Practical inference for typed-based termination in a polymorphic setting

Gilles Barthe[1], Benjamin Grégoire[1], and Fernando Pastawski[2⋆]

[1] INRIA Sophia-Antipolis, France
{Gilles.Barthe,Benjamin.Gregoire}@sophia.inria.fr
[2] FaMAF, Univ. Nacional de Córdoba, Argentina fernandopastawski@arnet.com.ar

Abstract. We introduce a polymorphic λ-calculus that features inductive types and that enforces termination of recursive definitions through typing. Then, we define a sound and complete type inference algorithm that computes a set of constraints to be satisfied for terms to be typable. In addition, we show that Subject Reduction fails in a naive use of typed-based termination for a λ-calculus *à la* Church, and we propose a general solution to this problem.

1 Introduction

Type-based termination is a method to enforce termination of recursive functions through typing, using an enriched type system that conveys size information about the inhabitants of inductive types. Tracing back to N. P. Mendler's work on inductive types [13], type-based termination offers several distinctive advantages over termination checking through syntactic criteria, e.g. the possibility for separate compilation. However, there is no widely used implementation of a type system that relies on type-based termination. There are two main reasons for the lack of such an implementation:

- termination of recursive definitions is mostly relevant for proof assistants, but less so for programming languages. However, proof assistants that enforce termination through syntactic means often rely on dependent type theories *à la* Church, for which type-based termination is not completely understood, despite preliminary work in this direction [4, 6, 11];
- the problem of size inference is seldom considered, and existing algorithms [7, 16] for inferring size annotations are complex, and typically return for every program an annotated type and a formula (in a decidable fragment of arithmetic) to be satisfied by the size variables.

The principal contribution of this article is to provide a practical inference algorithm for a polymorphic λ-calculus featuring type-based termination. The inference algorithm is scalable, in the sense that it computes size annotations efficiently, and returns types that are easily understood. We achieve practicality

⋆ Most of the work was done while supported by an INRIA International Internship.

by two means: first of all, we require recursive definitions to carry annotations about the dependency between the arguments and results of a recursive function definition. Due to the limited overhead they introduce (they should be compared to the {`struct` *id*} declarations in recursive definitions of Coq V8 [8]), we consider it perfectly acceptable to require users to provide such annotations when writing programs. Second of all, we consider a simple grammar for stage expressions: stage expressions are either stage variables, or ∞ (which denotes the maximal, closing stage of a datatype), or the successor of a stage expression. Thus our approach departs from the philosophy of many works on type-based termination which aim at typing as many terminating recursive definitions as possible. Nevertheless, our approach is more powerful than syntactic criteria for termination checking, see e.g. [5].

The combination of minimal annotations and a restricted language for stages enables us to guarantee that inferred types are of the form $C \Rightarrow \tau$, where τ is a type, and C is a finite conjunction of stage inequalities (our system features an obvious partial order on stages; hence the inference algorithm generates inequality constraints). The inference algorithm takes as input a pair (Γ, e) where Γ is a context and e is an expression, and returns an error if e is not typable in Γ, or a type τ *and* a set C of inequality between size expressions such that:

Soundness: for every stage substitution ρ satisfying C, we have $\rho\Gamma \vdash e : \rho\tau$.

Completeness: for every stage substitution ρ' such that $\rho'\Gamma \vdash e : \tau'$, there exists a stage substitution such that ρ satisfies C and $\rho\Gamma = \rho'\Gamma$ and $\rho\tau \preceq \tau'$, where the subtyping relation $\preceq$ is inherited from the partial order on stages.

Contents of the paper Section 2 introduces $F^{\frown}$, which gives the setting of our work. Section 3 introduces the inference algorithm and establishes its soundness and completeness. Section 3.5 briefly describes an implementation of the size inference algorithm for $F^{\frown}$. Section 4 provides an overview of related work. We conclude in Section 5.

2 System $F^{\frown}$

The system $F^{\frown}$ is an extension of System F with inductive types and type-based termination. As such, $F^{\frown}$ shares many features of $\lambda^{\frown}$ [5], from which it is inspired: simple stage arithmetic, implicit stage polymorphism, and a subtyping relation inherited from the partial order between stages. Unlike $\lambda^{\frown}$, the system $F^{\frown}$ features abstractions *à la* Church, i.e. of the form $\lambda x : \tau.e$, and $\mathsf{letrec}\ f : \sigma = e$.

2.1 Types and datatypes declarations

In order to achieve subject reduction and efficient type inference, we are led to consider three different families of types: size types, which record size information, bare types, which do not contain any size information and are used to tag variables in λ-abstractions, position types which are used to tag fixpoint variables in recursive definitions and indicate which positions in the type of a recursive function have the same stage variable as the recursive argument (we use a $\star$ to indicate those positions). The need for bare types and position types is further justified in Sections 2.4 and 3.1 below.

Types are built from type variables and datatypes using type constructors for universal quantification and function space. Formally, we assume given a set $\mathcal{X}$ of type variables, and a set $\mathcal{D}$ of datatype identifiers. Each datatype identifier comes equipped with an arity $\mathsf{ar}(d)$.

Definition 1 (Stages and types).

1. *The set* $\mathcal{S}$ *of* stage expressions *is given by the abstract syntax:*

$$s, r ::= \imath \mid \infty \mid \widehat{s}$$

where $\imath$ *ranges over stage variables. Stage substitution is defined in the obvious way, and we write* $s[\imath := s']$ *to denote the stage obtained by replacing* $\imath$ *by* s' *in* s*. Furthermore, the base stage of a stage expression is defined by the clauses* $\lfloor \imath \rfloor = \imath$ *and* $\lfloor \widehat{s} \rfloor = \lfloor s \rfloor$ *(the function is not defined on stages that contain* ∞*).*

2. *The sets* $\mathcal{T}$, $\overline{\mathcal{T}}$ *and* $\mathcal{T}^\star$ *of* bare types, (size) types*, and* position types *are given by the abstract syntaxes:*

$$\begin{array}{rl} \mathcal{T} ::= & \mathcal{X} \mid \mathcal{T} \to \mathcal{T} \mid \forall \mathcal{X}.\mathcal{T} \mid \mathcal{D}\ \boldsymbol{\mathcal{T}} \\ \overline{\mathcal{T}} ::= & \mathcal{X} \mid \overline{\mathcal{T}} \to \overline{\mathcal{T}} \mid \forall \mathcal{X}.\overline{\mathcal{T}} \mid \mathcal{D}^{\mathcal{S}}\ \overline{\boldsymbol{\mathcal{T}}} \\ \mathcal{T}^\star ::= & \mathcal{X} \mid \mathcal{T}^\star \to \mathcal{T}^\star \mid \forall \mathcal{X}.\mathcal{T}^\star \mid \mathcal{D}^\star\ \boldsymbol{\mathcal{T}}^\star \mid \mathcal{D}\ \boldsymbol{\mathcal{T}}^\star \end{array}$$

where in the clauses for datatypes, it is assumed that the length of the vectors $\boldsymbol{\mathcal{T}}$, $\overline{\boldsymbol{\mathcal{T}}}$, *and* $\boldsymbol{\mathcal{T}}^\star$ *is exactly the arity of the datatype.*

Types are related by a subtyping relation inherited from stages.

Definition 2 (Subtyping). *The relations* s is a substage of s', *written* $s \sqsubseteq s'$, *and* τ is a subtype of σ, *written* $\tau \preceq \sigma$, *are defined by the rules:*

$$\text{(refl)}\frac{}{s \sqsubseteq s} \qquad \text{(trans)}\frac{s \sqsubseteq r \quad r \sqsubseteq p}{s \sqsubseteq p} \qquad \text{(hat)}\frac{}{s \sqsubseteq \widehat{s}} \qquad \text{(infty)}\frac{}{s \sqsubseteq \infty}$$

$$\text{(refl)}\frac{}{\sigma \preceq \sigma} \qquad \text{(data)}\frac{s \sqsubseteq r \quad \boldsymbol{\tau} \preceq \boldsymbol{\tau'}}{d^s \boldsymbol{\tau} \preceq d^r \boldsymbol{\tau'}} \qquad \text{(func)}\frac{\tau' \preceq \tau \quad \sigma \preceq \sigma'}{\tau \to \sigma \preceq \tau' \to \sigma'} \qquad \text{(prod)}\frac{\sigma \preceq \sigma'}{\forall A.\sigma \preceq \forall A.\sigma'}$$

Remark that the rule *(infty)* imply that $\widehat{\infty} \sqsubseteq \infty$. We now turn to datatype declarations. We assume given a fixed set $\mathcal{C}$ of *constructors*, and a function $\mathsf{C} : \mathcal{D} \to \wp(\mathcal{C})$ such that $\mathsf{C}(d) \cap \mathsf{C}(d') = \emptyset$ for every distinct $d, d' \in \mathcal{D}$.

Definition 3.

1. *A size type* $\overline{\sigma} \in \overline{\mathcal{T}}$ *is a* $(d, \imath, \boldsymbol{A})$-constructor type *if it is of the form*

$$\forall \boldsymbol{B}.\overline{\theta}_1 \to \ldots \to \overline{\theta}_k \to d^{\widehat{\imath}}\ \boldsymbol{A}$$

and every $\overline{\theta}_i$ *belongs to the set* $\overline{\mathcal{T}}_{(d,\imath,\boldsymbol{A})} \subseteq \overline{\mathcal{T}}$ *defined by the syntax:*

$$\mathcal{X} \mid \overline{\mathcal{T}}_{(d,\imath,\boldsymbol{A})} \to \overline{\mathcal{T}}_{(d,\imath,\boldsymbol{A})} \mid \forall \mathcal{X}.\overline{\mathcal{T}}_{(d,\imath,\boldsymbol{A})} \mid d^{\imath}\ \boldsymbol{A} \mid d'^{\infty}\ \overline{\boldsymbol{\mathcal{T}}}_{(d,\imath,\boldsymbol{A})}$$

(Note that $\imath$ is the sole stage variable in $(d, \imath, \boldsymbol{A})$-constructor type.) In the sequel, we let $\mathsf{Inst}(\overline{\sigma}, s, \overline{\boldsymbol{\tau}}, \overline{\sigma}') = \overline{\theta}_1[\boldsymbol{A} := \overline{\boldsymbol{\tau}}, \imath := s] \to \ldots \to \overline{\theta}_k[\boldsymbol{A} := \overline{\boldsymbol{\tau}}, \imath := s] \to \sigma'$.

2. A datatype definition *is a declaration of the form:*

$$\textbf{Inductive}\ d\ \boldsymbol{A} := c_1 : \overline{\sigma}_1 \mid \ \ldots\ \mid c_n : \overline{\sigma}_n$$

where $\mathsf{C}(d) = \{c_1, \ \ldots\ , c_n\}$ *and* $\overline{\sigma}_1\ \ldots\ \overline{\sigma}_n$ *are* $(d, \imath, \boldsymbol{A})$*-constructor types in which* $\boldsymbol{A}, d^{\imath}\ \boldsymbol{A}$ *occur positively. We let* $\mathsf{Type}(c_i, s) = \forall \boldsymbol{A}.\overline{\sigma}_i[\imath := s]$ *and*

$$\mathsf{Inst}(c_i, s, \boldsymbol{\tau}, \sigma) = \mathsf{Inst}(\overline{\sigma}_i, s, \boldsymbol{\tau}, \sigma)$$

3. An environment is a set of datatype definitions $I_1\ \ldots I_n$ in which constructors of the datatype definition I_k only use datatypes introduced by $I_1 \ldots I_k$.

Note that the definition of constructor types and environments rules out heterogeneous and mutually recursive datatypes. Besides, the positivity requirement for $d^{\imath}\ \boldsymbol{A}$ is a standard assumption to guarantee strong normalization. Also, the positivity requirement for $\boldsymbol{A}$ is added to guarantee the soundness of the subtyping rule for datatypes, and to avoid considering polarity, as in e.g. [14]. See e.g. [5] for a definition of positivity.

In the remaining of the paper, we implicitly assume given an environment in which every $d \in \mathcal{D}$ has a corresponding datatype definition.

2.2 Terms and reduction

Terms are built from variables, abstractions, applications, constructors, case-expressions and recursive definitions. Formally we assume given a set $\mathcal{V}$ of *(object) variables.*

Definition 4 (Terms). *The set $\mathcal{E}$ of* terms *is given by the abstract syntax:*

$$e, e' ::= x \mid \lambda x : \tau.e \mid \Lambda A.e \mid e\ e' \mid e\ \tau \mid c \mid \mathsf{case}_{\tau}\ e'\ \mathsf{of}\ \{\boldsymbol{c} \Rightarrow \boldsymbol{e}\} \mid \mathsf{letrec}\ f : \tau^{\star} = e$$

Free and bound variables, substitution, etc. are defined as usual. We let $e[x := e']$ be the result of replacing all free occurrences of x in e with e'.

The reduction calculus is given by β-reduction for function application, $\imath$-reduction for case analysis and μ-reduction for unfolding recursive definitions, which is only allowed in the context of application to a constructor application.

Definition 5 (Reduction Calculus). *The notion of $\beta\imath\mu$-reduction $\to_{\beta\imath\mu}$ is defined as $\to_{\beta} \cup \to_{\imath} \cup \to_{\mu}$, where $\to_{\beta}$, $\to_{\imath}$, and μ-reduction are defined as the compatible closure of the rules*

$$\begin{array}{c}
(\lambda x : \tau.e)\ e' \ \to_{\beta}\ e[x := e'] \\
(\Lambda \alpha.e)\ \tau\ \to_{\beta}\ e[\alpha := \tau] \\
\mathsf{case}_{\tau}\ (c_i\ \boldsymbol{a})\ \mathsf{of}\ \{c_1 \Rightarrow e_1 \mid \ldots \mid c_n \Rightarrow e_n\}\ \to_{\imath}\ e_i\ \boldsymbol{a} \\
(\mathsf{letrec}\ f : \tau^{\star} = e)\ (c\ \boldsymbol{a})\ \to_{\mu}\ e[f := (\mathsf{letrec}\ f : \tau^{\star} = e)]\ (c\ \boldsymbol{a})
\end{array}$$

The rewrite system $\beta\imath\mu$ is orthogonal and confluent.

$$
\begin{array}{lll}
\text{(var)} & \dfrac{}{\overline{\Gamma}\vdash x:\overline{\sigma}} & \text{if } (x:\overline{\sigma})\in\overline{\Gamma} \\[2ex]
\text{(abs)} & \dfrac{\overline{\Gamma};x:\overline{\tau}\vdash e:\overline{\sigma}}{\overline{\Gamma}\vdash\lambda x:|\overline{\tau}|.e:\overline{\tau}\rightarrow\overline{\sigma}} & \\[2ex]
\text{(app)} & \dfrac{\overline{\Gamma}\vdash e:\overline{\tau}\rightarrow\overline{\sigma}\qquad\overline{\Gamma}\vdash e':\overline{\tau}}{\overline{\Gamma}\vdash e\;e':\overline{\sigma}} & \\[2ex]
\text{(T-abs)} & \dfrac{\overline{\Gamma}\vdash e:\overline{\sigma}}{\overline{\Gamma}\vdash\Lambda A.e:\forall A.\overline{\sigma}} & \text{if } A\notin\overline{\Gamma} \\[2ex]
\text{(T-app)} & \dfrac{\overline{\Gamma}\vdash e:\forall A.\overline{\sigma}}{\overline{\Gamma}\vdash e\;|\overline{\tau}|:\overline{\sigma}[A:=\overline{\tau}]} & \\[2ex]
\text{(cons)} & \dfrac{}{\overline{\Gamma}\vdash c:\mathsf{Type}(c,s)} & \text{if } c\in\mathsf{C}(d) \text{ for some } d \\[2ex]
\text{(case)} & \dfrac{\overline{\Gamma}\vdash e':d^{\widehat{s}}\overline{\boldsymbol{\tau}}\qquad\overline{\Gamma}\vdash e_i:\mathsf{Inst}(c_i,s,\overline{\boldsymbol{\tau}},\overline{\theta})\quad(1\leq i\leq n)}{\overline{\Gamma}\vdash\mathsf{case}_{|\overline{\theta}|}\;e'\;\mathsf{of}\;\{c_1\Rightarrow e_1\mid\ldots\mid c_n\Rightarrow e_n\}:\overline{\theta}} & \text{if } \mathsf{C}(d)=\{c_1,\ldots,c_n\} \\[2ex]
\text{(rec)} & \dfrac{\overline{\Gamma};f:d^{\imath}\overline{\boldsymbol{\tau}}\rightarrow\overline{\theta}\vdash e:d^{\widehat{\imath}}\overline{\boldsymbol{\tau}}\rightarrow\overline{\theta}[\imath:=\widehat{\imath}]}{\overline{\Gamma}\vdash\mathsf{letrec}\;f:\tau^{\star}=e:d^{s}\boldsymbol{\tau}\rightarrow\theta[\imath:=s]} & \text{if } \begin{cases}\imath \text{ not in } \overline{\Gamma},\overline{\boldsymbol{\tau}} \text{ and } \imath\;\mathsf{pos}\;\overline{\theta}\\ \tau^{\star} \text{ is } \imath-\text{compatible with } d^{\imath}\boldsymbol{\tau}\rightarrow\theta\end{cases} \\[2ex]
\text{(sub)} & \dfrac{\overline{\Gamma}\vdash e:\overline{\sigma}\qquad\overline{\sigma}\preceq\overline{\sigma}'}{\overline{\Gamma}\vdash e:\overline{\sigma}'} &
\end{array}
$$

Fig. 1. Typing rules for $F^{\frown}$

2.3 Typing rules

The type system is adapted from [5], and uses an erasure function $|.|$ that maps types to bare types by removing all size information.

Definition 6 (Contexts and judgments).

1. *A* context *is a finite sequence* $x_1 : \overline{\sigma}_1, \ldots, x_n : \overline{\sigma}_n$ *where* $x_1, \ldots, x_n$ *are pairwise disjoint (object) variables and* $\overline{\sigma}_1, \ldots, \overline{\sigma}_n$ *are (size) types.*
2. *A* typing judgment *is a triple of the form* $\overline{\Gamma} \vdash e : \overline{\sigma}$, *where* $\overline{\Gamma}$ *is a context,* e *is a term and* $\overline{\sigma}$ *is a (size) type.*
3. *A typing judgment is* derivable *if it can be inferred from the rules of Figure 1 where the positivity condition* $\imath\;\mathsf{pos}\;\overline{\sigma}$ *in the (rec) rule is defined in the usual way [5], and an annotated type* $\sigma^{\star}$ *is* $\imath$-compatible *with a size type* $\overline{\sigma}$ *if* $\sigma^{\star}$ *can be obtained from* $\overline{\sigma}$ *by replacing all stage annotations containing* $\imath$ *by* $\star$ *and by erasing all other size annotations.*
4. *A term* $e \in \mathcal{E}$ *is typable if* $\overline{\Gamma} \vdash e : \overline{\sigma}$ *is derivable for some context* $\overline{\Gamma}$ *and (size) type* $\overline{\sigma}$.

In the (rec) rule the side condition on $\imath$-compatibility is only relevant for the inference algorithm. Note also that the expression $\lambda x : \mathrm{Nat}.x$ has type $\mathrm{Nat}^s \rightarrow \mathrm{Nat}^s$ for all stages s. We refer to [5] for further explanations on the typing rules.

2.4 Subject reduction

The system $F^{\frown}$ enjoys subject reduction. The proof is routine.

Lemma 1. *If* $\overline{\Gamma} \vdash e : \overline{\tau}$ *and* $e \to_{\beta\iota\mu} e'$, *then* $\overline{\Gamma} \vdash e' : \overline{\tau}$.

On the contrary, subject reduction would fail if annotated terms were used to tag expressions. Indeed, consider for a moment that terms carry annotated types. With the obvious adaptation of the typing rules, we would be able to derive $y : \mathrm{Nat}^j \vdash M\ (\mathsf{s}\ y) : \mathrm{Nat}^j$, where M is

$$\mathsf{letrec}\ f : \mathrm{Nat}^{\imath} \to \mathrm{Nat}^{\imath} = \lambda x : \mathrm{Nat}^{\widehat{\imath}}.\ \mathsf{case}_{\mathrm{Nat}^{\imath}}\ x\ \mathsf{of}\ \{\mathsf{o} \Rightarrow \mathsf{o} \mid \mathsf{s}\ x' \Rightarrow (\lambda z : \mathrm{Nat}^{\imath}.z)\ x'\}$$

With the obvious adaptation of the reduction rules, one would also derive that $M\ (\mathsf{s}\ y)$ reduces to $(\lambda z : \mathrm{Nat}^{\imath}.z)y$. However, the latter is obviously ill-typed. Intuitively, the failure of subject reduction is caused by the combination of implicit stage polymorphism with explicit stage annotations in the tags of λ-expressions.

2.5 Strong normalization

Type-based termination is meant to enforce strong normalization of recursive definitions, therefore it is important to show that every typable expression in $F^{\frown}$ is strongly normalizing. The latter can be established by adapting standard model constructions based on saturated sets, see [5].

3 Inference algorithm

The purpose of this section is to define a sound and complete inference algorithm which computes for every context and expression a most general typing of the form $C \Rightarrow \overline{\sigma}$ where C is a set (i.e. conjunction) of constraints, and $\overline{\sigma}$ is an annotated type.

Definition 7 (Constraint and constraint systems).

1. *A* stage constraint *is a pair of stages, written* $s_1 \sqsubseteq s_2$.
2. *A* constraint system *is a finite set of stage constraints.*
3. *A stage substitution* ρ *satisfies a constraint system* C, *written* $\rho \models C$, *if for every constraint* $s_1 \sqsubseteq s_2$ *in* C, *we have* $\rho s_1 \sqsubseteq \rho s_2$.

Every constraint system has an obvious solution which maps all stage variables for ∞. Furthermore, each subtyping constraint $\overline{\sigma} \preceq \overline{\tau}$ which has a solution yields a constraint system C. In the sequel, we often write $\overline{\sigma} \preceq \overline{\tau}$ instead of C. Finally, we equate ∞ with $\widehat{\infty}$.

3.1 Discussion

Before describing the algorithm, we would like to illustrate two issues with efficient size annotation: the lack of unconstrained principal types, and the need for position types.

Lack of unconstrained principal types This issue is pervasive in type systems with subtyping, but we sketch how it can be cast it to our setting. Consider the expression twice, defined as $\lambda f : \mathrm{Nat} \to \mathrm{Nat}.\lambda x : \mathrm{Nat}.f\ (f\ x)$, and which can be given the typings $\tau_1 \equiv (\mathrm{Nat}^{\imath} \to \mathrm{Nat}^{\imath}) \to \mathrm{Nat}^{\imath} \to \mathrm{Nat}^{\imath}$ and also $\tau_2 \equiv (\mathrm{Nat}^{\imath} \to \mathrm{Nat}^{\infty}) \to \mathrm{Nat}^{\imath} \to \mathrm{Nat}^{\infty}$. If there would exist an unconstrained principal type $\tau_0 \equiv (\mathrm{Nat}^{s_1} \to \mathrm{Nat}^{s_2}) \to \mathrm{Nat}^{s_3} \to \mathrm{Nat}^{s_4}$, such that $\mathsf{twice} : \tau_0$ then we would have two stage substitutions ρ and ρ' such that $\rho\tau_0 \preceq \tau_1$ and $\rho'\tau_0 \preceq \tau_2$. Using the soundness of the inference algorithm (Proposition 5) and some elementary reasoning one can show that no such substitutions exist.
The same example can be used to show that the completeness statement of the inference algorithm must involve both subtyping and stage substitution.

Need for annotated types Consider the most general typings

$$\mathsf{letrec}\ f : \mathrm{Nat}^{\star} \to \mathrm{Nat}^{\star} = \lambda x : \mathrm{Nat}.\mathsf{o} : \mathrm{Nat}^{\imath} \to \mathrm{Nat}^{\imath}$$
$$\mathsf{letrec}\ f : \mathrm{Nat}^{\star} \to \mathrm{Nat} = \lambda x : \mathrm{Nat}.\mathsf{o} : \mathrm{Nat}^{\imath} \to \mathrm{Nat}^{\widehat{\jmath}}$$

Would we omit $\star$ in the tags of the recursive definitions, then $\mathsf{letrec}\ f : \mathrm{Nat} \to \mathrm{Nat} = \lambda x : \mathrm{Nat}.\mathsf{o}$ would not have a most general typing. Indeed, such a most general typing would be of the form $C \Rightarrow \mathrm{Nat}^{\imath_1} \to \mathrm{Nat}^{\imath_2}$ with the substitutions $[\imath_1 := \imath, \imath_2 := \imath] \models C$ and $[\imath_1 := \imath, \imath_2 := \widehat{\jmath}] \models C$. By elementary reasoning, we can show that it is impossible. In order to recover a most general typing, it will be required to allow both disjunctions and conjunctions of constraints in C. Unfortunately, disjunction makes types more complex to compute and to grasp.

3.2 Algorithm

The inference algorithm $\mathrm{Infer}(V, \overline{\Gamma}, t)$ take as input a context $\overline{\Gamma}$, an expression e and an auxiliary parameter V that represents the stage variables that have been previously used during inference (we need the latter to guarantee that we only introduce fresh variables). It returns a tuple $(V', C, \overline{\tau})$ where $\overline{\tau}$ is an annotated type, C is a constraint system, and V' is an extended set of stage variables that has been used by the algorithm. The invariants are $\mathsf{FV}(\overline{\Gamma}) \subseteq V$ and $V \subseteq V'$ and $\mathsf{FV}(C, \overline{\tau}) \subseteq V'$. For practical reasons, we also use a second algorithm $\mathrm{Check}(V, \overline{\Gamma}, t, \overline{\tau})$ which return a pair (V', C) ensuring that t has type $\overline{\tau}$ in environment $\overline{\Gamma}$. The invariants are $\mathsf{FV}(\overline{\Gamma}, \overline{\tau}) \subseteq V$ and $V \subseteq V'$ and $\mathsf{FV}(C) \subseteq V'$.

Definition 8. *The algorithms* Infer *and* Check *are defined in Figure 2.*

Note that the algorithms rely on auxiliary functions Annot, AnnotRec and RecCheck. $\mathrm{Annot}(\sigma, V) = (\overline{\sigma}, V')$ where V is a set of stage variables, σ a bare type. $\overline{\sigma}$ is an annotated version of σ with fresh stage variables (that do not belong to V) and $V' = V \bigcup \mathsf{FV}(\overline{\sigma})$. The only stage expressions that appear in $\overline{\sigma}$ are distinct and fresh stage variables. The second function AnnotRec is similar, but takes as input a position type instead of a bare type and returns $(\overline{\sigma}, V', V^{\star})$ where $V^{\star}$ is the set of fresh stage variables that appear in $\overline{\sigma}$ where there was a $\star$ in the original position type.

$$
\begin{array}{lcl}
\mathrm{Check}(V, \overline{\Gamma}, e, \overline{\tau}) & = & V_e, C_e \cup \overline{\tau}_e \preceq \overline{\tau} \\
 & & \text{where } (V_e, C_e, \overline{\tau}_e) := \mathrm{Infer}(\overline{\Gamma}, e) \\
\mathrm{Infer}(V, \overline{\Gamma}, x) & = & V, \emptyset, \overline{\Gamma}(x) \\
\mathrm{Infer}(V, \overline{\Gamma}, \lambda x : \tau_1.e) & = & V_e, C_e, \overline{\tau}_1 \to \overline{\tau}_2 \\
 & & \text{where } V_1, \overline{\tau}_1 := \mathrm{annot}(V, \tau_1) \\
 & & \qquad (V_e, C_e, \overline{\tau}_2) := \mathrm{Infer}(V_1, \overline{\Gamma}; x : \overline{\tau}_1, e) \\
\mathrm{Infer}(V, \overline{\Gamma}, \Lambda A.e) & = & V_e, C_e, \forall A.\overline{\tau} \\
 & & \text{where } (V_e, C_e, \overline{\tau}) := \mathrm{Infer}(V, \overline{\Gamma}, e) \\
 & & \text{if } A \text{ does not occur in } \overline{\Gamma} \\
\mathrm{Infer}(V, \overline{\Gamma}, e_1\, e_2) & = & V_2, C_1 \cup C_2, \overline{\tau} \\
 & & \text{where } (V_1, C_1, \overline{\tau}_2 \to \overline{\tau}) := \mathrm{Infer}(V, \overline{\Gamma}, e_1) \\
 & & \qquad (V_2, C_2) := \mathrm{Check}(V_1, \overline{\Gamma}, e_2, \overline{\tau}_2) \\
\mathrm{Infer}(V, \overline{\Gamma}, e\, \tau) & = & V_e, C_e, \overline{\tau}_e[A := \overline{\tau}] \\
 & & \text{where } V_1, \overline{\tau} := \mathrm{annot}(V, \tau) \\
 & & \qquad (V_e, C_e, \forall A.\overline{\tau}_e) := \mathrm{Infer}(V_1, \overline{\Gamma}, e) \\
\mathrm{Infer}(V, \overline{\Gamma}, c) & = & V \cup \{\alpha\}, , \mathsf{Type}(c, \alpha) \\
 & & \text{with } \alpha \notin V \\
\mathrm{Infer}(V, \overline{\Gamma}, \mathsf{case}_\theta\; e' \;\mathsf{of}\; \{\boldsymbol{c_i} \Rightarrow \boldsymbol{e_i}\}) & = & V_n, s \sqsubseteq \widehat{\alpha} \cup C_{e'} \cup \bigcup_{i=1}^{n} C_i, \overline{\theta} \\
 & & \text{where } \alpha \notin V \\
 & & \qquad V_\theta, \overline{\theta} := \mathrm{annot}(V \cup \{\alpha\}, \theta) \\
 & & \qquad (V_0, C_{e'}, d^s\overline{\boldsymbol{\tau}}) := \mathrm{Infer}(V_\theta, \overline{\Gamma}, e') \\
 & & \qquad (V_i, C_i) := \mathrm{Check}(V_{i-1}, \overline{\Gamma}, e_i, \mathsf{Inst}(c_i, \alpha, \overline{\boldsymbol{\tau}}, \overline{\theta})) \\
 & & \text{if } \mathsf{C}(d) = \{c_1, \ldots, c_n\} \\
\mathrm{Infer}(V, \overline{\Gamma}, \mathsf{letrec}\; f : d^\star\boldsymbol{\tau} \to \theta = e') & = & V_{e'}, C_f, d^\alpha\overline{\boldsymbol{\tau}} \to \overline{\theta} \\
 & & \text{where } (V_1, V^\star, d^\alpha\overline{\boldsymbol{\tau}} \to \overline{\theta}) := \mathrm{annotrec}(V, d^\star\boldsymbol{\tau} \to \theta) \\
 & & \qquad \widehat{\theta} := \overline{\theta}[\alpha_i := \widehat{\alpha_i}]_{\alpha_i \in V^\star} \\
 & & (V_{e'}, C_{e'}) := \mathrm{Check}(V_1, \overline{\Gamma}; f : d^\alpha\overline{\boldsymbol{\tau}} \to \overline{\theta}, e', d^{\widehat{\alpha}}\overline{\boldsymbol{\tau}} \to \widehat{\theta}) \\
 & & \qquad C_f := \mathrm{RecCheck}(\alpha, V^\star, V_1 \backslash V^\star, C_{e'} \cup \overline{\theta} \preceq \widehat{\theta})
\end{array}
$$

Fig. 2. Inference Algorithm

The function RecCheck is at the core of guaranteeing termination: it takes as input the set of constraints that has been inferred for the body of the recursive definition, and either returns an error if the definition is unsound w.r.t. the type system, or the set of constraints for the recursive definition, if the definition is sound w.r.t. the type system. Formally, the function RecCheck takes as input:

- the stage variable α which corresponds to the recursive argument, and which must be mapped to a fresh base stage $\imath$;
- a set of stage variables $V^\star$ that must be mapped to a stage expression with the same base stage as α. The set $V^\star$ is determined by the position types in the tag of the recursive definition. In particular, we have $\alpha \in V^\star$;
- a set of stage variables $V^{\neq}$ that must be mapped to a stage expression with a base stage different from $\imath$;

– a set of constraints C';

and returns an error or a set of constraints subject to the conditions below. To express these conditions concisely, we use $\rho \leq_{V_1}^{V_2} \rho'$ as a shorthand for $\rho(\alpha) \sqsubseteq \rho'(\alpha)$ for all $\alpha \in V_1$ and $\rho(\alpha) = \rho'(\alpha)$ for all $\alpha \in V_2$ and $\circ$ to denote the usual composition of substitutions. Furthermore, we freely use set notation.

Soundness (SRC): if $\mathrm{RecCheck}(\alpha, V^\star, V^{\neq}, C') = C$ then for all stage substitutions ρ such that $\rho \models C$ there exists a fresh stage variable $\imath$ and a stage substitution ρ' such that $\rho' \models C'$, and $\rho'(\alpha) = \imath$, and $[\imath := \rho(\alpha)] \circ \rho' \leq_{V^\star}^{V^{\neq}} \rho$, and $\lfloor \rho'(V^\star) \rfloor = \imath$, and $\lfloor \rho'(V^{\neq}) \rfloor \neq \imath$ (the last two formulas state respectively that for all $\alpha_i \in V^\star$, $\lfloor \rho'(\alpha_i) \rfloor = \imath$, and for all $\beta \in V^{\neq}$, $\lfloor \rho'(\beta) \rfloor \neq \imath$).

Completeness (CRC): if $\rho(\alpha) = \imath$ and $\lfloor \rho(V^\star) \rfloor = \imath$ and $\lfloor \rho(V^{\neq}) \rfloor \neq \imath$ and also $\rho \models C'$ then $\mathrm{RecCheck}(\alpha, V^\star, V^{\neq}, C')$ is defined and

$$\rho \models \mathrm{RecCheck}(\alpha, V^\star, V^{\neq}, C')$$

The algorithm for computing RecCheck, and the proofs of correctness, are given in Section 3.4.

3.3 Soundness and completeness

In order to establish soundness and completeness of the inference and checking algorithms, we must prove some preliminary lemmas. The first lemma provides a characterization of positivity which we use for dealing with the positivity constraints of the (rec) rule.

Lemma 2. $\imath \ \mathsf{pos}\ \overline{\tau} \Leftrightarrow \overline{\tau} \preceq \overline{\tau}[\imath := \widehat{\imath}]$

In the sequel, we use the notation $V \ \mathsf{pos}\ \overline{\tau}$ as a shorthand for $\overline{\tau} \preceq \overline{\tau}[\alpha := \widehat{\alpha}]_{\alpha \in V}$. The second and third lemmas are concerned with stage substitutions.

Lemma 3. *If* $\rho \leq_{V_1}^{V_2} \rho' \wedge \mathsf{FV}(\overline{\tau}) \subseteq V_1 \cup V_2 \wedge V_1 \ \mathsf{pos}\ \overline{\tau}$ *then* $\rho(\overline{\tau}) \preceq \rho'(\overline{\tau})$

Lemma 4. *If* $\lfloor \rho(V) \rfloor = \imath$ *then* $\rho(\overline{\tau})[\imath := \widehat{\imath}] = \rho(\overline{\tau}[\alpha_i := \widehat{\alpha_i}]_{\alpha_i \in V})$.

We now turn to soundness.

Lemma 5 (Soundness). *Check and Infer are sound:*

$$\begin{array}{rcl} \mathit{Check}(V, \overline{\Gamma}, e, \overline{\tau}) = (V', C) & \Rightarrow & \forall \rho \models C.\ \rho(\overline{\Gamma}) \vdash e : \rho(\overline{\tau}) \\ \mathit{Infer}(V, \overline{\Gamma}, e) = (V', C, \overline{\tau}) & \Rightarrow & \forall \rho \models C.\ \rho(\overline{\Gamma}) \vdash e : \rho(\overline{\tau}) \end{array}$$

Proof. We prove the two statements simultaneously. Note that the first statement can be deduced easily from the second one.

Indeed, assume that $\mathrm{Check}(V, \overline{\Gamma}, e, \overline{\tau}) = (V', C)$ and that $\rho \models C$. By definition of Check, we have $\mathrm{Infer}(V, \overline{\Gamma}, e) = (V', C_e, \overline{\tau}_e)$ and $\rho \models C_e \cup \overline{\tau}_e \preceq \overline{\tau}$, so $\rho \models C_e$ and $\rho(\overline{\tau}_e) \preceq \rho(\overline{\tau})$. Assuming the first statement, it follows that $\rho(\overline{\Gamma}) \vdash e : \rho(\overline{\tau}_e)$ and we conclude using the (sub) rule.

Therefore we focus on the proof of the second statement, by induction on the structure of e:

Cases $e \equiv x$ and $e \equiv \Lambda A.e_1$ $e \equiv c$. Trivial.

Cases $e \equiv \lambda x : \tau_1.e_1$ and $e \equiv t\,\tau_1$. The result follows from the induction hypothesis and the facts that both $|\text{annot}(\tau_1, V)| = \tau_1$ and $\rho(\overline{\tau}_e[A := \overline{\tau}_1]) = \rho(\overline{\tau}_e)[A := \rho(\overline{\tau}_1)]$.

Case $e \equiv e_1\, e_2$. By hypothesis, $\rho \models C_1 \cup C_2$ so $\rho \models C_1$ and $\rho \models C_2$. By the induction hypotheses, $\rho(\overline{\Gamma}) \vdash e_1 : \rho(\overline{\tau}_2) \to \rho(\overline{\tau})$ and $\rho(\overline{\Gamma}) \vdash e_2 : \rho(\overline{\tau}_2)$. We conclude using the (app) rule.

Case $e \equiv \mathsf{case}_\theta\ e'$ of $\{\boldsymbol{c_i} \Rightarrow \boldsymbol{e_i}\}$. By hypothesis, $\rho \models s \sqsubseteq \widehat{\alpha} \cup C_{e'} \cup \bigcup_{i=1}^{n} C_i$. By induction hypotheses, $\rho(\overline{\Gamma}) \vdash e' : \rho(d^s \boldsymbol{\tau})$ and for every $c_i \in \mathsf{C}(d)$

$$\rho(\overline{\Gamma}) \vdash e_i : \rho(\mathsf{Inst}(c_i, \alpha, \boldsymbol{\tau}, \overline{\theta})) = \mathsf{Inst}(c_i, \rho(\alpha), \rho(\boldsymbol{\tau}), \rho(\overline{\theta}))$$

Using the (sub) rule and the fact that $\rho(s) \sqsubseteq \rho(\widehat{\alpha})$, we derive

$$\rho(\overline{\Gamma}) \vdash e' : \rho(d^{\widehat{\alpha}} \boldsymbol{\tau})$$

and conclude using the (case) rule.

Case $e \equiv \mathsf{letrec}\ f : d^\star \boldsymbol{\tau} \to \theta = e'$. By hypothesis,

$$\rho \models \text{RecCheck}(\alpha, V^\star, V_1^{\neq}, C_{e'} \cup \overline{\theta} \preceq \widehat{\theta})$$

By (SRC), there exists $\imath$ and ρ' such that $\rho' \models C_{e'}$. Using the induction hypothesis with ρ', we get $\rho'(\overline{\Gamma}); f : d^\imath \rho'(\boldsymbol{\tau}) \to \rho'(\overline{\theta}) \vdash e' : d^{\widehat{\imath}} \rho'(\boldsymbol{\tau}) \to \rho'(\widehat{\theta})$. So by Lemma 4 : $\rho'(\overline{\Gamma}); f : d^\imath \rho'(\boldsymbol{\tau}) \to \rho'(\overline{\theta}) \vdash e' : d^\imath \rho'(\boldsymbol{\tau}) \to \rho'(\overline{\theta})[\imath := \widehat{\imath}]$.
Because $\rho' \models \overline{\theta} \preceq \widehat{\theta}$, we know that $\imath$ pos $\rho'(\overline{\theta})$ by lemma 2. We also know from (SRC) that $\imath \notin \rho'(\overline{\Gamma}), \rho'(\boldsymbol{\tau})$. Let s be $\rho(\alpha)$, by application of the (rec) typing rule we obtain $\rho'(\overline{\Gamma}) \vdash e : d^s \rho'(\boldsymbol{\tau}) \to \rho'(\overline{\theta})[\imath := s]$. Finally we use the facts that $[\imath := s] \circ \rho' \leq_{V^\star}^{V^{\neq}} \rho$, $V^\star$ pos $\overline{\theta}$ and Lemma 3 to conclude that e has the expected type.

We now turn to completeness. Note that, for the proofs to go through, the statement of completeness is slightly stronger than the one of the introduction: we require $\rho =_V \rho'$ instead of $\rho(\overline{\Gamma}) = \rho'(\overline{\Gamma})$, where $\rho =_V \rho'$ iff $\rho(\alpha) = \rho'(\alpha)$ for all $\alpha \in V$.

Lemma 6 (Completeness). *Check and Infer are complete:*

1. *If $\rho(\overline{\Gamma}) \vdash e : \rho\overline{\tau}$ and $\mathsf{FV}(\overline{\Gamma}, \overline{\tau}) \subseteq V$ then there exist V', C, ρ' such that $Check(V, \overline{\Gamma}, e, \overline{\tau}) = (V', C)$ and $\rho' \models C \wedge \rho =_V \rho'$.*
2. *If $\rho(\overline{\Gamma}) \vdash e : \overline{\theta}$ and $\mathsf{FV}(\overline{\Gamma}) \subseteq V$ there exist $V', C, \overline{\tau}, \rho'$ such that $Infer(V, \overline{\Gamma}, e) = (V', C, \overline{\tau})$ and $\rho' \models C$ and $\rho'(\overline{\tau}) \preceq \overline{\theta}$ and $\rho' =_V \rho$*

Proof. We prove the two statements simultaneously. Note that the first statement can be deduced easily from the second one. Indeed, assume that $\rho(\overline{\Gamma}) \vdash e : \rho\overline{\tau}$ with $\mathsf{FV}(\overline{\Gamma}, \overline{\tau}) \subseteq V$. Using the completeness of inference, there exists V', C, $\overline{\tau}_1$, ρ' such that $\text{Infer}(V, \overline{\Gamma}, e) = (V', C, \overline{\tau}_1)$, $\rho' \models C$, $\rho'(\overline{\tau}_1) \preceq \rho(\overline{\tau})$ and $\rho' =_V \rho$. Because $\mathsf{FV}(\overline{\tau}) \subseteq V$, we have $\rho(\overline{\tau}) = \rho'(\overline{\tau})$, so we conclude easily.

Therefore, we focus on the proof of the second statement, by induction on the typing derivation:

Cases (var) and (sub). Trivial.

Case (abs). Let $e = \lambda x : \overline{\tau}_1.e'$ and $(V_1, \overline{\tau}_1) = \text{annot}(V, |\overline{\sigma}_1|)$, by definition of annot, variables in $\overline{\tau}_1$ do not appear in V, so we can choose ρ_1 such that $\rho_1 =_V \rho$ and $\rho_1(\overline{\tau}_1) = \overline{\sigma}$. By induction hypotheses we have that $\text{Infer}(V_1, \overline{\Gamma}; x : \overline{\tau}_1, e') = (V', C, \overline{\tau}_2)$ and $\rho' \models C$, $\rho'(\overline{\tau}_2) \preceq \overline{\sigma}_2$ $\rho' =_{V_1} \rho_1$. It is easy to conclude.

Case (app), (T-abs), (T-app). Similar to (abs), just remark for the last one that $\overline{\tau} \preceq \overline{\sigma}$ implies $\overline{\tau}[A := \overline{\tau}'] \preceq \overline{\sigma}[A := \overline{\tau}']$.

Case (cons). Choose $\rho' = \rho[\alpha := s]$ and conclude.

Case (case). Let $e = \mathsf{case}_{|\overline{\sigma}|}\ e'\ \mathsf{of}\ \{\boldsymbol{c_i} \Rightarrow \boldsymbol{e_i}\}$ and $(V_\sigma, \overline{\sigma}') = \text{annot}(V \cup \{\alpha\}, |\overline{\sigma}|)$. Choose ρ_σ such that $\rho_\sigma =_V \rho$, $\rho_\sigma(\alpha) = s$ and $\rho_\sigma(\overline{\sigma}') = \overline{\sigma}$. By induction hypotheses, the type checking of branches succeeds. So we can easily conclude.

Case (rec). Let $e = \mathsf{fix}\ f : \tau^\star.e'$, with $\tau^\star$ is $\imath$-compatible with $d^\imath\boldsymbol{\overline{\tau}} \to \overline{\theta}$. Let $(V_1, V^\star, d^\alpha\boldsymbol{\overline{\tau}'} \to \overline{\theta}') = \text{annotrec}(V, \tau^\star)$ and choose ρ_1 such that $\rho_1 =_V \rho$, $\rho_1(d^\alpha\boldsymbol{\overline{\tau}'} \to \overline{\theta}') = d^\imath\boldsymbol{\overline{\tau}} \to \overline{\theta}$, in particular we have $\rho_1(\alpha) = \imath$. By induction hypothesis $\text{Check}(V_1, \overline{\Gamma}; f : d^\alpha\boldsymbol{\overline{\tau}'} \to \overline{\theta}', e', d^{\widehat{\alpha}}\boldsymbol{\overline{\tau}'} \to \widehat{\theta}') = (V_{e'}, C_{e'})$ and there exists $\rho_{e'} =_{V_1} \rho_1$ such that $\rho_{e'} \models C_{e'}$. By soundness,

$$\rho_{e'}(\overline{\Gamma}; f : d^\alpha\boldsymbol{\overline{\tau}'} \to \overline{\theta}') \vdash e' : \rho_{e'}(d^{\widehat{\alpha}}\boldsymbol{\overline{\tau}'} \to \widehat{\theta}')$$

By CRC, $\rho_{e'}$ is a solution for C_f. Choose $\rho' = [i := s] \circ \rho_{e'}$ to conclude.

3.4 Implementation of the RecCheck algorithm

Constraint systems can be represented by a graph where the nodes are stage variables and ∞, and the edges are integers. Concretely, each constraint in C is of the form $\infty \sqsubseteq \widehat{\alpha}^n$, or $\widehat{\alpha}_1^{n_1} \sqsubseteq \widehat{\alpha}_2^{n_2}$. In the first case, one adds an edge from α to ∞ labeled with 0, in the second case one adds an edge from α_2 to α_1 labeled with $n_2 - n_1$. Using Bellman's algorithm, one can then detect negative cycles in the graph, i.e. cycles where the sums of the edges are strictly negative. Such cycles imply $\widehat{\alpha}^{k+1} \sqsubseteq \alpha$, or equivalently $\infty \sqsubseteq \alpha$, for the stage variable α at the beginning of the cycle.

We now turn to the computation of $RecCheck(\alpha,\ V^\star,\ V^{\neq},\ C)$. Assuming that we intend to map α to a fresh stage variable $\imath$, the computation proceeds as follows:

1. it computes the downwards closed set $S_\imath$ of stage variables that must be mapped to a stage expression with base stage $\imath$. The rules are $V^\star \subseteq S_\imath$, and if $\alpha_1 \in S_\imath$ and $\widehat{\alpha}_2^{n_2} \sqsubseteq \widehat{\alpha}_1^{n_1} \in C$ then $\alpha_2 \in S_\imath$;
2. the algorithm must enforce that α is the smallest variable in $S_\imath$. It does so by adding to C the constraints $\alpha \sqsubseteq S_\imath$. Let $C_1 = C \cup \alpha \sqsubseteq S_\imath$;
3. the algorithm checks for negative cycles in the graph representation of C_1. Each time it finds such a cycle starting from β, the algorithm computes the set $V_{\sqsupseteq\beta}$ of variables greater or equal to β, removes all inequalities about variables in $V_{\sqsupseteq\beta}$ and adds the constraints $\infty \sqsubseteq V_{\sqsupseteq\beta}$. At the end of this step there are no more negative cycles in the graph, and we get a new set of constraints C_2;

4. the algorithm computes the upwards closed set $S_{\imath\sqsubseteq}$ of stage variables that must be mapped to ∞ or to a stage expression with base stage $\imath$. The rules are $S_\imath \subseteq S_{\imath\sqsubseteq}$ and if $\alpha_1 \in S_{\imath\sqsubseteq}$ and $\widehat{\alpha}_1^{n_1} \sqsubseteq \widehat{\alpha}_2^{n_2} \in C_2$ then $\alpha_2 \in S_{\imath\sqsubseteq}$;
5. the algorithm computes the upwards closed set $S_{\neg\imath}$ of stage variables that cannot be mapped to a stage expression with base stage $\imath$. The rules are $V^{\neq} \subseteq S_{\neg\imath}$ and if $\alpha_1 \in S_{\neg\imath}$ and $\widehat{\alpha}_1^{n_1} \sqsubseteq \widehat{\alpha}_2^{n_2} \in C_2$ then α_2 is in $S_{\neg\imath}$;
6. the algorithm sets all variables $\beta \in S_{\neg\imath} \cap S_{\imath\sqsubseteq}$ to ∞ (as in Step 3). At the end of this step we get a new set of constraints C_3;
7. the algorithm computes the upwards closed set S_∞ of stage variables that must be mapped to ∞. If $\infty \sqsubseteq \widehat{\beta}^k \in C_3$ then β is in S_∞, and if $\alpha_1 \in S_\infty$ and $\widehat{\alpha}_1^{n_1} \sqsubseteq \widehat{\alpha}_2^{n_2} \in C_3$ then α_2 is in S_∞.
8. if $S_\infty \cap S_\imath = \emptyset$ the algorithm returns the new set of constraints, else it fails.

Lemma 7 (Soundness of RecCheck). *The algorithm guarantees (SRC).*

Proof. Assume $\mathrm{RecCheck}(\alpha, V^\star, V^{\neq}, C') = C$ and let ρ be a stage substitution such that $\rho \models C$. Let $\imath$ be a fresh base stage that does not appear in the codomain of ρ, and let $C_\imath$ be the subset of C that only contains inequalities of the form $\widehat{\alpha}_1^{n_1} \sqsubseteq \widehat{\alpha}_2^{n_2}$ with both $\alpha_i \in S_{\imath\sqsubseteq}$ and $\rho(\alpha_i) \neq \infty$. By definition of the algorithm, there is no negative cycle in the graph representation of $C_\imath$. So we can use the Floyd-Warshall algorithm to compute the shortest path (SP) from α to each variable of $S_{\imath\sqsubseteq}$. Then we set

$$\rho'(\beta) = \begin{cases} \widehat{\imath}^{\mathrm{SP}(\beta)} \text{if } \beta \in S_\imath \vee (\beta \in S_{\imath\sqsubseteq} \wedge \rho(\beta) \neq \infty) \\ \rho(\beta) \text{ otherwise} \end{cases}$$

Let us show that ρ' has the expected properties. By construction, $\imath$ does not appear in $\rho(\beta)$ for $\beta \in V^{\neq}$ and ρ' verifies the first and last two properties required from it in (SRC); hence are left to verify that $\rho' \models C'$ and $[\imath := \rho(\alpha)] \circ \rho' \leq_{V^\star}^{V^{\neq}} \rho$.

We first prove $\rho' \models C'$. Note that constraints of the form $s \sqsubseteq \infty$ are trivially satisfied by ρ', and that $\rho'(s) = \rho(s)$ whenever there is a constraint of the form $\infty \sqsubseteq s$ in C' (in which case the constraint is derivable from C, and hence satisfied by ρ and ρ'). Hence we only need to focus on constraints of the form $\widehat{\beta}_1^{n_1} \sqsubseteq \widehat{\beta}_2^{n_2} \in C'$. We write $\beta \in S_{\mathrm{LP}}$ if $\beta \in S_\imath \vee (\beta \in S_{\imath\sqsubseteq} \wedge \rho(\beta) \neq \infty)$. We proceed by case analysis. There are three cases to treat:

- $\beta_1, \beta_2 \in S_{\mathrm{LP}}$. By definition of SP, the constraint is satisfied by ρ'.
- $\beta_1 \in S_{\mathrm{LP}}$ and $\beta_2 \notin S_{\mathrm{LP}}$. Since $\beta_2 \in S_{\imath\sqsubseteq}$, we must have $\rho(\beta_2) = \infty$. Therefore $\rho'(\beta_2) = \infty$ and the constraint is satisfied by ρ'.
- $\beta_1 \notin S_{\imath\sqsubseteq}$. Necessarily $\beta_2 \notin S_{\imath\sqsubseteq}$ and we conclude as for constraints of the form $\infty \sqsubseteq s$.

Now we prove that $[\imath := \rho(\alpha)] \circ \rho' \leq_{V^\star}^{V^{\neq}} \rho$:

- if $\beta \in V^{\neq}$, we must prove $\rho'(\beta) = \rho(\beta)$. By definition of ρ' it is sufficient to show for all $\beta \in V^{\neq}$ that $\beta \notin S_\imath$, and that if $\beta \in S_{\imath\sqsubseteq}$ then $\rho(\beta) = \infty$.
  For the first statement, we have $S_\imath \cap V^{\neq} = \emptyset$, and for the second statement, we have that if $\beta \in V^{\neq} \cap S_{\imath\sqsubseteq}$ then the algorithm has added a constraint $\infty \sqsubseteq \beta$ so that $\rho(\beta) = \infty$;

– if $\beta \in V^\star$, we must prove $\rho'(\beta)[\imath := \rho(\alpha)] \sqsubseteq \rho(\beta)$. As $\rho \models C$ we have

$$\rho'(\beta)[\imath := \rho(\alpha)] = \widehat{\rho(\alpha)}^{\mathrm{SP}(\beta)} \sqsubseteq \rho(\beta)$$

which concludes the proof.

Note that in the above SP is only used to establish the soundness of RecCheck.

Lemma 8 (Completeness of RecCheck). *The algorithm guarantees (CRC).*

Proof. Assume $\mathrm{RecCheck}(\alpha, V^\star, V^{\neq}, C') = C$ and let ρ be a stage substitution s.t. $\rho(\alpha) = \imath$ and $\lfloor \rho(V^\star) \rfloor = \imath$ and $\lfloor \rho(V^{\neq}) \rfloor \neq \imath$ and $\rho \models C'$. To show $\rho \models \mathrm{RecCheck}(\alpha, V^\star, V^{\neq}, C')$. We proceed by showing that the constraints introduced by each step of the algorithm are satisfied.

During Step (2), the algorithm adds the constraints $\alpha \sqsubseteq S_\imath$; because $\lfloor \rho(S_\imath) \rfloor = \imath$, and by construction of $S_\imath$, it follows that these constraints are satisfied by ρ.

During Step (3) the algorithm adds the constraints $\infty \sqsubseteq V_{\sqsupseteq \beta}$ each time it finds a negative cycle starting from β. By construction, negative cycles entail that $\widehat{\beta}^{k+1} \sqsubseteq \beta$ is derivable from C. Since ρ satisfies C, it must satisfy all inequalities derivable from C, so necessarily $\infty \sqsubseteq \rho(\beta)$, as expected.

During Step (6) the algorithm adds constraints $\infty \sqsubseteq S_{\neg\imath} \cap S_{\imath\sqsubseteq}$. However, a constraint $\infty \sqsubseteq \alpha$ is only added if one can derive from C that $\widehat{\beta}^k \sqsubseteq \widehat{\alpha}^{k'}$ with $\beta \in V^{\neq}$ and $\imath \sqsubseteq \alpha$. Since ρ satisfies C, it must satisfy all inequalities derivable from C, which forces $\infty \sqsubseteq \rho(\alpha)$, as expected.

3.5 Implementation issues and complexity

We have implemented the inference algorithm in OCaml. The implementation also supports global declarations, and performs some additional heuristics to increase readability and compactness of constraints. Roughly speaking, heuristics either try to remove constraints that involve stage variables not occurring in the inferred type (these stage variables are implicitly quantified existentially), or try to exploit positivity or negativity information about the positions in which stage variables occur (we do not loose any solutions by maximizing stage variables that occur negatively, or by minimizing stage variables that occur positively; in the first case it amounts replacing the variable by ∞).

The complexity of the RecCheck algorithm is in n^2 where n is the number of distinct stage variables used in the set of constraints, since it is equivalent to the complexity of Bellman algorithm.

4 Related work

The origins of type-based termination can be traced back to N. Mendler [13], who considered recursors and corecursors in a fixpoint-like style. The study of type-based termination was further pursued by H. Geuvers [9] who established a comparison between Mendler's formulation and the traditional formulation of recursors, and by E. Giménez [10, 11], who developed a variant of the Calculus of Inductive Definitions based on type-based termination, and established its relationship with the variant that uses a syntactic criterion for termination. Independently, J. Hughes, L. Pareto and A. Sabry [12] studied the use of size types to ensure productivity of reactive programs, and provided a type-checking algorithm for size types.

In an attempt to gain a better understanding of [11] and [12], several authors have studied the meta-theoretical properties of these systems or their variants. In particular, R. Amadio and S. Coupet-Grimal [2] considered type-based termination for coinductive types. Decidability of type-checking for this system is further established in [3]. More recently, G. Barthe, M.J. Frade, E. Giménez, L. Pinto and T. Uustalu [5] introduced $\lambda^{\widehat{}}$ and showed its strong normalization, but did not consider the issue of type checking. Furthermore, A. Abel [1] has studied type-based termination in a non-dependent setting and established decidability of type-checking for his type system (strictly speaking, his result only applies to terms in normal form, as a result of considering a calculus with domain-free abstractions). In a dependent setting, B. Barras [4] pursued the work of [11] by formalizing a variant of Giménez's system, and showed decidability of type-checking. Finally, F. Blanqui [6] considered a variant of the Calculus of Algebraic Constructions where termination of rewrite systems is enforced through typing.

However, these works do not consider the problem of inferring size annotations. To our best knowledge, the first inference algorithm for size types has been developed by W.-N. Chin and S.-C. Khoo [7], who provide an algorithm that infers for every program an annotated type and a formula in Presburger arithmetic to be satisfied by the size variables. Their type system is more powerful than ours, but has a greater complexity. In a similar line of work, H. Xi [16] proposes a system of restricted dependent types, built upon DML [17], to ensure program termination. As in [7], his system features a very rich system with stage arithmetic, and a notion of metric that is very useful to handle functions in several arguments. Expressiveness is achieved at the cost of practicality (some difficulties with scalability are discussed in [7]).

5 Conclusion

The main contribution of this paper is the definition and justification of a practical size inference algorithm for a polymorphic typed λ-calculus combining type-based termination and λ-abstractions *à la Church*. We have tried to motivate our choices in the design of the inference algorithm, and shown its soundness and completeness. Experimentations with an implementation of the algorithm suggests that the algorithm is practical.

In a companion article, we define a sound, complete and terminating size inference algorithm for a type-based variant of the Calculus of Inductive Constructions [15]. We also discuss some experiments with an implementation of the system, notably a formal proof of the correctness of sorting algorithms. Taken together, the results of the two articles demonstrate that it is feasible and indeed desirable that proof assistants such as Coq use type-based termination instead of syntactic termination criteria.

Future work includes implementing type-based termination for Coq, and experimenting with extensions that increase the expressiveness of the type system without compromising the practicality of the inference algorithm.

Acknowledgments Thanks to the anonymous referees and F. Blanqui for their comments on the paper.

References

1. A. Abel. Termination checking with types. *RAIRO– Theoretical Informatics and Applications*, 38:277–320, October 2004.
2. R. Amadio and S. Coupet-Grimal. Analysis of a guard condition in type theory. In M. Nivat, editor, *Proceedings of FOSSACS'98*, volume 1378 of *Lecture Notes in Computer Science*, pages 48–62. Springer-Verlag, 1998.
3. A. Bac. Un algorithme d'inférence de types pour les types coinductifs. Master's thesis, École Normale Supérieure de Lyon, 1998.
4. B. Barras. *Auto-validation d'un système de preuves avec familles inductives*. PhD thesis, Université Paris 7, 1999.
5. G. Barthe, M. J. Frade, E. Giménez, L. Pinto, and T. Uustalu. Type-based termination of recursive definitions. *Mathematical Structures in Computer Science*, 14:97–141, February 2004.
6. F. Blanqui. A type-based termination criterion for dependently-typed higher-order rewrite systems. In V. van Oostrom, editor, *Proceedings of RTA'04*, volume 3091 of *Lecture Notes in Computer Science*, pages 24–39, 2004.
7. W.-N. Chin and S.-C. Khoo. Calculating sized types. *Higher-Order and Symbolic Computation*, 14(2–3):261–300, September 2001.
8. Coq Development Team. *The Coq Proof Assistant User's Guide. Version 8.0*, January 2004.
9. H. Geuvers. Inductive and coinductive types with iteration and recursion. In B. Nordström, K. Pettersson, and G. Plotkin, editors, *Informal proceedings of Logical Frameworks'92*, pages 193–217, 1992.
10. E. Giménez. *Un calcul de constructions infinies et son application à la vérification de systèmes communicants*. PhD thesis, Ecole Normale Superieure de Lyon, 1996.
11. E. Giménez. Structural recursive definitions in Type Theory. In K.G. Larsen, S. Skyum, and G. Winskel, editors, *Proceedings of ICALP'98*, volume 1443 of *Lecture Notes in Computer Science*, pages 397–408. Springer-Verlag, 1998.
12. J. Hughes, L. Pareto, and A. Sabry. Proving the correctness of reactive systems using sized types. In *Proceedings of POPL'96*, pages 410–423. ACM Press, 1996.
13. N. P. Mendler. Inductive types and type constraints in the second-order lambda calculus. *Annals of Pure and Applied Logic*, 51(1-2):159–172, March 1991.
14. M. Steffen. *Polarized Higher-order Subtyping*. PhD thesis, Department of Computer Science, University of Erlangen, 1997.
15. B. Werner. *Méta-théorie du Calcul des Constructions Inductives*. PhD thesis, Université Paris 7, 1994.
16. H. Xi. Dependent Types for Program Termination Verification. In *Proceedings of LICS'01*, pages 231–242. IEEE Computer Society Press, 2001.
17. H. Xi and F. Pfenning. Dependent types in practical programming. In *Proceedings of POPL'99*, pages 214–227. ACM Press, 1999.