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

(Full version)

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. [BM97]; and it allows designing fine-grained type systems for advanced programming constructs, e.g. [Pot00a]. It is also well suited for extending standard type systems in order to perform some static analysis, such as detection of uncaught exceptions [AF97], data [PO95] or information [PS02] 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 [Mit84] and Cardelli [Car88]. 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.[1] Following the complexity study of Tiuryn [Tiu92] and Hoang and Mitchell [HM95], we will assume the atomic poset to be a lattice (our results may be easily extended to the case where it is a disjoint union of lattices).

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 [Mit84,Mit91] and improved for handling structural subtyping by Fuh and Mishra [FM88,FM89]. However, quoting Hoang and Mitchell [HM95], "*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 [Tiu92] 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 [HM95] proved the equivalence of constraint resolution with typability (in the simply typed λ-calculus with subtyping). Frey [Fre97] settled completely the complexity of the general case, showing it is *PSPACE-complete*. Lastly, Kuncak and Rinard [KR03] 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 [AF96,TS96,FF97,Pot01]. Rehof [Reh97] 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 [MW97,Fäh99,Pot00b,Fre98,Kod02].

Structural subtyping has also been studied throughout a series of specific applications. Among them, one may mention Foster's work about type quali-

[1] Structural subtyping cannot in general be viewed as a particular sub-case of some non-structural framework. For instance, operations described in [Pot01], are incorrect if the set of types is not a lattice, e.g. if there is no types $\bot$ and $\top$. Moreover, extending a unification-based type system with a non-structural order is likely to make typable expressions that are rejected by the original system.

fiers [FFA99,FTA02] and the introduction of boolean constraints for binding-time analysis [GSSS01]. 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 [Sim02] 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. [Pot00b], 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 tran-

sitive closure. It is interesting to draw a parallel with Heintze and McAllester's algorithm for control flow analysis (CFA) [HM97]. 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 [FM88,Tiu92]. 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 [SHO98], 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 [Rém92]. 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 [AF97,PL00]. Second, we deal with an original form of constraints, referred to as *weak inequalities*, which allow handling constraints such as *guards* [PS02] (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. Then, in Section **??**, we sketch how to extend the solving algorithm to decide constraint implication. The implemen-

tation and experimental measures are presented in Section 7. The paper ends with some discussion about possible extensions.

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. We let L range over co-finite subsets of $\mathcal{L}$ and if $\ell \notin L$, $\ell.L$ stands for $\{\ell\} \cup L$. *Ground terms* and *kinds* are defined as follows:

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

Ground terms include atomic constants a, which have kind Atom. For each type constructor c, we assume given a signature which is a finite list of kinds; we write $c :: \kappa_1 \otimes \cdots \otimes \kappa_n \Rightarrow \mathsf{Type}$ (the integer n is the *arity* of c). Then, provided a list of ground terms $\tau_1, \ldots, \tau_n$, whose kinds are respectively $\kappa_1, \ldots, \kappa_n$, $c(\tau_1, \ldots, \tau_n)$ is a *ground type* of kind Type. 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. (L is the *domain* of t and is denoted by $\mathsf{dom}(t)$. We write $\partial_L t$ for the constant row which maps labels in L to t (we omit L when it may be deduced from the context). In this paper, if t is a ground term of kind κ, we write $\vdash t : \kappa$. The formal definition of this predicate is reminded in figure 1; and in the remainder of the paper, we restrict our attention to well kinded ground terms. We also assume that there exists a type constructor whose signature has the form $\mathsf{Row}^*\ \mathsf{Atom} \otimes \cdots \otimes \mathsf{Row}^*\ \mathsf{Atom} \Rightarrow \mathsf{Type}$ (otherwise, the ground algebra contains no type). The *height* of a term is defined by:

$$\begin{aligned} \mathsf{h}(a) &= 0 \\ \mathsf{h}(c(t_1, \ldots, t_n)) &= 1 + \max\{\mathsf{h}(t_i) \mid i \in [\![1, n]\!]\} \\ \mathsf{h}(\{\ell \mapsto t_\ell\}_{\ell \in L}) &= 1 + \max\{\mathsf{h}(t_\ell) \mid \ell \in L\} \end{aligned}$$

Because recursive ground terms are not allowed and rows are quasi-constant, every ground term is of finite height.

3.2 Strong subtyping

The set of ground terms is equipped with the partial order $\leq$ defined in Figure 2, 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,

$$\vdash a : \mathsf{Atom} \qquad \frac{c :: \kappa_1 \otimes \cdots \otimes \kappa_n \Rightarrow \mathsf{Type} \qquad \forall i \ \ \vdash t_i : \kappa_i}{\vdash c(t_1, \ldots, t_n) : \mathsf{Type}} \qquad \frac{\forall \ell \in L \ \ \vdash t_\ell : \kappa}{\vdash \{\ell \mapsto t_\ell\}_{\ell \in L} : \mathsf{Row}_L\, \kappa}$$

Fig. 1: Kinding ground terms

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

Fig. 2: Subtyping over ground terms

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* and denoted by the meta-variable s. 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 [Tiu92].

Lemma 1. *Assume $t_1 \approx t_2$ or $t_1 \leq t_2$. Then t_1 and t_2 have the same kind and $\mathsf{h}(t_1) = \mathsf{h}(t_2)$.*

Proof. By induction on the definition of $\leq$ and $\approx$.

The height of a ground skeleton is therefore the common height of its ground terms.

Lemma 2. *Let s be a ground skeleton. $(s, \leq)$ is a lattice.*

Proof. By induction on the height h of skeletons. The base case if that of height 0: $(s, \leq)$ is the lattice $(\mathcal{A}, \leq_{\mathcal{A}})$. We now assume the above property to be verified by any skeleton whose height is at most h. Let s be a skeleton of height $h+1$. We consider two elements of s, t and t', and prove that they admit a least upper bound (and, by symmetry, a greatest lower one).

∘ *Case* s is a skeleton of types. t is $c(t_1, \ldots, t_n)$ and t' is $c(t_1', \ldots, t_n')$ for some $c \in \mathcal{C}$. Because $t \approx t'$ then, for every i, $t_i \approx t_i'$. By induction hypothesis, let $\check{t} = c(\check{t_1}, \ldots, \check{t_n})$ where $\check{t_i} = t_i \sqcup t_i'$ if $v_i(c) = \oplus$, $\check{t_i} = t_i \sqcap t_i'$ if $v_i(c) = \ominus$, and $\check{t_i} = t_i = t_i'$ if $v_i(c) = \odot$. Consider $t'' = c(t_1'', \ldots, t_n'')$ such that $t \leq t''$ and $t' \leq t''$. By definition of $\leq$, for every i, $t_i \leq^{v_i(c)} t_i''$ and $t_i' \leq^{v_i(c)} t_i''$. It follows $\check{t} \leq t''$. We conclude that $\check{t}$ is the least upper bound of t and t'.

$$\text{ATOM}\ \frac{a_1 \leq_{\mathcal{A}} a_2}{a_1 \sqsubset a_2} \qquad \text{TYPE-LEFT}\ \frac{\forall i \in l(c)\ \ t_i \sqsubset t'}{c(\bar{t}) \sqsubset t'} \qquad \text{TYPE-RIGHT}\ \frac{\forall i \in r(c)\ \ t' \sqsubset t_i}{t' \sqsubset c(\bar{t})} \qquad \text{ROW-LEFT}\ \frac{\forall \ell \in L\ \ t_\ell \sqsubset t'}{\{\ell \mapsto t_\ell\}_L \sqsubset t'} \qquad \text{ROW-RIGHT}\ \frac{\forall \ell \in L\ \ t' \sqsubset t_\ell}{t' \sqsubset \{\ell \mapsto t_\ell\}_L}$$

Fig. 3: Weak inequalities

∘ *Case* s is a skeleton of rows of domain L. Applying the induction hypothesis, we define $\check{t} = \{\ell \mapsto t(\ell) \sqcup t(\ell')\}_{\ell \in L}$ (because t and t' are quasi-constant, t'' is quasi-constant). Similarly to the previous case, we check that $\check{t}$ is the smallest upper bound of t and t'.

3.3 Weak inequalities

Type systems with structural subtyping may require constraints relating an arbitrary term to an atom, such as "*protected types*" [ABHR99] or "*guards*" ($\lhd$ or $\blacktriangleleft$) [PS02]. For instance, in [PS02], 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 3. $\sqsubset$ 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. For proving this, we define two mappings (or *destructors*) $\underline{\sqcup}$ and $\underline{\sqcap}$ from terms two atoms by the following equations:

$$\begin{aligned} \underline{\sqcup}\, a &= a & \underline{\sqcap}\, a &= a \\ \underline{\sqcup}\, c(\vec{\tau}) &= \sqcup\{\underline{\sqcup}\, \tau_i \mid i \in l(c)\} & \underline{\sqcap}\, c(\vec{\tau}) &= \sqcap\{\underline{\sqcap}\, \tau_i \mid i \in r(c)\} \\ \underline{\sqcup}\{\ell \mapsto \tau_\ell\}_L &= \sqcup\{\underline{\sqcup}\, \tau_\ell \mid \ell \in L\} & \underline{\sqcap}\{\ell \mapsto \tau_\ell\}_L &= \sqcap\{\underline{\sqcap}\, \tau_\ell \mid \ell \in L\} \end{aligned}$$

Then we have the following property.

Lemma 3. *$\tau_1 \sqsubset \tau_2$ if and only if $\underline{\sqcup}\, \tau_1 \leq_{\mathcal{A}} \underline{\sqcap}\, \tau_2$.*

Proof. By induction on the sum of the height of τ_1 and τ_2.

Moreover, it is worth noting that, 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 $t_1 \sqsubset t_3$. To conclude with weak inequalities, the following lemma points out that weak inequalities reduce to strong ones when the structure of terms is known.

Lemma 4. *Let s be a ground skeleton. For every atom a, there exists two unique ground terms in s, $\wr s \uparrow a \int_{\boxed{\vee}}$ and $\wr s \uparrow a \int_{\boxed{\wedge}}$ such that for all $t \in s$, $t \sqsubset a$ is equivalent to $t \leq \wr s \uparrow a \int_{\boxed{\vee}}$ and $a \sqsubset t$ is equivalent to $\wr s \uparrow a \int_{\boxed{\wedge}} \leq t$.*

Moreover, $\wr s \uparrow \cdot \int_{\boxed{\vee}}$ and $\wr s \uparrow \cdot \int_{\boxed{\wedge}}$ are increasing mappings and, for all atoms a, $\wr s \uparrow a \int_{\boxed{\wedge}} \leq \wr s \uparrow a \int_{\boxed{\vee}}$.

Proof. By induction on the height of skeletons.

A total function from atoms to ground types Π is a *morphism* if and only if it is increasing (i.e. $a_1 \leq_{\mathcal{A}} a_2$ implies $\Pi(a_1) \leq \Pi(a_2)$) and its composition with a destructor is the identity (i.e. $\boxed{\wedge}\,\Pi(a) = \boxed{\vee}\,\Pi(a) = a$).

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$\mathcal{V}_\kappa$ of *variables* of kind κ (let $\mathcal{V}$ stand for the union of all the $\mathcal{V}_\kappa$). Such variables are denoted by α or β. In the paper, the notation $\vec{\alpha}$ stands for a finite list of variables $\alpha_1, \ldots, \alpha_n$. 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 [Rém92] 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. We restrict our attention to well-formed terms and hand-sides, see Fig. 4.

Variables are interpreted in the model by *assignments* ρ that are total kind-preserving mappings from variables into ground terms (i.e. if $\alpha \in \mathcal{V}_\kappa$ then $\vdash \rho(\alpha) : \kappa$ holds); they are straightforwardly extended to terms and hand-sides by the equations: $\rho(c(\tau_1, \ldots, \tau_n)) = c(\rho(\tau_1), \ldots, \rho(\tau_n))$, $\rho(\ell\colon \tau, \tau') = \rho(\tau')[\ell \mapsto \rho(\tau)]$ and $\rho(a) = a$.

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

Terms

$$\frac{\alpha \in \mathcal{V}_\kappa}{\vdash \alpha : \kappa} \qquad \frac{c :: \kappa_1 \otimes \cdots \otimes \kappa_n \Rightarrow \mathsf{Type} \qquad \forall i \ \vdash \tau_i : \kappa_i}{\vdash c(\tau_1, \ldots, \tau_n) : \mathsf{Type}} \qquad \frac{\vdash \tau_\ell : \kappa \qquad \vdash \tau : \mathsf{Row}_L\, \kappa}{\vdash (\ell\colon \tau_\ell, \tau) : \mathsf{Row}_{\ell.L}\, \kappa}$$

Constraints

$$\frac{\forall \tau \in \bar{\bar{\tau}}_1 \cup \ldots \cup \bar{\bar{\tau}}_n \ \vdash \tau : \kappa}{\vdash \langle \bar{\bar{\tau}}_1 \rangle^* \approx \cdots \approx \langle \bar{\bar{\tau}}_n \rangle^*} \qquad \frac{\vdash \alpha_1 : \kappa \qquad \vdash \alpha_2 : \kappa}{\vdash \alpha_1 \leq^* \alpha_2} \qquad \vdash \phi_1 \sqsubset \phi_2$$

Fig. 4: Kinding

$$\frac{\begin{array}{c}\forall \tau, \tau' \in \bar{\bar{\tau}}_1 \cup \ldots \cup \bar{\bar{\tau}}_n \ \ \rho(\tau) \approx \rho(\tau') \\ \forall i \in [\![1, n]\!] \ \ \forall \tau, \tau' \in \bar{\bar{\tau}}_i \ \ \rho(\tau) = \rho(\tau')\end{array}}{\rho \vdash \langle \bar{\bar{\tau}}_1 \rangle^* \approx \cdots \approx \langle \bar{\bar{\tau}}_n \rangle^*} \qquad \frac{\rho(\alpha_1) \leq \rho(\alpha_2)}{\rho \vdash \alpha_1 \leq^* \alpha_2} \qquad \frac{\rho(\phi_1) \sqsubset \rho(\phi_2)}{\rho \vdash \phi_1 \leq^* \phi_2}$$

$$\rho \vdash \mathbf{true} \qquad \frac{\rho \vdash \Gamma_1 \qquad \rho \vdash \Gamma_2}{\rho \vdash \Gamma_1 \wedge \Gamma_2} \qquad \frac{\rho' \vdash \Gamma \qquad \rho' = \rho[\alpha \mapsto *]}{\rho \vdash \exists \alpha . \Gamma}$$

Fig. 5: Interpretation of constraints

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:

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

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. It is formally defined by the rules of Fig. 5 ($\rho' = \rho[\alpha \mapsto *]$ means that ρ' maps α to an arbitrary ground term α and agree with ρ for other variables).

We now give some explanations about the different forms of constraints and their interpretation. 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 $\tilde{\tilde{\tau}}$, or $\tilde{\tilde{\alpha}}$ when it is known to contain only

$$\mathbf{true} \wedge \Gamma \equiv \Gamma \qquad \Gamma_1 \wedge \Gamma_2 \equiv \Gamma_2 \wedge \Gamma_1 \qquad (\Gamma_1 \wedge \Gamma_2) \wedge \Gamma_3 \equiv \Gamma_1 \wedge (\Gamma_2 \wedge \Gamma_3)$$

$$\exists\alpha.\Gamma \equiv \exists\beta.\Gamma[\alpha/\beta] \ \ (\text{if } \beta \notin \mathsf{fv}(\Gamma)) \qquad \exists\alpha.(\Gamma_1 \wedge \Gamma_2) \equiv (\exists\alpha.\Gamma_1) \wedge \Gamma_2 \ \ (\text{if } \alpha \notin \mathsf{fv}(\Gamma_2))$$

Fig. 6: Constraint equivalence

variables. If $\tilde{\bar{\tau}}$ is $\langle\bar{\bar{\tau}}_1\rangle^{\iota_1} \approx \cdots \approx \langle\bar{\bar{\tau}}_n\rangle^{\iota_n}$ then $\langle\bar{\bar{\tau}}\rangle^{\iota} \approx \tilde{\bar{\tau}}$ stands for the multi-skeleton $\langle\bar{\bar{\tau}}\rangle^{\iota} \approx \langle\bar{\bar{\tau}}_1\rangle^{\iota_1} \approx \cdots \approx \langle\bar{\bar{\tau}}_n\rangle^{\iota_n}$. 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.

In the following, we restrict our attention to well-formed constraints (see Fig. 4). We consider constraints modulo the equivalence relation $\equiv$ introduced in Fig. 6 ($\mathsf{fv}(\Gamma)$ denotes the set of free variables in Γ, $\Gamma[\tau/\alpha]$ stands for the constraint obtained from Γ by replacing every free occurrence of α by τ). This equivalence expresses the commutativity and the associativity of $\wedge$; and allows α-conversion and scope-extrusion of existential quantifications. 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. In the following, we will often consider a constraint (class) Γ and choose one of its representative

member, eventually satisfying some condition. For the sake of mathematical rigor, one have to prove that every consideration made about the constraint is independent of the previous choice. However, we will often omit this check, because it is generally systematical and technical.

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 $\sqsubset$.

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 $\forall\vec{\alpha}[\Gamma].\tau$. The variables in $\vec{\alpha}$ are bound in σ; and schemes are considered equal modulo α-conversion. The kind of a scheme is that of its body. 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:

$$[\![\sigma]\!]_\rho = \uparrow\{\rho'(\tau) \mid \rho' \vdash \Gamma \text{ and } \rho' \overset{\mathcal{V}\setminus\vec{\alpha}}{=} \rho\}$$

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 $\sigma_1 \preccurlyeq \sigma_2$.

Lemma 5. *Let $\sigma_1 = \forall\vec{\alpha}[\Gamma_1].\tau$ and $\sigma_2 = \forall\vec{\alpha}[\Gamma_2].\tau$ be two schemes. If $\Gamma_1 \models \Gamma_2$ then $\sigma_2 \preccurlyeq \sigma_1$.*

Proof. Let ρ be an arbitrary assignment, and $t \in [\![\sigma_1]\!]_\rho$. There exists ρ' satisfying Γ_1 such that $\rho' \overset{\mathcal{V}\setminus\vec{\alpha}}{=} \rho$ and $\rho'(\tau) \leq t$. Because $\Gamma_1 \models \Gamma_2$, ρ' satisfies Γ_2 too. Then $\rho'(\tau) \in [\![\sigma_2]\!]_\rho$, and we conclude that $t \in [\![\sigma_2]\!]_\rho$.

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. [PS02].

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.

$$\text{GENERATE}^{\leq}\quad \frac{\alpha \leq^1 \beta}{\alpha \leq^0 \beta \wedge \alpha \approx \beta}\rightsquigarrow \qquad \text{FUSE}^{\approx}\quad \frac{\langle \alpha = \bar{\tau}\rangle^{\iota} \approx \tilde{\tau} \wedge \langle \alpha = \bar{\tau}'\rangle^{\iota'} \approx \tilde{\tau}'}{\langle \alpha = \bar{\tau} = \bar{\tau}'\rangle^{\iota\vee\iota'} \approx \tilde{\tau} \approx \tilde{\tau}'}\rightsquigarrow \qquad \text{FUSE}^{=}\quad \frac{\langle \alpha = \bar{\tau}\rangle^{\iota} \approx \langle \alpha = \bar{\tau}'\rangle^{\iota'} \approx \tilde{\tau}}{\langle \alpha = \bar{\tau} = \bar{\tau}'\rangle^{\iota\vee\iota'} \approx \tilde{\tau}}\rightsquigarrow$$

$$\text{DECOMPOSE}^{\approx}\quad \frac{\langle \bar{\tau} = f(\vec{\alpha})\rangle^1 \approx \langle \bar{\tau}' = f(\vec{\beta})\rangle^1 \approx \tilde{\tau}}{\langle \bar{\tau} = f(\vec{\alpha})\rangle^1 \approx \langle \bar{\tau}' = f(\vec{\beta})\rangle^0 \approx \tilde{\tau} \wedge_i \alpha_i \approx^{v_i(f)} \beta_i}\rightsquigarrow \qquad \text{DECOMPOSE}^{=}\quad \frac{\langle \bar{\tau} = f(\vec{\alpha}) = f(\vec{\beta})\rangle^{\iota} \approx \tilde{\tau}}{\langle \bar{\tau} = f(\vec{\alpha})\rangle^{\iota} \approx \tilde{\tau} \wedge_i \alpha_i = \beta_i}\rightsquigarrow$$

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

$$\text{GENERALIZE}\quad \frac{\tilde{\tau}[\tau/\alpha]}{\exists \alpha.[\tilde{\tau} \wedge \langle \alpha = \tau\rangle^1]}\rightsquigarrow \quad (\text{if } \alpha \in \mathsf{fv}(\tilde{\tau}) \setminus \mathsf{Terms}(\tilde{\tau}) \setminus \mathsf{fv}(\tau) \text{ and } \tau \notin \mathcal{V})$$

Fig. 7: Unification (rewriting system Ω_u)

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 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

$$\Gamma_i^o \vdash \frac{\Gamma_i}{\Gamma_i'}\rightsquigarrow$$

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

Expansion (rewriting system Ω_e)

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

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

Decomposition (rewriting system Ω_d)

$$\begin{array}{l}\alpha = f(\vec{\alpha})\\ \beta = f(\vec{\beta})\end{array} \vdash \frac{\alpha \leq \beta}{\wedge_i \alpha_i \leq^{v_i(f)} \beta_i}\rightsquigarrow^{\textsc{decompose}^{\leq}} \qquad \alpha = f(\vec{\alpha}) \vdash \frac{\alpha \sqsubset \phi}{\wedge_{i\in l(f)} \alpha_i \sqsubset \phi}\rightsquigarrow^{\textsc{decompose}^{l}} \qquad \alpha = f(\vec{\alpha}) \vdash \frac{\phi \sqsubset \alpha}{\wedge_{i\in r(f)} \phi \sqsubset \alpha_i}\rightsquigarrow^{\textsc{decompose}^{r}}$$

Fig. 8: Expansion and decomposition (rewriting system Ω_{ed})

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 [Rém92]. They are described by the rewriting system of figure 7, 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 $\textsc{generate}^{\leq}$, 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 $\textsc{fuse}^{\approx}$ merges two multi-skeletons which have a common variable and then $\textsc{fuse}^{=}$ 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 $\textsc{decompose}^{\approx}$ 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. ($\mathsf{Terms}(\tilde{\tilde{\tau}})$ stands for the set of top-level terms of $\tilde{\tilde{\tau}}$.)

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*.

As explained above the flags carried by multi-equations and inequalities are used in this step of the solving process in order to ensure termination. Indeed, the algorithm preserve the following invariant on the constraint $\Gamma \doteq \exists\vec{\alpha}.\Gamma'$ it manipulates:

(1) for every multi-skeleton $\langle\bar{\bar{\tau}}\rangle^0 \approx \tilde{\tilde{\tau}}$ in Γ', there exists a term $f(\vec{\alpha})$ in $\bar{\bar{\tau}}$ and a term $f'(\vec{\alpha}')$ in one of $\tilde{\tilde{\tau}}$'s multi-equations flagged 1 such that, either $f' <_{\mathcal{L}} f$ or $f = f'$ and for every i, $\alpha_i \approx^{v_i(f)} \alpha'_i \in \Gamma'$.
(2) for every strong inequality $\alpha \leq^{\iota} \beta$ in Γ', either ι is 1 or $\alpha \approx \beta \in \Gamma'$.

Any constraint which satisfies the above properties is said to be well-skeletonized. Obviously, any input constraint is assumed to be well-skeletonized; but, this is not restrictive because it may be enforced by setting all flags to 1.

Lemma 6 (Soundness). *The rewriting system Ω_u is sound.*

Proof. By inspection of rules of Figure 7 and the definitions of Section 3.

Lemma 7 (Termination). *The rewriting system Ω_u terminates.*

Proof. A label is said to be *apparent* in a constraint if it appears in at least one of its terms. All rules of Ω_u preserve the set of apparent labels, i.e. they do not cause new labels to appear. We measure the occurrence $(\ell\colon\tau_1, \tau_2)$ of a row label in constraints, by its *weight*: the pair (L, ℓ) where L is the codomain of τ_2. Weights are ordered by the lexicographical product of $\supseteq$ and $\leq_{\mathcal{L}}$, which does not admit any infinite strictly decreasing chain with a bounded set of apparent labels.

Let us now measure a constraint by the following quantities, ordered lexicographically:

(1) The number of strong inequalities carrying the flag 1,
(2) The multi-set of labels weights in the constraint,
(3) The number of type constructors in the constraint,
(4) The sum of *heights* of terms (including sub-terms) in the constraint,
(5) The number of multi-equations carrying the flag 1,
(6) The number of multi-equations.

Rule GENERATE$^{\leq}$ decreases (1) by switching the flag of an inequality from 1 to 0. All other rules do not deal with inequalities, so they keep (1) unchanged. Rules FUSE$^{\approx}$, FUSE$^{=}$ and DECOMPOSE$^{\approx}$ do not affect the terms appearing in the constraint, so they keep (2), (3) and (4) unchanged. Moreover FUSE$^{\approx}$ and FUSE$^{=}$ may not increase (5) (it decreases if both merged multi-equations are flagged 1) and must decrease (6). DECOMPOSE$^{\approx}$ always decreases (5). Rule DECOMPOSE$^{=}$ removes exactly one symbol of the constraint; hence reducing either (2) or (3). Rule MUTATE replaces an occurrence of a row term of weight (L, ℓ_b) by three strictly inferior ones: (L, ℓ_a), $(\ell_b.L, \ell_b)$ and $(\ell_a.L, \ell_a)$. It therefore decreases (2). Lastly, rule GENERALIZE may not increase (2) or (3) and decreases (4).

Because the order on measures has no infinite strictly decreasing chain with a finite number of apparent labels (as the lexicographical product of orders that satisfy the same property), we conclude that the rewriting system terminates.

The following lemma states that the rewriting system Ω_u preserves well-skeletonization throughout computation.

Lemma 8. *Assume Γ is a well-skeletonized constraint. If $\Gamma \xrightarrow{\Omega_u} \Gamma'$ then Γ' is well-skeletonized too.*

Proof. By inspection of rules of Figure 7.

Lemma 9 (Completeness). *Assume Γ is a well-skeletonized constraint. If Γ is Ω_u-normal then either Γ is unified or it is an unification error.*

Proof. Let us consider $\Gamma \doteq \exists\vec{\alpha}.\Gamma'$ a well-skeletonized constraint which is not a unification error. We now check that if Γ is Ω_u-normal then it satisfies the five properties defining unified constraints.

Because FUSE$^{\approx}$ does not apply, every variable appears in at most one multi-skeleton. Moreover, because FUSE$^{=}$ does not apply, every variable appears in at most one multi-equation, hence **(2)**. Because GENERALIZE does not apply, every term occurring in one of Γ''s multi-skeletons is small; **(5)** follows. Because Γ is not an unification error, two type constructors cannot appear in the same multi-skeleton, and because MUTATE does not apply and $\leq_{\mathcal{L}}$ is a total order, two different row labels cannot occur in the same multi-skeleton. Besides, assume $f(\alpha_1, \ldots, \alpha_n) \approx f(\beta_1, \ldots, \beta_n) \in \Gamma'$. By Lemma 8, Γ' is well-skeletonized. Because DECOMPOSE$^{\approx}$ does not apply, Γ' cannot contain any multi-skeleton of the form $\langle f(\alpha_1, \ldots, \alpha_n) = *\rangle^1 \approx \langle f(\alpha'_1, \ldots, \alpha'_n) = *\rangle^1 \approx *$. It follows that $\alpha_i \approx^{v_f(i)} \beta_i \in \Gamma'$ holds for all i. This yields **(3)**. Two terms appearing in the same multi-equation of Γ' must be of the form $f(\alpha_1, \ldots, \alpha_n)$ and $f(\alpha'_1, \ldots, \alpha'_n)$. Because DECOMPOSE$^{=}$, this cannot arise, hence we have **(4)**. Lastly, because Γ is well-skeletonized and GENERATE$^{\leq}$ does not apply, we have **(1)**.

Theorem 1 (Unification). *Assume Γ is a well-skeletonized constraint. Then Ω_u terminates on Γ. If $\Gamma \mathrel{-\Omega_u\twoheadrightarrow} \Gamma'$ then Γ' is equivalent to Γ and either unified or erroneous.*

Proof. By Lemmas 6, 7 and 9.

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 [Sim02], 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*. Let us consider an $\exists$-free constraint Γ. We define the sub-term relation $\prec_\Gamma$ between variables as the smallest binary relation such that: if $\alpha \approx \tau \in \Gamma$ (for some non-variable term τ), then $\beta \prec_\Gamma \alpha$ holds for every variable β which is in the same multi-skeleton that a variable appearing within τ.

(For the purpose of formulating the invariant preserved by the rewriting system Ω_{ed} (Section 5.3), we also define $\prec_\Gamma^*$ the smallest binary relation containing $\prec_\Gamma$ and such that $\beta' \approx \beta \in \Gamma$, $\beta \prec_\Gamma^* \alpha$ and $\alpha \approx \alpha' \in \Gamma$ imply $\beta' \prec_\Gamma^* \alpha'$. Let us remark that if Γ is unified then $\prec_\Gamma^*$ and $\prec_\Gamma$ coincide.)

Lemma 10. *Let Γ be an $\exists$-free constraint. If $\beta \prec_\Gamma \alpha$ then for all $\rho \vdash \Gamma$ we have either $\mathsf{h}(\rho(\beta)) < \mathsf{h}(\rho(\alpha))$ or α and β are row variables, $\mathsf{h}(\rho(\beta)) \leq \mathsf{h}(\rho(\alpha))$ and $\mathsf{dom}(\rho(\beta)) \subset \mathsf{dom}(\rho(\alpha))$.*

Proof. Assume τ is a non-variable sub-term involving the variable β and appearing in the same multi-skeleton as α. Because $\rho \vdash \Gamma$, $\rho(\tau) \approx \rho(\alpha)$ holds. If $\tau = (\ell_1 : *, \ldots, \ell_n : *, \beta)$ (for some $n \geq 1$) then $\mathsf{h}(\rho(\beta)) \leq \mathsf{h}(\rho(\alpha))$ and $\mathsf{dom}(\beta) \subset \mathsf{dom}(\alpha)$. Otherwise $\mathsf{h}(\rho(\beta)) < \mathsf{h}(\rho(\alpha))$. Conclude by Lemma 1.

Theorem 2 (Occur-check). *If $\Gamma \doteq \exists\vec{\alpha}.\Gamma'$ is satisfiable then $\prec_{\Gamma'}$ is acyclic.*

Proof. Let ρ' be an assignment which satisfies Γ'. Assume $\prec_{\Gamma'}$ has a cycle and apply Lemma 10 along it. This yields, for every variable α of the cycle, either $\mathsf{h}(\rho'(\alpha)) < \mathsf{h}(\rho'(\alpha))$ or $\mathsf{dom}(\rho'(\alpha)) \subset \mathsf{dom}(\rho'(\alpha))$. Hence the contradiction.

A constraint $\Gamma \doteq \exists\vec{\alpha}.\Gamma'$ *satisfies the occur-check* if and only if $\prec_{\Gamma'}$ is acyclic. A variable α is *terminal* in Γ' if it is minimal for $\prec_{\Gamma'}$, i.e. there is no β such that $\beta \prec_{\Gamma'} \alpha$. These are variables about the structure of whose Γ tells nothing. In particular, variables of kind Atom and variables that do not appear in the constraint are terminal. A term is said to be terminal if and only if it is a terminal variable.

Practically, the occur-check may be performed in linear time applying by standard graph algorithms. 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 subvariables 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 8) 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}. (In the premises of these three rules, $\alpha = f(\vec{\alpha})$ is a shorthand for $\langle\alpha = * = f(\vec{\alpha})\rangle^{*} \approx *$.)

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

We now address the correction proof of expansion and decomposition. A constraint $\Gamma \doteq \exists\vec{\alpha}.\Gamma'$ satisfies the *pre-occur-check* if the relation $\prec^*_{\Gamma'}$ is acyclic.

Lemma 11 (Soundness). *The rewriting system Ω_{ed} is sound, preserves the pre-occur-check and does not introduce unification errors.*

Proof. By inspection of rules of Figure 7 and the definitions of Section 3.

Lemma 12 (Termination). *The rewriting system Ω_{ed} terminates on every input which satisfies the occur-check.*

Proof. Because the rewriting system Ω_{ed} preserves the pre-occur-check (Lemma 11), we restrict our attention in this proof to constraints which satisfy it.

Let us consider a constraint $\Gamma \doteq \exists\vec{\alpha}.\Gamma'$. The *height* of a variable in Γ' is the integer $\max\{n \mid \exists\alpha_1 \cdots \alpha_n\ \alpha_n \prec^*_{\Gamma'} \cdots \prec^*_{\Gamma'} \alpha_1 \prec^*_{\Gamma'} \alpha\}$. All variables appearing in the same multi-equation in Γ' have the same height, so we define the height of a multi-equation as the height of its variables. Similarly, let the height of $\alpha_1 \leq \alpha_2$ be the common height of α_1 and α_2 an the height of $\phi_1 \sqsubset \phi_2$ the sum of the heights of ϕ_1 and ϕ_2 (by convention, the height of a constant hand-side is 0).

We now measure a constraint by the lexicographical product of the following quantities:

(1) The multi-set of the heights of its multi-equations containing only variables,
(2) The multi-set of the heights of its inequalities.

Rule EXPAND inserts a non-variable term in a multi-equation which contain only variables and adds only multi-equations of lesser height, so it decreases (1). EXP-FUSE$^{\approx}$ and EXP-FUSE$^{=}$ remove one multi-equation containing only variables and do not affect the height of other ones, so they also reduce (1). Lastly, rules DECOMPOSE$^{\leq}$, DECOMPOSE^{l} and DECOMPOSE^{r} replace an inequality with several ones of lesser height, so they decrease (2) and do not affect (1).

Because the order on measures is well-founded, we conclude that Ω_{ed} terminates on every input which satisfies the occur-check.

Lemma 13 (Completeness). *Assume Γ_1 is unified. If $\Gamma_1 \mathrel{-\!\!\Omega_{ed}\!\!\twoheadrightarrow} \Gamma_2$ then Γ_2 is reduced.*

Proof. By Lemma 11, Γ_2 satisfies the occur-check and is not an unification error.

We say that a multi-equation is *trivial* if it contains only variables. By inspecting rules of Figure 8, one check that Ω_{ed} preserves the properties (1) and (3) to (5) of unified constraints, and the following: **(2')** *Every variable appears in at most one* non-trivial *multi-equation.* Then Γ_2 satisfies these properties, and because it is Ω_{ed}-normal, we conclude that Γ_2 is unified.

Let $\Gamma_2 \doteq \exists\vec{\alpha}_2.\Gamma'_2$. Because Γ_2 is unified, for every variable α, there exists at most one non-variable term τ such that $\alpha = \tau \in \Gamma'_2$. Moreover, it appears in at most one multi-equation. If α is non-terminal, it must appear in one multi-equation and because EXPAND does not apply, this multi-equation must contain a non-variable term.

Lastly, because none of DECOMPOSE$^{\leq}$, DECOMPOSE^{l} and DECOMPOSE^{r} apply, each of Γ_2's inequalities involve only terminal variables.

Theorem 3 (Expansion and decomposition). *Let Γ be a unified constraint which satisfies the occur-check. Ω_{ed} terminates on Γ. If $\Gamma \xrightarrow{\Omega_{ed}}\!\!\!\!\rightarrow \Gamma'$ then Γ' is equivalent to Γ and reduced.*

Proof. By Lemmas 12, 11 and 13.

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.

We now assume that Γ is $\exists$-free and let ρ be an assignment. We define its *extension along* Γ, written $\lceil \rho \rceil_\Gamma$ by:

$$\lceil \rho \rceil_\Gamma(\alpha) = \begin{cases} \rho(\alpha) & \text{if } \alpha \text{ is terminal in } \Gamma \\ f(\lceil \rho \rceil_\Gamma(\alpha_1), \ldots, \lceil \rho \rceil_\Gamma(\alpha_n)) & \text{if } \alpha = f(\alpha_1, \ldots, \alpha_n) \in \Gamma \end{cases}$$

Because Γ is reduced, these equations properly define an assignment $\lceil \rho \rceil_\Gamma$: for every non-terminal variable α, there exists a unique term $\tau = f(\alpha_1, \ldots, \alpha_n)$ such that $\alpha = \tau \in \Gamma$; for all i, $\alpha_i \prec_\Gamma \alpha$; and $\prec_\Gamma$ is well-founded.

Lemma 14. *Let Γ be an $\exists$-free reduced constraint. If $\rho \vdash \lfloor \Gamma \rfloor$ then $\lceil \rho \rceil_\Gamma \vdash \Gamma$.*

Proof. Let $\rho' = \lceil \rho \rceil_\Gamma$. Because $\rho \vdash \lfloor \Gamma \rfloor$ and, for every terminal variable α, $\rho(\alpha) = \rho'(\alpha)$, ρ' satisfies every elementary constraint of Γ that involves terminal variables only. Moreover, because Γ is expanded and decomposed, only multi-skeletons in Γ may include non-terminal variables. Then, it is sufficient to prove that for all non-terminal (small) terms τ and τ', $\tau \approx \tau' \in \Gamma$ (resp. $\tau = \tau' \in \Gamma$) implies $\rho'(\tau) \approx \rho'(\tau')$ (resp. $\rho'(\tau) = \rho'(\tau')$). We proceed by well-founded induction on $\prec_\Gamma$.

∘ *Case* $f(\alpha_1, \ldots, \alpha_n) \approx f(\beta_1, \ldots, \beta_n) \in \Gamma$. Because Γ is unified, for all i, we have $\alpha_i \approx \beta_i \in \Gamma$. Conclude by induction hypothesis.

∘ *Case* $\alpha \approx f(\beta_1, \ldots, \beta_n) \in \Gamma$. Because Γ is expanded, we may consider $f(\alpha_1, \ldots, \alpha_n)$ such that $\alpha = f(\alpha_1, \ldots, \alpha_n) \in \Gamma$. Because Γ is unified, for all i, we have $\alpha_i \approx \beta_i \in \Gamma$. Conclude by induction hypothesis.

∘ *Case* $\alpha \approx \beta \in \Gamma$. Because Γ is expanded, we may consider $f(\alpha_1, \ldots, \alpha_n)$ and $f'(\beta_1, \ldots, \beta_{n'})$ such that $\alpha = f(\alpha_1, \ldots, \alpha_n) \in \Gamma$ and $\beta = f'(\beta_1, \ldots, \beta_{n'}) \in \Gamma$. Because Γ is not an unification error, $f = f'$ and $n = n'$. Because Γ is unified, for all i, we have $\alpha_i \approx \beta_i \in \Gamma$. Conclude by induction hypothesis.

∘ *Case* $\alpha = \beta \in \Gamma$. Because Γ is reduced, there exist a unique small term $f(\alpha_1, \ldots, \alpha_n)$ such that $\alpha = f(\alpha_1, \ldots, \alpha_n) \in \Gamma$ and $\beta = f(\alpha_1, \ldots, \alpha_n) \in \Gamma$ hold. By definition, $\rho'(\alpha) = \rho'(\beta) = f(\rho'(\alpha_1), \ldots, \rho'(\alpha_n))$.

∘ *Case* $\alpha = \tau \in \Gamma$. Because Γ is reduced, τ has the form $f(\alpha_1, \ldots, \alpha_n)$. By definition, $\rho'(\alpha) = f(\rho'(\alpha_1), \ldots, \rho'(\alpha_n))$. Hence $\rho'(\alpha) = \rho'(\tau)$.

∘ *Case* $\tau = \tau' \in \Gamma$. Because Γ is expanded, $\tau = \tau'$ must hold.

Theorem 4. *If Γ is reduced, then the satisfiability of Γ and $\lfloor \Gamma \rfloor$ are equivalent.*

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

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

Fig. 9: Syntactic implication

Proof. Because $\Gamma \models \lfloor \Gamma \rfloor$, if Γ is satisfiable then $\lfloor \Gamma \rfloor$ is satisfiable too. Conversely, if $\lfloor \Gamma \rfloor$ is satisfiable, by Lemma 14, Γ is satisfiable too.

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 9 and may be checked in linear time.

For the purpose of proving the correctness of this algorithm, we states a few auxiliary definitions and lemmas. First and foremost, we prove the existence of a morphism Π_0, which will be useful for defining particular solutions of constraints.

Lemma 15 (Morphism). *There exists a morphism Π_0.*

Proof. Let c_0 be a type constructor of signature $\mathsf{Row}^{k_1}\ \mathsf{Atom} \otimes \cdots \otimes \mathsf{Row}^{k_n}\ \mathsf{Atom} \Rightarrow \mathsf{Type}$ (we supposed the existence of such a constructor in Section 3.1). For every atom a, we define $\Pi_0(a) = c_0(\partial^{k_1} a_1, \ldots, \partial^{k_n} a_n)$ where a_i is a if $i \in l(c_0) \cup r(c_0)$ and $\bot$ otherwise. Because for $l(c_0) \cup r(c_0)$ is a non-empty subset of $\{i \mid v_i(c) = \oplus\}$, we check that Π_0 is a morphism.

We define the lower and upper bounds of a variable in Γ by: $\mathrm{lb}_\Gamma(\alpha) = \sqcup\{a \mid \Gamma \mathrel{|\!\!\approx} a \sqsubset \alpha\}$ and $\mathrm{ub}_\Gamma(\alpha) = \sqcap\{a \mid \Gamma \mathrel{|\!\!\approx} \alpha \sqsubset a\}$. If λ is a total mapping from variables to atoms, we extend λ to hand-sides by $\lambda(a) = a$. Given Π ranging from atoms to types, we also define an assignment $[\lambda/\Pi]$ by:

$$[\lambda/\Pi](\alpha) = \begin{cases} \partial^n \lambda(\alpha) & \text{if} \vdash \alpha : \mathsf{Row}^n\ \mathsf{Atom} \\ \partial^n \Pi(\lambda(\alpha)) & \text{if} \vdash \alpha : \mathsf{Row}^n\ \mathsf{Type} \end{cases}$$

Lemma 16. *Let Γ be a $\exists$-free and atomic constraint. If $\Gamma \mathrel{|\!\!\approx} \alpha_1 \leq \alpha_2$ (resp. $\Gamma \mathrel{|\!\!\approx} \phi_1 \sqsubset \phi_2$) then $\Gamma \models \alpha_1 \leq \alpha_2$ (resp. $\Gamma \models \phi_1 \sqsubset \phi_2$).*

Proof. By induction on the derivation of $\Gamma \models\!\approx \alpha_1 \leq \alpha_2$ (resp. $\Gamma \models\!\approx \phi_1 \sqsubset \phi_2$).

Lemma 17. *Let Γ be a $\exists$-free and atomic constraint. For all variable α, $\Gamma \models \mathrm{lb}_\Gamma(\alpha) \sqsubset \alpha$ and $\Gamma \models \alpha \sqsubset \mathrm{ub}_\Gamma(\alpha)$.*

Proof. By Lemma 16.

Lemma 18. *Let Γ be an $\exists$-free atomic constraint and λ be lb_Γ or ub_Γ. If for all a_1, a_2, $\Gamma \models\!\approx a_1 \sqsubset a_2$ implies $a_1 \leq_{\mathcal{A}} a_2$, then for all $\Gamma \models\!\approx \phi_1 \sqsubset \phi_2$ (resp. $\Gamma \models\!\approx \alpha_1 \leq \alpha_2$), $\lambda(\phi_1) \leq_{\mathcal{A}} \lambda(\phi_2)$ (resp. $\lambda(\alpha_1) \leq_{\mathcal{A}} \lambda(\alpha_2)$) holds.*

Proof. Let us assume $\lambda = \mathrm{lb}_\Gamma$ (the case $\lambda = \mathrm{ub}_\Gamma$ is symmetrical).

- *Case* $\Gamma \models\!\approx a_1 \sqsubset a_2$. By hypothesis, $a_1 \leq_{\mathcal{A}} a_2$ holds.
- *Case* $\Gamma \models\!\approx a \sqsubset \alpha$. By the definition of lb_Γ, this implies $a \leq_{\mathcal{A}} \mathrm{lb}_\Gamma(\alpha)$.
- *Case* $\Gamma \models\!\approx \alpha \sqsubset a$. If $\Gamma \models\!\approx a' \sqsubset \alpha$ then, applying the weak transitivity rule, we obtain $\Gamma \models\!\approx a' \sqsubset a$. By hypothesis, this yields $a' \leq_{\mathcal{A}} a$. We conclude that $\mathrm{lb}_\Gamma(\alpha) \leq_{\mathcal{A}} a$.
- *Case* $\Gamma \models\!\approx \alpha_1 \sqsubset \alpha_2$ or $\Gamma \models\!\approx \alpha_1 \leq \alpha_2$. If $\Gamma \models\!\approx a \sqsubset \alpha_1$ then, applying one of Figure 9's transitivity rules, $\Gamma \models\!\approx a \sqsubset \alpha_2$. It follows that $\mathrm{lb}_\Gamma(\alpha_1) \leq_{\mathcal{A}} \mathrm{lb}_\Gamma(\alpha_2)$.

Lemma 19. *Let Γ be an $\exists$-free atomic constraint. If $\Gamma \models\!\approx \phi_1 \sqsubset \phi_2$ (resp. $\Gamma \models\!\approx \alpha_1 \leq \alpha_2$) implies $\lambda(\phi_1) \leq_{\mathcal{A}} \lambda(\phi_2)$ (resp. $\lambda(\alpha_1) \leq_{\mathcal{A}} \lambda(\alpha_2)$) and Π is a morphism then $[\lambda/\Pi] \vdash \Gamma$.*

Proof. Let $\rho = [\lambda/\Pi]$. By inspection of the definition of $[\lambda/\Pi]$, we check that if α_1 and α_2 have the same kind then $\rho(\alpha_1) \approx \rho(\alpha_2)$ holds. Besides, for every variable α, $\sqcup\, \rho(\alpha) = \sqcap\, \rho(\alpha) = \lambda(\alpha)$.

If $\alpha_1 \approx \alpha_2 \in \Gamma$ then $\rho(\alpha_1) \approx \rho(\alpha_2)$. If $\alpha_1 = \alpha_2 \in \Gamma$ then $\Gamma \models\!\approx \alpha_1 \leq \alpha_2$ and $\Gamma \models\!\approx \alpha_2 \leq \alpha_1$. This yields $\lambda(\alpha_1) = \lambda(\alpha_2)$ and hence $\rho(\alpha_1) = \rho(\alpha_2)$. Lastly, if $\phi_1 \sqsubset \phi_2 \in \Gamma$ then $\lambda(\phi_1) \leq_{\mathcal{A}} \lambda(\phi_2)$ and $\rho(\phi_1) \sqsubset \rho(\phi_2)$. Similarly, if $\alpha_1 \leq \alpha_2 \in \Gamma$ then $\lambda(\alpha_1) \leq_{\mathcal{A}} \lambda(\alpha_2)$ and $\rho(\alpha_1) \leq \rho(\alpha_2)$.

The following theorem states the criterion of satisfiability of atomic constraints involved by our algorithm.

Theorem 5. *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, $\Gamma' \models\!\approx a_1 \sqsubset a_2$ implies $a_1 \leq_{\mathcal{A}} a_2$.*

Proof. Γ is satisfiable if and only if Γ' is satisfiable. By Lemma 16, $\Gamma' \models\!\approx a_1 \sqsubset a_2$ implies $\Gamma' \models a_1 \sqsubset a_2$. We conclude that if $a_1 \not\leq_{\mathcal{A}} a_2$ then Γ is not satisfiable.

We now assume that for every atoms a_1 and a_2, $\Gamma' \models\!\approx a_1 \sqsubset a_2$ implies $a_1 \leq_{\mathcal{A}} a_2$. By Lemma 18, if $\Gamma' \models\!\approx \phi_1 \sqsubset \phi_2$ (resp. $\Gamma' \models\!\approx \alpha_1 \leq \alpha_2$) then $\mathrm{lb}_{\Gamma'}(\phi_1) \leq_{\mathcal{A}} \mathrm{lb}_{\Gamma'}(\phi_2)$ (resp. $\mathrm{lb}_{\Gamma'}(\alpha_1) \leq_{\mathcal{A}} \mathrm{lb}_{\Gamma'}(\alpha_2)$). Then, by Assumption 15 and lemma 19, $[\mathrm{lb}_{\Gamma'}/\Pi_0] \vdash \Gamma'$.

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 [Pot03].

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) [Tar75]. 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 [FFSA98], 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 [FM89,TS96,Pot01] 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:

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

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.

Given two assignments ρ_1, ρ_2 and a scheme σ, we write $\rho_1 \leq^{[\sigma]} \rho_2$ if and only if $\sigma_1 \vdash \alpha : +$ (resp. $\sigma_1 \vdash \alpha : -$) implies $\rho_1(\alpha) \leq \rho_2(\alpha)$ (resp. $\rho_2(\alpha) \leq \rho_1(\alpha)$)

Lemma 20 (Scheme comparison). *Let $\sigma_1 = \forall\vec{\alpha}[\Gamma_1].\tau$ and $\sigma_2 = \forall\vec{\alpha}[\Gamma_2].\tau$ be two schemes. If for all $\rho_2 \vdash \Gamma_2$ there exists $\rho_1 \vdash \Gamma_1$ such that $\rho_1 \leq^{[\sigma]} \rho_2$ then $\sigma_1 \preccurlyeq \sigma_2$.*

Proof. Let ρ be an arbitrary assignment. We consider $t \in [\![\sigma_2]\!]_\rho$. There exists ρ_2 satisfying Γ_2 such that $\rho_2 \overset{\mathcal{V}\setminus\vec{\alpha}}{=} \rho$ and $\rho_2(\tau) \leq t$. By hypothesis, there exists $\rho_1 \vdash \Gamma_1$ such that $\rho_1(\tau) \leq \rho_2(\tau)$ and, for all $\alpha \in \vec{\alpha}$, $\rho_1(\alpha) = \rho_2(\alpha)$. This yields $t \in [\![\sigma_2]\!]_\rho$. We conclude that, for all ρ, $[\![\sigma_2]\!]_\rho \subseteq [\![\sigma_1]\!]_\rho$.

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* [EST95] and, by Aiken and Fähndrich [AF96] 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 6 (Chains). *Let $\sigma = \forall\vec{\alpha}[\Gamma \wedge \langle\bar{\bar{\alpha}}\rangle \approx \langle\bar{\bar{\tau}}\rangle \approx \bar{\bar{\tau}}].\tau$ 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 $\forall\vec{\alpha}[\Gamma \wedge \langle\bar{\bar{\alpha}} = \bar{\bar{\tau}}\rangle \approx \bar{\bar{\tau}}].\tau$.*

Proof. We address the case of a non-negative multi-equation which has a unique predecessor; the other one is symmetric. Let $\Gamma_1 = \Gamma \wedge \langle\bar{\bar{\alpha}}\rangle \approx \langle\bar{\bar{\tau}}\rangle \approx \bar{\bar{\tau}}$ and $\Gamma_2 = \Gamma \wedge \langle\bar{\bar{\alpha}} = \bar{\bar{\tau}}\rangle \approx \bar{\bar{\tau}}$. We have $\Gamma_2 \models \Gamma_1$, and hence $\forall\vec{\alpha}[\Gamma_1].\tau \preccurlyeq \forall\vec{\alpha}[\Gamma_2].\tau$.

We now consider $\rho_1 \vdash \Gamma_1$. Let V be the set of variables which are "above" $\bar{\bar{\alpha}}$ in Γ, namely $\{\beta \mid \exists \alpha \in \bar{\bar{\alpha}}\ \alpha \prec_\Gamma^+ \beta\}$. We define the assignment ρ_2 by:

$$\rho_2(\beta) = \begin{cases} f(\rho_2(\beta_1), \ldots, \rho_2(\beta_n)) & \text{if } \beta \in V \text{ with } \beta = f(\beta_1, \ldots, \beta_n) \in \Gamma \\ \rho_1(\bar{\bar{\tau}}) & \text{if } \beta \in \bar{\bar{\alpha}} \\ \rho_1(\beta) & \text{otherwise} \end{cases}$$

By hypothesis, $\bar{\bar{\tau}} \leq \bar{\bar{\alpha}} \in \Gamma$. Then $\rho_1 \vdash \Gamma$ yields $\rho_1(\bar{\bar{\tau}}) \leq \rho_1(\bar{\bar{\alpha}})$ and, hence, $\rho_2(\bar{\bar{\alpha}}) \leq \rho_1(\bar{\bar{\alpha}})$. Γ_1 is expanded down to $\bar{\bar{\alpha}}$. So, by induction on $\prec_\Gamma$, and because $\bar{\bar{\alpha}}$ is not negative, we establish that $\rho_2 \leq^{[\sigma]} \rho_1$.

We now check that ρ_2 is a solution for Γ_2. Assume $\tau_1 \approx \tau_2 \in \Gamma_2$. We have $\tau_1 \approx \tau_2 \in \Gamma_1$ and hence $\rho_1(\tau_1) \approx \rho_1(\tau_2)$. Because $\rho_2 \leq^{[\sigma]} \rho_1$, this yields $\rho_2(\tau_1) \approx \rho_2(\tau_2)$. Similarly, suppose $\tau_1 = \tau_2 \in \Gamma_2$. If $\tau_1 = \tau_2 \in \Gamma_1$, $\rho_2(\tau_1) = \rho_2(\tau_2)$ follows because Γ_1 is unified and expanded down to $\bar{\bar{\alpha}}$. If $\tau_1 = \tau_2 \notin \Gamma_1$ then one of τ_1 and τ_2 belongs to $\bar{\bar{\alpha}}$ and the other to $\bar{\bar{\tau}}$, what yields $\rho_2(\tau_1) = \rho_1(\bar{\bar{\tau}}) = \rho_2(\tau_2)$.

We now consider $\beta_1 \leq \beta_2 \in \Gamma_2$. We have $\beta_1 \leq \beta_2 \in \Gamma_1$ and $\rho_1(\beta_1) \leq \rho_1(\beta_2)$. If β_2 belongs to $\bar{\bar{\alpha}}$ then β_1 is in $\bar{\bar{\tau}}$ and $\rho_2(\beta_1) = \rho_2(\beta_2)$. Otherwise, because Γ is expanded and decomposed down to $\bar{\bar{\alpha}}$, β_1 and β_2 are not in V. Thus $\rho_2(\beta_2) = \rho_1(\beta_2)$ and $\rho_2(\beta_1) \leq \rho_1(\beta_1)$. This yields $\rho_2(\beta_1) \leq \rho_2(\beta_2)$. The case $\phi_1 \sqsubset \phi_2 \in \Gamma'$ is similar.

Then, we have $\rho_2 \leq^{[\sigma]} \rho_1$ and $\rho_2 \vdash \Gamma_2$. By Lemma 20, we conclude $\forall \vec{\alpha}[\Gamma_2].\tau \preccurlyeq \forall \vec{\alpha}[\Gamma_1].\tau$.

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 [TS96] in the case of non-structural subtyping.

Let $\sigma = \forall \vec{\alpha}[\Gamma].\tau$ be a scheme with Γ reduced and satisfiable. 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 $\mathrm{ub}_\Gamma\, \alpha \neq \top$

Theorem 7 (Garbage collection). *$GC(\sigma)$ is equivalent to σ.*

Proof. Let $\sigma = \forall \vec{\alpha}[\Gamma].\tau$ and $GC(\sigma) = \forall \vec{\alpha}[\Gamma'].\tau$. By Lemmas 16 and 17, $\Gamma \models \Gamma'$ holds. This yields $GC(\sigma) \preccurlyeq \sigma$. Let ρ' be an assignment which satisfies Γ'.

For every variable α, we consider the set of positive or negative variables β such that $\alpha \approx \beta \in \Gamma$. All these variables must be interpreted by ρ' within the same ground skeleton. So, we define $\bot(\alpha)$ the bottom element of this ground skeleton. (In the case where the previous set is empty, we arbitrarily choose $\bot(\alpha) = \Pi_0(\bot)$.) Let us also introduce

$$\begin{aligned} \mathrm{spred}(\alpha) &= \{\beta \mid \Gamma \mathrel{|\!\!\approx} \beta \leq \alpha \text{ and } \sigma \vdash \beta : -\} \\ \mathrm{wpred}(\alpha) &= \{\beta \mid \Gamma \mathrel{|\!\!\approx} \beta \sqsubset \alpha \text{ and } \sigma \vdash \beta : -\} \end{aligned}$$

Then we let $\rho(\alpha)$ be the union of $\sqcup\{\rho'(\beta) \mid \beta \in \mathrm{spred}(\alpha)\}$ and $\wr \bot(\alpha) \uparrow \sqcup\{\asymp \rho'(\beta) \mid \beta \in \mathrm{wpred}(\alpha\}\int_{\boxed{\wr}}$ and $\wr \bot(\alpha) \uparrow \mathrm{lb}_\Gamma(\alpha)\int_{\boxed{\wr}}$.

We now check that $\rho \vdash \lfloor \Gamma \rfloor$. If $\alpha_1 \approx \alpha_2 \in \lfloor \Gamma \rfloor$ then, because Γ is unified, $\bot(\alpha_1) = \bot(\alpha_2)$. This yields $\rho(\alpha_1) \approx \rho(\alpha_2)$. Similarly, if $\alpha_1 = \alpha_2 \in \lfloor \Gamma \rfloor$ then $\bot(\alpha_1) = \bot(\alpha_2)$, $\mathrm{spred}(\alpha_1) = \mathrm{spred}(\alpha_2)$, $\mathrm{wpred}(\alpha_1) = \mathrm{wpred}(\alpha_2)$ and $\mathrm{lb}_\Gamma(\alpha_1) = \mathrm{lb}_\Gamma(\alpha_2)$. $\rho(\alpha_1) = \rho(\alpha_2)$ follows. If $\alpha_1 \leq \alpha_2 \in \lfloor \Gamma \rfloor$ then $\bot(\alpha_1) = \bot(\alpha_2)$, $\mathrm{spred}(\alpha_1) \subseteq \mathrm{spred}(\alpha_2)$, $\mathrm{wpred}(\alpha_1) \subseteq \mathrm{wpred}(\alpha_2)$ and $\mathrm{lb}_\Gamma(\alpha_1) \leq_{\mathcal{A}} \mathrm{lb}_\Gamma(\alpha_2)$. Again, this yields $\rho(\alpha_1) \leq \rho(\alpha_2)$. Lastly, consider $\phi_1 \sqsubset \phi_2 \in \lfloor \Gamma \rfloor$.

∘ *Case* $\phi_1 = a_1$ and $\phi_2 = a_2$. Because Γ is satisfiable, by Theorem 5, $a_1 \leq_{\mathcal{A}} a_2$ must hold.

∘ *Case* $\phi_1 = a_1$ and $\phi_2 = \alpha_2$. We have $a_1 \leq \mathrm{lb}_\Gamma(\alpha_2)$, which yields, by Lemma 4, $a_1 \sqsubset \wr \bot(\alpha_2) \uparrow \mathrm{lb}_\Gamma(\alpha_2)\int_{\boxed{\wr}}$. $a_1 \sqsubset \rho(\alpha_2)$ follows.

∘ *Case* $\phi_1 = \alpha_1$ and $\phi_2 = a_2$. Because Γ is satisfiable, by Theorem 5 and Lemma 17, $\mathrm{lb}_\Gamma(\alpha_1) \leq_{\mathcal{A}} a_2$. Moreover, for every $\beta \in \mathrm{spred}(\alpha_1) \cup \mathrm{wpred}(\alpha_1)$, $\Gamma' \mathrel{|\!\!\approx} \beta \sqsubset a_2$ and $\rho'(\beta) \sqsubset a_2$ holds. We conclude that $\rho(\alpha_1) \sqsubset a_2$.

∘ *Case* $\phi_1 = \alpha_1$ and $\phi_2 = \alpha_2$. We have $\mathrm{lb}_\Gamma(\alpha_1) \leq_{\mathcal{A}} \mathrm{lb}_\Gamma(\alpha_2)$, $\mathrm{spred}(\alpha_1) \cup \mathrm{wpred}(\alpha_1) \subseteq \mathrm{wpred}(\alpha_2)$. This yields $\rho(\alpha_1) \sqsubset \rho(\alpha_2)$.

We conclude that $\rho \vdash \lfloor \Gamma \rfloor$.

We now prove that $\rho \leq^{[\sigma]} \rho'$. If α is a negative variable then $\alpha \in \mathrm{spred}(\alpha)$. This yields $\rho'(\alpha) \leq \rho(\alpha)$. We now assume that α is a positive variable. For every variable β in $\mathrm{spred}(\alpha)$ (resp. $\mathrm{wpred}(\alpha)$), $\beta \leq \alpha \in \Gamma'$ (resp. $\beta \sqsubset \alpha \in \Gamma'$), which implies $\rho'(\beta) \leq \rho'(\alpha)$ (resp. $\rho'(\beta) \sqsubset \rho'(\alpha)$). Moreover, if $\mathrm{lb}_\Gamma(\alpha) \neq \bot$ then $\mathrm{lb}_\Gamma(\alpha) \sqsubset \alpha \in \Gamma'$. Thus, $\mathrm{lb}_\Gamma(\alpha) \sqsubset \rho'(\alpha)$. We conclude that $\rho(\alpha) \leq \rho'(\alpha)$.

By Lemma 14, we obtain $\lceil \rho \rceil_\Gamma \vdash \Gamma$ and, by induction on $\prec_\Gamma$, we check that $\lceil \rho \rceil_\Gamma \leq^{[\sigma]} \rho'$

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). Once constraints have been decomposed, it may be performed here in two steps: a first one detecting equivalent terminal variables and a second one of hash-consing.

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

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

Proof. In this proof, we write $\alpha_1 \sim_\sigma \alpha_2$ if and only if $\bar{\bar{\alpha}}_1 \sim_\sigma \bar{\bar{\alpha}}_2$ holds for some $\bar{\bar{\alpha}}_1$ and $\bar{\bar{\alpha}}_2$ such that α_1 in $\bar{\bar{\alpha}}_1$ and α_2 in $\bar{\bar{\alpha}}_2$. Let $\sigma = \forall\vec{\alpha}[\Gamma].\tau$ and $M(\sigma) = \forall\vec{\alpha}[\Gamma'].\tau$. By construction, $\Gamma' \models \Gamma$; this yields $\sigma \preccurlyeq M(\sigma)$. Let us now consider an assignment ρ which satisfies Γ. Because $\beta \sim_\sigma \alpha$ implies $\rho(\beta) \approx \rho(\alpha)$, thanks to Lemma 2, we may define ρ' by:

$$\rho'(\alpha) = \begin{cases} \sqcup\{\rho(\beta) \mid \beta \sim_\sigma \alpha\} & \text{if } \alpha \text{ is negative non-positive} \\ \sqcap\{\rho(\beta) \mid \beta \sim_\sigma \alpha\} & \text{if } \alpha \text{ is positive non-negative} \\ \rho(\alpha) & \text{otherwise} \end{cases}$$

We now check that $\rho' \vdash \Gamma'$. By construction, for every α, $\rho(\alpha) \approx \rho'(\beta)$. If $\alpha \approx \beta \in \Gamma'$ then $\alpha \approx \beta \in \Gamma$. Because $\rho \vdash \Gamma$, this yields $\rho(\alpha) \approx \rho(\beta)$ and $\rho'(\alpha) \approx \rho'(\beta)$.

Assume $\alpha_1 = \alpha_2 \in \Gamma'$. If $\alpha_1 = \alpha_2 \in \Gamma$ then α_1 and α_2 have the same polarities (in σ) and for every β, $\alpha_1 \sim_\sigma \beta$ is equivalent to $\alpha_2 \sim_\sigma \beta$. This yields $\rho'(\alpha_1) = \rho'(\alpha_2)$. Otherwise, if $\alpha_1 = \alpha_2 \notin \Gamma$, then $\alpha_1 \sim_\sigma \alpha_2$ holds. Then α_1 and α_2 are either two positive non-negative variables or two negative non-positive variables. In both cases, because $\sim_\sigma$ is transitive, this yields $\rho'(\alpha_1) = \rho'(\alpha_2)$.

Assume $\alpha_1 \leq \alpha_2 \in \Gamma'$. By definition, $\alpha_1 \leq \alpha_2 \in \Gamma$ holds too and, by hypothesis, $\rho(\alpha_1) \leq \rho(\alpha_2)$, $\sigma \vdash \alpha_1 : -$ and $\sigma \vdash \alpha_2 : +$. Then we have $\rho'(\alpha_1) = \sqcup\{\rho(\beta) \mid \beta \sim_\sigma \alpha_1\}$ and $\rho'(\alpha_2) = \sqcap\{\rho(\beta) \mid \beta \sim_\sigma \alpha_2\}$ (even if there are also respectively positive or negative). Moreover, if $\beta_1 \sim_\sigma \alpha_1$ and $\beta_2 \sim_\sigma \alpha_2$ then $\beta_1 \leq \beta_2 \in \Gamma$ and $\rho(\beta_1) \leq \rho(\beta_2)$. We conclude that $\rho'(\alpha_1) \leq \rho'(\alpha_2)$.

The case of weak inequalities is similar.

Hash-consing aims at propagating the equivalence found by minimization between terminal variables to non-terminal ones: it unifies non-terminal skeletons

$$\alpha_1 \approx \beta_1 \;\; \cdots \;\; \alpha_n \approx \beta_n \vdash \frac{\langle \bar{\bar{\tau}} = f(\alpha_1, \ldots, \alpha_n) \rangle^{\iota} \approx \bar{\bar{\tau}} \qquad \langle \bar{\bar{\tau}}' = f(\beta_1, \ldots, \beta_n) \rangle^{\iota} \approx \bar{\bar{\tau}}'}{\langle \bar{\bar{\tau}} = f(\alpha_1, \ldots, \alpha_n) \rangle^{\iota} \approx \bar{\bar{\tau}} \approx \langle \bar{\bar{\tau}}' = f(\beta_1, \ldots, \beta_n) \rangle^{\iota} \approx \bar{\bar{\tau}}'} \; \text{HASH}^{\approx} \; \rightsquigarrow$$

$$\alpha_1 = \beta_1 \;\; \cdots \;\; \alpha_n = \beta_n \vdash \frac{\langle \bar{\bar{\tau}} = f(\alpha_1, \ldots, \alpha_n) \rangle^{\iota} \approx \langle \bar{\bar{\tau}}' = f(\beta_1, \ldots, \beta_n) \rangle^{\iota'} \approx \bar{\bar{\tau}}}{\langle \bar{\bar{\tau}} = f(\alpha_1, \ldots, \alpha_n) = \bar{\bar{\tau}}' = f(\beta_1, \ldots, \beta_n) \rangle^{\iota \vee \iota'} \approx \bar{\bar{\tau}}} \; \text{HASH}^{=} \; \rightsquigarrow$$

Fig. 10: Hash-consing (rewriting system Ω_h)

or variables which share the same sub-terms, as performed by the rewriting system Ω_h described by the two rules in Figure 10: the first one intends to fuse multi-skeletons and the second one aims at merging multi-equations within each multi-skeleton. 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.

Theorem 9 (Hash-consing). *The rewriting system Ω_h is sound and terminates.*

Proof. We measure a constraint according to (1) its number of multi-skeletons and (2) its number of multi-equations. Rule $\text{HASH}^{\approx}$ fuses two multi-skeletons, so it decreases (1) and keeps (2) unchanged. Rule $\text{HASH}^{=}$ merges two multi-equations belonging to the same multi-skeleton, reducing (2) without affecting (1). We conclude that the rewriting system Ω_h terminates.

7 Implementation and experiments

The `Dalton` library [Sim02] 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.

	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%)	1424 (2%)	1840 (3%)
	(9% of n.t.)	(8% of n.t.)	(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

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` [Sim03], 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 (either by fusing with another multi-equation or by garbage collection). 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 [TS96], where all type schemes are *closed*, i.e. have no free type variables, but contain a local environment. Hence their meaning does not depend on any external assumption. 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 [Rém92]. 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 [FRD00].

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. The expressiveness of weak inequalities may also be extended in two ways. First, it might be interesting for some applications to have several weak inequalities equipped with different decomposition rules. It should be straightforward to extend our algorithms by labeling the $\sqsubset$ symbol and providing appropriate transitivity rules. Besides, in this paper, $\sqsubset$ is only allowed to consider *covariant* arguments of type constructors. However, in [PS02], 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

[ABHR99] Martín Abadi, Anindya Banerjee, Nevin Heintze, and Jon G. Riecke. A core calculus of dependency. In *Proceedings of the 26th ACM Symposium on Principles of Programming Languages*, pages 147–160, San Antonio, Texas, January 1999. ACM Press. URL: `http://www.soe.ucsc.edu/~abadi/Papers/flowpopl.ps`.

[AF96] Alexander S. Aiken and Manuel Fähndrich. Making set-constraint based program analyses scale. Technical Report CSD-96-917, University of Cal-

ifornia, Berkeley, September 1996. URL: http://http.cs.berkeley.edu/~manuel/papers/scw96.ps.gz.

[AF97] Alexander S. Aiken and Manuel Fähndrich. Program analysis using mixed term and set constraints. In *Proceedings of the 4th International Static Analysis Symposium*, Lecture Notes in Computer Science, pages 114–126, Paris, France, September 1997. Springer Verlag. URL: http://www.cs.berkeley.edu/~aiken/publications/papers/sas97.ps.

[BM97] François Bourdoncle and Stephan Merz. Type checking higher-order polymorphic multi-methods. In *Proceedings of the 24th ACM Symposium on Principles of Programming Languages*, pages 302–315, Paris, January 1997. ACM Press. URL: http://www.cma.ensmp.fr/Francois.Bourdoncle/popl97.ps.Z.

[Car88] Luca Cardelli. A semantics of multiple inheritance. *Information and Computation*, 76(2/3):138–164, February 1988.

[EST95] Jonathan Eifrig, Scott Smith, and Valery Trifonov. Sound polymorphic type inference for objects. *ACM SIGPLAN Notices*, 30(10):169–184, 1995. URL: http://rum.cs.yale.edu/trifonov/papers/sptio.ps.gz.

[FF97] Cormac Flanagan and Matthias Felleisen. Componential set-based analysis. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, pages 235–248, Las Vegas, Nevada, June 1997. ACM Press. URL: http://www.cs.rice.edu/CS/PLT/Publications/pldi97-ff.ps.gz.

[FFA99] Jeffrey S. Foster, Manuel Fähndrich, and Alexander Aiken. A Theory of Type Qualifiers. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, pages 192–203, Atlanta, Georgia, May 1999. URL: http://www.cs.umd.edu/~jfoster/papers/pldi99.ps.gz.

[FFSA98] Manuel Fähndrich, Jeffrey S. Foster, Zhendong Su, and Alexander S. Aiken. Partial online cycle elimination in inclusion constraint graphs. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, pages 85–96, June 1998. URL: http://research.microsoft.com/users/maf/pldi98.ps.

[FM88] You-Chin Fuh and Prateek Mishra. Type inference with subtypes. In H. Ganzinger, editor, *Proceedings of the European Symposium on Programming*, volume 300 of *Lecture Notes in Computer Science*, pages 94–114. Springer Verlag, 1988.

[FM89] You-Chin Fuh and Prateek Mishra. Polymorphic subtype inference: Closing the theory-practice gap. In J. Díaz and F. Orejas, editors, *Proceedings of the European Joint Conference on Theory and Practice of Software Development*, volume 352 of *Lecture Notes in Computer Science*, pages 167–183, Berlin, March 1989. Springer Verlag.

[FRD00] Manuel Fähndrich, Jakob Rehof, and Manuvir Das. Scalable context-sensitive flow analysis using instantiation constraints. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, June 2000. URL: http://research.microsoft.com/~maf/pldi00.ps.

[Fre97] Alexandre Frey. Satisfying subtype inequalities in polynomial space. In Pascal Van Hentenryck, editor, *Proceedings of the 4th International Static Analysis Symposium*, number 1302 in Lecture Notes in Computer Science, pages 265–277, Paris, France, September 1997. Springer Verlag.

[Fre98] Alexandre Frey. *Jazz*. URL: http://www.exalead.com/jazz/, December 1998.

[FTA02] Jeffrey S. Foster, Tachio Terauchi, and Alex Aiken. Flow-sensitive type qualifiers. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, 2002. URL: `http://http.cs.berkeley.edu/~jfoster/papers/pldi02.ps.gz`.

[Fäh99] Manuel Fähndrich. Bane*, A Library for Scalable Constraint-Based Program Analysis*. PhD thesis, University of California at Berkeley, 1999. URL: `http://research.microsoft.com/~maf/diss.ps`.

[GSSS01] Kevin Glynn, Peter J. Stuckey, Martin Sulzmann, and Harald Søndergaard. Boolean constraints for binding-time analysis. In O. Danvy and A. Filinsky, editors, *Proceedings of the Programs as Data Objects*, volume 2053 of *Lecture Notes in Computer Science*, pages 39–63. Springer Verlag, 2001.

[HM95] My Hoang and John C. Mitchell. Lower bounds on type inference with subtypes. In *Proceedings of the 22nd ACM Symposium on Principles of Programming Languages*, pages 176–185, New York, NY, USA, January 1995. ACM Press.

[HM97] Nevin Heintze and David McAllester. Linear-time subtransitive control flow analysis. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, pages 261–272. ACM Press, June 1997. URL: `http://www.autoreason.com/PLDI97.ps`.

[Kod02] John Kodumal. Banshee*, a toolkit for building constraint-based analyses*. PhD thesis, University of California at Berkeley, 2002. URL: `http://www.cs.berkeley.edu/Research/Aiken/banshee/`.

[KR03] Viktor Kuncak and Martin Rinard. Structural subtyping of non-recursive types is decidable. In *Proceedings of the 18th IEEE Symposium on Logic in Computer Science*, June 2003.

[Mit84] John C. Mitchell. Coercion and type inference. In *Proceedings of the 11th ACM Symposium on Principles of Programming Languages*, pages 175–185, Salt Lake City, January 1984. ACM Press.

[Mit91] John C. Mitchell. Type inference with simple subtypes. *Journal of Functional Programming*, 1(3):245–286, 1991.

[MW97] Simon Marlow and Philip Wadler. A practical subtyping system for Erlang. In *Proceedings of the* , pages 136–149, June 1997.

[PL00] François Pessaux and Xavier Leroy. Type-based analysis of uncaught exceptions. *ACM Transactions on Programming Languages and Systems*, 22(2):340–377, 2000. URL: `http://pauillac.inria.fr/~xleroy/publi/exceptions-toplas.ps.gz`.

[PO95] Jens Palsberg and Patrick M. O'Keefe. A type system equivalent to flow analysis. In *Proceedings of the 22nd ACM Symposium on Principles of Programming Languages*, San Francisco, California, January 1995. ACM Press. URL: `ftp://ftp.daimi.aau.dk/pub/palsberg/papers/popl95.ps.Z`.

[Pot00a] Fran cois Pottier. A versatile constraint-based type inference system. *Nordic Journal of Computing*, 7(4):312–347, November 2000. URL: `http://pauillac.inria.fr/~fpottier/publis/fpottier-njc-2000.ps.gz`.

[Pot00b] François Pottier. *Wallace*, an efficient implementation of type inference with subtyping. URL: `http://pauillac.inria.fr/~fpottier/wallace/`, February 2000.

[Pot01] Fran cois Pottier. Simplifying subtyping constraints: a theory. *Information and Computation*, 170(2):153–183, November 2001. URL: `http://pauillac.inria.fr/~fpottier/publis/fpottier-ic01.ps.gz`.

[Pot03] François Pottier. A constraint-based presentation and generalization of rows. In *Proceedings of the 18th IEEE Symposium on Logic in Computer Science*, June 2003. URL: `http://pauillac.inria.fr/~fpottier/publis/fpottier-lics03.ps.gz`.

[PS02] François Pottier and Vincent Simonet. Information flow inference for ML. In *Proceedings of the 29th ACM Symposium on Principles of Programming Languages*, pages 319–330, Portland, Oregon, January 2002. ACM Press. URL: `http://cristal.inria.fr/~simonet/publis/fpottier-simonet-popl02.ps.gz`.

[Reh97] Jakob Rehof. Minimal typings in atomic subtyping. In *Proceedings of the 24th ACM Symposium on Principles of Programming Languages*, pages 278–291, Paris, France, January 1997. ACM Press. URL: `http://research.microsoft.com/~rehof/popl97.ps`.

[Rém92] Didier Rémy. Extending ML type system with a sorted equational theory. Research Report 1766, Institut de Recherche en Informatique et en Automatique, 1992. URL: `ftp://ftp.inria.fr/INRIA/Projects/cristal/Didier.Remy/eq-theory-on-types.ps.gz`.

[SHO98] Bratin Saha, Nevin Heintze, and Dino Oliva. Subtransitive CFA using types. Technical report, Yale University, Department of Computer Science, October 1998. URL: `http://flint.cs.yale.edu/flint/publications/cfa.ps.gz`.

[Sim02] Vincent Simonet. *Dalton*, an efficient implementation of type inference with structural subtyping. URL: `http://cristal.inria.fr/~simonet/soft/dalton/`, October 2002.

[Sim03] Vincent Simonet. *Flow Caml*, information flow inference in Objective Caml. URL: `http://cristal.inria.fr/~simonet/soft/flowcaml/`, July 2003.

[Tar75] Robert Endre Tarjan. Efficiency of a good but not linear set union algorithm. *Journal of the ACM*, 22(2):215–225, April 1975.

[Tiu92] Jerzy Tiuryn. Subtype inequalities. In Andre Scedrov, editor, *Proceedings of the 7th IEEE Symposium on Logic in Computer Science*, pages 308–317, Santa Cruz, CA, June 1992. IEEE Computer Society Press.

[TS96] Valery Trifonov and Scott Smith. Subtyping constrained types. In *Proceedings of the 3rd International Static Analysis Symposium*, volume 1145 of *Lecture Notes in Computer Science*, pages 349–365. Springer Verlag, September 1996. URL: `http://rum.cs.yale.edu/trifonov/papers/subcon.ps.gz`.