ of length $|\boldsymbol{t}| = n$.

Every term variable x is equipped with a sort $\mathtt{s}_x$ and, as usual, terms equivalent modulo sort-preserving renaming of bound variables are identified. Let $\mathcal{V}(t)$ be the set of size variables in t, and $\mathrm{FV}(t)$ be the set of term variables free in t. Let $\theta, \sigma, \ldots$ denote term substitutions, *i.e.* functions from $\mathcal{X}$ to $\mathcal{T}$. For our previous examples, we have $\mathcal{CF}^{\square} = \{nat, list, bool\}$ and $\mathcal{F} = \mathcal{CF}^{\square} \cup \{0, s, /, nil, cons, insert, sort\}$.

Rewriting. Terms only built from variables and symbol applications $f\boldsymbol{t}$ are said to be *algebraic*. We assume given a set $\mathcal{R}$ of *rewrite rules* $l \to r$ such that l is algebraic, $l = f\boldsymbol{l}$ with $f \notin \mathcal{CF}^{\square}$ and $\mathrm{FV}(r) \subseteq \mathrm{FV}(l)$. Note that, while left hand-sides are algebraic and thus require syntactic matching only, right hand-sides may have abstractions and products. β-reduction and rewriting are defined as usual: $C[[x : T]u\ v] \to_{\beta} C[u\{x \mapsto v\}]$ and $C[l\sigma] \to_{\mathcal{R}} C[r\sigma]$ if $l \to r \in \mathcal{R}$. Let $\to = \to_{\beta} \cup \to_{\mathcal{R}}$ and $\to^*$ be its reflexive and transitive closure. Let $t \downarrow u$ iff there exists v such that $t \to^* v \ {}^*\!\leftarrow u$.

Typing. We assume that every symbol f is equipped with a sort $\mathtt{s}_f$ and a type $\tau_f = (\boldsymbol{x} : \boldsymbol{T})U$ such that, for all rules $f\boldsymbol{l} \to r \in \mathcal{R}$, $|\boldsymbol{l}| \leq |\boldsymbol{T}|$ (f is not applied to more arguments than the number of arguments given by τ_f). Let $\mathcal{F}^{\mathtt{s}}$ (resp. $\mathcal{X}^{\mathtt{s}}$) be the set of symbols (resp. variables) of sort $\mathtt{s}$. As usual, we distinguish the following classes of terms where t is any term:

- objects: $o ::= x \in \mathcal{X}^{\star} \mid f \in \mathcal{F}^{\star} \mid [x : t]o \mid ot$
- predicates: $p ::= x \in \mathcal{X}^{\square} \mid C^a \in \mathcal{CF}^{\square} \mid f \in \mathcal{F}^{\square} \setminus \mathcal{CF}^{\square} \mid [x : t]p \mid (x : t)p \mid pt$
- kinds: $K ::= \star \mid (x : t)K$

Examples of objects are the constructors of inductive types $0, s, nil, cons, \ldots$ and the function symbols $-, /, insert, sort, \ldots$. Their types are predicates: inductive types $bool, nat, list, \ldots$, logical connectors $\wedge, \vee, \ldots$, universal quantifications $(x : T)U, \ldots$ The types of predicates are kinds: $\star$ for types like $bool$ or nat, $\star \Rightarrow nat \Rightarrow \star$ for $list$, $\ldots$

An *environment* Γ is a sequence of variable-term pairs. An environment is *valid* if a term is typable in it. The typing rules of CACSA are given in Figure 4 and its subtyping rules in Figure 3. In (symb), φ is an arbitrary size substitution.

This reflects the fact that, in type declarations, size variables are implicitly universally quantified, like in ML. In contrast with [12], subtyping uses no sorting judgment. This simplification is justified in [5].

In comparison with [5], we added the side condition $\mathcal{V}(\boldsymbol{t}) = \emptyset$ in (size). It does not affect the properties proved in [5] and ensures that the size ordering is compatible with subtyping (Lemma 2). By the way, one could think of taking the more general rule $C^a\boldsymbol{t} \leq C^b\boldsymbol{u}$ with $\boldsymbol{t} \simeq_{\mathcal{A}} \boldsymbol{u}$. This would eliminate the need for equality constraints and thus simplify a little bit the constraint solving procedure. More generally, one could think in taking into account the monotony of type constructors by having, for instance, $list\ nat^a \leq list\ nat^b$ whenever $a \leq_{\mathcal{A}} b$. This requires extensions to Chen's work [12] and proofs of many non trivial properties of [5] again, like Theorem 1 below or subject reduction for β.

Fig. 3. Subtyping rules

$$\text{(refl)}\quad T \leq T \qquad \text{(size)}\quad C^a\boldsymbol{t} \leq C^b\boldsymbol{t} \quad (C \in \mathcal{CF}^{\square},\ a \leq_{\mathcal{A}} b,\ \mathcal{V}(\boldsymbol{t}) = \emptyset)$$

$$\text{(prod)}\quad \frac{U' \leq U \quad V \leq V'}{(x:U)V \leq (x:U')V'} \qquad \text{(conv)}\quad \frac{T' \leq U'}{T \leq U} \quad (T \downarrow T',\ U' \downarrow U)$$

$$\text{(trans)}\quad \frac{T \leq U \quad U \leq V}{T \leq V}$$

Fig. 4. Typing rules

$$\text{(ax)}\quad \vdash \star : \square \qquad \text{(prod)}\quad \frac{\Gamma \vdash U : \mathtt{s} \quad \Gamma, x:U \vdash V : \mathtt{s}'}{\Gamma \vdash (x:U)V : \mathtt{s}'}$$

$$\text{(size)}\quad \frac{\vdash \tau_C : \square}{\vdash C^a : \tau_C} \quad (C \in \mathcal{CF}^{\square},\ a \in \mathcal{A}) \qquad \text{(symb)}\quad \frac{\vdash \tau_f : \mathtt{s}_f}{\vdash f : \tau_f\varphi} \quad (f \notin \mathcal{CF}^{\square})$$

$$\text{(var)}\ \frac{\Gamma \vdash T : \mathtt{s}_x}{\Gamma, x:T \vdash x : T}\ (x \notin \mathrm{dom}(\Gamma)) \qquad \text{(weak)}\ \frac{\Gamma \vdash t : T \quad \Gamma \vdash U : \mathtt{s}_x}{\Gamma, x:U \vdash t : T}\ (x \notin \mathrm{dom}(\Gamma))$$

$$\text{(abs)}\quad \frac{\Gamma, x:U \vdash v : V \quad \Gamma \vdash (x:U)V : \mathtt{s}}{\Gamma \vdash [x:U]v : (x:U)V} \qquad \text{(app)}\quad \frac{\Gamma \vdash t : (x:U)V \quad \Gamma \vdash u : U}{\Gamma \vdash tu : V\{x \mapsto u\}}$$

$$\text{(sub)}\quad \frac{\Gamma \vdash t : T \quad \Gamma \vdash T' : \mathtt{s}}{\Gamma \vdash t : T'} \quad (T \leq T')$$

∞-Terms. An *∞-term* is a term whose only size annotations are ∞. In particular, it has no size variable. An *∞-environment* is an environment made of ∞-terms. This class of terms is isomorphic to the class of (unannotated) CAC terms. Our goal is to be able to infer annotated types for these terms, by using the size annotations given in the type declarations of constructors and function symbols $0, s, /, nil, cons, insert, sort, \ldots$

Since size variables are intended to occur in object type declarations only, and since we do not want matching to depend on size annotations, we assume that rules and type declarations of predicate symbols $nat, bool, list, \ldots$ are made of ∞-terms. As a consequence, we have:

Lemma 1. *– If $t \to_{\mathcal{R}} t'$ then, for all φ, $t\varphi \to_{\mathcal{R}} t'\varphi$.*
– If $\Gamma \vdash t : T$ then, for all φ, $\Gamma\varphi \vdash t\varphi : T\varphi$.

We make three important assumptions:

(1) $\mathcal{R}$ preserves typing: for all $l \to r \in \mathcal{R}$, Γ, T and σ, if $\Gamma \vdash l\sigma : T$ then $\Gamma \vdash r\sigma : T$. It is generally not too difficult to check this by hand. However, as already mentioned in [7], finding sufficient conditions for this to hold in general does not seem trivial.

(2) $\beta \cup \mathcal{R}$ is confluent. This is for instance the case if $\mathcal{R}$ is confluent and left-linear [24], or if $\beta \cup \mathcal{R}$ is terminating and $\mathcal{R}$ is locally confluent.

(3) $\beta \cup \mathcal{R}$ is terminating. In [7], it is proved that $\beta \cup \mathcal{R}$ is terminating essentially if, in every rule $f\boldsymbol{l} \to r \in \mathcal{R}$, recursive calls in r are made on terms whose size – by typing – are smaller than $\boldsymbol{l}$, by using lexicographic and multiset comparisons. Note that, with type-level rewriting, confluence is necessary for proving termination [8].

Important remark. One may think that there is some vicious circle here: we assume the termination for proving the decidability of type-checking, while type-checking is used for proving termination! The point is that termination checks are done incrementally. At the beginning, we can check that some set of rewrite rules $\mathcal{R}_1$ is terminating in the system with β only. Indeed, we do not need to use $\mathcal{R}_1$ in the type conversion rule (conv) for typing the terms of $\mathcal{R}_1$. Then, we can check in $\beta \cup \mathcal{R}_1$ that some new set of rules $\mathcal{R}_2$ is terminating, and so on...

Various properties of CACSA have already been studied in [5]. We refer the reader to this paper if necessary. For the moment, we just mention two important and non trivial properties based on Chen's work on subtyping with dependent types [12]: subject reduction for β and transitivity elimination:

Theorem 1 ([5]). *$T \leq U$ iff $T{\downarrow} \leq_s U{\downarrow}$, where $\leq_s$ is the restriction of $\leq$ to (refl), (size) and (prod).*

We now give some properties of the size and substitution orderings. Let $\to_{\mathcal{A}}$ be the confluent and terminating relation on $\mathcal{A}$ generated by the rule $s\infty \to \infty$.

Lemma 2. *Let $a{\downarrow}$ be the normal form of a w.r.t. $\to_{\mathcal{A}}$.*
– $a \simeq_{\mathcal{A}} b$ iff $a{\downarrow} = b{\downarrow}$.
– If $\infty \leq_{\mathcal{A}} a$ or $s^{k+1}a \leq_{\mathcal{A}} a$ then $a{\downarrow} = \infty$.
– If $a \leq_{\mathcal{A}} b$ and $\varphi \leq_{\mathcal{A}} \psi$ then $a\varphi \leq_{\mathcal{A}} b\psi$.
– If $\varphi \leq_{\mathcal{A}} \psi$ and $U \leq V$ then $U\varphi \leq V\psi$.

Note that ∞-terms are in $\mathcal{A}$-normal form. The last property (compatibility of size ordering wrt subtyping) follows from the restriction $\mathcal{V}(\boldsymbol{t}) = \emptyset$ in (size).

3 Decidability of typing

In this section, we prove the decidability of type inference and type-checking for ∞-terms under general assumptions that will be proved in Section 4. We begin with some informal explanations.

How to do type inference? The critical cases are (symb) and (app). In (symb), a symbol f can be typed by any instance of τ_f, and two different instances may be necessary for typing a single term (*e.g.* $s(sx)$). For type inference, it is therefore necessary to type f by its most general type, namely a renaming of τ_f with fresh variables, and to instantiate it later when necessary.

Assume now that we want to infer the type of an application tu. We naturally try to infer a type for t and a type for u using distinct fresh variables. Assume that we get T and U' respectively. Then, tu is typable if there is a size substitution φ and a product type $(x : P)Q$ such that $T\varphi \leq (x : P)Q$ and $U'\varphi \leq P$.

After Theorem 1, checking whether $A \leq B$ amounts to check whether $A{\downarrow} \leq_s B{\downarrow}$, and checking whether $A \leq_s B$ amounts to apply the (prod) rule as much as possible and then to check that (refl) or (size) holds. Hence, $T\varphi \leq (x : P)Q$ only if $T{\downarrow}$ is a product. Thus, the application tu is typable if $T{\downarrow} = (x : U)V$ and there exists φ such that $U'{\downarrow}\varphi \leq_s U\varphi$. Finding φ such that $A\varphi \leq_s B\varphi$ amounts to apply the (prod) rule on $A \leq_s B$ as much as possible and then to find φ such that (refl) or (size) holds. So, a subtyping problem can be transformed into a constraint problem on size variables.

We make this precise by first defining the constraints that can be generated.

Definition 1 (Constraints). Constraint problems *are defined as follows:*

$$\mathcal{C} ::= \bot \mid \top \mid \mathcal{C} \wedge \mathcal{C} \mid a = b \mid a \leq b$$

where $a, b \in \mathcal{A}$, $=$ *is commutative,* $\wedge$ *is associative and commutative,* $\mathcal{C} \wedge \mathcal{C} = \mathcal{C} \wedge \top = \mathcal{C}$ *and* $\mathcal{C} \wedge \bot = \bot$. *A finite conjunction* $\mathcal{C}_1 \wedge \ldots \wedge \mathcal{C}_n$ *is identified with* $\top$ *if* $n = 0$. *A constraint problem is in canonical form if it is neither of the form* $\mathcal{C} \wedge \top$, *nor of the form* $\mathcal{C} \wedge \bot$, *nor of the form* $\mathcal{C} \wedge \mathcal{C} \wedge \mathcal{D}$. *In the following, we always assume that constraint problems are in canonical form. An* equality *(resp.* inequality*)* problem *is a problem having only equalities (resp. inequalities). An inequality* $\infty \leq \alpha$ *is called an* ∞-inequality. *An inequality* $s^p\alpha \leq s^q\beta$ *is called a* linear inequality. *Solutions to constraint problems are defined as follows:*

- $S(\bot) = \emptyset$,
- $S(\top)$ *is the set of all size substitutions,*
- $S(\mathcal{C} \wedge \mathcal{D}) = S(\mathcal{C}) \cap S(\mathcal{D})$,
- $S(a = b) = \{\varphi \mid a\varphi = b\varphi\}$,
- $S(a \leq b) = \{\varphi \mid a\varphi \leq_{\mathcal{A}} b\varphi\}$.

Let $S^\ell(\mathcal{C}) = \{\varphi \mid \forall \alpha,\, \alpha\varphi{\downarrow} \neq \infty\}$ *be the set of* linear solutions.

We now prove that a subtyping problem can be transformed into constraints.

Lemma 3. *Let* $S(U, V)$ *be the set of substitutions* φ *such that* $U\varphi \leq_s V\varphi$. *We have* $S(U, V) = S(\mathcal{C}(U, V))$ *where* $\mathcal{C}(U, V)$ *is defined as follows:*

– $\mathcal{C}((x:U)V,(x:U')V') = \mathcal{C}(U',U) \wedge \mathcal{C}(V,V')$,
– $\mathcal{C}(C^a\boldsymbol{u}, C^b\boldsymbol{v}) = a \leq b \wedge \mathcal{E}^0(u_1,v_1) \wedge \ldots \wedge \mathcal{E}^0(u_n,v_n)$ *if* $|\boldsymbol{u}| = |\boldsymbol{v}| = n$,
– $\mathcal{C}(U,V) = \mathcal{E}^1(U,V)$ *in the other cases,*

and $\mathcal{E}^i(U,V)$ *is defined as follows:*

– $\mathcal{E}^i((x{:}U)V,(x{:}U')V') = \mathcal{E}^i([x{:}U]V,[x{:}U']V') = \mathcal{E}^i(UV,U'V')$
$= \mathcal{E}^i(U,U') \wedge \mathcal{E}^i(V,V')$,
– $\mathcal{E}^1(C^a,C^b) = a = b$,
– $\mathcal{E}^0(C^a,C^b) = a = b \wedge \infty \leq a$,
– $\mathcal{E}^i(c,c) = \top$ *if* $c \in \mathcal{S} \cup \mathcal{X} \cup \mathcal{F} \setminus \mathcal{CF}^\Box$,
– $\mathcal{E}^i(U,V) = \bot$ *in the other cases.*

Proof. First, we clearly have $\varphi \in S(\mathcal{E}^1(U,V))$ iff $U\varphi = V\varphi$, and $\varphi \in S(\mathcal{E}^0(U,V))$ iff $U\varphi = V\varphi$ and $\mathcal{V}(U\varphi) = \emptyset$. Thus, $S(U,V) = S(\mathcal{C}(U,V))$. □

Fig. 5. Type inference rules

$$\text{(ax)}\quad \Gamma \vdash^{\mathcal{Y}}_{\mathrm{a}} \star : \Box \qquad \text{(prod)}\quad \frac{\Gamma \vdash^{\mathcal{Y}}_{\mathrm{a}} U : \mathbf{s}_x \quad \Gamma, x:U \vdash^{\mathcal{Y}}_{\mathrm{a}} V : \mathbf{s}'}{\Gamma \vdash^{\mathcal{Y}}_{\mathrm{a}} (x:U)V : \mathbf{s}'}$$

$$\text{(size)}\quad \Gamma \vdash^{\mathcal{Y}}_{\mathrm{a}} C^\infty : \tau_C \quad (C \in \mathcal{CF}^\Box) \qquad \text{(symb)}\quad \Gamma \vdash^{\mathcal{Y}}_{\mathrm{a}} f : \tau_f\rho_{\mathcal{Y}} \quad (f \notin \mathcal{CF}^\Box)$$

$$\text{(var)}\quad \Gamma \vdash^{\mathcal{Y}}_{\mathrm{a}} x : x\Gamma \quad (x \in \mathrm{dom}(\Gamma)) \qquad \text{(abs)}\quad \frac{\Gamma \vdash^{\mathcal{Y}}_{\mathrm{a}} U : \mathbf{s}_x \quad \Gamma, x:U \vdash^{\mathcal{Y}}_{\mathrm{a}} v : V}{\Gamma \vdash^{\mathcal{Y}}_{\mathrm{a}} [x:U]v : (x:U)V}\ (V \neq \Box)$$

$$\text{(app)}\quad \frac{\Gamma \vdash^{\mathcal{Y}}_{\mathrm{a}} t : T \quad \Gamma \vdash^{\mathcal{Y}\cup\mathcal{V}(T)}_{\mathrm{a}} u : U'}{\Gamma \vdash^{\mathcal{Y}}_{\mathrm{a}} tu : V\varphi\rho_{\mathcal{Y}}\{x \mapsto u\}} \quad \begin{array}{c}(T{\downarrow} = (x:U)V,\ \mathcal{C} = \mathcal{C}(U'{\downarrow},U),\\ S(\mathcal{C}) \neq \emptyset,\ \varphi = mgs(\mathcal{C}))\end{array}$$

For renaming symbol types with variables outside some finite set of already used variables, we assume given a function ρ which, to every finite set $\mathcal{Y} \subseteq \mathcal{Z}$, associates an injection $\rho_{\mathcal{Y}}$ from $\mathcal{Y}$ to $\mathcal{Z} \setminus \mathcal{Y}$. In Figure 5, we define a type inference algorithm $\vdash^{\mathcal{Y}}_{\mathrm{a}}$ parametrized by a finite set $\mathcal{Y}$ of (already used) variables under the following assumptions:

(1) It is decidable whether $S(\mathcal{C})$ is empty or not.
(2) If $S(\mathcal{C}) \neq \emptyset$ then $\mathcal{C}$ has a most general solution $mgs(\mathcal{C})$.
(3) If $S(\mathcal{C}) \neq \emptyset$ then $mgs(\mathcal{C})$ is computable.

It would be interesting to try to give a modular presentation of type inference by clearly separating constraint generation from constraint solving, as it is done for ML in [25] for instance. However, for dealing with dependent types, one at least needs higher-order pattern unification. Indeed, assume that we have a constraint generation algorithm which, for a term t and a type (meta-)variable X, computes a set $\mathcal{C}$ of constraints on X whose solutions provide valid instances of X, *i.e.* valid types for t. Then, in (app), if the constraint generation gives $\mathcal{C}_1$ for $t : Y$ and $\mathcal{C}_2$ for $u : Z$, then it should give something like $\mathcal{C}_1 \wedge \mathcal{C}_2 \wedge (\exists U.\exists V.\ Y =_{\beta\eta} (x:U)Vx \wedge Z \leq U \wedge X =_{\beta\eta} Vu)$ for $tu : X$.

We now prove the correctness, completeness and minimality of $\vdash_{\mathrm{a}}^{\mathcal{Y}}$, assuming that symbol types are well sorted ($\vdash \tau_f : \mathtt{s}_f$ for all f).

Theorem 2 (Correctness). *If Γ is a valid ∞-environment and $\Gamma \vdash_{\mathrm{a}}^{\mathcal{Y}} t : T$, then $\Gamma \vdash t : T$, t is an ∞-term and $\mathcal{V}(T) \cap \mathcal{Y} = \emptyset$.*

Proof. By induction on $\vdash_{\mathrm{a}}^{\mathcal{Y}}$. We only detail the (app) case.

(app) By induction hypothesis, $\Gamma \vdash t : T$, $\Gamma \vdash u : U'$ and t and u are ∞-terms. Thus, tu is an ∞-term. By Lemma 1, $\Gamma \vdash t : T\varphi$ and $\Gamma \vdash u : U'\varphi$. Since $T\varphi{\downarrow} = (x : U\varphi)V\varphi$, we have $T\varphi \neq \Box$ and $\Gamma \vdash T\varphi : \mathtt{s}$. By subject reduction, $\Gamma \vdash (x : U\varphi)V\varphi : \mathtt{s}$. Hence, by (sub), $\Gamma \vdash t : (x : U\varphi)V\varphi$. By Lemma 3, $S(\mathcal{C}) = S(U'{\downarrow}, U)$ and $U'{\downarrow}\varphi \leq_s U\varphi$. Since $\Gamma \vdash U\varphi : \mathtt{s}'$, by (sub), $\Gamma \vdash u : U\varphi$. Therefore, by (app), $\Gamma \vdash tu : V\varphi\{x \mapsto u\}$ and $\Gamma \vdash tu : V\varphi\rho_{\mathcal{Y}}\{x \mapsto u\}$ since $\mathcal{V}(u) = \emptyset$. □

Theorem 3 (Completeness and minimality). *If Γ is an ∞-environment, t is an ∞-term and $\Gamma \vdash t : T$, then there are T' and ψ such that $\Gamma \vdash_{\mathrm{a}}^{\mathcal{Y}} t : T'$ and $T'\psi \leq T$.*

Proof. By induction on $\vdash$. We only detail some cases.

(symb) Take $T' = \tau_f\rho_{\mathcal{Y}}$ and $\psi = \rho_{\mathcal{Y}}^{-1}\varphi$.

(app) By induction hypothesis, there exist T, ψ_1, U' and ψ_2 such that $\Gamma \vdash_{\mathrm{a}}^{\mathcal{Y}} t : T$, $T\psi_1 \leq (x : U)V$, $\Gamma \vdash_{\mathrm{a}}^{\mathcal{Y}\cup\mathcal{V}(T)} u : U'$ and $U'\psi_2 \leq U$. By Lemma 2, $\mathcal{V}(U') \cap \mathcal{V}(T) = \emptyset$. Thus, $\mathrm{dom}(\psi_1) \cap \mathrm{dom}(\psi_2) = \emptyset$. So, let $\psi = \psi_1 \uplus \psi_2$. By Lemma 1, $T{\downarrow}\psi \leq_s (x : U{\downarrow})V{\downarrow}$. Thus, $T{\downarrow} = (x : U_1)V_1$, $U{\downarrow} \leq U_1\psi$ and $V_1\psi \leq V{\downarrow}$. Since $U'\psi \leq U$ and $U{\downarrow} \leq U_1\psi$, we have $U'{\downarrow}\,\psi \leq U_1\psi$ and, by Lemma 1, $U'{\downarrow}\,\psi \leq_s U_1\psi$. Thus, $\psi \in S(U'{\downarrow}, U_1)$. By Lemma 3, $S(U'{\downarrow}, U_1) = S(\mathcal{C})$ with $\mathcal{C} = \mathcal{C}(U'{\downarrow}, U_1)$. Thus, $S(\mathcal{C}) \neq \emptyset$ and there exists $\varphi = mgs(\mathcal{C})$. Hence, $\Gamma \vdash_{\mathrm{a}}^{\mathcal{Y}} tu : V_1\varphi\rho_{\mathcal{Y}}\theta$ where $\theta = \{x \mapsto u\}$. We are left to prove that there exists φ' such that $V_1\varphi\rho_{\mathcal{Y}}\theta\varphi' \leq V\theta$. Since $\varphi = mgs(\mathcal{C})$, there exists ψ' such that $\varphi\psi' \leq_{\mathcal{A}} \psi$. So, let $\varphi' = \rho_{\mathcal{Y}}^{-1}\psi'$. Since $\mathcal{V}(u) = \emptyset$, θ commutes with size substitutions. Since $V_1\psi \leq V{\downarrow} \leq V$, by Lemma 2, $V_1\varphi\rho_{\mathcal{Y}}\theta\varphi' = V_1\varphi\psi'\theta \leq V_1\psi\theta \leq V\theta$. □

Theorem 4 (Decidability of type-checking). *Let Γ be an ∞-environment, t be an ∞-term and T be a type such that $\Gamma \vdash T : \mathtt{s}$. Then, the problem of knowing whether there is ψ such that $\Gamma \vdash t : T\psi$ is decidable.*

Proof. The decision procedure consists in (1) trying to compute the type T' such that $\Gamma \vdash_{\mathrm{a}}^{\mathcal{Y}} t : T'$ by taking $\mathcal{Y} = \mathcal{V}(T)$, and (2) trying to compute $\psi = mgs(\mathcal{C}(T', T))$. Every step is decidable.

We prove its correctness. Assume that $\Gamma \vdash_{\mathrm{a}}^{\mathcal{Y}} t : T'$, $\mathcal{Y} = \mathcal{V}(T)$ and $\psi = mgs(\mathcal{C}(T', T))$. Then, $T'\psi \leq T\psi$ and, by Theorem 2, $\Gamma \vdash t : T'$. By Lemma 1, $\Gamma \vdash t : T'\psi$. Thus, by (sub), $\Gamma \vdash t : T\psi$.

We now prove its completeness. Assume that there is ψ such that $\Gamma \vdash t : T\psi$. Let $\mathcal{Y} = \mathcal{V}(T)$. Since Γ is valid and $\mathcal{V}(\Gamma) = \emptyset$, by Theorem 3, there are T' and φ such that $\Gamma \vdash_{\mathrm{a}}^{\mathcal{Y}} t : T'$ and $T'\varphi \leq T\psi$. This means that the decision procedure cannot fail ($\psi \uplus \varphi \in S(T', T)$). □

4 Solving constraints

In this section, we prove that the satisfiability of constraint problems is decidable, and that a satisfiable problem has a smallest solution. The proof is organized as follows. First, we introduce simplification rules for equalities similar to usual unification procedures (Lemma 4). Second, we introduce simplification rules for inequalities (Lemma 5). From that, we can deduce some general result on the form of solutions (Lemma 7). We then prove that a conjunction of inequalities has always a linear solution (Lemma 8). Then, by using linear algebra techniques, we prove that a satisfiable inequality problem has always a smallest solution (Lemma 11). Finally, all these results are combined in Theorem 5 for proving the assumptions of Section 3.

Let a *state* $\mathbb{S}$ be $\bot$ or a triplet $\mathcal{E}|\mathcal{E}'|\mathcal{C}$ where $\mathcal{E}$ and $\mathcal{E}'$ are conjunctions of equalities and $\mathcal{C}$ a conjunction of inequalities. Let $S(\bot) = \emptyset$ and $S(\mathcal{E}|\mathcal{E}'|\mathcal{C}) = S(\mathcal{E} \wedge \mathcal{E}' \wedge \mathcal{C})$ be the solutions of a state. A conjunction of equalities $\mathcal{E}$ is in *solved form* if it is of the form $\alpha_1 = a_1 \wedge \ldots \wedge \alpha_n = a_n$ $(n \geq 0)$ with the variables α_i distinct from one another and $\mathcal{V}(\boldsymbol{a}) \cap \{\boldsymbol{\alpha}\} = \emptyset$. It is identified with the substitution $\{\boldsymbol{\alpha} \mapsto \boldsymbol{a}\}$.

Fig. 6. Simplification rules for equalities

$$
\begin{array}{lrcl}
(1) & \mathcal{E} \wedge sa = sb \mid \mathcal{E}' \mid \mathcal{C} & \rightsquigarrow & \mathcal{E} \wedge a = b \mid \mathcal{E}' \mid \mathcal{C} \\
(2) & \mathcal{E} \wedge a = a \mid \mathcal{E}' \mid \mathcal{C} & \rightsquigarrow & \mathcal{E} \mid \mathcal{E}' \mid \mathcal{C} \\
(3) & \mathcal{E} \wedge a = s^{k+1}a \mid \mathcal{E}' \mid \mathcal{C} & \rightsquigarrow & \bot \\
(4) & \mathcal{E} \wedge \infty = s^{k+1}a \mid \mathcal{E}' \mid \mathcal{C} & \rightsquigarrow & \bot \\
(5) & \mathcal{E} \wedge \alpha = a \mid \mathcal{E}' \mid \mathcal{C} & \rightsquigarrow & \mathcal{E}\{\alpha \mapsto a\} \mid \mathcal{E}'\{\alpha \mapsto a\} \wedge \alpha = a \mid \mathcal{C}\{\alpha \mapsto a\} \text{ if } \alpha \notin \mathcal{V}(a)
\end{array}
$$

The simplification rules on equalities given in Figure 6 correspond to the usual simplification rules for first-order unification [19], except that substitutions are propagated into the inequalities.

Lemma 4. *The relation of Figure 6 terminates and preserves solutions: if* $\mathbb{S}_1 \rightsquigarrow \mathbb{S}_2$ *then* $S(\mathbb{S}_1) = S(\mathbb{S}_2)$. *Moreover, any normal form of* $\mathcal{E}|\top|\mathcal{C}$ *is either* $\bot$ *or of the form* $\top|\mathcal{E}'|\mathcal{C}'$ *with* $\mathcal{E}'$ *in solved form and* $\mathcal{V}(\mathcal{C}') \cap \mathrm{dom}(\mathcal{E}') = \emptyset$.

We now introduce a notion of graphs due to Pratt [26] that allows us to detect the variables that are equivalent to ∞. In the following, we use other standard techniques from graph combinatorics and linear algebra. Note however that we apply them on symbolic constraints, while they are generally used on numerical constraints. What we are looking for is substitutions, not numerical solutions. In particular, we do not have the constant 0 in size expressions (although it could be added without having to change many things). Yet, for proving that satisfiable problems have most general solutions, we will use some isomorphism between symbolic solutions and numerical ones (see Lemma 10).

Definition 2 (Dependency graph). *To a conjunction of linear inequalities* $\mathcal{C}$, *we associate a graph* $G_{\mathcal{C}}$ *on* $\mathcal{V}(\mathcal{C})$ *as follows. To every constraint* $s^p\alpha \leq s^q\beta$,

we associate the labeled edge $\alpha \xrightarrow{p-q} \beta$. *The* cost *of a path* $\alpha_1 \xrightarrow{p_1} \ldots \xrightarrow{p_k} \alpha_{k+1}$ *is* $\Sigma_{i=1}^{k} p_i$. *A* cyclic path *(*i.e. *when* $\alpha_{k+1} = \alpha_1$*) is* increasing *if its cost is* > 0.

Fig. 7. Simplification rules for inequalities

$$
\begin{array}{lrcll}
(1) & \mathcal{C} \wedge a \leq s^k \infty & \rightsquigarrow & \mathcal{C} & \\
(2) & \mathcal{C} \wedge \mathcal{D} & \rightsquigarrow & \mathcal{C} \wedge \{\infty \leq \alpha \mid \alpha \in \mathcal{V}(\mathcal{D})\} & \text{if } G_{\mathcal{D}} \text{ is increasing} \\
(3) & \mathcal{C} \wedge s^k \infty \leq s^l \alpha & \rightsquigarrow & \mathcal{C}\{\alpha \mapsto \infty\} \wedge \infty \leq \alpha & \text{if } \alpha \in \mathcal{V}(\mathcal{C})
\end{array}
$$

A conjunction of inequalities $\mathcal{C}$ is in *reduced form* if it is of the form $\mathcal{C}_\infty \wedge \mathcal{C}_\ell$ with $\mathcal{C}_\infty$ a conjunction of ∞-inequalities, $\mathcal{C}_\ell$ a conjunction of linear inequalities with no increasing cycle, and $\mathcal{V}(\mathcal{C}_\infty) \cap \mathcal{V}(\mathcal{C}_\ell) = \emptyset$.

Lemma 5. *The relation of Figure 7 on inequality problems terminates and preserves solutions. Moreover, any normal form is in reduced form.*

Lemma 6. *If* $\mathcal{C}$ *is a conjunction of inequalities then* $S(\mathcal{C}) \neq \emptyset$. *Moreover, if* $\mathcal{C}$ *is a conjunction of* ∞*-inequalities then* $S(\mathcal{C}) = \{\varphi \mid \forall \alpha \in \mathcal{V}(\mathcal{C}), \alpha\varphi{\downarrow} = \infty\}$.

Lemma 7. *Assume that* $\mathcal{E}|\top|\mathcal{C}$ *has normal form* $\top|\mathcal{E}'|\mathcal{C}'$ *by the rules of Figure 6, and* $\mathcal{C}'$ *has normal form* $\mathcal{D}$ *by the rules of Figure 7. Then,* $S(\mathcal{E} \wedge \mathcal{C}) \neq \emptyset$, $\mathcal{E}' = mgs(\mathcal{E})$ *and every* $\varphi \in S(\mathcal{E} \wedge \mathcal{C})$ *is of the form* $\mathcal{E}'(\upsilon \uplus \psi)$ *with* $\upsilon \in S(\mathcal{D}_\infty)$ *and* $\psi \in S(\mathcal{D}_\ell)$.

Proof. The fact that, in this case, $S(\mathcal{E}) \neq \emptyset$ and $\mathcal{E}' = mgs(\mathcal{E})$ is a well known result on unification [19]. Since $S(\mathcal{E} \wedge \mathcal{C}) = S(\mathcal{E}' \wedge \mathcal{D})$, $\mathcal{V}(\mathcal{E}') \cap \mathcal{V}(\mathcal{D}) = \emptyset$ and $S(\mathcal{D}) \neq \emptyset$, we have $S(\mathcal{E} \wedge \mathcal{C}) \neq \emptyset$. Furthermore, every $\varphi \in S(\mathcal{E} \wedge \mathcal{C})$ is of the form $\mathcal{E}'\varphi'$ since $S(\mathcal{E}' \wedge \mathcal{D}) \subseteq S(\mathcal{E}')$. Now, since $\mathcal{V}(\mathcal{D}_\infty) \cap \mathcal{V}(\mathcal{D}_\ell) = \emptyset$, $\varphi' = \upsilon \uplus \psi$ with $\upsilon \in S(\mathcal{D}_\infty)$ and $\psi \in S(\mathcal{D}_\ell)$. □

Hence, the solutions of a constraint problem can be obtained from the solutions of the equalities, which is a simple first-order unification problem, and from the solutions of the linear inequalities resulting of the previous simplifications.

In the following, let $\mathcal{C}$ be a conjunction of K linear inequalities with no increasing cycle, and L be the biggest label in absolute value in $G_\mathcal{C}$. We first prove that $\mathcal{C}$ has always a linear solution by using Bellman-Ford's algorithm.

Lemma 8. $S^\ell(\mathcal{C}) \neq \emptyset$.

Proof. Let $succ(\alpha) = \{\beta \mid \alpha \xrightarrow{p} \beta \in G_\mathcal{C}\}$ and $succ^*$ be the reflexive and transitive closure of $succ$. Choose $\gamma \in \mathcal{Z} \setminus \mathcal{V}(\mathcal{C})$, a set R of vertices in $G_\mathcal{C}$ such that $succ^*(R)$ covers $G_\mathcal{C}$, and a minimal cost $q_\beta \geq KL$ for every $\beta \in R$. Let the cost of a vertex α_{k+1} along a path $\alpha_1 \xrightarrow{p_1} \alpha_2 \xrightarrow{p_2} \ldots \alpha_{k+1}$ with $\alpha_1 \in R$ be $q_{\alpha_1} + \Sigma_{i=1}^{k} p_i$. Now, let ω_β be the maximal cost for β along all the possible paths from a vertex in R. We have $\omega_\beta \geq 0$ since there is no increasing cycle. Hence, for all edge $\alpha \xrightarrow{p} \beta \in G_\mathcal{C}$, we have $\omega_\alpha + p \leq \omega_\beta$. Thus, the substitution $\varphi = \{\alpha \mapsto s^{\omega_\alpha}\gamma \mid \alpha \in \mathcal{V}(\mathcal{C})\} \in S^\ell(\mathcal{C})$. □

We now prove that any solution has a more general linear solution. This implies that inequality problems are always satisfiable and that the satisfiability of a constraint problem only depends on its equalities.

Lemma 9. *If $\varphi \in S(\mathcal{C})$ then there exists $\psi \in S^\ell(\mathcal{C})$ such that $\psi \leq_{\mathcal{A}} \varphi$.*

We now prove that $S^\ell(\mathcal{C})$ has a smallest element. To this end, assume that inequalities are ordered and that $\mathcal{V}(\mathcal{C}) = \{\alpha_1, \ldots, \alpha_n\}$. We associate to $\mathcal{C}$ an adjacency-like matrix $M = (m_{i,j})$ with K lines and n columns, and a vector $v = (v_i)$ of length K as follows. Assume that the i-th inequality of $\mathcal{C}$ is of the form $s^p\alpha_j \leq s^q\alpha_k$. Then, $m_{i,j} = 1$, $m_{i,k} = -1$, $m_{i,l} = 0$ if $l \notin \{j, k\}$, and $v_i = q - p$. Let $P = \{z \in \mathbb{Q}^n \mid Mz \leq v, z \geq 0\}$ and $P' = P \cap \mathbb{Z}^n$.

To a substitution $\varphi \in S^\ell(\mathcal{C})$, we associate the vector z^φ such that z^φ_i is the natural number p such that $\alpha_i\varphi = s^p\beta$.

To a vector $z \in P'$, we associate a substitution φ_z as follows. Let $\{G_1, \ldots, G_s\}$ be the connected components of $G_{\mathcal{C}}$. For all i, let c_i be the component number to which α_i belongs. Let $\beta_1, \ldots, \beta_s$ be variables distinct from one another and not in $\mathcal{V}(\mathcal{C})$. We define $\alpha_i\varphi_z = s^{z_i}\beta_{c_i}$.

We then study the relations between symbolic and numerical solutions.

Lemma 10.
- *If $\varphi \in S^\ell(\mathcal{C})$ then $z^\varphi \in P'$. Furthermore, if $\varphi \leq_{\mathcal{A}} \varphi'$ then $z^\varphi \leq z^{\varphi'}$.*
- *If $z \in P'$ then $\varphi_z \in S^\ell(\mathcal{C})$. Furthermore, if $z \leq z'$ then $\varphi_z \leq_{\mathcal{A}} \varphi_{z'}$.*
- *$z^{\varphi_z} = z$ and $\varphi_{z^\varphi} \sqsubseteq \varphi$.*

Finally, we are left to prove that P' has a smallest element. The proof uses techniques from linear algebra.

Lemma 11. *There is a unique $z^* \in P'$ such that, for all $z \in P'$, $z^* \leq z$.*

An efficient algorithm for computing the smallest solution of a set of linear inequalities with at most two variables per inequality can be found in [23]. A more efficient algorithm can perhaps be obtained by taking into account the specificities of our problems.

Gathering all the previous results, we get the decidability.

Theorem 5 (Decidability). *Let $\mathcal{C}$ be a constraint problem. Whether $S(\mathcal{C})$ is empty or not can be decided in polynomial time w.r.t. the size of equalities in $\mathcal{C}$. Furthermore, if $S(\mathcal{C}) \neq \emptyset$ then $S(\mathcal{C})$ has a smallest solution that is computable in polynomial time w.r.t. the size of inequalities.*

5 Conclusion and related works

In Section 3, we give a general algorithm for type inference with size annotations based on constraint solving, that does not depend on the size algebra. For having completeness, we require satisfiable sets of constraints to have a computable most general solution. In Section 4, we prove that this is the case if the size algebra is

built from the symbols s and ∞ which, although simple, captures usual inductive definitions (since then the size corresponds to the number of constructors) and much more (see the introduction and [7]).

A natural extension would be to add the symbol $+$ in the size algebra, for typing list concatenation in a more precise way for instance. We think that the techniques used in the present work can cope with this extension. However, without restrictions on symbol types, one may get constraints like $1 \leq \alpha+\beta$ and loose the unicity of the smallest solution. We think that simple and general restrictions can be found to avoid such constraints to appear. Now, if symbols like $\times$ are added to the size algebra, then we lose linearity and need more sophisticated mathematical tools.

The point is that, because we consider dependent types and subtyping, we are not only interested in satisfiability but also in minimality and unicity, in order to have completeness of type inference [12]. There exist many works on type inference and constraint solving. We only mention some that we found more or less close to ours: Zenger's indexed types [32], Xi's Dependent[1] ML [30], Odersky *et al* 's ML with constrained types [25], Abel's sized types [1], and Barthe *et al* 's staged types [4]. We note the following differences:

Terms. Except [4], the previously cited works consider λ-terms *à la* Curry, *i.e.* without types in λ-abstractions. Instead, we consider λ-terms *à la* Church, *i.e.* with types in λ-abstractions. Note that type inference with λ-terms *à la* Curry and polymorphic or dependent types is not decidable. Furthermore, they all consider functions defined by fixpoint and matching on constructors. Instead, we consider functions defined by rewrite rules with matching both on constructor and defined symbols (*e.g.* associativity and distributivity rules).

Types. If we disregard constraints attached to types, they consider simple or polymorphic types, and we consider fully polymorphic and dependent types. Now, our data type constructors carry no constraints: constraints only come up from type inference. On the other hand, the constructors of Zenger's indexed data types must satisfy polynomial equations, and Xi's index variables can be assigned boolean propositions that must be satisfiable in some given model (*e.g.* Presburger arithmetic). Explicit constraints allow a more precise typing and more function definitions to be accepted. For instance (see [7]), in order for *quicksort* to have type $list^{\alpha} \Rightarrow list^{\alpha}$, we need the auxiliary *pivot* function to have type $nat^{\infty} \Rightarrow list^{\alpha} \Rightarrow list^{\beta} \times list^{\gamma}$ with the constraint $\alpha = \beta+\gamma$. And, if *quicksort* has type $list^{\infty} \Rightarrow list^{\infty}$ then a rule like $f\ (cons\ x\ l) \rightarrow g\ x\ (f\ (quicksort\ l))$ is rejected since $(quicksort\ l)$ cannot be proved to be smaller than $(cons\ x\ l)$. The same holds in [1,4].

Constraints. In contrast with Xi and Odersky *et al* who consider the constraint system as a parameter, giving DML(C) and HM(X) respectively, we consider a fixed constraint system, namely the one introduced in [3]. It is close to the one considered by Abel whose size algebra does not have ∞ but whose types have explicit bounded quantifications. Inductive types are indeed interpreted in the same way. We already mentioned also that Zenger considers polynomial

[1] By "dependent", Xi means constrained types, not full dependent types.

equations. However, his equivalence on types is defined in such a way that, for instance, $list^{\alpha}$ is equivalent to $list^{2\alpha}$, which is not very natural. So, the next step in our work would be to consider explicit constraints from an abstract constraint system. By doing so, Odersky *et al* get general results on the completeness of inference. Sulzmann [28] gets more general results by switching to a fully constrained-based approach. In this approach, completeness is achieved if every constraint can be represented by a type. With term-based inference and dependent types, which is our case, completeness requires minimality which is not always possible [12].

Constraint solving. In [4], Barthe *et al* consider system F with ML-like definitions and the same size annotations. Since they have no dependent type, they only have inequality constraints. They also use dependancy graphs for eliminating ∞, and give a specific algorithm for finding the most general solution. But they do not study the relations between linear constraints and linear programming. So, their algorithm is less efficient than [23], and cannot be extended to size annotations like $a + b$, for typing addition or concatenation.

Inference of size annotations. As already mentioned in the introduction, we do not infer size annotations for function symbols like [13,4]. We just check that function definitions are valid wrt size annotations, and that they preserve termination. However, finding annotations that satisfy these conditions can easily be expressed as a constraint problem. Thus, the techniques used in this paper can certainly be extended for inferring size annotations too. For instance, if we take $- : nat^{\alpha} \Rightarrow nat^{\beta} \Rightarrow nat^{X}$, the rules of $-$ given in the introduction are valid whenever $0 \leq X$, $\alpha \leq X$ and $X \leq X$, and the most general solution of this constraint problem is $X = \alpha$.

Acknowledgments. I would like to thank very much Miki Hermann, Hongwei Xi, Christophe Ringeissen and Andreas Abel for their comments on a previous version of this paper.

References

1. A. Abel. Termination checking with types. *Theoretical Informatics and Applications*, 38(4):277–319, 2004.
2. H. Barendregt. Lambda calculi with types. In S. Abramsky, D. Gabbay, and T. Maibaum, editors, *Handbook of logic in computer science*, volume 2. Oxford University Press, 1992.
3. G. Barthe, M. J. Frade, E. Giménez, L. Pinto, and T. Uustalu. Type-based termination of recursive definitions. *Mathematical Structures in Computer Science*, 14(1):97–141, 2004.
4. G. Barthe, B. Grégoire, and F. Pastawski. Practical inference for type-based termination in a polymorphic setting. In *Proceedings of the 7th International Conference on Typed Lambda Calculi and Applications*, Lecture Notes in Computer Science 3461, 2005.
5. F. Blanqui. Full version of [7]. Available on the web.

6. F. Blanqui. Rewriting modulo in Deduction modulo. In *Proceedings of the 14th International Conference on Rewriting Techniques and Applications*, Lecture Notes in Computer Science 2706, 2003.
7. F. Blanqui. A type-based termination criterion for dependently-typed higher-order rewrite systems. In *Proceedings of the 15th International Conference on Rewriting Techniques and Applications*, Lecture Notes in Computer Science 3091, 2004.
8. F. Blanqui. Definitions by rewriting in the Calculus of Constructions. *Mathematical Structures in Computer Science*, 15(1):37–92, 2005.
9. F. Blanqui. Full version. See `http://www.loria.fr/~blanqui/`, 2005.
10. F. Blanqui. Inductive types in the Calculus of Algebraic Constructions. *Fundamenta Informaticae*, 65(1-2):61–86, 2005.
11. V. Breazu-Tannen. Combining algebra and higher-order types. In *Proceedings of the 3rd IEEE Symposium on Logic in Computer Science*, 1988.
12. G. Chen. *Subtyping, Type Conversion and Transitivity Elimination*. PhD thesis, Université Paris VII, France, 1998.
13. W. N. Chin and S. C. Khoo. Calculating sized types. *Journal of Higher-Order and Symbolic Computation*, 14(2-3):261–300, 2001.
14. H. Comon. Solving symbolic ordering constraints. *International Journal of Foundations of Computer Science*, 1(4):387–412, 1990.
15. Coq-Development-Team. *The Coq Proof Assistant Reference Manual - Version 8.0*. INRIA Rocquencourt, France, 2004. `http://coq.inria.fr/`.
16. T. Coquand. An algorithm for testing conversion in type theory. In G. Huet and G. Plotkin, editors, *Logical Frameworks*, pages 255–279. Cambridge University Press, 1991.
17. T. Coquand and G. Huet. The Calculus of Constructions. *Information and Computation*, 76(2-3):95–120, 1988.
18. T. Coquand and C. Paulin-Mohring. Inductively defined types. In *Proceedings of the International Conference on Computer Logic*, Lecture Notes in Computer Science 417, 1988.
19. N. Dershowitz and J.-P. Jouannaud. Rewrite systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, chapter 6. North-Holland, 1990.
20. E. Giménez. Structural recursive definitions in type theory. In *Proceedings of the 25th International Colloquium on Automata, Languages and Programming*, Lecture Notes in Computer Science 1443, 1998.
21. J. Hughes, L. Pareto, and A. Sabry. Proving the correctness of reactive systems using sized types. In *Proceedings of the 23th ACM Symposium on Principles of Programming Languages, 1996*.
22. J.-P. Jouannaud and A. Rubio. The Higher-Order Recursive Path Ordering. In *Proceedings of the 14th IEEE Symposium on Logic in Computer Science*, 1999.
23. G. Lueker, N. Megiddo, and V. Ramachandran. Linear programming with two variables per inequality in poly-log time. *SIAM Journal on Computing*, 19(6):1000–1010, 1990.
24. F. Müller. Confluence of the lambda calculus with left-linear algebraic rewriting. *Information Processing Letters*, 41(6):293–299, 1992.
25. M. Odersky, M. Sulzmann, and M. Wehr. Type inference with constrained types. *Theory and Practice of Object Systems*, 5(1):35–55, 1999.
26. V. Pratt. Two easy theories whose combination is hard. Technical report, MIT, United States, 1977.
27. A. Schrijver. *Theory of linear and integer programming*. Wiley-Interscience Series in Discrete Mathematics and Optimization. John Wiley and Sons, 1986.

28. M. Sulzmann. A general type inference framework for Hindley/Milner style systems. In *Proceedings of the 5th Fuji International Symposium on Functional and Logic Programming*, Lecture Notes in Computer Science 2024, 2001.
29. D. Walukiewicz-Chrząszcz. Termination of rewriting in the Calculus of Constructions. *Journal of Functional Programming*, 13(2):339–414, 2003.
30. H. Xi. *Dependent types in practical programming.* PhD thesis, Carnegie-Mellon, Pittsburgh, United States, 1998.
31. H. Xi. Dependent types for program termination verification. *Journal of Higher-Order and Symbolic Computation*, 15(1):91–131, 2002.
32. C. Zenger. Indexed types. *Theoretical Computer Science*, 187(1-2):147–165, 1997.