Type inference with structural subtyping: A faithful formalization of an efficient constraint solver

Vincent Simonet

INRIA Rocquencourt
Vincent.Simonet@inria.fr

Abstract. We are interested in type inference in the presence of *structural* subtyping from a pragmatic perspective. This work combines theoretical and practical contributions: first, it provides a faithful description of an efficient algorithm for solving and simplifying constraints; whose correctness is formally proved. Besides, the framework has been implemented in Objective Caml, yielding a generic type inference engine. Its efficiency is assessed by a complexity result and a series of experiments in realistic cases.

1 Introduction

1.1 Subtyping

Subtyping is a key feature of many type systems for programming languages. Previous works have shown how to integrate it into languages such as the simply typed λ-calculus, ML or Haskell. It appears as the basis of many object-oriented languages, e.g. [1]; and it allows designing fine-grained type systems for advanced programming constructs, e.g. [2]. It is also well suited for extending standard type systems in order to perform some static analysis, such as detection of uncaught exceptions [3], data [4] or information [5] flow analyses.

In all cases, subtyping consists of a partial order $\leq$ on types and a subsumption rule that allows every expression which has some type to be used with any greater type, as proposed by Mitchell [6] and Cardelli [7]. The subtyping order may reflect a variety of concepts: inclusion of mathematical domains, class hierarchy in object oriented languages, principals in security analyses, for instance.

As a consequence, the definition of the subtyping order itself varies. In this paper, we are interested in the case of *structural* subtyping, where comparable types must have the same shape and can only differ by their *atomic leaves*. (This contrasts with *non-structural* subtyping where different type constructors may be comparable; in particular, a least type $\bot$ and a greatest type $\top$ may be supplied. This also differs from *atomic* subtyping where there is no type constructor but only *atoms* belonging to some poset.) Structural subtyping is of particular interest when one intends to enrich a unification-based type system, such as ML's, with annotations belonging to a poset of *atoms*. In this case, the subtyping order may be simply defined by lifting the poset order along the existing type structure. Following the complexity study of Tiuryn [8] and Hoang and Mitchell [9], we will assume the atomic poset to be a lattice.

1.2 Type inference

Type inference consists in automatically determining the possible types of a piece of code. First, it allows type errors to be detected at compile time, without forcing programmers to include type annotations in programs. Second, a variety of program analyzes may be described as type inference processes.

The first algorithm for type inference with atomic subtyping was proposed by Mitchell [6, 10] and improved for handling structural subtyping by Fuh and Mishra [11, 12]. However, quoting Hoang and Mitchell [9], "*it has seen little if any practical use*", mostly because "*it is inefficient and the output, even for relatively simple input expressions, appears excessively long and cumbersome to read*". The theoretical complexity of the problem has been largely studied. Tiuryn [8] showed that deciding satisfiability of subtyping constraints between atoms is *PSPACE-hard* in the general case; and in the common case where the atomic poset is a disjoint union of lattices, it is solvable in linear time. Hoang and Mitchell [9] proved the equivalence of constraint resolution with typability (in the simply typed λ-calculus with subtyping). Frey [13] settled completely the complexity of the general case, showing it is *PSPACE-complete*. Lastly, Kuncak and Rinard [14] recently showed the first order theory of structural subtyping of non-recursive types to be decidable. Besides, in an attempt to overcome the practical limitation of the first algorithms, several researchers have investigated simplification heuristics in a variety of settings [15–18]. Rehof [19] also studies this question from a theoretical viewpoint, proving the existence of *minimal* typings in atomic subtyping and setting an exponential lower bound on their size. Simplification is useful because it may speed up type inference and make type information more readable for the user. However, to the best of our knowledge, only a small number of realistic type inference systems with *let*-polymorphism and (a flavor of) subtyping have been published [20–24].

Structural subtyping has also been studied throughout a series of specific applications. Among them, one may mention Foster's work about type qualifiers [25, 26] and the introduction of boolean constraints for binding-time analysis [27]. Both involve a type system with structural subtyping; however their type inference algorithm consists, in short, in expanding type structure and decomposing constraints without performing simplifications until obtaining a problem which involves only atoms and can be handled by an external solver. This contrasts with our approach which emphasizes the interlacing of resolution and simplification.

Type systems with subtyping associate not only a type to an expression but also a set of *constraints*, stating assumptions about the subtype order under which the typing is valid. Its presence is required by the desire to obtain most general (i.e. principal) statements which summarize all possible typings for a given expression, with a type compatibility notion which is not restricted to type instantiation, but includes all the coercions allowed by the subtyping order.

It is wise to decompose the design of a type synthesizer for a given language in two steps, whose implementation is in principle independent (although their execution is interlaced for efficiency). The first step consists in traversing the abstract syntax tree and generating types and constraints; this is in general straightforward but specific to the analysis at hand. The second step requires deciding the satisfiability of the generated constraints, and possibly rewriting them into a better but equivalent form. This remains independent from the language or even the type system itself, and relies only on the *constraint logic*. Hence, it may be delegated to a generic library taking care of constraint management.

2 Overview

In this paper, we address this question from a pragmatic perspective, our main objective resides in providing a generic, practical and scalable type inference engine featuring structural subtyping and polymorphism. Firstly, we give a faithful description of the algorithm and explain the main design choices, motivated by efficiency. Secondly, because previous works have proved this a delicate task and correctness of type inference is crucial, we provide a complete, formal proof of the framework. Lastly, we believe that experience is necessary for designing efficient techniques and advocating their scalability; so a complete implementation of the algorithms presented in this paper has been carried out in Objective Caml [28] and experienced on realistic cases. On our tests, it has shown to be slower only by a factor of 2 to 3 in comparison with standard unification (see Section 7 for details). Despite the large amount of work about structural subtyping in the literature, the faithful description of a complete type inference engine—which aims at efficiency, has been proved correct and also implemented apart from any particular application—forms an original contribution of the current work. We hope this will help such system to be widely used.

Our solving strategy sharply contrasts with some previous works about solving of subtyping constraints, e.g. [22], where a transitive closure of subtyping constraints is intertwined with decomposition. This yields a cubic-time algorithm, which cannot be improved because it performs a sort of dynamic transitive closure. It is interesting to draw a parallel with Heintze and McAllester's algorithm for control flow analysis (CFA) [29]. Whereas, in the standard approach for CFA, a transitive closure is dynamically interlaced with constraint generation, they propose a framework which first builds a certain graph and then performs a (demand-driven) closure. This gives a linear-time algorithm under the hypothesis of bounded-types. Similarly, our strategy for structural subtyping consists in postponing closure by first expanding the term structure and decomposing constraints until obtaining an atomic problem, as described in [11, 8]. Although expansion may theoretically introduce a number of variables exponential in the size of the input constraint, this behavior is rare and, under the hypothesis of bounded terms, this strategy remains *quasi-linear* under the hypothesis of bounded-terms (see section 5.5). This hypothesis may be discussed [30], but we believe it to be a good approximation for the practical examples: it captures the intuition that functions generally have limited number of arguments and order.

However, a simple algorithm performing expansion and decomposition on inequalities does not scale to the type-checking of real programs: despite the linear bound, expansion may in practice severely increase the size of the problem, affecting too much the overhead of the algorithm. To avoid this, we refine the strategy in several ways. Firstly, in order to deal with expansion and decomposition in an efficient manner, we enrich the original constraint language, which basically allows expressing a conjunction of inequalities, with new constructions, named *multi-skeletons*, that allow taking advantage of the structurality of subtyping by re-using techniques found in standard constraint-based unification frameworks [31]. Roughly speaking, multi-skeletons simultaneously express standard equalities between terms (written $=$) as well as equalities between term *structures* or *shapes* (written $\approx$); hence they allow performing *unification* on both. Secondly, in order to reduce as much as possible the number of variables to be expanded, we introduce several simplifications which must be performed *throughout the expansion process*. Lastly, we provide another set of simplifications which can be realized only at the end of solving. They allow the output of concise and readable typing information, and are most beneficial in the presence of *let*-polymorphism: generalization and instantiation require duplicating constraints, hence they must be made as compact as possible beforehand.

In addition to standard structural subtyping, our system is also equipped with less common features. First, it provides rows, which increase the expressiveness of the language and are useful for type-and-effect systems, see [3, 32]. Second, we deal with an original form of constraints, referred to as *weak inequalities*, which allow handling constraints such as *guards* [5] (see Section 3.3).

The remainder of the article is organized as follows. We begin by introducing the ground model of terms (Section 3) and, then, the first order logic in which constraints are expressed (Section 4). Section 5 describes the heart of the constraint resolution algorithm, and, in Section 6, we incorporate into this process simplification techniques, as discussed above. The implementation and experimental measures are presented in Section 7. The paper ends with some discussion about possible extensions. By lack of space, proofs of theorems are omitted, they can be found in the full version of the paper [33].

3 The ground algebra

The ground algebra is a logical model for interpreting constraints and type schemes. It consists in a set of *ground terms* and two binary relations: a subtyping order, $\leq$, and a “weak inequality”, $\sqsubset$.

3.1 Ground terms

Let $(\mathcal{A}, \leq_{\mathcal{A}})$ be a lattice whose elements, denoted by a, are the *atoms*. Let $\mathcal{C}$ and $\mathcal{L}$ be two denumerable sets of *type constructors* c and *row labels* ℓ, respectively. *Ground terms* and *kinds* are defined as follows:

$$\begin{array}{rl@{\qquad\qquad}rl}
t ::= & a & \kappa ::= & \mathsf{Atom} \\
\mid & c(t, \ldots, t) & \mid & \mathsf{Type} \\
\mid & \{\ell \mapsto t\}_{\ell \in L} & \mid & \mathsf{Row}_L\, \kappa
\end{array}$$

$$\frac{a \leq_{\mathcal{A}} a'}{a \leq a'} \qquad \frac{\begin{array}{c}\forall i \;\; v_i(c) \in \{\oplus, \odot\} \Rightarrow \tau_i \leq \tau_i' \\ \forall i \;\; v_i(c) \in \{\ominus, \odot\} \Rightarrow \tau_i \geq \tau_i'\end{array}}{c(\tau_1, \ldots, \tau_n) \leq c(\tau_1', \ldots, \tau_n')} \qquad \frac{\forall \ell \in L \;\; t_\ell \leq t_\ell'}{\{\ell \mapsto t_\ell\}_{\ell \in L} \leq \{\ell \mapsto t_\ell'\}_{\ell \in L}}$$

Fig. 1: Subtyping over ground terms

Ground terms include atomic constants a, which have kind Atom. Every type constructor c has a fixed arity n and a signature $\kappa_1, \ldots, \kappa_n$ which gives the expected kinds of its arguments. A *row* of kind $\mathsf{Row}_L\,\kappa$ is a mapping from labels in L to terms of kind κ, which is constant but on a finite subset of L.

3.2 Strong subtyping

The set of ground terms is equipped with the partial order $\leq$ defined in Figure 1, called *subtyping*, which relates terms of the same kind. On atoms, it is the order $\leq_{\mathcal{A}}$. Two comparable types must have the same head constructor c; moreover, their sub-terms must be related according to the *variances* of c: for i ranging from 1 to c's arity, $v_i(c)$ is one of $\oplus$ (*covariant*), $\ominus$ (*contravariant*) or $\odot$ (*invariant*).

This subtyping relation is *structural*: two comparable terms must share the same structure or *skeleton* and only their atomic annotations may differ. This leads us to introduce an equivalence relation $\approx$ on ground terms, which is nothing but the symmetric, transitive closure of $\leq$: $t_1 \approx t_2$ (read: t_1 *has the same skeleton as* t_2) if and only if $t_1\ (\leq \cup \geq)^*\ t_2$. Equivalence classes are referred to as *ground skeletons*. Roughly speaking, two terms have the same skeleton if they are equal, except within some non-invariant atomic annotations. In the absence of invariant arguments, skeletons would be identical to Tiuryn's shapes [8].

3.3 Weak inequalities

Type systems with structural subtyping may require constraints relating an arbitrary term to an atom, such as "*protected types*" [34] or "*guards*" ($\lhd$ or $\blacktriangleleft$) [5]. For instance, in [5], a constraint of the form $a \lhd t$ requires the constant a to be less than or equal to one or several atoms appearing in t, whose "positions" depend on the particular structure of t: $a \lhd \mathsf{int}^{a_1}$ and $a \lhd (t_1 + t_2)^{a_1}$ are equivalent to $a \leq a_1$ while $a \lhd \mathsf{int}^{a_1} \times \mathsf{int}^{a_2}$ holds if and only if $a \leq a_1$ and $a \leq a_2$, i.e. $a \leq a_1 \sqcap a_2$.

Our framework handles such constraints in an abstract and (as far as possible) general manner by the *weak inequality* $\sqsubset$ defined in figure 2. $\sqsubset$ relates ground terms of arbitrary kind by decomposing them until atoms are obtained, which are dealt with by rule ATOM. The other rules govern decomposition of the left-hand-side (TYPE-LEFT and ROW-LEFT) and the right-hand-side (TYPE-RIGHT and ROW-RIGHT) of a weak inequality. On a type constructor c, decomposition occurs on some of the sub-terms: we assume to non-disjoint subsets of $\{i \mid v_i(c) = \oplus\}$, $l(c)$ and $r(c)$. In short, a constraint $\tau_1 \sqsubset \tau_2$ is decomposable in a set of inequalities between some of the atoms appearing in τ_1 and some of τ_2 which are given by the respective structure of the two terms.

Although the rules defining $\sqsubset$ are not syntax-directed, they are however equivalences. In other words, all strategies for decomposing a weak inequality produce the same set of atomic inequalities. Moreover, it is worth noting that,

$$\frac{a_1 \leq_{\mathcal{A}} a_2}{a_1 \sqsubset a_2}\,\textsc{atom} \qquad \frac{\forall i \in l(c)\ \ t_i \sqsubset t'}{c(\bar{t}) \sqsubset t'}\,\textsc{type-left} \qquad \frac{\forall i \in r(c)\ \ t' \sqsubset t_i}{t' \sqsubset c(\bar{t})}\,\textsc{type-right} \qquad \frac{\forall \ell \in L\ \ t_\ell \sqsubset t'}{\{\ell \mapsto t_\ell\}_L \sqsubset t'}\,\textsc{row-left} \qquad \frac{\forall \ell \in L\ \ t' \sqsubset t_\ell}{t' \sqsubset \{\ell \mapsto t_\ell\}_L}\,\textsc{row-right}$$

Fig. 2: Weak inequalities

because $l(c)$ and $r(c)$ are non-disjoint and $\sqsubset$ matches only covariant sub-terms, it is transitive and $t_1 \sqsubset t_2 \leq t_3$ or $t_1 \leq t_2 \sqsubset t_3$ imply $t_1 \sqsubset t_3$.

4 The syntactic algebra

4.1 The first order logic

Terms and constraints are part of a first order logic interpreted in the ground algebra of Section 3. For every kind κ, we assume given a distinct denumerable set of *variables* denoted by α or β. On top of variables, we build two syntactic classes, *terms* and *hand-sides*:

$$\tau ::= \alpha \mid c(\tau, \ldots, \tau) \mid (\ell\colon \tau, \tau) \qquad\qquad \phi ::= \alpha \mid a$$

Terms include variables, type terms made up of a type constructor and a list of sub-terms, and row terms. For the latter, Rémy's [31] syntax is adopted: the term $(\ell\colon \tau, \tau')$ represents the row whose entry at index ℓ is τ and whose other entries are given by τ'. Hand-sides, which are either a variable (of arbitrary kind) or an atomic constant, shall appear in weak inequalities. Variables are interpreted in the model by *assignments* ρ that are total kind-preserving mappings from variables into ground terms; they are straightforwardly extended to terms and hand-sides.

Because constructed types and row terms are handled similarly in most of our development, it is convenient to introduce a common notation for them. (Indeed, only the unification algorithm described in Section 5.1, through the rule MUTATE, requires distinguishing them.) For this purpose, we let a *symbol* f be either a type constructor or a row label. If $f = c$ then $f(\tau_1, \ldots, \tau_n)$ stands for the type $c(\tau_1, \ldots, \tau_n)$ and, if $f = \ell$, then $f(\tau_1, \tau_2)$ stands for the row $(\ell\colon \tau_1, \tau_2)$. The notations for variance and weak inequality propagation introduced in Section 3 are extended to symbols accordingly. A *small term* is a term of height 0 or 1, i.e. either a variable α or a symbol with variable arguments $f(\alpha_1, \ldots, \alpha_n)$.

The formulas of the first order logic are *constraints* Γ:

$$\begin{aligned} \Gamma ::= {} & \langle \tau = \cdots = \tau \rangle^{\iota} \approx \cdots \approx \langle \tau = \cdots = \tau \rangle^{\iota} \\ \mid {} & \alpha \leq^{\iota} \alpha \mid \phi \sqsubset \phi \mid \mathbf{true} \mid \mathbf{false} \mid \Gamma \wedge \Gamma \mid \exists \alpha.\Gamma \end{aligned}$$

Constraints are interpreted in the ground algebra by a two place predicate $\cdot \vdash \cdot$ whose first argument is an assignment and whose second argument is a constraint. Its formal description can be found in the full version of the paper [33]; here, we prefer to explain their semantics in words.

First, $\langle \bar{\bar{\tau}}_1 \rangle^{\iota_1} \approx \cdots \approx \langle \bar{\bar{\tau}}_n \rangle^{\iota_n}$ is a *multi-skeleton*. It is a multi-set of *multi-equations* $\bar{\bar{\tau}}_1, \ldots, \bar{\bar{\tau}}_n$ each of which is decorated with a boolean flag $\iota_1, \ldots, \iota_n$. A multi-equation $\bar{\bar{\tau}}_i$ is a multi-set of terms written $\tau_{i,1} = \cdots = \tau_{i,n_i}$. All terms appearing in a multi-skeleton must have the same kind. The flags carried by multi-equations have no logical meaning; they are just needed by one step of constraint solving to store some termination-related data. Such a multi-skeleton is interpreted as follows: it requires that all the terms appearing in the multi-skeleton belong to a single ground skeleton and, moreover, that all terms of each multi-equation have the same interpretation. In this paper, a multi-skeleton is denoted by the meta-variable $\overset{\approx}{\tau}$, or $\overset{\approx}{\alpha}$ when it is known to contain only variables. If $\overset{\approx}{\tau}$ is $\langle\overset{=}{\tau}_1\rangle^{\iota_1} \approx \cdots \approx \langle\overset{=}{\tau}_n\rangle^{\iota_n}$ then $\langle\overset{=}{\tau}\rangle^{\iota} \approx \overset{\approx}{\tau}$ stands for the multi-skeleton $\langle\overset{=}{\tau}\rangle^{\iota} \approx \langle\overset{=}{\tau}_1\rangle^{\iota_1} \approx \cdots \approx \langle\overset{=}{\tau}_n\rangle^{\iota_n}$. We adopt similar notations for multi-equations and we write $\tau_1 \approx \tau_2$ and $\tau_1 = \tau_2$ for $\langle\tau_1\rangle^0 \approx \langle\tau_2\rangle^0$ and $\langle\tau_1 = \tau_2\rangle^0$, respectively.

A *strong inequality* $\alpha_1 \leq^{\iota} \alpha_2$ involves a pair of variables of the same kind and a boolean flag (which has the same use as those carried by multi-equations) interpreted in the ground algebra by the subtyping order $\leq$. A weak inequality $\phi_1 \sqsubset \phi_2$ consists of a pair of hand-sides. Atomic constants, which are not part of the terms grammar, can be encoded by these constraints: the atom a may be represented in the syntactic algebra by a "fresh" variable α of kind Atom and the pair of constraints $a \sqsubset \alpha$ and $\alpha \sqsubset a$.

Other constructs allow conjunction and existential quantification of constraints. The latter allows the introduction of intermediate variables, by the type checker during the constraint generation or by the solver itself for the purpose of resolution.

One may argue that the constraint language is not minimal. Indeed, multi-equations and multi-skeletons may be encoded using strong inequalities: on may prove that $\tau_1 \approx \tau_2$ and $\tau_1 = \tau_2$ are respectively equivalent to $\exists\beta.[\beta \leq \tau_1 \wedge \beta \leq \tau_2]$ and $\tau_1 \leq \tau_2 \wedge \tau_2 \leq \tau_1$. However, such an encoding is not practical, because multi-skeletons and multi-equations allow much more efficient representation and manipulation of constraints: they allow to benefit of the efficiency of unification-based algorithms, throughout the solving process. Indeed, in most applications, the client of the solver generates inequalities, while multi-skeletons belong only to the solver's internal representation of constraints and are introduced throughout the solving process. There is another slight redundancy in the logic: if α_1 and α_2 are two variables of kind Atom, the constraints $\alpha_1 \leq \alpha_2$ and $\alpha_1 \sqsubset \alpha_2$ are equivalent. By convention, in the remainder of the paper, any occurrence of the former must be read as an instance of the latter.

We omit the sorting rules necessary to ensure that terms and constraints are well-formed, which are standard. Moreover, we consider constraints modulo the commutativity and associativity of $\wedge$; and α-conversion and scope-extrusion of existential quantifications. (A complete specification of constraint kinding and equivalence is given in [33].) A constraint is $\exists$-*free* if it is a conjunction of multi-skeletons and inequalities. In the following, we write $\Gamma \doteq \exists\vec{\alpha}.\Gamma'$ if and only if $\exists\vec{\alpha}.\Gamma'$ is a representative of Γ such that Γ' is $\exists$-free. Every constraint admits such a representation.

Given an $\exists$-free constraint Γ, we let the predicate $\tau \approx \tau' \in \Gamma$ (resp. $\tau = \tau' \in \Gamma$) hold if and only if τ and τ' appear in the same multi-skeleton (resp. multi-equation) in Γ. Similarly, we write $\alpha_1 \leq \alpha_2 \in \Gamma$ if a constraint $\alpha_1 \leq^{\iota} \alpha_2$ appears within Γ, and $\bar{\bar{\alpha}}_1 \leq \bar{\bar{\alpha}}_2 \in \Gamma$ if $\alpha_1 \leq \alpha_2 \in \Gamma$ for some α_1 in $\bar{\bar{\alpha}}_1$ and α_2 in $\bar{\bar{\alpha}}_2$. The corresponding notations are also adopted for $\sqsubset$.

$$\textsc{generate}^{\leq}\ \ \frac{\alpha \leq^1 \beta}{\alpha \leq^0 \beta \wedge \alpha \approx \beta}\rightsquigarrow
\qquad
\textsc{fuse}^{\approx}\ \ \frac{\langle \alpha = \overset{=}{\tau} \rangle^{\iota} \approx \overset{\approx}{\tau} \wedge \langle \alpha = \overset{=}{\tau}' \rangle^{\iota'} \approx \overset{\approx}{\tau}'}{\langle \alpha = \overset{=}{\tau} = \overset{=}{\tau}' \rangle^{\iota \vee \iota'} \approx \overset{\approx}{\tau} \approx \overset{\approx}{\tau}'}\rightsquigarrow
\qquad
\textsc{fuse}^{=}\ \ \frac{\langle \alpha = \overset{=}{\tau} \rangle^{\iota} \approx \langle \alpha = \overset{=}{\tau}' \rangle^{\iota'} \approx \overset{\approx}{\tau}}{\langle \alpha = \overset{=}{\tau} = \overset{=}{\tau}' \rangle^{\iota \vee \iota'} \approx \overset{\approx}{\tau}}\rightsquigarrow$$

$$\textsc{decompose}^{\approx}\ \ \frac{\langle \overset{=}{\tau} = f(\vec{\alpha}) \rangle^1 \approx \langle \overset{=}{\tau}' = f(\vec{\beta}) \rangle^1 \approx \overset{\approx}{\tau}}{\langle \overset{=}{\tau} = f(\vec{\alpha}) \rangle^1 \approx \langle \overset{=}{\tau}' = f(\vec{\beta}) \rangle^0 \approx \overset{\approx}{\tau} \wedge_i \alpha_i \approx^{v_i(f)} \beta_i}\rightsquigarrow
\qquad
\textsc{decompose}^{=}\ \ \frac{\langle \overset{=}{\tau} = f(\vec{\alpha}) = f(\vec{\beta}) \rangle^{\iota} \approx \overset{\approx}{\tau}}{\langle \overset{=}{\tau} = f(\vec{\alpha}) \rangle^{\iota} \approx \overset{\approx}{\tau} \wedge_i \alpha_i = \beta_i}\rightsquigarrow$$

$$\textsc{mutate}\ \ \frac{\overset{\approx}{\tau} \approx \langle \overset{=}{\tau} = \ell_b \colon \tau_b, \tau' \rangle^{\iota}}{\exists \alpha\alpha'.[\overset{\approx}{\tau} \approx \langle \overset{=}{\tau} = \ell_a \colon \tau_a, \alpha' \rangle^1 \wedge \langle \alpha' = \ell_b \colon \tau_b, \alpha \rangle^1 \wedge \langle \tau' = \ell_a \colon \tau_a, \alpha \rangle^1]}\rightsquigarrow
\quad
\begin{array}{r}(\text{if } \ell_a \in \mathsf{Roots}(\overset{=}{\tau}, \overset{\approx}{\tau}) \\ \text{and } \ell_a <_{\mathcal{L}} \ell_b)\end{array}$$

$$\textsc{generalize}\ \ \frac{\overset{\approx}{\tau}[\tau/\alpha]}{\exists \alpha.[\overset{\approx}{\tau} \wedge \langle \alpha = \tau \rangle^1]}\rightsquigarrow
\quad
\begin{array}{l}(\text{if } \tau \text{ is not a variable and} \\ \alpha \text{ occurs in a non-variable term of } \overset{\approx}{\tau} \text{ but not in } \tau)\end{array}$$

Fig. 3: Unification (rewriting system Ω_u)

Let Γ_1 and Γ_2 be two constraints. Γ_1 *implies* Γ_2 (we write: $\Gamma_1 \models \Gamma_2$) if and only if every assignment which satisfies Γ_1 also satisfies Γ_2. Γ_1 and Γ_2 are *equivalent* ($\Gamma_1 \simeq \Gamma_2$) if $\Gamma_1 \models \Gamma_2$ and $\Gamma_2 \models \Gamma_1$.

4.2 Schemes

A (type) *scheme* σ is a triple of a set of quantifiers $\vec{\alpha}$, a constraint Γ and a body τ, written $\forall\vec{\alpha}[\Gamma].\tau$. Given an assignment ρ setting the interpretation of free variables, a scheme denotes a set of ground terms, which is obtained by applying all solutions of the constraint to the body:

$$[\![\sigma]\!]_\rho = \uparrow\{\rho'(\tau) \mid \rho' \vdash \Gamma \text{ and } \forall\beta \notin \vec{\alpha}\ \ \rho'(\beta) = \rho(\beta)\}$$

The upward-closure operator (written $\uparrow$) reflects the subsumption rule equipping systems with subtyping: any program of type t may be given a super-type t'.

A scheme σ_1 is more general than σ_2 if and only if it represents a larger set of ground terms under any context: $\sigma_1 \preccurlyeq \sigma_2$ holds if and only if for every assignment ρ, $[\![\sigma_2]\!]_\rho \subseteq [\![\sigma_1]\!]_\rho$. We write $\sigma_1 \simeq \sigma_2$ if σ_1 and σ_2 are *equivalent*, i.e. $\sigma_2 \preccurlyeq \sigma_1$ and $\sigma_1 \preccurlyeq \sigma_2$. In particular, if Γ_1 and Γ_2 are equivalent constraints then the schemes $\forall\vec{\alpha}[\Gamma_1].\tau$ and $\forall\vec{\alpha}[\Gamma_2].\tau$ are equivalent as well.

5 Solving constraints

We are done introducing terms and constraints. We do not present an instance of this logic dedicated to a particular program analysis, or specify how schemes are associated to programs: this is merely out of the topic of this paper, several examples can be found in the literature, e.g. [5].

We now describe an algorithm to decide whether a scheme has an instance, and so determine whether a program is well-typed. For efficiency, the algorithm must also simplify the scheme at hand, by reducing the size of the constraint. Naturally, solving must be interlaced with simplifications, so that the former benefits from the latter. However, for the sake of clarity, we divide our presentation in two parts. The first one handles a single constraint which is rewritten into an equivalent one whose satisfiability can be immediately decided. The second

$$\textsc{expand}\quad \frac{\langle\bar{\bar{\alpha}}\rangle \approx \langle\bar{\bar{\tau}} = f(\vec{\beta})\rangle \approx \overset{\approx}{\tau}'}{\exists\vec{\alpha}.[\langle\bar{\bar{\alpha}} = f(\vec{\alpha})\rangle \approx \langle\bar{\bar{\tau}} = f(\vec{\beta})\rangle \approx \overset{\approx}{\tau}' \wedge_i \beta_i \approx^{v_i(f)} \alpha_i]}\rightsquigarrow$$

$$\textsc{exp-fuse}^{\approx}\quad \frac{\langle\alpha = \bar{\bar{\tau}}\rangle \approx \overset{\approx}{\tau} \wedge \langle\alpha = \bar{\bar{\alpha}}\rangle \approx \overset{\approx}{\alpha}}{\langle\alpha = \bar{\bar{\tau}} = \bar{\bar{\alpha}}\rangle \approx \overset{\approx}{\tau} \approx \overset{\approx}{\alpha}}\rightsquigarrow \qquad \textsc{exp-fuse}^{=}\quad \frac{\langle\alpha = \bar{\bar{\tau}}\rangle \approx \langle\alpha = \bar{\bar{\alpha}}\rangle \approx \overset{\approx}{\tau}}{\langle\alpha = \bar{\bar{\tau}} = \bar{\bar{\alpha}}\rangle \approx \overset{\approx}{\tau}}\rightsquigarrow$$

$$\begin{matrix}\alpha = f(\vec{\alpha})\\ \beta = f(\vec{\beta})\end{matrix} \left|\, \overset{\textsc{decompose}^{\leq}}{\frac{\alpha \leq \beta}{\wedge_i\, \alpha_i \leq^{v_i(f)} \beta_i}}\right.\rightsquigarrow \qquad \alpha = f(\vec{\alpha}) \left|\, \overset{\textsc{decompose}^{l}}{\frac{\alpha \sqsubset \phi}{\wedge_{i\in l(f)}\, \alpha_i \sqsubset \phi}}\right.\rightsquigarrow \qquad \alpha = f(\vec{\alpha}) \left|\, \overset{\textsc{decompose}^{r}}{\frac{\phi \sqsubset \alpha}{\wedge_{i\in r(f)}\, \phi \sqsubset \alpha_i}}\right.\rightsquigarrow$$

Fig. 4: Expansion and decomposition (rewriting system Ω_{ed})

part consists in a series of simplification techniques which consider the constraint in its context, i.e. a scheme. They are described in Section 6 and intended to be integrated throughout the solving process, as we will explain.

The algorithm for solving constraints is made of several steps; some of them are formalized as *rewriting systems*. A rewriting system Ω consists in a reduction relation, $-\Omega\rightarrow$, defined by a set of rules of the form

$$\Gamma_i^o \left|\, \frac{\Gamma_i}{\Gamma_i'}\right.\rightsquigarrow$$

Then $-\Omega\rightarrow$ is the smallest congruence (w.r.t. $\wedge$ and $\exists$) such that, for all i, $\Gamma_i^o \wedge \Gamma_i -\Omega\rightarrow \Gamma_i^o \wedge \Gamma_i'$. Ω is *sound* if it preserves the semantics of constraints, i.e. $\Gamma -\Omega\rightarrow \Gamma'$ implies $\Gamma \simeq \Gamma'$. It *terminates* on Γ if it has no infinite derivation whose origin is Γ. We write $\Gamma -\Omega\twoheadrightarrow \Gamma'$ if and only if $\Gamma -\Omega\rightarrow^* \Gamma'$ and Γ' is an normal form for $-\Omega\rightarrow$.

Rewriting systems are convenient to describe algorithms in a concise and precise manner, and to reason about them. Moreover, they allow abstracting away from some details of an implementation, such as an evaluation strategy specifying the order in which rules have to be applied.

5.1 Unification

The first step of the solving algorithm is made of two interlaced unification processes: one for skeletons (multi-skeletons) and the other for terms (multi-equations). Each of them is to some extent similar to unification in equational theories [31]. They are described by the rewriting system of figure 3, that intends to rewrite the input constraint into an equivalent one $\Gamma \doteq \exists\vec{\alpha}.\Gamma'$ which is *unified.* We now explain the properties which define unified constraints and how the rules make the constraint at hand satisfying them.

First and foremost, reflecting the inclusion of $\leq$ in $\approx$, **(1)** *strong inequalities must be propagated to skeletons*: if $\alpha \leq \beta \in \Gamma'$ then α and β must have the same skeleton, and $\alpha \approx \beta \in \Gamma'$ must hold. This is realized by GENERATE$^{\leq}$, which generates a multi-skeleton from every strong inequality. As a side-effect, the flag carried by the inequality decreases from 1 to 0 preventing multiple applications of the rule on the same constraint. Then, **(2)** *the multi-equations of a unified constraint must be fused*, i.e. every variable can appear at most in one of them.

This is made possible by the transitivity of $\approx$ and $=$: rule FUSE$^{\approx}$ merges two multi-skeletons which have a common variable and then FUSE$^{=}$ operates on pairs of multi-equations within a multi-skeleton.

Furthermore, constraints involving non-variable terms must be propagated to sub-terms. This concerns **(3)** *multi-skeletons that must be decomposed*: two non-variable terms in the same multi-skeleton must have the same root symbol and if $f(\vec{\alpha}) \approx f(\vec{\beta}) \in \Gamma'$ then, for all i, $\alpha_i \approx^{v_i(f)} \beta_i \in \Gamma'$. An application of DECOMPOSE$^{\approx}$ propagates *same-skeleton* constraints between two non-variable terms with the same head symbol to their sub-terms. This is recorded by changing the flag of one of the two multi-equations from 1 to 0: once again, this prevents successive applications of the rule on the same pair of terms. (In this rule, $\alpha_i \approx^{v_i(f)} \beta_i$ stands for $\alpha_i = \beta_i$ if $v_i(f) = \odot$ and $\alpha_i \approx \beta_i$ otherwise. Furthermore, $\Gamma \wedge_i \alpha_i \approx^{v_i(f)} \beta_i$ where i ranges from 1 to n is a shorthand for $\Gamma \wedge \alpha_1 \approx^{v_1(f)} \beta_1 \wedge \cdots \wedge \alpha_n \approx^{v_n(f)} \beta_n$.) Decomposition also occurs for multi-equations: in a unified constraint, **(4)** *every multi-equation must involve at most one non-variable term.* This is enforced thanks to DECOMPOSE$^{=}$, which is similar DECOMPOSE$^{\approx}$; however, one of the two terms may be removed here, which is sufficient to ensure termination. Besides, when a multi-equation contains two row terms with different labels, it is necessary to permute one of them by a *mutation* (rule MUTATE) in order to be able to apply DECOMPOSE$^{\approx}$ or DECOMPOSE$^{=}$. ($\mathsf{Roots}(\bar{\bar{\tau}})$ stands for the set of symbols f such that $f(\vec{\tau}) \in \bar{\bar{\tau}}$ for some $\vec{\tau}$.) In the purpose of orienting the permutation of labels, MUTATE assumes an arbitrary well-founded order $\leq_{\mathcal{L}}$ on labels. Lastly, **(5)** *a unified constraint can involve only small terms.* Thus, GENERALIZE replaces a deep occurrence of a non-variable term τ in a multi-skeleton by a fresh variable α and adds the constraint $\alpha = \tau$. This allows in particular decomposition to apply to small terms only and prevents duplicating structure.

Unification may fail if rewriting produces a constraint where two different type constructors appear as roots in the same multi-skeleton. Such a constraint is said to be a *unification error* and is not satisfiable. A constraint which satisfies the conditions (1) to (5) above and is not a unification error is *unified.*

Theorem 1 (Unification). *Assume every flag in Γ is* 1*. Then Ω_u terminates on Γ. If $\Gamma \xrightarrow{\Omega_u}\!\!\!\rightarrow \Gamma'$ then Γ' is equivalent to Γ and either unified or erroneous.*

This theorem states the soundness and the completeness of the unification algorithm. Because flags carried by multi-equations are no longer used by the following steps of the algorithm, we omit them in the remainder of the paper.

In our implementation [28], unification is performed during constraint construction by the type checker. This allows unification errors to be detected immediately; thus they may be reported by the same techniques than those used in unification-based systems. What is more, only unified constraints may be stored in the solver: indeed, every multi-skeleton or multi-equation is represented by a single node. Multi-equations carry a pointer to the multi-skeleton they belong to, so that fusing can be performed efficiently by union-find. Moreover, every node has pointers to its “sub-nodes”, if any; which allow decomposition. Lastly, inequalities are stored as the edges of a graph between nodes.

5.2 Occur-check

Because recursive terms are not valid solutions for constraints in the model, one must ensure that the multi-skeletons of a constraint do not require cyclic term structures. This verification is commonly referred to as the *occur-check.* By lack of space, we omit its formal description, which is standard and can be found in [33], and just introduce notions that are useful for the following. In short, considering a constraint Γ, it requires to compute a topological ordering $\prec_\Gamma$ of variables according to the term structure imposed by Γ's multi-skeletons. Roughly speaking $\beta \prec_\Gamma \alpha$ means that β appears in a *sub-skeleton* of α. Variables which are minimal for $\prec_\Gamma$ are *terminal.* These are variables about whose structure Γ tells nothing.

It is worth noting that, because unification terminates even in the presence of cyclic structures, it is not necessary to perform an occur-check every time two terms are unified; a single invocation at the end is sufficient and more efficient.

5.3 Expansion and decomposition

Let us first illustrate this step by an example: consider the multi-skeleton $\langle\alpha\rangle \approx \langle\beta = c(\beta_1, \ldots, \beta_n)\rangle$. Every solution of this constraint maps α to a c type; hence we can *expand* α and rewrite the constraint as $\exists\vec{\alpha}.[\langle\alpha = c(\alpha_1, \ldots, \alpha_n)\rangle \approx \langle\beta = c(\beta_1, \ldots, \beta_n)\rangle]$. Besides, taking advantage of the previous expansion, it is possible to *decompose* the inequality $\alpha \leq \beta$ as a series of inequalities relating the sub-variables according to c's variances, i.e. $\alpha_1 \leq^{v_1(c)} \beta_1 \wedge \cdots \wedge \alpha_n \leq^{v_n(c)} \beta_n$.

Formally, a variable α is *expanded* in an $\exists$-free constraint Γ if there exists a non-variable term τ such that $\alpha = \tau \in \Gamma$ holds. It is *decomposed* if it does not appear in any of Γ's inequalities. We say Γ is *expanded down to* α if and only if every variable β such that $\alpha \prec_\Gamma^+ \beta$ is expanded. A constraint $\Gamma \doteq \exists\vec{\alpha}.\Gamma'$ is expanded if and only if Γ' is expanded down to all its terminal variables. We adopt the same terminology for decomposition. A unified, expanded and decomposed constraint which satisfies the occur-check is *reduced.*

The rewriting system Ω_{ed} (Figure 4) rewrites a unified constraint which satisfies the occur-check into an equivalent reduced one. Rule EXPAND performs the expansion of a non-terminal variable. Fresh variables are introduced as arguments of the symbol, with the appropriate $\approx$ and $=$ constraints. These are merged with existing multi-skeletons by rules EXP-FUSE$^\approx$ and EXP-FUSE$^=$, respectively (which are particular cases of FUSE$^\approx$ and FUSE$^=$), allowing the constraint to remain unified. Strong inequalities are decomposed by DECOMPOSE$^\leq$. In this rule, $\alpha_i \leq^{v_i(f)} \beta_i$ stands for $\alpha_i \leq \beta_i$ (resp. $\beta_i \leq \alpha_i$) if $v_i(c) = \oplus$ (resp. $\ominus$). In the case where $v_i(c) = \odot$, it must be read as the constraint **true**, which is sufficient because the equation $\alpha_i = \beta_i$ has already been generated during unification by GENERATE$^\leq$ and DECOMPOSE$^\approx$. Weak inequalities are decomposed by DECOMPOSE^l and DECOMPOSE^r.

Termination of expansion and decomposition relies on the the occur-check; that is the reason why we did not allow recursive types in the model. As an implementation strategy, it is wise to expand and decompose multi-skeletons by considering them in the topological order exposed by the occur-check; so that

$$\frac{\alpha_1 = \alpha_2 \in \Gamma \text{ or } \alpha_1 \leq \alpha_2 \in \Gamma}{\Gamma \mathrel{|\!\!\approx} \alpha_1 \leq \alpha_2} \qquad \frac{\phi_1 \sqsubset \phi_2 \in \Gamma}{\Gamma \mathrel{|\!\!\approx} \phi_1 \sqsubset \phi_2}$$

$$\frac{\begin{array}{c}\Gamma \mathrel{|\!\!\approx} \alpha_1 \leq \alpha_2 \\ \Gamma \mathrel{|\!\!\approx} \alpha_2 \leq \alpha_3\end{array}}{\Gamma \mathrel{|\!\!\approx} \alpha_1 \leq \alpha_3} \qquad \frac{\begin{array}{c}\Gamma \mathrel{|\!\!\approx} \phi_1 \sqsubset \phi_2 \\ \Gamma \mathrel{|\!\!\approx} \phi_2 \sqsubset \phi_3\end{array}}{\Gamma \mathrel{|\!\!\approx} \phi_1 \sqsubset \phi_3} \qquad \frac{\begin{array}{c}\Gamma \mathrel{|\!\!\approx} \phi_1 \sqsubset \alpha_2 \\ \Gamma \mathrel{|\!\!\approx} \alpha_2 \leq \alpha_3\end{array}}{\Gamma \mathrel{|\!\!\approx} \phi_1 \sqsubset \beta_3} \qquad \frac{\begin{array}{c}\Gamma \mathrel{|\!\!\approx} \alpha_1 \leq \alpha_2 \\ \Gamma \mathrel{|\!\!\approx} \alpha_2 \sqsubset \phi_3\end{array}}{\Gamma \mathrel{|\!\!\approx} \alpha_1 \sqsubset \phi_3}$$

Fig. 5: Syntactic implication

fresh variables and inequalities are generated only within skeletons that have not yet been dealt with.

Theorem 2 (Expansion and decomposition). *Let Γ be a unified constraint which satisfies the occur-check. Ω_{ed} terminates on Γ. If $\Gamma \, -_{\Omega_{ed}}\!\twoheadrightarrow \Gamma'$ then Γ' is equivalent to Γ and reduced.*

A constraint is *atomic* if and only if all its terms are terminal. Given a reduced constraint Γ we define its *atomic part* as the constraint $\lfloor \Gamma \rfloor$ obtained from Γ by removing all multi-skeletons which contain non-variable terms.

Theorem 3. *If Γ is reduced, then the satisfiability of Γ and $\lfloor \Gamma \rfloor$ are equivalent.*

This theorem shows that the satisfiability of a reduced constraint is equivalent to that of its atomic part. As a consequence, it is now sufficient to provide an algorithm for solving atomic constraints, which we do in the following subsection.

5.4 Solving atomic constraints

The algorithm for solving a $\exists$-free atomic constraint Γ consists in checking that, in the graph defined by Γ's inequalities, there is no path between two constants a_1 and a_2 such that $a_1 \not\leq_{\mathcal{A}} a_2$. Paths are formally defined by the predicates $\Gamma \mathrel{|\!\!\approx} \cdot \sqsubset \cdot$ and $\Gamma \mathrel{|\!\!\approx} \cdot \leq \cdot$ introduced in Figure 5 and may be checked in linear time. The following theorem states the criterion of satisfiability of atomic constraints involved by our algorithm.

Theorem 4. *Let $\Gamma \doteq \exists\vec{\alpha}.\Gamma'$ be an atomic constraint. Γ is satisfiable if and only if for all atoms a_1 and a_2, $\Gamma' \mathrel{|\!\!\approx} a_1 \sqsubset a_2$ implies $a_1 \leq_{\mathcal{A}} a_2$.*

5.5 Complexity analysis

We now informally discuss the theoretical complexity of the solving algorithm, i.e. the four steps described in Section 5.1 to 5.4. The input of the algorithm, a constraint Γ, is measured as its size n which is the sum of the sizes of all the involved terms, which is generally proportional to the size of the studied program. As we have explained, we make the hypothesis that the height of terms is bounded: we let h be the length of the longest chain (w.r.t. $\prec_\Gamma$) found by occur-check and a the maximal arity of constructors. For the sake of simplicity, we exclude rows of our study, whose analysis is more delicate [35].

The first step of the algorithm is the combination of two unification algorithms, one applying on multi-skeletons and the other on multi-equations, which may be—in the absence of row terms—performed separately. Hence, using a union-find algorithm, it requires time $O(n\alpha(n))$ (where α is related to an inverse of Ackermann's function) [36]. Occur-check is equivalent to a topological ordering of the graph of multi-skeletons, so it is linear in their number, which is $O(n)$. Then, under the hypothesis of bounded-terms, expansion generates at most a^{h} new variables for each variable in the input problem, and, the decomposition of an inequality is similarly bounded. So, these two steps cost $O(\mathsf{a}^{\mathsf{h}}n)$. Lastly, checking paths in the atomic graph can be done in linear time by a topological walk, provided that lattice operations on atoms (i.e. $\leq$, $\sqcup$ and $\sqcap$) can be computed in constant time.

6 Simplifying constraints and type schemes

We are done in describing the corpus of the solving algorithm. We now have to introduce throughout this process a series of heuristics whose purpose is to reduce the size of the problem at hand, and hence improve the efficiency of solving. Simplification is a subtle problem: it must be correct (i.e. the result must be equivalent to the input) as well as efficient in computation time and effective in constraint size reduction. Following our pragmatic motivation, we do not address here the question of optimality or completeness of simplification. Instead, we present a series of techniques which we have experienced to be effective: finely combined into the solving algorithm, they allow to notably improve its efficiency.

The most critical part of solving is expansion, which is likely to introduce a lot of new variables. As we have explained, expansion is performed one multi-skeleton at a time, in the order found by the occur-check. So, in attempt to minimize its possible impact, we will apply those of our techniques that are *local* to a multi-skeleton (Section 6.1 and 6.3) just before its variables are expanded, in order to reduce the number of variables to be introduced. A second group of simplifications (Section 6.4 and 6.5) needs to consider the whole graph of atomic constraints. As a result, they are performed only once, at the end of solving. Their purpose is to overcome another costly part of type inference in presence of *let*-polymorphism: generalization and instantiation require duplicating schemes, hence they must be made as compact as possible. They also allow obtaining human-readable typing information.

Basically, there are two ways to reduce the size of schemes. The first one (featured in Sections 6.1, 6.3 and 6.5) consists in identifying variables. Formally, this consists in replacing inequalities by multi-equations, which is effective since dealing with the latter is much more efficient than with the former. The second one (Section 6.4) removes intermediate variables which are no longer useful for the final result (e.g. because they are unreachable).

In this section, we restrict our attention to schemes whose constraint is $\exists$-free. (This is not restrictive because $\forall\vec{\alpha}[\exists\vec{\beta}.\Gamma].\tau$ can be rewritten into $\forall\vec{\alpha}\vec{\beta}[\Gamma].\tau$.)

6.1 Collapsing cycles

Cycle detection allows replacing a cycle of inequalities by a multi-equation in a constraint. A cycle consists in a list of multi-equations $\bar{\bar{\alpha}}_0, \ldots, \bar{\bar{\alpha}}_n$ such that $\bar{\bar{\alpha}}_1 \leq \bar{\bar{\alpha}}_2 \in \Gamma, \ldots, \bar{\bar{\alpha}}_n \leq \bar{\bar{\alpha}}_1 \in \Gamma$. Clearly, any solution ρ for Γ satisfies $\rho(\bar{\bar{\alpha}}_1) = \cdots = \rho(\bar{\bar{\alpha}}_n)$. Thus, the multi-equations $\bar{\bar{\alpha}}_0, \ldots, \bar{\bar{\alpha}}_n$ can be fused

In [37], Fähndrich *et al.* proposed a partial on-line cycle detection algorithm, which permits to collapse cycles incrementally at the same time as inequalities are generated by some closure algorithm. However, in the current paper, all the variables in a cycle of a unified constraint must belong to the same multi-skeleton. This allows considering each multi-skeleton separately—before its expansion—and thus using a standard graph algorithm for detecting cycles in linear time, which is more efficient.

6.2 Polarities

The remaining simplification techniques need to consider the constraint at hand in its context, i.e. a whole scheme describing a (piece of) program. Indeed, this allows distinguishing the type variables which represent an "input" of the program from those which stand for an "output": the former are referred to as *negative* while the latter are *positive*. Because we remain abstract from the programming language and the type system itself, these notions are only given by the variances of type constructors: roughly speaking, one may say that a covariant argument describes an output while a contravariant one stands for an input. (This is reflected by the variances commonly attributed to the $\rightarrow$ type constructor in λ-calculus or ML.) Because non-positive variables are not related to the *output* produced by the program, we are not interested with their lower bounds. Similarly, the upper bounds of non-negative variables do not matter.

For this purpose, we assign polarities [12, 16, 18] to variables in a scheme $\sigma = \forall\vec{\alpha}[\Gamma].\tau$: we write $\sigma \vdash \alpha : +$ (resp. $\sigma \vdash \alpha : -$) if α is *positive* (resp. *negative*) in σ. (The same variable can simultaneously be both.) Regarding variances of symbols, polarities are extended to terms by the following rule

$$\frac{\forall i \;\; v_i(f) \in \{\oplus, \odot\} \Rightarrow \sigma \vdash \tau_i : + \qquad \forall i \;\; v_i(f) \in \{\ominus, \odot\} \Rightarrow \sigma \vdash \tau_i : -}{\sigma \vdash f(\tau_1, \ldots, \tau_n) : +}$$

and its symmetric counterpart for proving $\sigma \vdash f(\tau_1, \ldots, \tau_n) : -$. Then, $\sigma \vdash \cdot : +$ and $\sigma \vdash \cdot : -$ are defined as the smallest predicates such that:

$$\sigma \vdash \tau : + \qquad \frac{\alpha \notin \vec{\alpha}}{\sigma \vdash \alpha : \pm} \qquad \frac{\sigma \vdash \alpha : + \qquad \alpha = \tau' \in \Gamma}{\sigma \vdash \tau' : +} \qquad \frac{\sigma \vdash \alpha : - \qquad \alpha = \tau' \in \Gamma}{\sigma \vdash \tau' : -}$$

The first rule reflects the fact that the body describes the result produced by the associated piece of code, hence it is positive. The second rule makes every free variable bipolar, because it is likely to be related to any other piece of code. The last two rules propagate polarities throughout the structure of terms. Polarities can be computed by a simple propagation during expansion.

6.3 Reducing chains

Constraint generation yields a large number of chains of inequalities: because subsumption is allowed at any point in a program, the type synthesizer usually generates inequalities for all of them; but many are not really used by the program at hand. Chains reduction intends to detect and remove these intermediate variables and constraints, as proposed by Eifrig *et al* [38] and, by Aiken and Fähndrich [15] in the setting of set constraints. Here, we adapt their proposal to the case of structural subtyping.

We say that $\bar{\bar{\tau}}$ is the unique predecessor of $\bar{\bar{\alpha}}$ in $\forall\vec{\alpha}[\Gamma].\tau$ if and only if $\bar{\bar{\tau}} \leq \bar{\bar{\alpha}} \in \Gamma$ and it is the only inequality involving $\bar{\bar{\alpha}}$ as right-hand-side. Symmetrically, we define unique successors. The following theorem states that a non-positive (resp. non-negative) multi-equation may be fused with its unique successor (resp. predecessor).

Theorem 5 (Chains). *Let $\sigma = \forall\vec{\alpha}[\Gamma \wedge \langle\bar{\bar{\alpha}}\rangle \approx \langle\bar{\bar{\tau}}\rangle \approx \tilde{\bar{\bar{\tau}}}].\tau$ be a unified scheme, satisfying the occur-check, expanded and decomposed down to $\bar{\bar{\alpha}}$. If $\bar{\bar{\alpha}}$ is non-positive (resp. non-negative) and $\bar{\bar{\tau}}$ is its unique successor (resp. predecessor) then σ is equivalent to $\forall\vec{\alpha}[\Gamma \wedge \langle\bar{\bar{\alpha}} = \bar{\bar{\tau}}\rangle \approx \tilde{\bar{\bar{\tau}}}].\tau$.*

6.4 Polarized garbage collection

Computing the scheme which describes a piece of code typically yields a large number of variables. Many of them are useful only during intermediate steps of the type generation, but are no longer essential once it is over. Garbage collection is designed to keep only polar variables in the scheme at hand, and paths from variables which are related to some input of the program (i.e. negative ones) to those which are related to some output (i.e. positive ones). Indeed, it rewrites the input constraint into a *closed* one such that: (1) every variable appearing in a multi-skeleton is polar, (2) for every inequality $\alpha_1 \leq \alpha_2$ or $\alpha_1 \sqsubset \alpha_2$, α_1 is negative and α_2 is positive, (3) only positive (resp. negative) variables may have a constant lower (resp. upper) bound, which—if it exists—is unique. This idea has been introduced by Trifonov and Smith [16] in the case of non-structural subtyping.

Let $\sigma = \forall\vec{\alpha}[\Gamma].\tau$ be a scheme with Γ reduced and satisfiable. We define the lower and upper bounds of α in Γ by: $\mathrm{lb}_\Gamma(\alpha) = \sqcup\{a \mid \Gamma \models\!\!\approx a \sqsubset \alpha\}$ and $\mathrm{ub}_\Gamma(\alpha) = \sqcap\{a \mid \Gamma \models\!\!\approx \alpha \sqsubset a\}$. A multi-equation is said to be polar if it contains a variable which is negative or positive. Then, $GC(\sigma)$ is $\forall\vec{\alpha}[\Gamma'].\tau$ where Γ' is the conjunction of the following constraints:

- $\langle\bar{\bar{\tau}}_1\rangle \approx \cdots \approx \langle\bar{\bar{\tau}}_n\rangle$, for all $\bar{\bar{\tau}}_1, \ldots, \bar{\bar{\tau}}_n$ which are the polar multi-equations of one of Γ's multi-skeletons,
- $\alpha \leq \beta$, for all α and β such that $\Gamma \models\!\!\approx \alpha \leq \beta$ and $\sigma \vdash \alpha : -$ and $\sigma \vdash \beta : +$,
- $\alpha \sqsubset \beta$, for all α and β such that $\Gamma \models\!\!\approx \alpha \sqsubset \beta$ and $\sigma \vdash \alpha : -$ and $\sigma \vdash \beta : +$,
- $\mathrm{lb}_\Gamma\,\alpha \sqsubset \alpha$, for all α such that $\sigma \vdash \alpha : +$ and $\mathrm{lb}_\Gamma\,\alpha \neq \bot$
- $\alpha \sqsubset \mathrm{ub}_\Gamma\,\alpha$, for all α such that $\sigma \vdash \alpha : -$ and $\mathrm{ub}_\Gamma\,\alpha \neq \top$

It is worth noting that, once garbage collection is performed, a scheme involves only polar variables. Hence, using a suitable substitution, it may be rewritten in a body giving the whole term structure, and a constraint (consisting in a conjunction of $\approx$, $\leq$ and $\sqsubset$) relating variables of the body, without any intermediate one. This form is most suitable for giving human-readable type information.

6.5 Minimization

This simplification intends to reduce the number of distinct variables or terms in a constraint by detecting some equivalences. It is called *minimization* because it is similar to that of an automaton (which detects equivalent states).

Let $\sigma = \forall\vec{\alpha}[\Gamma].\tau$ be a unified scheme. Two terminal multi-equations $\bar{\bar{\alpha}}_1$ and $\bar{\bar{\alpha}}_2$ of the same multi-skeleton are equivalent in σ (we write $\bar{\bar{\alpha}}_1 \sim_\sigma \bar{\bar{\alpha}}_2$) if

	Caml Light		Flow Caml
	library	compiler	library
A.s.t. nodes	14002	22996	13123
1. Type inference[1]			
Unification	0.346 s	0.954 s	
Structural subtyping (`Dalton`)	0.966 s	2.213 s	n.a.
ratio	2.79	2.31	
2. Statistics about `Dalton`[2]			
Multi-equations	30345	65946	73328
Collapsing cycles	501 (2%)	1381 (2%)	1764 (2%)
Chain reduction	9135 (30%)	15967 (24%)	17239 (24%)
Garbage collection	15288 (50%)	31215 (47%)	18460 (25%)
Minimization	424 (1%)	644 (1%)	815 (1%)
Expanded variables[3]	948 (3%) (9% of n.t.)	1424 (2%) (8% of n.t.)	1840 (3%) (14% of n.t.)

[1] Benchmarks realized on a Pentium III 1 GHz (average of 100 runs)
[2] Percentages relative to the total number of multi-equations
[3] The 2nd percentage is relative to the number of non-terminal multi-eq. considered by expansion

Table 1. Experimental measures

- either they are non-positive and have the same successors (i.e. $\{\beta \mid \bar{\bar{\alpha}}_1 \leq \beta \in \Gamma\} = \{\beta \mid \bar{\bar{\alpha}}_2 \leq \beta \in \Gamma\}$ and $\{\phi \mid \bar{\bar{\alpha}}_1 \sqsubset \phi \in \Gamma\} = \{\phi \mid \bar{\bar{\alpha}}_2 \sqsubset \phi \in \Gamma\}$),
- or they are non-negative and have the same predecessors (i.e. $\{\beta \mid \beta \leq \bar{\bar{\alpha}}_1 \in \Gamma\} = \{\beta \mid \beta \leq \bar{\bar{\alpha}}_2 \in \Gamma\}$ and $\{\phi \mid \phi \sqsubset \bar{\bar{\alpha}}_1 \in \Gamma\} = \{\phi \mid \phi \sqsubset \bar{\bar{\alpha}}_2 \in \Gamma\}$).

Minimization consists in fusing every pair of equivalent multi-equations. So, we define $M(\sigma) = \forall\vec{\alpha}[\Gamma \wedge \{\bar{\bar{\alpha}}_1 = \bar{\bar{\alpha}}_2 \mid \bar{\bar{\alpha}}_1 \sim_\sigma \bar{\bar{\alpha}}_2\}].\tau$.

Theorem 6 (Minimization). *Let σ be a reduced and closed scheme. $M(\sigma)$ is equivalent to σ.*

Once minimization has been performed, the equivalences found between the terminal variables can be propagated throughout the term structure thanks to *hash-consing*. This is described in the full version of the paper [33]. However, this does not improve readability of the terms printed by the solver (because they are generally displayed without exhibiting sharing between internal nodes) and, according to our experiments (see Section 7), it has only a little impact on the practical efficiency.

7 Implementation and experiments

The `Dalton` library [28] is a real-size implementation in Objective Caml of the algorithms described in the current paper. In this library, the constraint solver comes as a *functor* parametrized by a series of modules describing the client's type system. Hence, we hope it will be a suitable type inference engine for a variety of applications.

We have experimented with this toolkit in two prototypes. First, we designed an implementation of the Caml Light compiler that is modular w.r.t. the type system and the constraints solver used for type inference. We equipped this prototype with two engines:

– A standard unification-based solver, which implements the same type system as Caml Light,
– An instance of the `Dalton` library, which features an extension of the previous type system with structural subtyping, where each type constructor carry an extra atomic annotation belonging to some arbitrary lattice.

This second type system has no interest for itself, but is a good representative—in terms of constraints resolution—of real ones featuring structural subtyping in the purpose of performing some static analysis on programs, such as a data or information flow analysis. We ran them on several sets of source code files, including the Caml Light compiler and its standard library; the resulting measures appear in the first two columns of Table 1. To compare our framework with standard unification, we measure the computation time of the typing phase of compilation: on our tests, `Dalton` appears to be slower than unification only by a factor comprised between 2 to 3. Such measures must be interpreted carefully. However, unification is recognized to be efficient and is widely used; so we believe them to be a point assessing the practicality and the scalability of our framework. Besides, we used our solver as the type inference engine of `Flow Caml` [39], an information flow analyzer for the Caml language. The measures obtained during the analysis of its library appear in the last column of Table 1.

These experiments also provide information about the behavior of the solving algorithm and the efficiency of simplification techniques. We measured the total number of multi-equations generated throughout type generation and constraints solving, and of those which are collected by one of the simplification techniques. Chain reduction appears as a key optimization, since it approximatively eliminates one quarter of multi-equations—that are variables—*before* expansion. The direct contribution of collapsing cycles is ten times less; however, we observed that skipping this simplification affects chain reduction. Hence, expansion becomes marginal: the number of variables that are expanded represents only a few percents of the total number of multi-equations, and about a tenth of the non-terminal multi-equations considered by expansion. Simplifying before expansion is crucial: if we modify our implementation by postponing chain reduction after expansion, the number of expanded variables grow by a factor around 20. Lastly, our measures show that the contribution of garbage collection is comparable to that of chain reduction; minimization has less impact on the size of the constraints but appears crucial for readability.

8 Discussion

Our implementation handles polymorphism in a manner inspired by Trifonov and Smith [16], where all type schemes are *closed*, i.e. have no free type variables, but contain a local environment. The interest lies in the fact that generalization and instantiation simply consist in making fresh copies of schemes. This approach turns out to be reasonable in practice, mostly because, thanks to simplification, the size of copied structures is limited. However, it should also be possible to deal with polymorphism in a more standard way, by using numeric *ranks* for distinguishing generalizable variables [31]. This would require making copies of constraints *fragments*, which yields more complicated machinery. However, in both approaches, we are still faced with the problem of constraints duplication. This is largely similar to the difficulty encountered in ML, whose practical impact is limited. Furthermore, this question has been studied for the setting of a flow analysis in [40].

Several possible extensions of the system may be mentioned. An interesting question lies in the introduction of recursive terms. This should mostly require to adapt expansion which relies on the finiteness of the term structure. Besides, in this paper, $\sqsubset$ is only allowed to consider *covariant* arguments of type constructors. However, in [5], the combination of polymorphic equality and mutable cells requires weak inequalities to be decomposed on *invariant* arguments too. Such an extension requires introducing weak inequalities on *skeletons*. This is experimentally handled by our implementation.

References

1. Bourdoncle, F., Merz, S.: Type checking higher-order polymorphic multi-methods. In: 24th Principles of Programming Languages. (1997)
2. Pottier, F.: A versatile constraint-based type inference system. Nordic Journal of Computing **7** (2000)
3. Aiken, A.S., Fähndrich, M.: Program analysis using mixed term and set constraints. In: 4th Static Analysis Symposium. (1997)
4. Palsberg, J., O'Keefe, P.M.: A type system equivalent to flow analysis. In: 22nd Principles of Programming Languages. (1995)
5. Pottier, F., Simonet, V.: Information flow inference for ML. In: 29th Principles of Programming Languages. (2002)
6. Mitchell, J.C.: Coercion and type inference. In: 11th Principles of Programming Languages. (1984)
7. Cardelli, L.: A semantics of multiple inheritance. Information and Computation **76** (1988)
8. Tiuryn, J.: Subtype inequalities. In: 7th IEEE Symposium on Logic in Computer Science. (1992)
9. Hoang, M., Mitchell, J.C.: Lower bounds on type inference with subtypes. In: 22nd Principles of Programming Languages. (1995)
10. Mitchell, J.C.: Type inference with simple subtypes. Journal of Functional Programming **1** (1991)
11. Fuh, Y.C., Mishra, P.: Type inference with subtypes. In: European Symposium on Programming. Volume 300. (1988)
12. Fuh, Y.C., Mishra, P.: Polymorphic subtype inference: Closing the theory-practice gap. In: European Joint Conference on Theory and Practice of Software Development. Volume 352. (1989)
13. Frey, A.: Satisfying subtype inequalities in polynomial space. In: 4th Static Analysis Symposium. Number 1302 (1997)
14. Kuncak, V., Rinard, M.: Structural subtyping of non-recursive types is decidable. In: 18th IEEE Symposium on Logic in Computer Science. (2003)
15. Aiken, A.S., Fähndrich, M.: Making set-constraint based program analyses scale. Technical Report CSD-96-917, University of California, Berkeley (1996)
16. Trifonov, V., Smith, S.: Subtyping constrained types. In: 3rd Static Analysis Symposium. Volume 1145. (1996)

17. Flanagan, C., Felleisen, M.: Componential set-based analysis. In: Programming Language Design and Implementation. (1997)
18. Pottier, F.: Simplifying subtyping constraints: a theory. Information and Computation **170** (2001)
19. Rehof, J.: Minimal typings in atomic subtyping. In: 24th Principles of Programming Languages. (1997)
20. Marlow, S., Wadler, P.: A practical subtyping system for Erlang. In: International Conference on Functional Programming. (1997)
21. Fähndrich, M.: *Bane*, A Library for Scalable Constraint-Based Program Analysis. PhD thesis (1999)
22. Pottier, F.: *Wallace*, an efficient implementation of type inference with subtyping. URL: `http://pauillac.inria.fr/~fpottier/wallace/` (2000)
23. Frey, A.: *Jazz*. URL: `http://www.exalead.com/jazz/` (1998)
24. Kodumal, J.: *Banshee*, a toolkit for building constraint-based analyses. PhD thesis (2002)
25. Foster, J.S., Fähndrich, M., Aiken, A.: A Theory of Type Qualifiers. In: Programming Language Design and Implementation. (1999)
26. Foster, J.S., Terauchi, T., Aiken, A.: Flow-sensitive type qualifiers. In: Programming Language Design and Implementation. (2002)
27. Glynn, K., Stuckey, P.J., Sulzmann, M., Søndergaard, H.: Boolean constraints for binding-time analysis. In: Programs as Data Objects. Volume 2053. (2001)
28. Simonet, V.: *Dalton*, an efficient implementation of type inference with structural subtyping. URL: `http://cristal.inria.fr/~simonet/soft/dalton/` (2002)
29. Heintze, N., McAllester, D.: Linear-time subtransitive control flow analysis. In: Programming Language Design and Implementation. (1997)
30. Saha, B., Heintze, N., Oliva, D.: Subtransitive CFA using types. Technical report, Yale University (1998)
31. Rémy, D.: Extending ML type system with a sorted equational theory. Research Report 1766, Institut de Recherche en Informatique et en Automatique (1992)
32. Pessaux, F., Leroy, X.: Type-based analysis of uncaught exceptions. ACM Transactions on Programming Languages and Systems **22** (2000)
33. Simonet, V.: Type inference with structural subtyping: A precise formalization of an efficient constraint solver. Full version. URL: `http://cristal.inria.fr/~simonet/publis/simonet-structural-subtyping-full.ps.gz` (2003)
34. Abadi, M., Banerjee, A., Heintze, N., Riecke, J.G.: A core calculus of dependency. In: 26th Principles of Programming Languages. (1999)
35. Pottier, F.: A constraint-based presentation and generalization of rows. In: 18th IEEE Symposium on Logic in Computer Science. (2003)
36. Tarjan, R.E.: Efficiency of a good but not linear set union algorithm. Journal of the ACM **22** (1975)
37. Fähndrich, M., Foster, J.S., Su, Z., Aiken, A.S.: Partial online cycle elimination in inclusion constraint graphs. In: Programming Language Design and Implementation. (1998)
38. Eifrig, J., Smith, S., Trifonov, V.: Sound polymorphic type inference for objects. ACM SIGPLAN Notices **30** (1995)
39. Simonet, V.: *Flow Caml*, information flow inference in Objective Caml. URL: `http://cristal.inria.fr/~simonet/soft/flowcaml/` (2002)
40. Fähndrich, M., Rehof, J., Das, M.: Scalable context-sensitive flow analysis using instantiation constraints. In: Programming Language Design and Implementation. (2000)