Nordic Journal of Computing

A Versatile Constraint-Based Type Inference System

François Pottier*
Francois.Pottier@inria.fr

Abstract. The combination of *subtyping*, *conditional constraints* and *rows* yields a powerful *constraint-based type inference* system. We illustrate this claim by proposing solutions to three delicate type inference problems: "accurate" pattern matchings, record concatenation, and first-class messages. Previously known solutions involved a different technique in each case; our theoretical contribution is in using only a single set of tools. On the practical side, this allows all three problems to benefit from a common set of constraint simplification techniques, a formal description of which is given in an appendix.

CR Classification: F.3.3 [Logics and Meanings of Programs]: Studies of Program Constructs—Type Structure.

Key words: Constraint-based type inference. Subtyping. Rows. Conditional constraints.

1. Introduction

Type inference is the task of examining a program which lacks some (or even all) type annotations, and recovering enough type information to make it acceptable by a type checker. Its original, and most obvious, application is to free the programmer from the burden of manually providing these annotations, thus making static typing a less dreary discipline. However, type inference has also seen heavy use as a simple, modular way of formulating program analyses.

The design of a type inference system can be influenced by its purpose. When used as a user-visible way of enforcing a coding discipline, it might be desirable to make it simple and somewhat rigid. When used invisibly as part of a compiler's optimization process, on the other hand, maximum precision may be desired. Regardless of this distinction, however, powerful type inference techniques are often made a necessity by the advanced features found in many recent programming languages.

This paper presents a common solution to several seemingly unrelated type inference problems, using an existing framework for *subtyping-constraint-based type inference* [14], equipped with *conditional constraints* inspired by Aiken, Wimmers and Lakshman [2] and with *rows* à la Rémy [19, 21].

*INRIA Rocquencourt, BP 105, 78153 Le Chesnay Cedex, France.

Received 20th November 2000.

Constraint-Based Type Inference

Subtyping is a partial order on types, defined so that an object of a subtype may safely be supplied wherever an object of a supertype is expected. Type inference in the presence of subtyping reflects this basic principle. Every time a piece of data is passed from a producer to a consumer, the former's output type is required to be a *subtype* of the latter's input type. This requirement is explicitly recorded by creating a symbolic *subtyping constraint* between these types. Thus, each potential data flow discovered in the program yields one constraint. This fact allows viewing a constraint set as a directed approximation of the program's data flow graph – regardless of our particular definition of subtyping.

Various type inference systems based on subtyping constraints exist. One may cite works by Aiken et al. [1, 2, 5], the present author [16, 17], Trifonov and Smith [29], as well as Odersky et al.'s abstract framework HM(X) [14, 28, 27]. Related systems include set-based analysis [9, 6] and type inference systems based on feature constraints [11, 12] or predicate constraints [10].

Conditional Constraints

In many constraint-based systems, the expression `if` e_0 `then` e_1 `else` e_2 is, at best, described by

$$\alpha_1 \leq \alpha \quad \wedge \quad \alpha_2 \leq \alpha$$

where α_i stands for e_i's type, and α stands for the whole expression's type. This amounts to stating that "the value of e_1 (resp. e_2) may become the value of the whole expression", regardless of the test's outcome. A more precise description – "if e_0 may evaluate to `true` (resp. `false`), then the value of e_1 (resp e_2) may become the value of the whole expression" – can be given using *conditional constraints*:

$$\texttt{true} \leq \alpha_0 \,?\, \alpha_1 \leq \alpha \quad \wedge \quad \texttt{false} \leq \alpha_0 \,?\, \alpha_2 \leq \alpha$$

Introducing tests into constraints allows keeping track of some of the program's *control* flow – that is, mirroring, at the level of types, the way evaluation is affected by the outcome of a test.

Conditional set expressions were introduced by Reynolds [25] as a means of solving set constraints involving strict type constructors and destructors. Heintze [9] uses them to formulate an analysis which ignores "dead code". He also introduces *case constraints*, which allow ignoring the effect of a branch, in a `case` construct, unless it is actually liable to be taken. Aiken, Wimmers and Lakshman [2] use *conditional types*, together with intersection types, for this purpose.

In the present paper, we suggest a single notion of *conditional constraint*, which is comparable in expressive power to the above constructs, and lends itself to a simple and efficient implementation. (A similar choice was made

independently by Fähndrich [5].) We emphasize its use as a way not only of introducing some *control* into types, but also of *delaying* type computations, thus introducing some "laziness" into type inference.

Rows

Designing a type system for a programming language with records, or objects, requires some way of expressing labelled products of types, where labels are field or method names. Dually, if the language allows manipulating structured data, then its type system is likely to require labelled sums, where labels are names of data constructors.

Wand [30] and Rémy [19, 21] elegantly deal with both problems at once by introducing notation to express denumerable, indexed families of types, called *rows*:

$$\rho ::= \alpha, \beta, \ldots, \varphi, \psi, \ldots \mid a : \tau;\ \rho \mid \partial\tau$$

(Here, τ ranges over types, and $a, b, \ldots$ range over indices.) An unknown row may be represented by a *row variable*, exactly as in the case of types. (By lack of symbols, we will not syntactically distinguish plain type variables and row variables.) The term $a : \tau;\ \rho$ represents a row whose element at index a is τ, and whose other elements are given by ρ. The term $\partial\tau$ stands for a row whose element at any index is τ. These statements are given formal meaning by interpreting rows in a logical model where the following equations hold:

$$a : \tau_a;\ (b : \tau_b;\ \rho) = b : \tau_b;\ (a : \tau_a;\ \rho)$$
$$\partial\tau = a : \tau;\ \partial\tau$$

If desired, some type constructors may be *lifted* to the level of rows, i.e. viewed as row constructors as well. For instance, to lift the type constructor $\rightarrow$, we extend the syntax of rows:

$$\rho ::= \ldots \mid \rho \rightarrow \rho$$

The term $\rho \rightarrow \rho'$ is logically interpreted as the row obtained by applying the type constructor $\rightarrow$, point-wise, to the rows ρ and ρ'. As a result, the logical model satisfies the following equations:

$$(a : \tau;\ \rho) \rightarrow (a : \tau';\ \rho') = a : (\tau \rightarrow \tau');\ (\rho \rightarrow \rho')$$
$$\partial\tau \rightarrow \partial\tau' = \partial(\tau \rightarrow \tau')$$

More details are given in Section 2.

Rows offer a particularly straightforward way of describing operations which treat all labels (except possibly a finite number thereof) uniformly. Because every facility available at the level of types (e.g. constructors, constraints) can also be made available at the level of rows, a description of the operation's effect on a single label, written using types, can also be read as a description of the entire operation, written using rows. This interesting point will be developed further in the paper.

Putting It All Together

Our point is to show that the combination of the three concepts discussed above yields a very expressive system, which allows type inference for a number of advanced language features. Among these, "accurate" pattern matching constructs, record concatenation, and first-class messages will be discussed in this paper. Our system allows performing type inference for all of these features at once. Furthermore, efficiency issues concerning constraint-based type inference systems have already been studied [5, 17]. This existing knowledge benefits our system, which may thus be used to *efficiently* perform type inference for all of the above features.

In this paper, we focus on *applications* of our type system, i.e. we show how it allows solving each of the problems mentioned above. Formal definitions of our constraint resolution and simplification algorithms appear in Appendix A. Furthermore, a robust prototype implementation is publicly available [18]. We do not prove that the types given to the three problematic operations discussed in this paper are sound, but we believe this is a straightforward task.

The paper is organized as follows. Section 2 gives a detailed technical presentation of the type system. Section 3 gives an informal explanation of the potential costs and benefits of using conditional constraints. Sections 4, 5, and 6 discuss type inference for "accurate" pattern matchings, record concatenation, and first-class messages, respectively, within our system. Section 7 gives several examples, which show what inferred types look like in practice. Section 8 sums up our contribution. Lastly, Appendix A gives definitions and proofs for several constraint manipulation algorithms.

2. Formal Presentation of the System

This section gives an in-depth formal presentation of our type system, in its most general form. Much of it may be skipped on a first reading – the following sections describe the system in a more gentle fashion. The reader may wish to come back to this section at a later stage.

We define our type system as an instance of the parametric framework HM(X) [14, 28, 27]. To do so, we simply define a *constraint system*, called SRC (for *subtyping-rows-conditionals*), giving rise to HM(SRC). By re-using existing material, we save definitions and proofs, and emphasize the fact that our approach is standard.

In order to retain a measure of generality, SRC is itself parameterized by a *ground signature*, which is a succinct description of a type algebra and of its intended subtype ordering. Ground signatures are defined in Section 2.1. Given such a ground signature, we explicitly define the syntax of types and constraints (Section 2.2), a logical model within which they may be interpreted (Section 2.3), and the interpretation itself (Section 2.4).

2.1 Assumptions

A ground signature consists of three components: a series of *symbol* lattices, indexed by *kinds*, a set of *parameter labels* (each of which is either co- or contra-variant, describes either a row or a plain type parameter, and has a fixed kind), and a description of each symbol's *arity* as a finite set of parameter labels.

DEFINITION 1. *Let $\mathcal{K}$ be a finite set of* kinds. *For every kind $\kappa \in \mathcal{K}$, let $\mathcal{S}_\kappa$ be a lattice of* symbols, *with operations $\bot_\kappa$, $\top_\kappa$, $\leq_\kappa$, $\sqcup_\kappa$ and $\sqcap_\kappa$. Define $\mathcal{S} = \uplus_{\kappa \in \mathcal{K}} \mathcal{S}_\kappa$.*

Let $\mathcal{L}^+$ and $\mathcal{L}^-$ be denumerable sets of parameter labels. *Define $\mathcal{L} = \mathcal{L}^+ \uplus \mathcal{L}^-$. Let $\mathcal{L}_{\text{row}} \subseteq \mathcal{L}$ be a distinguished subset of* row *parameter labels. Let* kind *be a total mapping of $\mathcal{L}$ into $\mathcal{K}$.*

Let a be a total mapping from $\mathcal{S}$ to finite subsets of $\mathcal{L}$, such that:

- *for all $s_0, s_1, s_2 \in \mathcal{S}_\kappa$, $s_0 \leq_\kappa s_1 \leq_\kappa s_2$ implies $a(s_0) \cap a(s_2) \subseteq a(s_1)$;*
- *for any finite subset S of $\mathcal{S}_\kappa$, $a(\sqcup_\kappa S)$ and $a(\sqcap_\kappa S)$ are subsets of $\cup a(S)$. Note that this implies $a(\bot_\kappa) = a(\top_\kappa) = \varnothing$.*

The information described above forms a ground signature.

The first condition bearing on a is necessary to guarantee that the orderings $\leq_\kappa$ do give rise to an ordering on ground types (defined in Section 2.3). The second one makes the definition of some constraint manipulation algorithms more convenient (see Definition 23 in Appendix A).

EXAMPLE 1. Assume there is only one kind $\star$. Define $\mathcal{S}_\star = \{\bot, \to, \top\}$, where $\bot \leq_\star \to \leq_\star \top$. Let $\mathcal{L}^- = \{dom\}$, $\mathcal{L}^+ = \{rng\}$ and $\mathcal{L}_{\text{row}} = \varnothing$. Define $a(\bot) = a(\top) = \varnothing$ and $a(\to) = \{dom, rng\}$. This defines a ground signature, which allows typing the pure λ-calculus.

EXAMPLE 2. Define three kinds N, R and V, corresponding to *normal*, *record field* and *variant field* types, respectively. Let $\mathcal{S}_{\text{N}}$ be the flat lattice whose elements other than $\bot$ and $\top$ are $\to$, $\{\cdot\}$ and $[\cdot]$. Let $\mathcal{S}_{\text{R}}$ be the lattice with least element `Bot`, greatest element `Any`, and whose other elements are `Abs`, `Pre` and `Either`, ordered by `Abs` $\leq_{\text{R}}$ `Either` and `Pre` $\leq_{\text{R}}$ `Either`. Let $\mathcal{S}_{\text{V}}$ be the lattice with least element `Abs`, greatest element `Any`, and whose only other element is `Pre`. (By abuse of language, we are giving identical names to symbols in $\mathcal{S}_{\text{R}}$ and in $\mathcal{S}_{\text{V}}$. This remains non-ambiguous as long as all terms considered have known kinds.) Let $\mathcal{L}^- = \{dom\}$, $\mathcal{L}^+ = \{content, contents, rng\}$ and $\mathcal{L}_{\text{row}} = \{contents\}$. Define $a(\to) = \{dom, rng\}$, $a(\{\cdot\}) = a([\,\cdot\,]) = \{contents\}$, $a(\texttt{Pre}) = a(\texttt{Either}) = \{content\}$, and $a(\bot) = a(\top) = a(\texttt{Bot}) = a(\texttt{Abs}) = a(\texttt{Any}) = \varnothing$. This defines a ground signature, which is expressive enough to describe all programming language features considered in this paper. In particular, all of its expressive power will be exploited to describe first-class messages in Section 6.

$$\tau ::= \alpha, \beta, \varphi, \psi, \ldots \mid s(\tau_l)_{l \in a(s)} \mid r : \tau;\ \tau \mid \partial\tau$$
$$C ::= \mathbf{true} \mid C \wedge C \mid \exists \bar{\alpha}.C \mid \tau \leq \tau \mid s \leq \tau \,?\, \tau \leq \tau \quad (s \text{ prime in } \mathcal{S}_\kappa)$$

Figure 1: Syntax of types and constraints

In the rest of this formal presentation, we assume given a fixed, arbitrary ground signature. In Sections 4–7, we will use the ground signature described in Example 2 above, but we will re-introduce it step by step.

2.2 Syntax of Types and Constraints

The (raw) syntax of types and constraints is given in Fig. 1. $\alpha, \beta, \varphi, \psi, \ldots$ denote type variables. A type term $s(\bar{\tau})$ can be formed by picking a symbol $s \in \mathcal{S}$ and a family of type parameters $\bar{\tau}$, indexed according to the arity of s, i.e. $\bar{\tau}$ must be of the form $(\tau_l)_{l \in a(s)}$. Lastly, types may also be *rows*, which denote families of types indexed by a denumerable set of *row labels* $\mathcal{R}$. The term $r : \tau;\ \tau'$ (where $r \in \mathcal{R}$) represents a row whose element at index r is τ, and whose other elements are given by the row τ'. The term $\partial\tau$ stands for a row whose element at any index is τ.

The constraint language offers standard constructs (truth, conjunction, projection [14]), subtyping constraints, and conditional constraints. The latter are of the form $s \leq \tau \,?\, \tau \leq \tau$, where s must satisfy the following condition: for any finite subset S of $\mathcal{S}_\kappa$, $s \leq_\kappa (\sqcup_\kappa S)$ implies $\exists s' \in S \quad s \leq_\kappa s'$. In other words, s must be a *prime* element of its symbol lattice $\mathcal{S}_\kappa$. This ensures that a conditional constraint bearing on the least upper bound of a set of variables, e.g. $(s \leq \alpha_1 \sqcup \ldots \sqcup \alpha_n) \,?\, c$, is equivalent to a conjunction of conditional constraints bearing on its members: $\bigwedge_{i=1}^{n} (s \leq \alpha_i \,?\, c)$. It is a necessary condition for the correctness of the *garbage collection* algorithm (see Theorem 3 in Appendix A).

Our definition of conditional constraints is dissymmetric. Indeed, conditions must be of the form $s \leq \tau$; conditions of the form $\tau \leq s$ are disallowed. The motivation for this decision is to allow the constraint solving algorithm to ignore conditional constraints unless their condition *must* be satisfied (see Definition 19 in Appendix A). If both forms of conditions were allowed to coexist, the language would become expressive enough to encode disjunctions of constraints, making constraint solving more costly.

To ensure that only meaningful types and constraints can be built, we equip them with *kinding* and *sorting* rules. The grammar of *sorts* is defined by $\varsigma ::= \mathit{Type} \mid \mathit{Row}(R)$, where R ranges over finite subsets of $\mathcal{R}$. For every kind κ and every sort ς, we assume given a distinct, denumerable set $\mathcal{V}_\kappa^\varsigma$ of type variables. We define judgements of the form $\vdash \tau : \kappa$ (resp. $\vdash \tau : \varsigma$), meaning that the type τ has kind κ (resp. sort ς), and judgements of the

$$\frac{\alpha \in \mathcal{V}_\kappa}{\vdash \alpha : \kappa} \qquad \frac{s \in \mathcal{S}_\kappa \quad \forall l \in a(s) \quad \vdash \tau_l : \mathrm{kind}(l)}{\vdash s(\tau_l)_{l \in a(s)} : \kappa} \qquad \frac{\vdash \tau_1 : \kappa \quad \vdash \tau_2 : \kappa}{\vdash (r : \tau_1;\ \tau_2) : \kappa} \qquad \frac{\vdash \tau : \kappa}{\vdash \partial\tau : \kappa} \qquad \vdash \mathbf{true}$$

$$\frac{\vdash C_1 \quad \vdash C_2}{\vdash C_1 \wedge C_2} \qquad \frac{\vdash C}{\vdash \exists \bar{\alpha}.C} \qquad \frac{\vdash \tau_1 : \kappa \quad \vdash \tau_2 : \kappa}{\vdash \tau_1 \leq \tau_2} \qquad \frac{s \in \mathcal{S}_\kappa \quad \vdash \tau_0 : \kappa \quad \vdash \tau_1 \leq \tau_2}{\vdash s \leq \tau_0\,?\,\tau_1 \leq \tau_2}$$

Figure 2: Kinding rules

$$\frac{\alpha \in \mathcal{V}^\varsigma}{\vdash \alpha : \varsigma} \qquad \frac{a(s) \cap \mathcal{L}_{\mathrm{row}} = \varnothing \quad \forall l \in a(s) \quad \vdash \tau_l : \varsigma}{\vdash s(\tau_l)_{l \in a(s)} : \varsigma} \qquad \frac{a(s) \cap \mathcal{L}_{\mathrm{row}} \neq \varnothing \quad \forall l \in a(s) \setminus \mathcal{L}_{\mathrm{row}} \quad \vdash \tau_l : \mathit{Type} \quad \forall l \in a(s) \cap \mathcal{L}_{\mathrm{row}} \quad \vdash \tau_l : \mathit{Row}(\varnothing)}{\vdash s(\tau_l)_{l \in a(s)} : \mathit{Type}}$$

$$\frac{\vdash \tau_1 : \mathit{Type} \quad \vdash \tau_2 : \mathit{Row}(R \uplus \{r\})}{\vdash (r : \tau_1;\ \tau_2) : \mathit{Row}(R)} \qquad \frac{\vdash \tau : \mathit{Type}}{\vdash \partial\tau : \mathit{Row}(R)} \qquad \vdash \mathbf{true} \qquad \frac{\vdash C_1 \quad \vdash C_2}{\vdash C_1 \wedge C_2}$$

$$\frac{\vdash C}{\vdash \exists \bar{\alpha}.C} \qquad \frac{\vdash \tau_1 : \varsigma \quad \vdash \tau_2 : \varsigma}{\vdash \tau_1 \leq \tau_2} \qquad \frac{\vdash \tau_0 : \varsigma \quad \vdash \tau_1 : \varsigma \quad \vdash \tau_2 : \varsigma}{\vdash s \leq \tau_0\,?\,\tau_1 \leq \tau_2}$$

Figure 3: Sorting rules

form $\vdash C$, meaning that the constraint C is well-kinded (resp. well-sorted). The kinding rules, given in Fig. 2, simply enforce the kind discipline required by the ground signature. The sorting rules, displayed in Fig. 3, ensure that only meaningful row terms are built. Intuitively, the sort *Type* describes plain types, while the sort $\mathit{Row}(R)$ describes families of types indexed by $\mathcal{R} \setminus R$. In other words, a row of sort $\mathit{Row}(R)$ gives information about all row labels *except* those in R. For more details, we refer the reader to [21] or to [20, section 5].

Before moving on, let us point out that a term may have several sorts, for two distinct reasons. First, a uniform row $\partial\tau$ may be viewed as describing any (co-finite) number of entries, i.e. it may have any sort $\mathit{Row}(R)$. As a result, the row term $r_1 : \tau_1;\ \ldots;\ r_n : \tau_n;\ \partial\tau$ may have any sort $\mathit{Row}(R)$, provided $\{r_1, \ldots, r_n\} \cap R = \varnothing$. Such a term will be required to have sort $\mathit{Row}(\varnothing)$ only when used as the l-parameter of a type constructor s expecting a full row in l-position (i.e. $l \in a(s) \cap \mathcal{L}_{\mathrm{row}}$). Second, a type constructor s with non-row parameters (i.e. $a(s) \cap \mathcal{L}_{\mathrm{row}} = \varnothing$) can be used at any sort ς. For instance, if $r : \tau_0;\ \tau_0'$ and $r : \tau_1;\ \tau_1'$ have sort $\mathit{Row}(R)$, then $(r : \tau_0;\ \tau_0') \to (r : \tau_1;\ \tau_1')$ has sort $\mathit{Row}(R)$ as well. Its logical interpretation will be the same as that of $r : \tau_0 \to \tau_1;\ \tau_0' \to \tau_1'$.

This point makes the logical interpretation of terms, given in Section 2.4,

slightly more subtle: the meaning of a term depends on the sort at which it is viewed. Fortunately, the meaning of a constraint will remain independent of the sort of its components.

2.3 Logical Model

We now define the logical model within which our constraints are interpreted. Informally speaking, it is the term algebra generated by the ground signature at hand. However, things are made more complex by our desire to have recursive types[1] and by the presence of rows.

DEFINITION 2. *Let $\mathcal{A}$ be the alphabet formed of all letters $l \in \mathcal{L} \setminus \mathcal{L}_{\text{row}}$ and all composite letters $l \cdot r$, where $l \in \mathcal{L}_{\text{row}}$ and $r \in \mathcal{R}$. To every $l \in \mathcal{L}$, we associate a subset $\mathcal{A}_l$ of the alphabet, defined by $\mathcal{A}_l = \{l\}$ if $l \in \mathcal{L} \setminus \mathcal{L}_{\text{row}}$, and $\mathcal{A}_l = \{l \cdot r \,;\, r \in \mathcal{R}\}$ otherwise.*

A path *p is a finite string over the alphabet $\mathcal{A}$, i.e. an element of $\mathcal{A}^*$. The letter ϵ denotes the empty path. A* ground tree *t is a partial function from $\mathcal{A}^*$ into $\mathcal{S}$, whose domain is non-empty and prefix-closed, such that, for all paths $p \in \text{dom}(t)$ and for all labels $l \in \mathcal{L}$,*

- *if $l \in a(t(p))$, then $p.\mathcal{A}_l$ is a subset of $\text{dom}(t)$, whose image through t is a subset of $\mathcal{S}_\kappa$, where $\kappa = \text{kind}(l)$;*
- *otherwise, $p.\mathcal{A}_l$ lies outside of $\text{dom}(t)$.*

The head constructor *of a ground term t, written $\text{hd}(t)$, is $t(\epsilon)$. Given $p \in \text{dom}(t)$, the* subtree of t rooted at p, *written $t.p$, is the tree $q \mapsto t(p.q)$. Given p, l such that $l \in a(t(p)) \cap \mathcal{L}_{\text{row}}$, the* subrow of t rooted at (p, l) *is the function $r \in \mathcal{R} \mapsto t(p.(l \cdot r))$. A function is said to be* quasi-constant *iff its co-restriction to some finite set is a constant function. A ground tree is* regular *iff it has a finite number of subtrees. A ground tree t is a* ground type *iff it is regular and all of its subrows are quasi-constant. We denote the set of ground types by $\mathbb{T}$. A ground type t has kind κ if and only if $t(\epsilon) \in \mathcal{S}_\kappa$. We denote the set of ground types of kind κ by $\mathbb{T}_\kappa$.*

Then, we equip every $\mathbb{T}_\kappa$ with an ordering $\leq$. Because ground types are infinite trees, $\leq$ cannot be defined easily by structural induction; instead, it is defined as the limit of a decreasing sequence of pre-orders.

DEFINITION 3. *A family of pre-orders over every $\mathbb{T}_\kappa$ is defined as follows. Let $\leq_0$ be uniformly true over every $\mathbb{T}_\kappa$. Then, for any $k \in \mathbb{N}$ and $t, t' \in \mathbb{T}_\kappa$, define $t \leq_{k+1} t'$ as the conjunction of the following conditions:*

- $t(\epsilon) \leq_\kappa t'(\epsilon)$;
- $\forall l \in a(t(\epsilon)) \cap a(t'(\epsilon)) \setminus \mathcal{L}_{\text{row}} \quad t.l \leq_k^l t'.l$;
- $\forall l \in a(t(\epsilon)) \cap a(t'(\epsilon)) \cap \mathcal{L}_{\text{row}} \quad \forall r \in \mathcal{R} \quad t.(l \cdot r) \leq_k^l t'.(l \cdot r)$.

[1] The presence of recursive types removes the need to check whether all solutions of a constraint are cyclic, which, in the presence of subtyping relationships between type constructors of different arities, may be difficult.

(We let $t \leq_k^l t'$ stand for $t \leq_k t'$ when $l \in \mathcal{L}^+$ and $t' \leq_k t$ when $l \in \mathcal{L}^-$.) Subtyping, denoted by $\leq$, is the intersection of these pre-orders; it is a lattice on every $\mathbb{T}_\kappa$.

The subtyping relationship is *structural*: t and t' are related if and only if their head constructors $t(\epsilon)$ and $t'(\epsilon)$ are related in the lattice of symbols and, for every label l defined by both t and t', their l-sub-terms are related (either co- or contra-variantly, depending on the variance of l). It is, in general, *non-atomic*: type constructors of different arities may be related.

2.4 Logical Interpretation

There remains to give an interpretation of types and constraints within the model. It is parameterized by a *ground substitution*, which gives meaning to any free type variables. It maps types to ground types, or to families thereof (according to their sort), and constraints to Boolean values.

DEFINITION 4. *A* ground substitution *ϕ is a function of domain $\mathcal{V}$, which maps $\mathcal{V}_\kappa^{Type}$ into $\mathbb{T}_\kappa$, and which maps $\mathcal{V}_\kappa^{Row(R)}$ into the set of quasi-constant functions of $\mathcal{R} \setminus R$ into $\mathbb{T}_\kappa$.*

DEFINITION 5. *The* interpretation *of a type τ of sort ς, under a ground substitution ϕ, written $\phi(\tau^\varsigma)$, or simply $\phi(\tau)$ when ς can be determined from the context, is defined as follows.*

- *If τ is a type variable α, then $\phi(\tau^\varsigma)$ is the image of α through ϕ.*
- *If τ is of the form $s(\tau_l)_{l \in a(s)}$ and $\varsigma = Type$, then $\phi(\tau^\varsigma)$ is the ground type t such that $t(\epsilon) = s$, $t.l = \phi(\tau_l)$ whenever $l \in a(s) \setminus \mathcal{L}_{\text{row}}$ and $t.(l \cdot r) = \phi(\tau_l)(r)$ whenever $l \in a(s) \cap \mathcal{L}_{\text{row}}$ and $r \in \mathcal{R}$.*
- *If τ is of the form $s(\tau_l)_{l \in a(s)}$ and $\varsigma = Row(R)$, then, for every $r \in \mathcal{R} \setminus R$, $\phi(\tau^\varsigma)(r)$ is the ground type t such that $t(\epsilon) = s$ and $t.l = \phi(\tau_l)(r)$ whenever $l \in a(s)$.*
- *If τ is of the form $r : \tau_1;\ \tau_2$ and $\varsigma = Row(R)$, then $\phi(\tau^\varsigma)(r) = \phi(\tau_1)$ and, for every $r' \in \mathcal{R} \setminus (R \cup \{r\})$, $\phi(\tau^\varsigma)(r') = \phi(\tau_2)(r')$.*
- *If τ is of the form $\partial\tau_0$ and $\varsigma = Row(R)$, then, for every $r \in \mathcal{R} \setminus R$, $\phi(\tau^\varsigma)(r) = \phi(\tau_0)$.*

DEFINITION 6. *The* constraint satisfaction *predicate $\vdash$, whose arguments are a ground substitution ϕ and a well-sorted constraint C, is defined as follows.*

- *$\phi \vdash$ **true** holds.*
- *$\phi \vdash C_1 \wedge C_2$ holds iff $\phi \vdash C_1$ and $\phi \vdash C_2$ hold.*
- *$\phi \vdash \exists\bar{\alpha}.C$ holds iff there exists a ground substitution ϕ', which coincides with ϕ outside of $\bar{\alpha}$, such that $\phi' \vdash C$ holds.*
- *If $\vdash \tau_1, \tau_2 : Type$, then $\phi \vdash \tau_1 \leq \tau_2$ holds iff $\phi(\tau_1) \leq \phi(\tau_2)$ holds.*

- *If* $\vdash \tau_1, \tau_2 : \mathit{Row}(R)$*, then* $\phi \vdash \tau_1 \leq \tau_2$ *holds iff, for every* $r \in \mathcal{R} \setminus R$*,* $\phi(\tau_1)(r) \leq \phi(\tau_2)(r)$ *holds.*
- *If* $\vdash \tau_0, \tau_1, \tau_2 : \mathit{Type}$*, then* $\phi \vdash s \leq \tau_0 \,?\, \tau_1 \leq \tau_2$ *holds iff* $s \leq_{\mathcal{S}} \phi(\tau_0)(\epsilon)$ *implies* $\phi(\tau_1) \leq \phi(\tau_2)$*.*
- *If* $\vdash \tau_0, \tau_1, \tau_2 : \mathit{Row}(R)$*, then* $\phi \vdash s \leq \tau_0 \,?\, \tau_1 \leq \tau_2$ *holds iff, for every* $r \in \mathcal{R} \setminus R$*,* $s \leq_{\mathcal{S}} \phi(\tau_0)(r)(\epsilon)$ *implies* $\phi(\tau_1)(r) \leq \phi(\tau_2)(r)$*.*

This definition is well-formed because, even though the types which appear in a constraint may have several admissible sorts, all of them give rise to the same interpretation.

Lastly, constraint entailment is given its usual definition: $C \Vdash C'$ holds if and only if, for every ground substitution ϕ, $\phi \vdash C$ implies $\phi \vdash C'$.

2.5 The Type System HM(SRC)

We refer to the constraint logic defined in Sections 2.1–2.4 as SRC. It is a *sound constraint system* in the sense of [14]; thus, it gives rise to a type system, namely HM(SRC), for the λ-calculus with `let`.

We do not repeat the typing rules of HM(X) in this paper. For our purposes, suffice it to recall that *type schemes* are of the form $\sigma ::= \forall \bar{\alpha}[C].\tau$. When all of a type scheme's variables are universally quantified, we usually write "τ where C".

The λ-calculus with `let` is a limited programming language. To extend it, we will define new primitive operations, equipped with operational semantics and appropriate type schemes. However, no extension to the type system itself will be necessary. This explains why we do not describe it further. Instead, we will focus our interest on *writing expressive type schemes.*

3. About Conditional Constraints

The content of this section is informal. It shows how conditional constraints can be used to gain extra typing flexibility, and why we might want to use them only sparingly.

In a call-by-value language, if an expression e_2 diverges, then so does any application $(e_1\, e_2)$. In particular, if e_2 has type $\bot$, then $(e_1\, e_2)$ may safely be given type $\bot$ as well. In other words, if it can be proven that e_1 will never be called, then its return type can be discarded.

It is possible to make a type system aware of this fact. To do so, one merely introduces a new typing rule:

$$\frac{C, (\Gamma; x : \bot) \vdash e : \tau}{C, \Gamma \vdash \lambda x.e : \bot \to \bot}$$

As a result, the type system is no longer syntax-directed: typing a λ-abstraction involves a choice between this rule and the usual λ-abstraction rule. However, a practical type inference algorithm must not explore both

cases separately, since that would have exponential cost. Instead, a natural solution is to use a single type inference rule, which emits a conditional constraint, along the lines of

$$\frac{C, (\Gamma; x : \alpha) \vdash_{\mathrm{I}} e : \tau \qquad \alpha, \beta \text{ fresh}}{(\bot < \alpha \,?\, \tau \leq \beta) \wedge C, \Gamma \vdash_{\mathrm{I}} \lambda x.e : \alpha \to \beta}$$

As long as the function isn't invoked, $\bot$ remains an admissible solution for its argument type α. So, the conditional constraint has no effect, and β remains unconstrained, meaning that the function produces no result. However, if a call to this function is later discovered, then α will be constrained to some value greater than $\bot$. This will trigger the conditional constraint, and $\alpha \to \tau$ will become a lower bound for the function's type, meaning that the function produces a result of type τ.

This technique allows designing a "lazy" type inference system, which ignores the type of an expression unless it appears liable to be evaluated. Heintze [9] uses conditional types for this very purpose. In fact, it is possible to carry this idea even further, and to ignore not only the expression's type, but also its effect on the typing environment. This would involve replacing $(\bot < \alpha \,?\, \tau \leq \beta) \wedge C$ above with $\bot < \alpha \,?\, (\tau \leq \beta \wedge C)$; thus, the constraint C, which describes the requirements of the function concerning its environment, would also be subject to the condition $\bot < \alpha$. This idea appears, under a different formulation, in e.g. [26].

Despite their theoretical appeal, though, these proposals seem a bit extreme. They produce a large number of conditional constraints, making type inference less efficient, because potential constraint simplifications are delayed. Thus, in a practical system, "laziness" should be used only sparingly. We propose to build it into the types of a few primitive operations, rather than to hard-wire it into the typing rules. We will illustrate this principle in the following sections.

4. Accurate Analysis of Pattern Matchings

When faced with a pattern matching construct, most existing type inference systems adopt a simple, conservative approach: assuming that each branch may be taken, they let it contribute to the whole expression's type. A more accurate system should use types to prove that certain branches cannot be taken, and prevent them from contributing.

In this section, we describe such a system. The essential idea – introducing a conditional construct at the level of types – is due to [9, 2]. Some novelty resides in our two-step presentation, which we believe helps isolate independent concepts. First, we consider the case where only *one* data constructor exists. Then, we easily move to the general case, by enriching the type algebra with rows.

4.1 The Basic Case

We assume the language allows building and accessing tagged values.

$$e ::= \ldots \mid \texttt{Pre} \mid \texttt{Pre}^{-1}$$

A single data constructor, `Pre`, allows building tagged values, while the destructor $\texttt{Pre}^{-1}$ allows accessing their contents. This relationship is expressed by the following reduction rule:

$$\texttt{Pre}^{-1}\, v_1\, (\texttt{Pre}\, v_2) \quad \text{reduces to} \quad (v_1\, v_2)$$

The rule states that $\texttt{Pre}^{-1}$ first takes the tag off the value v_2, then passes it to the function v_1.

At the level of types, we introduce a (unary) variant type constructor $[\cdot]$. Also, we establish a distinction between so-called "normal types," written τ, and "field types," written ϕ.

$$\begin{aligned} \tau &::= \alpha, \beta, \gamma, \ldots \mid \bot \mid \top \mid \tau \to \tau \mid [\,\phi\,] \\ \phi &::= \varphi, \psi, \ldots \mid \texttt{Abs} \mid \texttt{Pre}\ \tau \mid \texttt{Any} \end{aligned}$$

A subtype ordering over field types is defined straightforwardly: `Abs` is its least element, `Any` is its greatest, and `Pre` is a covariant type constructor.

The data constructor `Pre` is given the following type scheme:

$$\texttt{Pre} \quad : \quad \alpha \to [\,\texttt{Pre}\ \alpha\,]$$

Notice that there is no way of building a value of type $[\,\texttt{Abs}\,]$. Thus, if an expression has this type, then it must diverge. This explains our choice of names. If an expression has type $[\,\texttt{Abs}\,]$, then its value must be "absent"; if it has type $[\,\texttt{Pre}\ \tau\,]$, then some value of type τ may be "present".

The data destructor $\texttt{Pre}^{-1}$ is described as follows:

$$\begin{array}{lll} \texttt{Pre}^{-1} & : & (\alpha \to \beta) \to [\,\varphi\,] \to \gamma \\ \text{where} & & \varphi \leq \texttt{Pre}\ \alpha \\ & & \texttt{Pre} \leq \varphi \,?\, \beta \leq \gamma \end{array}$$

The conditional constraint allows $(\texttt{Pre}^{-1}\, e_1\, e_2)$ to receive type $\bot$ when e_2 has type $[\,\texttt{Abs}\,]$, reflecting the fact that $\texttt{Pre}^{-1}$ isn't invoked until e_2 produces some value. Indeed, as long as φ equals `Abs`, the constraint is vacuously satisfied, so γ is unconstrained and assumes its most precise value, namely $\bot$. However, as soon as $\texttt{Pre} \leq \varphi$ holds, $\beta \leq \gamma$ must be satisfied as well. Then, $\texttt{Pre}^{-1}$'s type becomes equivalent to $(\alpha \to \beta) \to [\,\texttt{Pre}\ \alpha\,] \to \beta$, which is its usual ML type.

4.2 The General Case

We now move to a language with a denumerable set of data constructors.

$$e ::= \ldots \mid K \mid K^{-1} \mid \texttt{close}$$

(We let K, L, ... stand for data constructors.) An expression may be tagged, as before, by applying a data constructor to it. Accessing tagged values becomes slightly more complex, because multiple tags exist. The semantics of the elementary data destructor, K^{-1}, is given by the following reduction rules:

$$\begin{array}{lll} K^{-1}\, v_1\, v_2\, (K\, v_3) & \text{reduces to} & (v_1\, v_3) \\ K^{-1}\, v_1\, v_2\, (L\ v_3) & \text{reduces to} & (v_2\, (L\, v_3)) \quad \text{when } K \neq L \end{array}$$

According to these rules, if the value v_3 carries the expected tag, then it is passed to the function v_1. Otherwise, the value – still carrying its tag – is passed to the function v_2. Lastly, a special value, `close`, is added to the language, but no additional reduction rule is defined for it.

How do we modify our type algebra to accommodate multiple data constructors? In Section 4.1, we used field types to encode information about a tagged value's presence or absence. Here, we need exactly the same information, but this time *about every tag*. So, we need to manipulate a family of field types, indexed by tags. To do so, we add one layer to the type algebra: *rows* of field types.

$$\begin{array}{l} \tau ::= \alpha, \beta, \gamma, \ldots \mid \bot \mid \top \mid \tau \to \tau \mid [\,\rho\,] \\ \rho ::= \varphi, \psi, \ldots \mid K : \phi;\ \rho \mid \partial\phi \\ \phi ::= \varphi, \psi, \ldots \mid \texttt{Abs} \mid \texttt{Pre}\ \tau \mid \texttt{Any} \end{array}$$

We can now extend the previous section's proposal, as follows:

$$\begin{array}{rcl} K & : & \alpha \to [\,K : \texttt{Pre}\ \alpha;\ \partial\texttt{Abs}\,] \\ K^{-1} & : & (\alpha \to \beta) \to ([\,K : \texttt{Abs};\ \psi\,] \to \gamma) \to [\,K : \varphi;\ \psi\,] \to \gamma \\ \text{where} & & \varphi \leq \texttt{Pre}\ \alpha \\ & & \texttt{Pre} \leq \varphi\,?\,\beta \leq \gamma \\ \texttt{close} & : & [\,\partial\texttt{Abs}\,] \to \bot \end{array}$$

K^{-1}'s type scheme involves the same constraints as in the basic case. Using a single row variable, namely ψ, in two distinct positions allows expressing the fact that values carrying any tag other than K will be passed unmodified to K^{-1}'s second argument.

`close`'s argument type is $[\,\partial\texttt{Abs}\,]$, which prevents it from ever being invoked. This accords with the fact that `close` does not have an associated reduction rule. It plays the role of a function defined by zero cases.

This system offers *extensible* pattern matchings: any k-ary *case* construct can be written in terms of k nested destructor applications, terminated by

`close`; it will receive the desired, accurate type. (This fact is illustrated by Example 3 in Section 7.) Thus, no specific language construct or type inference rule is needed to deal with them.

5. Record Concatenation

Static typing for record operations is a widely studied problem [4, 15]. Common operations include selection, extension, restriction, and concatenation. The latter comes in two flavors: symmetric and asymmetric. The former requires its arguments to have disjoint sets of fields, whereas the latter gives precedence to the second one when a conflict occurs.

Of these operations, concatenation is probably the most difficult to deal with, because its behavior varies according to the presence or absence of each field in its two arguments. This has led many authors to restrict their attention to type checking, and to not address the issue of type inference [8]. An inference algorithm for asymmetric concatenation was suggested by Wand [30]. He uses *disjunctions* of constraints, however, which gives his system exponential complexity. Rémy [22] suggests an encoding of concatenation into λ-abstraction and record extension, whence an inference algorithm may be derived. Unfortunately, its power is somewhat decreased by subtle interactions with ML's restricted polymorphism; furthermore, the encoding is exposed to the user. In later work [23], Rémy suggests a direct, constraint-based algorithm, which involves a special form of constraints. Sulzmann [27] follows a similar route and creates a custom instance of HM(X), again involving special-purpose concatenation constraints. Our approach is inspired from Rémy's later paper, but re-formulated in terms of conditional constraints, thus showing that no ad hoc constraint forms are necessary.

Again, our presentation is in two steps. The basic case, where records only have one field, is tackled using subtyping and conditional constraints. Then, rows allow us to easily transfer our results to the case of multiple fields.

5.1 The Basic Case

We assume a language equipped with one-field records, whose unique field may be either "absent" or "present". More precisely, we assume a constant data constructor `Abs`, and a unary data constructor `Pre`; a "record" is a value built with one of these constructors. A data destructor, $\texttt{Pre}^{-1}$, allows accessing the contents of a non-empty record. Lastly, the language offers asymmetric and symmetric concatenation primitives, written @ and @@, respectively.

$$e ::= \ldots \mid \texttt{Abs} \mid \texttt{Pre} \mid \texttt{Pre}^{-1} \mid @ \mid @@$$

The relationship between record creation and record access is expressed by a simple reduction rule:

$$\texttt{Pre}^{-1}\,(\texttt{Pre}\,v) \quad \text{reduces to} \quad v$$

The semantics of asymmetric record concatenation is given as follows:

$$\begin{array}{rcl} v_1 \mathbin{@} \texttt{Abs} & \text{reduces to} & v_1 \\ v_1 \mathbin{@} (\texttt{Pre}\ v_2) & \text{reduces to} & \texttt{Pre}\ v_2 \end{array}$$

(In each of these rules, the value v_1 is required to be a record.) Lastly, symmetric concatenation is defined by

$$\begin{array}{rcl} \texttt{Abs} \mathbin{@@} v_2 & \text{reduces to} & v_2 \\ v_1 \mathbin{@@} \texttt{Abs} & \text{reduces to} & v_1 \end{array}$$

(In these two rules, v_1 and v_2 are required to be records.)

The construction of our type algebra is similar to the one performed in Section 4.1. We introduce a (unary) record type constructor, as well as a distinction between normal types and field types:

$$\begin{array}{l} \tau ::= \alpha, \beta, \gamma, \ldots \mid \bot \mid \top \mid \tau \to \tau \mid \{\phi\} \\ \phi ::= \varphi, \psi, \ldots \mid \texttt{Bot} \mid \texttt{Abs} \mid \texttt{Pre}\ \tau \mid \texttt{Either}\ \tau \mid \texttt{Any} \end{array}$$

Let us explain, step by step, our definition of field types. Our first, natural step is to introduce type constructors `Abs` and `Pre`, which allow describing values built with the data constructors `Abs` and `Pre`. The former is a constant type constructor, while the latter is unary and covariant.

Many type systems for record languages define `Pre` τ to be a subtype of `Abs`. This allows a record whose field is present to pretend it is not, leading to a classic theory of records whose fields may be "forgotten" via subtyping. However, when the language offers record concatenation, such a definition isn't appropriate. Why? Concatenation – asymmetric or symmetric – involves a choice between two reduction rules, which is performed by matching one, or both, of the arguments against the data constructors `Abs` and `Pre`. If, at the level of types, we allow a non-empty record to masquerade as an empty one, then it becomes impossible, based on the arguments' types, to find out which rule applies, and to determine the type of the operation's result. In summary, in the presence of record concatenation, no subtyping relationship must exist between `Pre` τ and `Abs`. (This problem is well described – although not solved – in [4].)

This leads us to making `Abs` and `Pre` *incomparable*. Once this choice has been made, completing the definition of field types is rather straightforward. Because our system requires type constructors to form a lattice, we define a least element `Bot`, and a greatest element `Any`. Lastly, we introduce a unary, covariant type constructor, `Either`, which we define as the least upper bound of `Abs` and `Pre`, so that `Abs` $\sqcup$ (`Pre` τ) equals `Either` τ. This optional refinement allows us to keep track of a field's type, even when its presence is not ascertained. (These ideas are due to Rémy, who carries them further in the case of objects [24].) The lattice of field types is shown in Fig. 4.

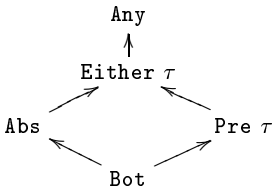

Figure 4: The lattice of record field types

Let us now assign types to the primitive operations offered by the language. Record creation and access receive their usual types:

$$\begin{array}{rcl} \texttt{Abs} & : & \{\texttt{Abs}\} \\ \texttt{Pre} & : & \alpha \to \{\texttt{Pre}\ \alpha\} \\ \texttt{Pre}^{-1} & : & \{\texttt{Pre}\ \alpha\} \to \alpha \end{array}$$

There remains to come up with correct, precise types for both flavors of record concatenation. The key idea is simple. As shown by its operational semantics, (either flavor of) record concatenation is really a function defined by cases over the data constructors `Abs` and `Pre` – and Section 4 has shown how to accurately describe such a function. Let us begin, then, with asymmetric concatenation:

$$\begin{array}{rcl} @ & : & \{\varphi_1\} \to \{\varphi_2\} \to \{\varphi_3\} \\ \text{where} & & \varphi_2 \leq \texttt{Either}\ \alpha_2 \\ & & \texttt{Abs} \leq \varphi_2 \,?\, \varphi_1 \leq \varphi_3 \\ & & \texttt{Pre} \leq \varphi_2 \,?\, \texttt{Pre}\ \alpha_2 \leq \varphi_3 \end{array}$$

Clearly, each conditional constraint mirrors one of the reduction rules. In the second conditional constraint, we assume α_2 is the type of the second record's field – if it has one. The first subtyping constraint represents this assumption. Notice that we use $\texttt{Pre}\ \alpha_2$, rather than φ_2, as the second branch's result type; this is strictly more precise, because φ_2 may be of the form $\texttt{Either}\ \alpha_2$.

Lastly, we turn to symmetric concatenation:

$$\begin{array}{rcl} @@ & : & \{\varphi_1\} \to \{\varphi_2\} \to \{\varphi_3\} \\ \text{where} & & \texttt{Abs} \leq \varphi_1 \,?\, \varphi_2 \leq \varphi_3 \\ & & \texttt{Abs} \leq \varphi_2 \,?\, \varphi_1 \leq \varphi_3 \\ & & \texttt{Pre} \leq \varphi_1 \,?\, \varphi_2 \leq \texttt{Abs} \\ & & \texttt{Pre} \leq \varphi_2 \,?\, \varphi_1 \leq \texttt{Abs} \end{array}$$

Again, each of the first two constraints mirrors a reduction rule. The last two constraints disallow the case where both arguments are non-empty records.

(The careful reader will notice that any one of these two constraints would in fact suffice; both are kept for symmetry.)

In both cases, the operation's description in terms of constraints closely resembles its operational definition. Automatically deriving the former from the latter seems possible; this is an area for future research.

5.2 The General Case

We now move to a language with a denumerable set of record labels, written l, m, etc. The language allows creating the empty record, as well as any one-field record; it also offers selection and concatenation operations. Extension and restriction can be easily added, if desired.

$$e ::= \varnothing \mid \{l = e\} \mid e.l \mid @ \mid @@$$

We do not give the semantics of the language, which should hopefully be clear enough.

At the level of types, we again introduce rows of field types, denoted by ρ. Furthermore, we introduce rows of normal types, denoted by ϱ. Lastly, we lift the five field type constructors to the level of rows.

$$\begin{aligned}
\tau &::= \alpha, \beta, \gamma, \ldots \mid \bot \mid \top \mid \tau \to \tau \mid \{\rho\} \\
\phi &::= \varphi, \psi, \ldots \mid \texttt{Bot} \mid \texttt{Abs} \mid \texttt{Pre}\ \tau \mid \texttt{Either}\ \tau \mid \texttt{Any} \\
\varrho &::= \alpha, \beta, \gamma, \ldots \mid l : \tau;\ \varrho \mid \partial\tau \\
\rho &::= \varphi, \psi, \ldots \mid l : \phi;\ \rho \mid \partial\phi \mid \texttt{Bot} \mid \texttt{Abs} \mid \texttt{Pre}\ \varrho \mid \texttt{Either}\ \varrho \mid \texttt{Any}
\end{aligned}$$

This allows writing complex constraints between rows, such as $\varphi \leq \texttt{Pre}\ \alpha$, where φ and α are row variables. A constraint between rows is interpreted as an infinite family of constraints between types, obtained component-wise. That is, $(l : \varphi';\ \varphi'') \leq \texttt{Pre}\ (l : \alpha';\ \alpha'')$ has the same logical meaning as $(\varphi' \leq \texttt{Pre}\ \alpha') \wedge (\varphi'' \leq \texttt{Pre}\ \alpha'')$. (See Section 2 for details.)

We may now give types to the primitive record operations. Creation and selection are easily dealt with:

$$\begin{aligned}
\varnothing &: \{\partial\texttt{Abs}\} \\
\{l = \cdot\} &: \alpha \to \{l : \texttt{Pre}\ \alpha;\ \partial\texttt{Abs}\} \\
\cdot.l &: \{l : \texttt{Pre}\ \alpha;\ \partial\texttt{Any}\} \to \alpha
\end{aligned}$$

Interestingly, the types of both concatenation operations are *unchanged* from the previous section – at least, syntactically. (We do not repeat them here.) A subtle difference lies in the fact that all variables involved must now be read as row variables, rather than as type variables. In short, the previous section exhibited constraints which describe concatenation, at the level of a single record field; here, the row machinery allows us to replicate these constraints over an infinite set of labels. This increase in power comes almost for free: it does not add any complexity to our notion of subtyping.

6. First-Class Messages

In many current object-oriented languages, messages do not have first-class status. That is, whenever a message is sent to an object, its name is fixed and must be explicitly mentioned; only the message parameters and the receiver object are allowed to vary dynamically. Some languages, however, allow first-class (also known as "dynamic") messages. That is, they allow messages to exist as autonomous entities, which may be computed in arbitrary ways before being sent to an object.

We will view objects as records of functions, and messages as tagged values, made up of a *label* and a *parameter*. Indeed, this simple view suffices to exhibit the type inference problem we are interested in. Thus, we consider a language with records and data constructors, as described in Sections 4.2 and 5.2. Furthermore, we let record labels and data constructors range over a single name space, that of message labels. A primitive message-send operation, written #, is defined as follows:

$$\#\,\{m = v_1; \ldots\}\,(m\,v_2) \quad \text{reduces to} \quad (v_1\,v_2)$$

In plain words, # examines its second argument, which must be some message m with parameter v_2. It then looks up the method named m in the receiver object, and applies the method's code, v_1, to the message parameter. Put another way, if r is a record of functions, then $(\#\,r)$ acts as a function defined by cases. Thus, # is nothing but a witness of the well-known isomorphism which exists between these representations.

6.1 The Problem

In a language without first-class messages, every message-send operation must involve a fixed message label. So, instead of a single, generic operation such as #, the language provides a family of primitive message-send operations, indexed by message labels.

In our view of objects as records of functions, these operations are definable within the language. Indeed, the operation $\#m$, which allows sending the message m to the object o with parameter p, may be defined as $\lambda o.\lambda p.(o.m\ p)$. Then, type inference yields

$$\#m \quad : \quad \{m : \mathtt{Pre}\ (\alpha \to \beta);\ \partial\mathtt{Any}\} \to \alpha \to \beta$$

Because the message label m is statically known, it can be explicitly mentioned in the type scheme, making it easy to require the receiver object to carry an appropriate method.

In a language with first-class messages, on the other hand, m is no longer known. As a result of this difficulty, much of the initial work on typed object-oriented languages has ignored the issue of first-class messages. Gaster [7, chapter 7] studies a static type system where # (under the name of *sumElim*) is a primitive operation. However, in his system, $(\#\,r)$ is well-typed

only if all functions stored in r have the same return type. This condition is clearly too restrictive for our purposes: an object must be allowed to contain methods with different return types. Nishimura [13] suggests a type inference system for an object-oriented language with first-class messages, in the style of Ohori's second-order typed record calculus [15]. It is later re-formulated by Müller and Nishimura [12]. The new presentation is based on feature constraints, including a new form of constraints, specifically intended to model the behavior of a generic message-send operation. Bugliesi and Crafa [3] also attempt to present a simplified view of Nishimura's original work. However, they choose a higher-order type system, thus abandoning type inference.

6.2 A Solution

We said above that, given a record r, the partial application $(\# r)$ yields a function defined by cases. Indeed, given a tagged value $(m\, v)$, it will invoke an appropriate piece of code, selected according to the label m. Good point – this paper is precisely concerned with ways of giving accurate types to functions defined by cases. We have shown how conditional constraints allow ignoring (the return type of) a branch, unless it is liable to be taken. In object-oriented terms, they allow ignoring (the return type of) any method which is provably unrelated with the message at hand. This solves the crucial problem with first-class messages.

Here, we choose to deal directly with the case of multiple message labels, even though the two-step presentation adopted in Sections 4 and 5 would still make sense here. Therefore, we propose:

$$\begin{array}{rcl} \# & : & \{\varphi\} \to [\,\psi\,] \to \beta \\ \text{where} & & \psi \leq \mathtt{Pre}\ \alpha \\ & & \mathtt{Pre} \leq \psi\,?\,\varphi \leq \mathtt{Pre}\ (\alpha \to \partial\beta) \end{array}$$

(Here, all variables except β are row variables.) The operation's first (resp. second) argument is required to be an object (resp. a message), whose contents (resp. possible values) are described by the row variable φ (resp. ψ). The first constraint merely lets α stand for the type of the message parameter. The conditional constraint, which involves three rows, should again be understood as a family, indexed by message labels, of conditional constraints between field types. The conditional constraint associated with some label m will be triggered only if ψ's element at index m is of the form $\mathtt{Pre}\ _$, i.e. only if the message's label may be m. When it is triggered, its right-hand side becomes active, with a three-fold effect. First, φ's element at index m must be of the form $\mathtt{Pre}\ (_ \to _)$, i.e. the receiver object must carry a method labeled m. Second, the method's argument type must be (a supertype of) α's element at label m, i.e. the method must be able to accept the message's parameter. Third, the method's result type must be (a subtype of) β, i.e. the result type of the whole operation will be (at least) the join of the return types of all potentially invoked methods.

This proposal shows that type inference for first-class messages can be performed using existing tools, with no need for dedicated theoretical machinery. It also shows that first-class messages are naturally compatible with all operations on records, including concatenation – a question left unanswered by Nishimura [13].

7. Examples

This section illustrates the proposals made in the previous sections with short examples.

EXAMPLE 3. We consider lists built out of two data constructors, N and C. The function *car*, which returns the first element of a list, if it exists, and E otherwise (where E is another data constructor, standing for *error*), is defined as follows:

$$\begin{array}{rcl} car & = & (N^{-1}\,E \\ & & (C^{-1}\,(\lambda(x,r).x) \\ & & \texttt{close})) \end{array}$$

Then, *car*'s inferred type scheme is

$$\begin{array}{rcl} car & : & [\,N : \varphi;\ C : \psi;\ \partial\texttt{Abs}\,] \to \gamma \\ \text{where} & & \varphi \leq \texttt{Pre}\ \alpha \\ & & \texttt{Pre} \leq \varphi\,?\,[\,E : \texttt{Pre}\ \alpha;\ \partial\texttt{Abs}\,] \leq \gamma \\ & & \psi \leq \texttt{Pre}\ (\beta \times \top) \\ & & \texttt{Pre} \leq \psi\,?\,\beta \leq \gamma \end{array}$$

(Because the language only offers unary constructors, N and E must carry some argument, which remains unspecified here; α stands for its type. Usually, one identifies α with some `unit` type.) The first conditional constraint above tells that the first branch of *car*'s definition – namely E – will not be taken unless φ is "present", i.e. unless *car*'s argument is tagged N. The next one tells that the second branch – namely $\lambda(x,r).x$ – will not be taken unless it is tagged C. No other tags are allowed, because *car*'s argument type involves the row $(N : \varphi;\ C : \psi;\ \partial\texttt{Abs})$, whose projection on any tag other than N and C is `Abs`.

What happens when applying *car*? The type inferred for the expression $car\,(C\,(1, C\,(\texttt{true}, N\,())))$, where it is passed a heterogeneous, 2-element list, is `int`. In other words, this expression is statically found not to produce E, because the first conditional constraint is not triggered.

EXAMPLE 4. We define a function which reads the field l out of a record r and returns a default value d if r has no such field. It is given by `extract` =

$\lambda d.\lambda r.(\{l = d\} @ r).l$. In our system, `extract`'s inferred type is

$$\begin{array}{lll} \texttt{extract} & : & \alpha \to \{l : \varphi;\ \psi\} \to \gamma \\ \quad \text{where} & & \varphi \leq \texttt{Either}\ \beta \qquad\qquad \psi \leq \texttt{Either}\ \epsilon \\ & & \texttt{Abs} \leq \varphi\,?\,\alpha \leq \gamma \qquad\qquad \texttt{Abs} \leq \psi\,?\,\texttt{Abs} \leq \texttt{Any} \\ & & \texttt{Pre} \leq \varphi\,?\,\beta \leq \gamma \qquad\qquad \texttt{Pre} \leq \psi\,?\,\texttt{Pre}\ \epsilon \leq \texttt{Any} \end{array}$$

The first constraint retrieves $r.l$'s type and names it β, regardless of the field's presence. (If the field turns out to be absent, β will be unconstrained.) The left-hand conditional constraints clearly specify the dependency between the field's presence and the function's result.

The right-hand conditional constraints have tautologous conclusions – therefore, they are superfluous. They remain only because our current constraint simplification algorithms are "lazy" and ignore any conditional constraints whose condition has not yet been fulfilled. This problem could be fixed by making the simplification algorithm slightly more aggressive, i.e. by allowing it to check whether the conclusion of a conditional constraint is redundant, regardless of its condition.

The type inferred for `extract` $0\ \{l = 1\}$ and `extract` $0\ \{m = 1\}$ is `int`. Thus, in many cases, one need not be aware of the complexity hidden in `extract`'s type.

Example 5. We assume given an object o, of the following type:

$$\begin{array}{lll} o : & \{ & \texttt{getText} : \texttt{Pre}\ (\texttt{unit} \to \texttt{string}); \\ & & \texttt{setText} : \texttt{Pre}\ (\texttt{string} \to \texttt{unit}); \\ & & \texttt{select} : \texttt{Pre}\ (\texttt{int} \times \texttt{int} \to \texttt{unit}); \\ & & \partial\texttt{Abs} \quad \} \end{array}$$

o may represent, for instance, an editable text field in a graphic user interface system. Its methods allow programmatically getting and setting its contents, as well as selecting a portion of text.

Next, we assume a list data structure, equipped with a simple iterator:

$$\mathit{iter} : (\alpha \to \texttt{unit}) \to \alpha\ \texttt{list} \to \texttt{unit}$$

The following expression creates a list of messages, and uses *iter* to send each of them in turn to o:

$$\mathit{iter}\ (\#\, o)\,[\,\texttt{setText}\ \text{"Hello!"};\ \texttt{select}\,(0,5)\,]$$

This expression is well-typed, because o contains appropriate methods to deal with each of these messages, and because these methods return `unit`, as expected by *iter*. The expression's type is of course `unit`, *iter*'s return type.

Here is a similar expression, which involves a `getText` message:

$$\mathit{iter}\ (\#\, o)\,[\,\texttt{setText}\ \text{"Hello!"};\ \texttt{getText}\,()\,]$$

This time, it is ill-typed. Indeed, sending a `setText` message to o produces a result of type `unit`, while sending it a `getText` message produces a result of type `string`. Thus, $(\#\, o)$'s result type must be $\top$, the join of these types. This makes $(\#\, o)$ an unacceptable argument for *iter*, since the latter expects a function whose return type is `unit`.

8. Conclusion

In this paper, we have advocated the use of a constraint-based type inference system equipped with subtyping, rows and conditional constraints. This provides a common solution to several difficult type inference problems, which, so far, had been addressed using special forms of constraints. From a practical point of view, it allows them to benefit from known constraint simplification techniques (see Appendix A), leading to an efficient inference algorithm [18].

Our system subsumes Rémy's proposal for record concatenation [23], as well as Müller and Nishimura's view of first-class messages [12]. Aiken, Wimmers and Lakshman's "soft" type system [2] is more precise than ours, because it interprets constraints in a richer logical model, but otherwise offers similar features.

The design of a type inference system involves two orthogonal components: a set of typing rules and a constraint language (together with its logical interpretation). As to the former, we have suggested using HM(X) [14, 28, 27], whose formulation appears most elegant, but other choices would be possible (see e.g. [10, 17]). The focus of the paper is really on the latter: our aim was to find a constraint language expressive enough to accurately describe the features of the programming language at hand. One should emphasize the fact that we do not, a priori, view the constraint system SRC as better (simpler, more elementary, more canonical, etc.) than its competitors. We merely take its wide applicability as evidence of the fact that it is comparatively more general-purpose (less ad hoc) than some of its predecessors.

To conclude, we hope this paper illustrates how a small number of well-understood logic mechanisms allow building an advanced type inference system.

Acknowledgements

Thanks to Jacques Garrigue, Martin Müller, Didier Rémy and Martin Sulzmann for stimulating discussions. Didier Rémy also proof-read a version of the manuscript. Lastly, I would like to thank the anonymous referees for their suggestions.

Appendix A. Algorithms and Proofs

This appendix contains a formal description of constraint resolution and simplification algorithms, in the presence of atomic and conditional subtyping constraints. Resolution is required in determining whether a program is type-correct; simplification is key to achieving reasonable efficiency.

The system described in this appendix does not have *rows*, or a separation of types into distinct *kinds*, but otherwise has all features presented in the body of this paper. Adding rows to this formal description would require work, but should not pose any foreseeable difficulty, since the concept of row is essentially orthogonal to the notion of subtyping. Adding kinds should be routine. A reference implementation of the full system, including rows and kinds, is available [18].

This appendix describes an extension of [17] with conditional constraints. Thus, most proofs presented here are partial, and describe only the modifications required to accommodate conditional constraints. However, all definitions and statements are complete.

This appendix is laid out as follows. First, we review all necessary concepts, including ground types, types, constraints, and type schemes. Then, we give a constraint resolution algorithm, and three constraint simplification algorithms.

Throughout this appendix, we use a couple of notational shortcuts. If P is a logic predicate, then

$$\begin{array}{llll} \forall \rho \vdash C & P(\rho) & \text{stands for} & \forall \rho \quad (\rho \vdash C) \Rightarrow P(\rho) \\ \exists \rho \vdash C & P(\rho) & \text{stands for} & \exists \rho \quad (\rho \vdash C) \wedge P(\rho) \end{array}$$

Appendix A.1 Ground Types

As in Section 2, our formal development is parameterized with an arbitrary ground signature (see Definition 1). We assume that it defines only one kind, so we write $\mathcal{S}$ and $\mathbb{T}$ instead of $\mathcal{S}_\kappa$ and $\mathbb{T}_\kappa$. We write $\bot_\mathcal{S}$, $\top_\mathcal{S}$, $\leq_\mathcal{S}$, $\sqcup_\mathcal{S}$ and $\sqcap_\mathcal{S}$ instead of $\bot_\kappa$, $\top_\kappa$, $\leq_\kappa$, $\sqcup_\kappa$ and $\sqcap_\kappa$. We also assume $\mathcal{L}_{\text{row}} = \varnothing$. The model $(\mathbb{T}, \leq)$, is defined as in Definitions 2 and 3.

In this appendix, we use the letter τ to denote either a ground type, or a type, and sometimes both at the same time (see e.g. Definition 17 and Theorem 2). We will try to preserve a clear distinction whenever possible.

THEOREM 1. *$\mathbb{T}$, equipped with $\leq$, is a lattice. Its lattice operations, denoted by $\sqcup$ and $\sqcap$, are characterized by the following identities:*

$$(\tau_1 \sqcap \tau_2)(\epsilon) = \tau_1(\epsilon) \sqcap_\mathcal{S} \tau_2(\epsilon)$$
$$\forall l \in \text{dom}(\tau_1 \sqcap \tau_2) \quad (\tau_1 \sqcap \tau_2).l = \tau_1.l \sqcap^l \tau_2.l$$

where $\sqcap$ may stand for $\sqcup$ or $\sqcap$. (We let $\sqcap^l$ stand for $\sqcap$ when $l \in \mathcal{L}^+$; when $l \in \mathcal{L}^-$, $\sqcup^l$ stands for $\sqcap$ and $\sqcap^l$ stands for $\sqcup$.) In the right-hand side of the

second equation, $\tau_1.l$ (resp. $\tau_2.l$) may be undefined; in such a case, it should be read as the neutral element of $\Box^l$.

Note that, because of the last requirement of Definition 1, at least one of $\tau_1.l$ and $\tau_2.l$ must be defined in the second equation above.

We let $\bot$ (resp. $\top$) stand for the ground type τ such that $\mathrm{dom}(\tau) = \{\epsilon\}$ and $\tau(\epsilon) = \bot_{\mathcal{S}}$ (resp. $\top_{\mathcal{S}}$).

Appendix A.2 Types

Types are defined as in Section 2.2, except row terms are disallowed.

DEFINITION 7. *Let $\mathcal{V}$ be a denumerable set of* type variables*, denoted by α, β, etc. The set of* types*, denoted by $\mathcal{T}$, is the term algebra $T(\Sigma, \mathcal{V})$. In other words, a type τ is either a type variable, or a* constructed *term, of the form $s(\tau_l)_{l \in a(s)}$, where $s \in \mathcal{S}$ is τ's* head constructor*, also written* $\mathrm{hd}(\tau)$.

DEFINITION 8. *A* ground substitution *ρ is a total mapping from type variables to ground types. Ground substitutions are straightforwardly extended to types.*

Appendix A.3 Constraints and Type Schemes

We now give syntax and semantics for three kinds of constraints: *atomic* constraints, *conditions* and *conditional* constraints. In each case, the notation $\rho \vdash_k c$ means that the ground substitution ρ k-satisfies the constraint c. The notation $\rho \vdash c$ means that ρ satisfies c, and holds, by definition, if and only if $\rho \vdash_k c$ holds for all $k \in \mathbb{N}^+$.

DEFINITION 9. *An* atomic constraint *is a pair of types, written $\tau_1 \leq \tau_2$. A ground substitution ρ* k-satisfies *it iff $\rho(\tau_1) \leq_k \rho(\tau_2)$.*

DEFINITION 10. *A* condition *is a pair of a symbol $s \in \mathcal{S}$ and of a type τ, written $s \leq \tau$, where s must be a* prime *element of $\mathcal{S}$. A ground substitution ρ* satisfies *$s \leq \tau$ iff $s \leq_{\mathcal{S}} \mathrm{hd}(\rho(\tau))$.*

DEFINITION 11. *A* conditional constraint *is a pair of a condition and of an atomic constraint, written $s \leq \tau \,?\, \tau_1 \leq \tau_2$. A ground substitution ρ* k-satisfies *it iff $\rho \vdash s \leq \tau$ implies $\rho \vdash_k \tau_1 \leq \tau_2$.*

Having defined constraints, we may define notions of satisfaction and entailment on constraint sets. They are defined in the usual way. We also introduce a non-standard notion of *pre-satisfaction* (resp. *pre-entailment*), which is logically weaker (resp. stronger) than its standard counterpart, because it ignores conditional constraints. These notions are purely technical; they are used only within our proofs.

DEFINITION 12. *Let C be a set of constraints, both atomic and conditional. A ground substitution ρ is a* pre-solution *of C iff $\rho \vdash c$ holds for all* atomic *$c \in C$. ρ is a* solution *of C iff $\rho \vdash c$ holds for* all *$c \in C$. We write $\rho \vdash^{\text{pre}} C$ in the former case, and $\rho \vdash C$ in the latter.*

Let c be a constraint. C pre-entails *c, which we write $C \Vdash^{\text{pre}} c$, iff $\forall \rho \vdash^{\text{pre}} C \quad \rho \vdash c$. C* entails *c, which we write $C \Vdash c$, iff $\forall \rho \vdash C \quad \rho \vdash c$.*

We now define *type schemes*. They are constrained polymorphic types, i.e. types containing variables whose possible instantiations are restricted by a constraint set. For simplicity, we only consider *closed* type schemes, i.e. type schemes which have no free type variables. Although somewhat uncommon, type systems exist which respect this restriction (see [29, 17]). It should also be possible to extend our results to the case of arbitrary type schemes.

DEFINITION 13. *A* type scheme *is a pair of a type τ and of a constraint set C, written $\forall C.\, \tau$.*

A type scheme σ represents a set of ground types, which we call its *denotation*. Each of these ground types represent one possible correct behavior of the program described by σ. A type scheme whose denotation is empty (i.e. whose constraint set has no solution) thus represents an ill-typed program.

DEFINITION 14. *The* denotation *$[\![\sigma]\!]$ of a type scheme σ is the union of the upper cones generated by its ground instances with respect to $\leq$. That is,*

$$[\![\forall C.\, \tau]\!] = \{\tau' ;\, \exists \rho \vdash C \quad \rho(\tau) \leq \tau'\}$$

A type scheme whose denotation is bigger represents a larger set of possible behaviors; thus, it is more general. This notion allows comparing type schemes, while accounting for polymorphism and subtyping at the same time. It was introduced in [29], where it was written $\leq^{\forall}$; we denote it $\preccurlyeq$.

DEFINITION 15. *Given two type schemes σ_1 and σ_2, the former is said to be* more general *than the latter iff $[\![\sigma_1]\!] \supseteq [\![\sigma_2]\!]$; we shall then write $\sigma_1 \preccurlyeq \sigma_2$. In other words, σ_1 is more general than σ_2 iff for any ground instance of σ_2, there exists a smaller ground instance of σ_1. Formally,*

$$(\forall C_1.\, \tau_1) \preccurlyeq (\forall C_2.\, \tau_2)$$

is thus equivalent to

$$\forall \rho_2 \vdash C_2 \quad \exists \rho_1 \vdash C_1 \quad \rho_1(\tau_1) \leq \rho_2(\tau_2)$$

We write $\sigma_1 \approx \sigma_2$ when $\sigma_1 \preccurlyeq \sigma_2$ and $\sigma_2 \preccurlyeq \sigma_1$.

The relation $\approx$ offers a specification of constraint simplification. Indeed, a type scheme σ can be *simplified* into a type scheme σ' only if $\sigma \approx \sigma'$. One would also expect σ' to have a smaller textual representation than σ, but

that is not a requirement; it is rather to be viewed as an implementation detail.

We conclude this section with a definition of what it means for a type scheme to be *made up of small terms*. All of the algorithms defined here will expect this property to hold, and will preserve it, making it a global invariant. This choice simplifies definitions and proofs. Furthermore, from a practical point of view, it allows enforcing maximum sharing, since it requires every sub-term to be "named" by a type variable, allowing our minimization algorithm to identify sub-terms.

DEFINITION 16. *A* small term *is a constructed type term whose strict sub-terms are type variables. A type scheme* $\forall C.\, \tau$ *is* made up of small terms *iff it satisfies the following conditions:*

- τ *is a type variable;*
- *for all* $(\tau_1 \leq \tau_2) \in C$*, either* τ_1 *and* τ_2 *are type variables, or one is a variable and the other is a small term.*
- *for all* $(s \leq \tau \,?\, \tau_1 \leq \tau_2) \in C$*,* τ*,* τ_1 *and* τ_2 *are type variables.*

Every type scheme can be turned into an equivalent type scheme which is made up of small terms. (In practice, this would be done when converting type schemes input by the user into some internal representation.)

Appendix A.4 Solving Constraints

We begin with a fundamental technical result, which describes a weak, sufficient condition for a constraint set to have a solution. It shall form the basis for the proof of the closure algorithm. We prove a fairly powerful version of this result, allowing ground constants to appear in constraints. (If these constraints were to be written, some finite representation of these constants would be required; however, such is not the case here.) Thanks to this generalization, this result will also form the basis for the proof of the garbage collection algorithm.

DEFINITION 17. *A constraint set* with ground constants *is a constraint set* C*, where atomic constraints may involve either two variables, one variable and a small term, or one variable and a ground type, and where conditional constraints have their usual form. Define the assertion* $C \Vdash^{+1} \tau_1 \leq \tau_2$ *to mean*

$$\forall k \geq 0 \quad \forall \rho \vdash_k^{\mathrm{pre}} C \quad \rho \vdash_{k+1} \tau_1 \leq \tau_2$$

Define $C^{\downarrow}(\alpha) = \{\tau \,;\, \tau \notin \mathcal{V} \wedge \tau \leq \alpha \in C\}$ *and* $C^{\uparrow}(\alpha) = \{\tau \,;\, \tau \notin \mathcal{V} \wedge \alpha \leq \tau \in C\}$*.* C *is said to be* weakly closed *iff the following conditions are met:*

(1) $\alpha \leq \beta \in C$ *and* $\beta \leq \gamma \in C$ *imply* $\alpha \leq \gamma \in C$*;*

(2) $\alpha \leq \beta \in C$ *and* $\tau \in C^{\downarrow}(\alpha)$ *imply* $\exists \tau' \in C^{\downarrow}(\beta) \quad C \Vdash^{+1} \tau \leq \tau'$*;*

(3) $\alpha \leq \beta \in C$ *and* $\tau' \in C^{\uparrow}(\beta)$ *imply* $\exists \tau \in C^{\uparrow}(\alpha) \quad C \Vdash^{+1} \tau \leq \tau'$*;*

(4) $\tau \in C^{\downarrow}(\alpha)$ *and* $\tau' \in C^{\uparrow}(\alpha)$ *imply* $C \Vdash^{+1} \tau \leq \tau'$*;*

(5) $\alpha \leq \beta \in C$ *and* $s \leq \beta \,?\, c \in C$ *imply* $s \leq \alpha \,?\, c \in C$*;*

(6) $s \leq \alpha \,?\, c \in C$*,* $\tau \in C^{\downarrow}(\alpha)$ *and* $s \leq_{\mathcal{S}} \mathrm{hd}(\tau)$ *imply* $C \Vdash^{\mathrm{pre}} c$*.*

$C^{\downarrow}(\alpha)$ contains all lower bounds of α which are not type variables; thus, every $\tau \in C^{\downarrow}(\alpha)$ must be either a small type term, or a ground type. A similar remark holds concerning $C^{\uparrow}(\alpha)$.

Conditions 1 and 5 above are purely syntactic transitivity conditions. Conditions 2 to 4 also involve transitivity, but the use of $\Vdash^{+1}$ allows expressing these conditions in a logical, rather than syntactic, way, making them less restrictive.

THEOREM 2. *Let C be a constraint set with ground constants. If C is weakly closed, then C has a solution.*

PROOF. Note that this proof only uses Conditions 2, 4 and 6 of Definition 17. The other conditions shall be required by further theorems, such as the correctness proof of garbage collection.

The first step of the proof consists in exhibiting a ground substitution ρ such that, for all $\alpha \in \mathrm{fv}(C)$, $\rho(\alpha)$ equals $\sqcup\{\rho(\tau)\,;\, \tau \in C^{\downarrow}(\alpha)\}$, and proving that ρ is a pre-solution of C. In fact, this step coincides with the classic proof performed in the absence of conditional constraints [17]; we shall not repeat it here.

The second step consists in proving that ρ is a full solution of C. Pick a conditional constraint $s \leq \alpha \,?\, c \in C$. Assume $\rho \vdash s \leq \alpha$. By definition of ρ, this statement can be written

$$\begin{aligned} s \leq_{\mathcal{S}} \mathrm{hd}(\rho(\alpha)) &= \mathrm{hd}(\sqcup\{\rho(\tau)\,;\, \tau \in C^{\downarrow}(\alpha)\}) \\ &= \sqcup_{\mathcal{S}}\{\mathrm{hd}(\rho(\tau))\,;\, \tau \in C^{\downarrow}(\alpha)\} \\ &= \sqcup_{\mathcal{S}}\{\mathrm{hd}(\tau)\,;\, \tau \in C^{\downarrow}(\alpha)\} \end{aligned}$$

(The identity $\mathrm{hd}(\rho(\tau)) = \mathrm{hd}(\tau)$ stems from the fact that τ is either a small term, or a ground constant, with a fixed head constructor.) Considering that s is prime (see Definition 10), this entails $s \leq_{\mathcal{S}} \mathrm{hd}(\tau)$ for some $\tau \in C^{\downarrow}(\alpha)$. We can then apply Condition 6 of Definition 17, which yields $C \Vdash^{\mathrm{pre}} c$. Since ρ is a pre-solution of C, this implies $\rho \vdash c$. We have thus verified $\rho \vdash s \leq \alpha \,?\, c$, proving that ρ is a solution of C. □

Equipped with this technical result, we are now ready to define a constraint resolution algorithm. It is based on a simple *closure* computation. We begin by defining an auxiliary constraint decomposition function, which breaks a constraint down into a set of equivalent constraints.

DEFINITION 18. *Given types* τ_1 *and* τ_2*,* $\mathrm{subc}(\tau_1 \leq \tau_2)$ *is defined as*

- $\{\tau_1 \leq \tau_2\}$*, if* τ_1 *or* τ_2 *is a variable;*

- $\{\tau_1.l \leq^l \tau_2.l\,;\, l \in \mathrm{dom}(\tau_1) \cap \mathrm{dom}(\tau_2)\}$, *if* τ_1 *and* τ_2 *are constructed terms such that* $\mathrm{hd}(\tau_1) \leq_{\mathcal{S}} \mathrm{hd}(\tau_2)$.

Note that $\mathrm{subc}(\tau_1 \leq \tau_2)$ is undefined when $\mathrm{hd}(\tau_1) \not\leq_{\mathcal{S}} \mathrm{hd}(\tau_2)$; indeed, such a constraint is clearly unsatisfiable.

Using this auxiliary function, we can now describe the closure conditions:

DEFINITION 19. *Let C be a constraint set, made up of small terms. C is said to be* closed *iff*

(1) $\tau \leq \alpha \in C$ *and* $\alpha \leq \tau' \in C$ *imply* $\mathrm{subc}(\tau \leq \tau') \subseteq C$;

(2) $\alpha \leq \beta \in C$ *and* $s \leq \beta\,?\,c \in C$ *imply* $s \leq \alpha\,?\,c \in C$;

(3) $s \leq \alpha\,?\,c \in C$, $\tau \in C^{\downarrow}(\alpha)$ *and* $s \leq_{\mathcal{S}} \mathrm{hd}(\tau)$ *imply* $c \in C$.

Condition 1 is the classic closure condition, found e.g. in [17]; it involves transitivity and structural decomposition. Condition 2 is a transitivity condition concerning conditional constraints. Condition 3 requires that the conclusion of a conditional constraint whose condition must be satisfied be discharged into the constraint set.

It is easy to check that closure implies weak closure [17]. This yields an algorithm to decide whether a constraint set C has a solution: attempt to compute the smallest closed set C^* containing it, by repeated application of the above three rules. Each rule preserves the set's solution space. So, if the computation succeeds, then C has a solution; if, on the other hand, it fails (because subc is applied outside of its domain), then C has no solution.

Consider a conditional constraint $s \leq \alpha\,?\,c$. According to the closure rules above, the atomic constraint c will have no effect on the constraint resolution process until it is discharged by rule 3. That is, c will be ignored until the algorithm discovers some evidence that the condition $s \leq \alpha$ *must* be satisfied. This explains why conditional constraints *delay* type computations, as mentioned in the body of this paper. The algorithm will not speculate about the consequence of the conditional constraint, should its condition be satisfied; rather, it waits until it has no choice but satisfy c.

Appendix A.5 Polarity

We now define how to associate a *polarity* with each type variable in a type scheme whose constraint set if (weakly) closed. This notion will be used in the definition of all three constraint simplification algorithms.

DEFINITION 20. *Consider a type scheme $\sigma = \forall C.\,\delta$, made up of small terms, where C is weakly closed. The set of* positive *variables of σ, and the set of* negative *variables of σ, respectively denoted by $\mathrm{fv}^+(\sigma)$ and $\mathrm{fv}^-(\sigma)$, are the smallest subsets P and N of $\mathrm{fv}(\sigma)$ such that*

- $\delta \in P$;
- $\forall \alpha \in P \quad \forall \tau \in C^{\downarrow}(\alpha) \quad \mathrm{split}(\tau) \subseteq (N, P)$;

- $\forall \alpha \in N \quad \forall \tau \in C^{\uparrow}(\alpha) \quad \text{split}(\tau) \subseteq (P, N)$;
- $\forall \alpha \in N \quad s \leq \alpha \,?\, \beta \leq \gamma \in C \;\Rightarrow\; \beta \in P \wedge \gamma \in N$.

where the auxiliary function split *maps a small term* τ *to an element of* $2^{\mathcal{V}} \times 2^{\mathcal{V}}$*, as follows:*

$$\text{split}(\tau) = (\{\tau.l \,;\, l \in \mathcal{L}^-\}, \{\tau.l \,;\, l \in \mathcal{L}^+\})$$

A type variable is said to be *bipolar* if it is positive and negative, and *neutral* if it is neither. $\text{fv}^+(\sigma)$ and $\text{fv}^-(\sigma)$ can be computed in time linear in the size of σ, using a simple fix-point calculation. Every type scheme is equivalent to a type scheme with no bipolar variables; we do not prove this result here.

Appendix A.6 Garbage Collection

Knowing the polarity of each variable allows us to throw away many redundant constraints, as shown by the following definition and theorem.

Definition 21. *Consider* σ *as in Definition 20. The image of* σ *through* garbage collection*, denoted by* $\text{GC}(\sigma)$*, is the type scheme* $\forall D.\, \delta$*, where* D *is a subset of* C *defined as follows:*

- $\alpha \leq \beta \in D$ *iff* $\alpha \leq \beta \in C$*,* $\alpha \in \text{fv}^-(\sigma)$ *and* $\beta \in \text{fv}^+(\sigma)$*;*
- $D^{\downarrow}(\alpha)$ *equals* $C^{\downarrow}(\alpha)$ *if* $\alpha \in \text{fv}^+(\sigma)$*, and* $\varnothing$ *otherwise;*
- $D^{\uparrow}(\alpha)$ *equals* $C^{\uparrow}(\alpha)$ *if* $\alpha \in \text{fv}^-(\sigma)$*, and* $\varnothing$ *otherwise;*
- $s \leq \alpha \,?\, \beta \leq \gamma \in D$ *iff* $s \leq \alpha \,?\, \beta \leq \gamma \in C$ *and* $\alpha \in \text{fv}^-(\sigma)$.

This definition is mostly identical to the one in [17]; only the fourth point is new, and specifies that a conditional constraint is redundant unless it bears on a negative variable. In operational terms, a conditional constraint $s \leq \alpha \,?\, c$ can be triggered only if α receives a lower bound which exceeds s. Considering that only negative variables can receive new lower bounds in the future, this constraint has no effect unless α is negative.

Theorem 3. *Consider* σ *as in Definition 21. Then* $\sigma \approx \text{GC}(\sigma)$.

Proof. Write $\sigma' = \text{GC}(\sigma)$. Since σ' has fewer constraints, it is clear that $\sigma' \preccurlyeq \sigma$. So, we need to prove $\sigma \preccurlyeq \sigma'$. According to Definition 15, this is equivalent to

$$\forall \rho' \vdash D \quad \exists \rho \vdash C \quad \rho(\delta) \leq \rho'(\delta)$$

Pick some $\rho' \vdash D$. We now wish to prove that $C \cup \{\delta \leq \rho'(\delta)\}$ admits a solution. This is a constraint set *with ground constants*, as per Definition 17. We shall meet our goal by proving that the following constraint set—a superset of the previous one—is weakly closed:

$$\begin{aligned} C \cup \{\rho'(\beta) \leq \alpha \,;\, \beta \in \text{fv}^-(\sigma) \wedge \beta \leq \alpha \in C^r\} \\ \cup \{\alpha \leq \rho'(\beta) \,;\, \beta \in \text{fv}^+(\sigma) \wedge \alpha \leq \beta \in C^r\} \end{aligned}$$

(C^r denotes the reflexive closure of C, i.e. $\alpha \leq \beta \in C^r$ iff $\alpha = \beta$ or $\alpha \leq \beta \in C$.) Let E denote this set.

That E should satisfy Conditions 1 to 4 of Definition 17 is a classic result, proved in [17]. The novelty, in the presence of conditional constraints, is to check that E also satisfies Conditions 5 and 6.

To check Condition 5, assume $\alpha \leq \beta \in E$ and $s \leq \beta \,?\, c \in E$. Considering the definition of E, these constraints must in fact belong to C. Since C itself is weakly closed, $s \leq \alpha \,?\, c$ belongs to C, which is a subset of E.

To check Condition 6, assume $s \leq \alpha \,?\, c \in E$, $\tau \in E^{\downarrow}(\alpha)$ and $s \leq_{\mathcal{S}} \mathrm{hd}(\tau)$. As above, $s \leq \alpha \,?\, c$ must in fact belong to C. Furthermore, if $\tau \in C^{\downarrow}(\alpha)$, then it is again easy to conclude, considering that C itself is weakly closed. Thus, let us assume $\tau \notin C^{\downarrow}(\alpha)$. Considering the definition of E, we must have $\tau = \rho'(\beta)$, $\beta \in \mathrm{fv}^-(\sigma)$ and $\beta \leq \alpha \in C^r$, for some $\alpha, \beta \in \mathrm{fv}(C)$.

We have $s \leq \alpha \,?\, c \in C$ and $\beta \leq \alpha \in C^r$. Because C satisfies Condition 5 of Definition 17, this entails $s \leq \beta \,?\, c \in C$. Furthermore, since $\beta \in \mathrm{fv}^-(\sigma)$, this constraint is preserved by garbage collection; formally, Definition 21 states that $s \leq \beta \,?\, c \in D$. Since ρ' is a solution of D, we have $\rho' \vdash s \leq \beta \,?\, c$. Finally, recall that $s \leq_{\mathcal{S}} \mathrm{hd}(\tau) = \mathrm{hd}(\rho'(\beta))$, which can be written $\rho' \vdash s \leq \beta$. By bringing both results together, we obtain $\rho' \vdash c$.

Let us now write γ_1 (resp. γ_2) for the left-hand (resp. right-hand) side of c. The assertion $\rho' \vdash c$ can be re-stated $\rho'(\gamma_1) \leq \rho'(\gamma_2)$. Besides, we have $\beta \in \mathrm{fv}^-(\sigma)$ and $s \leq \beta \,?\, \gamma_1 \leq \gamma_2 \in C$; according to Definition 20, this entails $\gamma_1 \in \mathrm{fv}^+(\sigma)$ and $\gamma_2 \in \mathrm{fv}^-(\sigma)$. Then, according to the definition of E, the constraints $\gamma_1 \leq \rho'(\gamma_1)$ and $\rho'(\gamma_2) \leq \gamma_2$ must appear in E. It follows that any pre-solution of E satisfies $\gamma_1 \leq \gamma_2$. In other words, $E \Vdash^{\mathrm{pre}} c$. □

Appendix A.7 Canonization

DEFINITION 22. *A constraint set C is in* canonical form *iff each variable $\alpha \in \mathrm{fv}(C)$ has exactly one constructed lower (resp. upper) bound, i.e. iff $C^{\downarrow}(\alpha)$ and $C^{\uparrow}(\alpha)$ are singletons.*

We now define an algorithm which turns an arbitrary type scheme σ into an equivalent type scheme in canonical form.

DEFINITION 23. *Let $\sigma = \forall C.\, \delta$ be a type scheme, made up of small terms, with no bipolar variables, such that $\sigma = \mathrm{GC}(\sigma)$.*

Let V (resp. W) range over non-empty subsets of $\mathrm{fv}^-(\sigma)$ (resp. $\mathrm{fv}^+(\sigma)$). For each such V (resp. W) of cardinality greater than 1, pick a fresh variable γ_V (resp. λ_W). (By fresh variables, we mean that these variables are pairwise distinct, and distinct from σ's variables.)

Define the rewriting functions *r^- and r^+ according to Fig. 5. The first three lines define r^- (resp. r^+) over non-empty sets of negative (resp. positive) variables; the next two extend them to sets of negative (resp. positive) small terms, ranged over by T. v stands for either $+$ or $-$; $\sqcup_{\mathcal{S}}^v$ stands for $\sqcup_{\mathcal{S}}$ when $v = +$, and for $\sqcap_{\mathcal{S}}$ when $v = -$; v^l stands for v when $l \in \mathcal{L}^+$, and*

$$
\begin{aligned}
r^v(\{\alpha\}) &= \alpha \\
r^+(W) &= \lambda_W \text{ when } |\, W \,| > 1 \\
r^-(V) &= \gamma_V \text{ when } |\, V \,| > 1 \\
\mathrm{hd}(r^v(T)) &= \sqcup^v_{\mathcal{S}} \mathrm{hd}(T) \\
\forall l \in a(\sqcup^v_{\mathcal{S}} \mathrm{hd}(T)) \quad r^v(T).l &= r^{v^l}(T.l)
\end{aligned}
$$

Figure 5: Definition of the rewriting functions

$$r^-(V) \leq r^+(W) \in D \text{ iff } \exists \alpha \in V \quad \exists \beta \in W \quad \alpha \leq \beta \in C$$

$$
\begin{aligned}
D^{\downarrow}(\alpha) &= \{r^+(C^{\downarrow}(\alpha))\} & D^{\uparrow}(\alpha) &= \{r^-(C^{\uparrow}(\alpha))\} \\
D^{\downarrow}(\gamma_V) &= \{\bot\} & D^{\uparrow}(\gamma_V) &= \{r^-(\cup C^{\uparrow}(V))\} \\
D^{\downarrow}(\lambda_W) &= \{r^+(\cup C^{\downarrow}(W))\} & D^{\uparrow}(\lambda_W) &= \{\top\}
\end{aligned}
$$

$$s \leq r^-(V) \,?\, c \in D \text{ iff } \exists \alpha \in V \quad s \leq \alpha \,?\, c \in C$$

Figure 6: Canonization

for the opposite of v when $l \in \mathcal{L}^-$. On the last line of Fig. 5, $T.l$ stands for $\{\tau.l\,;\, \tau \in T\}$. The expression $r^{v^l}(T.l)$ is well-defined, because $T.l$ is a non-empty *set of variables. Indeed, l belongs to $a(\sqcup^v_{\mathcal{S}} \mathrm{hd}(T))$. According to the last condition of Definition 1, this must be a subset of $\cup a(\mathrm{hd}(T))$; that is, there must exist some $\tau \in T$ such that $\tau.l$ is defined.*

The image of σ through canonization, *denoted by* $\mathrm{Can}(\sigma)$, *is $\forall D.\, \delta$, where the constraint set D is given by Fig. 6. It is clear that* $\mathrm{Can}(\sigma)$ *is in canonical form.*

Considering our strong hypotheses on σ, one easily proves that $\mathrm{Can}(\sigma)$ is closed. One can also give a conservative approximation of the polarity of each variable in $\mathrm{Can}(\sigma)$. Indeed, if a variable α is positive (resp. negative, neutral) in σ, then it is *at most* positive (resp. negative, neutral) in $\mathrm{Can}(\sigma)$. Furthermore, any λ_W (resp. γ_V) is at most positive (resp. negative) in $\mathrm{Can}(\sigma)$.

THEOREM 4. *Consider σ as in Definition 23. Then $\sigma \approx \mathrm{Can}(\sigma)$.*

PROOF. Let us use the notations of Definition 23. We first show that $\mathrm{Can}(\sigma) \preccurlyeq \sigma$, i.e.

$$\forall \rho \vdash C \quad \exists \rho' \vdash D \quad \rho'(\delta) \leq \rho(\delta)$$

Pick some $\rho \vdash C$. Define ρ' by

$$\rho'(\alpha) = \rho(\alpha) \qquad \rho'(\gamma_V) = \sqcap \rho(V) \qquad \rho'(\lambda_W) = \sqcup \rho(W)$$

Clearly, for any W, $\rho'(r^+(W)) = \sqcup \rho(W)$. Similarly, $\rho'(r^-(V)) = \sqcap \rho(V)$. Extending these assertions to sets of small terms, rather than sets of variables, is straightforward. Using these results, it is a matter of routine to ascertain that ρ' satisfies D. Here, we shall only check that all conditional constraints of D are satisfied by ρ'. Consider such a constraint; it must be of the form $s \leq r^-(V) \,?\, c$, where $s \leq \alpha \,?\, c \in C$ for some $\alpha \in V$. Assume $\rho' \vdash s \leq r^-(V)$. This can be written

$$\begin{aligned} s &\leq_{\mathcal{S}} \mathrm{hd}(\rho'(r^-(V))) \\ &= \mathrm{hd}(\sqcap \rho(V)) \\ &= \sqcap_{\mathcal{S}} \mathrm{hd}(\rho(V)) \\ &\leq_{\mathcal{S}} \mathrm{hd}(\rho(\alpha)) \qquad \text{since } \alpha \in V \end{aligned}$$

So, $\rho \vdash s \leq \alpha$ holds. Because $s \leq \alpha \,?\, c$ appears in C, and because ρ satisfies C, we must then have $\rho \vdash c$. However, ρ and ρ' coincide over $\mathrm{fv}(\sigma)$; so, $\rho' \vdash c$ holds as well. Thus, we have checked that ρ' satisfies $s \leq r^-(V) \,?\, c$, as desired.

The other direction of the proof is slightly more difficult, because D does not entail C; in fact, our definition of canonization contains a built-in garbage collection step. We introduce an intermediate type scheme $\sigma' = \forall E.\, \delta$, where E is defined by

$$E = D \cup \{\alpha \leq \lambda_W \,;\, \alpha \in W\} \cup \{\gamma_V \leq \alpha \,;\, \alpha \in V\}$$

This time, thanks to the added constraints, it is easy enough to prove that E entails C, which implies $\sigma \preccurlyeq \sigma'$. There remains to prove that $\sigma' \preccurlyeq \mathrm{Can}(\sigma)$. We shall do so by noticing that the constraints in $E \setminus D$ are superfluous, according to garbage collection. The result shall then follow from Theorem 3. Our first objective is to prove that E is weakly closed, which entitles us to apply garbage collection to σ'.

Proving that E satisfies Conditions 1 to 4 of Definition 17 is (tedious) routine; we refer the interested reader to [17].

To check that E satisfies Condition 5, assume $s \leq \psi \,?\, c \in E$ and $\phi \leq \psi \in E$. Considering the form of the conditional constraints which appear in E (see Fig. 6), ψ must be the image of some set of negative variables through r^-. But then, considering the form of the constraints between variables in E, $\phi \leq \psi$ must be of the form $\gamma_V \leq \alpha$, where $\alpha \in V$. So, ψ coincides with α. Thus, $s \leq \alpha \,?\, c$ appears in E; according to Fig. 6, it also appears in C.

Since $\alpha \in V$, another look at Fig. 6 indicates that $s \leq \gamma_V \,?\, c$ appears in D, hence in E. This was our goal, since γ_V is none other than ϕ.

To check that E satisfies Condition 6, assume $s \leq r^-(V) \,?\, c \in E$, $\tau \in E^{\downarrow}(r^-(V))$ and $s \leq_{\mathcal{S}} \mathrm{hd}(\tau)$.

First, we prove that $|\, V \,| = 1$. Indeed, if $|\, V \,| > 1$ were true, then $r^-(V)$ would be γ_V. Then, we would have $\tau \in E^{\downarrow}(\gamma_V) = \{\bot\}$, so $\tau = \bot$. Since $s \leq_{\mathcal{S}} \mathrm{hd}(\tau)$, it would follow that $s = \bot_{\mathcal{S}}$, which is forbidden by Definition 10: $\bot_{\mathcal{S}}$ is not a prime element of $\mathcal{S}$.

So, V must be a singleton set, say $\{\alpha\}$. Then, $E^{\downarrow}(\alpha) = \{r^+(C^{\downarrow}(\alpha))\}$, so τ coincides with $r^+(C^{\downarrow}(\alpha))$. Thus,

$$s \leq_{\mathcal{S}} \mathrm{hd}(\tau) = \sqcup_{\mathcal{S}} \mathrm{hd}(C^{\downarrow}(\alpha))$$

by definition of r^+ (see Fig. 5). Because s is prime (see Definition 10), this implies $s \leq_{\mathcal{S}} \mathrm{hd}(\tau')$ for some $\tau' \in C^{\downarrow}(\alpha)$. Besides, since $V = \{\alpha\}$, $s \leq \alpha \,?\, c$ appears in E, hence also in C. In light of the fact that C itself satisfies Condition 6 of Definition 17, all this implies $C \Vdash^{\mathrm{pre}} c$. However, by definition of E, every pre-solution of E is also a pre-solution of C. So, $E \Vdash^{\mathrm{pre}} c$ also holds. This was our goal.

We have verified that E is weakly closed. Thus, according to Theorem 3, we may throw away some of σ''s constraints, as allowed by polarity, and obtain an equivalent type scheme. One discovers, in fact, that all constraints in $E \setminus D$ are actually superfluous (see [17]). As a result, $\sigma' \approx \mathrm{Can}(\sigma)$. This concludes the proof. □

Appendix A.8 Minimization

We now give an algorithm which separates the variables of a type scheme into a number of equivalence classes, in such a way that all variables in a single class can be merged without affecting the type scheme's denotation. We begin with a couple of auxiliary definitions:

DEFINITION 24. *Let V be a set of type variables. Any equivalence relation $\equiv$ over V is extended to small terms whose variables are in V, as follows:*

$$\tau_1 \equiv \tau_2 \iff \mathrm{hd}(\tau_1) = \mathrm{hd}(\tau_2) \wedge (\forall l \in a(\mathrm{hd}(\tau_1)) \quad \tau_1.l \equiv \tau_2.l)$$

DEFINITION 25. *Let C be a constraint set. For $\alpha \in \mathcal{V}$, define*

$$\mathrm{pred}_C(\alpha) = \{\beta\,;\, \beta \leq \alpha \in C\}$$
$$\mathrm{succ}_C(\alpha) = \{\beta\,;\, \alpha \leq \beta \in C\}$$

Then, we give a series of requirements about equivalence relations, and show that they are sufficient to meet our goal.

DEFINITION 26. *Let $\sigma = \forall C.\, \delta$ be a type scheme in canonical form, made up of small terms, with no bipolar variables, such that $\sigma = \mathrm{GC}(\sigma)$. For any $\alpha \in \mathrm{fv}(\sigma)$, $C^{\downarrow}(\alpha)$ (resp. $C^{\uparrow}(\alpha)$) is a singleton set; by abuse of language, we shall use the same notation to refer to its unique element.*

An equivalence relation $\equiv$ over $\mathrm{fv}(\sigma)$ is compatible with σ *iff $\alpha \equiv \beta$ implies all of the following:*

(1) $\{\alpha, \beta\} \subseteq \mathrm{fv}^+(\sigma)$ *or* $\{\alpha, \beta\} \subseteq \mathrm{fv}^-(\sigma)$;

(2) $\mathrm{pred}_C(\alpha) = \mathrm{pred}_C(\beta)$ *and* $\mathrm{succ}_C(\alpha) = \mathrm{succ}_C(\beta)$;

(3) $C^{\downarrow}(\alpha) \equiv C^{\downarrow}(\beta)$ *and* $C^{\uparrow}(\alpha) \equiv C^{\uparrow}(\beta)$;

(4) $s \leq \alpha \,?\, \gamma_1 \leq \gamma_2 \in C$ *implies* $\exists \delta_1 \equiv \gamma_1 \quad \exists \delta_2 \equiv \gamma_2 \quad s \leq \beta \,?\, \delta_1 \leq \delta_2 \in C$.

DEFINITION 27. *Consider σ as in Definition 26; let $\equiv$ be a partition compatible with σ. The quotient $\sigma/_{\equiv}$ is defined—up to a renaming—as $\pi(\sigma)$, where π is any mapping of $\mathrm{fv}(\sigma)$ into $\mathcal{V}$ such that*

$$\forall \alpha, \beta \in \mathrm{fv}(\sigma) \quad \alpha \equiv \beta \iff \pi(\alpha) = \pi(\beta)$$

THEOREM 5. *Consider σ and $\equiv$ as in Definition 27. Then, $\sigma/_{\equiv} \approx \sigma$.*

PROOF. The assertion $\sigma \preccurlyeq \sigma/_{\equiv}$ clearly holds, because the latter is the image of the former through the substitution π. Reciprocally, let us show that $\sigma/_{\equiv} \preccurlyeq \sigma$. Let ρ be a solution of C. We need to exhibit a solution ρ' of $\pi(C)$ such that $\rho'(\pi(\delta)) \leq \rho(\delta)$.

Consider an equivalence class of $\equiv$. Because of Condition 1 of Definition 26, it must be either a subset of $\mathrm{fv}^-(\sigma)$, or a subset of $\mathrm{fv}^+(\sigma)$. We denote it by V (resp. W) in the former (resp. latter) case. We denote the image of its elements through π by φ_V (resp. φ_W). Define ρ' by

$$\rho'(\varphi_V) = \sqcup\, \rho(V) \qquad \rho'(\varphi_W) = \sqcap\, \rho(W)$$

We remark that for any $\alpha \in \mathrm{fv}^+(\sigma)$, $\rho'(\pi(\alpha)) \leq \rho(\alpha)$ holds; symmetrically, for any $\alpha \in \mathrm{fv}^-(\sigma)$, we have $\rho(\alpha) \leq \rho'(\pi(\alpha))$.

There remains to check that ρ' satisfies $\pi(C)$ and $\pi(\delta) \leq \rho(\delta)$. This is straightforward; as before, we shall deal with the case of conditional constraints explicitly, and refer the reader to [17] for the other cases. Consider a conditional constraint in $\pi(C)$. It has the form $s \leq \pi(\alpha) \,?\, \pi(\beta) \leq \pi(\gamma)$, where $s \leq \alpha \,?\, \beta \leq \gamma \in C$. Note that, necessarily, α and γ belong to $\mathrm{fv}^-(\sigma)$, while β belongs to $\mathrm{fv}^+(\sigma)$. Let V stand for α's equivalence class, i.e. $V = \pi^{-1}(\pi(\alpha))$. Assume $\rho' \vdash s \leq \pi(\alpha)$. This can be written

$$\begin{aligned} s &\leq_{\mathcal{S}} \mathrm{hd}(\rho'(\pi(\alpha))) \\ &= \mathrm{hd}(\sqcup\, \rho(V)) \\ &= \sqcup_{\mathcal{S}}\, \mathrm{hd}(\rho(V)) \end{aligned}$$

Because s is prime (see Definition 10), this implies $s \leq_{\mathcal{S}} \mathrm{hd}(\rho(\alpha'))$, for some $\alpha' \in V$. In other words, $\rho \vdash s \leq \alpha'$ holds. Furthermore, we have $\alpha \equiv \alpha'$;

since $\equiv$ is compatible with σ, Condition 4 yields $s \leq \alpha' \,?\, \beta' \leq \gamma' \in C$, for some $\beta' \equiv \beta$ and $\gamma' \equiv \gamma$. Both facts, combined, yield $\rho \vdash \beta' \leq \gamma'$, because ρ is a solution of C. Next, note that, necessarily, γ' belongs to $\mathrm{fv}^-(\sigma)$, while β' belongs to $\mathrm{fv}^+(\sigma)$. Thus, $\rho \vdash \beta' \leq \gamma'$ implies $\rho' \vdash \pi(\beta) \leq \pi(\gamma)$, by definition of ρ'. We have proved that ρ' satisfies $s \leq \pi(\alpha) \,?\, \pi(\beta) \leq \pi(\gamma)$. □

References

[1] Aiken, Alexander S. and Wimmers, Edward L. 1993. Type Inclusion Constraints and Type Inference. In *Functional Programming & Computer Architecture*. ACM Press, 31–41.

[2] Aiken, Alexander S., Wimmers, Edward L., and Lakshman, T. K. 1994. Soft Typing with Conditional Types. In *Principles of Programming Languages*, 163–173.

[3] Bugliesi, Michele and Crafa, Silvia. 1999. Object Calculi for Dynamic Messages. In *The Sixth International Workshop on Foundations of Object-Oriented Languages, FOOL 6, San Antonio, Texas*.

[4] Cardelli, Luca and Mitchell, John. 1991. Operations on Records. *Mathematical Structures in Computer Science 1*, 3–48.

[5] Fähndrich, Manuel. 1999. Bane: *A Library for Scalable Constraint-Based Program Analysis*. PhD thesis, University of California at Berkeley.

[6] Flanagan, Cormac and Felleisen, Matthias. 1997. Componential Set-Based Analysis. In *Proceedings of the ACM SIGPLAN '97 Conference on Programming Language Design and Implementation*. Las Vegas, Nevada, 235–248.

[7] Gaster, Benedict R. 1998. *Records, variants and qualified types*. PhD thesis, University of Nottingham.

[8] Harper, Robert and Pierce, Benjamin. 1991. A Record Calculus Based on symmetric Concatenation. In *Conference Record of the 18th Annual ACM Symposium on Principles of Programming Languages (POPL '91)*. ACM Press, Orlando, Florida, 131–142.

[9] Heintze, Nevin. 1993. Set Based Analysis of ML Programs. Tech. Report CMU-CS-93-193, Carnegie Mellon University, School of Computer Science.

[10] Jones, Mark P. 1994. *Qualified Types: Theory and Practice*. Cambridge University Press.

[11] Müller, Martin, Niehren, Joachim, and Podelski, Andreas. 2000. Ordering Constraints over Feature Trees. *Constraints, an International Journal 5*, 1–2, 7–42.

[12] Müller, Martin and Nishimura, Susumu. 2000. Type Inference for First-Class Messages with Feature Constraints. *International Journal of Foundations of Computer Science 11*, 1, 29–63.

[13] Nishimura, Susumu. 1998. Static Typing for Dynamic Messages. In *Conference Record of POPL '98: The 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. San Diego, California, 266–278.

[14] Odersky, Martin, Sulzmann, Martin, and Wehr, Martin. 1999. Type Inference with Constrained Types. *Theory and Practice of Object Systems 5*, 1, 35–55.

[15] Ohori, Atsushi. 1995. A Polymorphic Record Calculus and Its Compilation. *ACM Transactions on Programming Languages and Systems 17*, 6 (Nov.), 844–895.

[16] Pottier, François. 1998. Type inference in the presence of subtyping: from theory to practice. Tech. Report 3483, INRIA.

[17] Pottier, François. 2000. Simplifying subtyping constraints: a theory. To appear in *Information & Computation*. URL: `http://pauillac.inria.fr/~fpottier/publis/fpottier-ic-2000.ps.gz`.

[18] Pottier, François. 2000. `Wallace`: an efficient implementation of type inference with subtyping. URL: `http://pauillac.inria.fr/~fpottier/wallace/`.
[19] Rémy, Didier. 1992. Projective ML. In *1992 ACM Conference on Lisp and Functional Programming*. ACM Press, New-York, 66–75.
[20] Rémy, Didier. 1993. Syntactic Theories and the Algebra of Record Terms. Research Report 1869, INRIA.
[21] Rémy, Didier. 1993. Type Inference for Records in a Natural Extension of ML. In *Theoretical Aspects Of Object-Oriented Programming. Types, Semantics and Language Design*, Gunter, Carl A. and Mitchell, John C., Editors. MIT Press.
[22] Rémy, Didier. 1993. Typing Record Concatenation for Free. In *Theoretical Aspects Of Object-Oriented Programming. Types, Semantics and Language Design*, Gunter, Carl A. and Mitchell, John C., Editors. MIT Press.
[23] Rémy, Didier. 1995. A case study of typechecking with constrained types: Typing record concatenation. Presented at the workshop on Advances in Types for Computer Science at the Newton Institute, Cambridge, UK.
[24] Rémy, Didier. 1998. From Classes to Objects via Subtyping. In *Proceedings of the 1998 European Symposium On Programming (ESOP'98)*, Volume 1381 of *Lecture Notes in Computer Science*. Springer-Verlag, 200–220.
[25] Reynolds, John C. 1969. Automatic Computation of Data Set Definitions. In *Information Processing 68*, Volume 1. North-Holland, Amsterdam, 456–461.
[26] Smith, Scott and Wang, Tiejun. 2000. Polyvariant Flow Analysis with Constrained Types. In *Proceedings of the 2000 European Symposium on Programming (ESOP'00)*, Volume 1782 of *Lecture Notes in Computer Science*. Springer Verlag, 382–396.
[27] Sulzmann, Martin. 2000. *A general framework for Hindley/Milner type systems with constraints*. PhD thesis, Yale University, Department of Computer Science.
[28] Sulzmann, Martin, Müller, Martin, and Zenger, Christoph. 1999. Hindley/Milner style type systems in constraint form. Research Report ACRC–99–009, University of South Australia, School of Computer and Information Science.
[29] Trifonov, Valery and Smith, Scott. 1996. Subtyping Constrained Types. In *Proceedings of the Third International Static Analysis Symposium*, Volume 1145 of *LNCS*. SV, 349–365.
[30] Wand, Mitchell. 1991. Type Inference for Record Concatenation and Multiple Inheritance. *Information and Computation 93*, 1 (July), 1–15.