A Constraint-Based Presentation and Generalization of Rows

François Pottier
INRIA

Abstract

We study the combination of possibly conditional non-structural subtyping constraints with rows. We give a new presentation of rows, where row terms disappear; instead, we annotate constraints with filters. *We argue that, in the presence of subtyping, this approach is simpler and more general. In the case where filters are finite or cofinite sets of row labels, we give a constraint solving algorithm whose complexity is $O(n^3 m \log m)$, where n is the size of the constraint and m is the number of row labels that appear in it. We point out that this allows efficient type inference for record concatenation. Furthermore, by varying the nature of filters, we obtain several natural generalizations of rows.*

1 Introduction

Records are an important feature of programming languages, not only because of their ubiquitous use in defining data structures, but also because they form a basic layer on top of which other features, such as objects and modules, may be built. Records are finite associations of values to labels; typical primitive operations on records are access (extracting the contents of a field), update (modifying the contents of an existing field), extension (adding a new field), restriction (removing an existing field), and concatenation (merging two existing records into a new one) [1].

Rows [16, 14] offer syntax to describe infinite families of types, indexed by labels. Rows offer random access to any single component. Furthermore, they offer uniform access to *every* component at once; that is, by imposing a constraint on rows, one effectively imposes a family of constraints, point-wise, on their components. Equality constraints on rows allow assigning accurate types to record access, update, extension, and restriction.

Subtyping is a common way of increasing a type system's expressiveness. In the presence of record types, subtyping is usually *non-structural*—that is, a type and its subtypes may not have the same shape—because it allows forgetting whether certain fields are present in a record. In the presence of rows, this phenomenon occurs at the level of field types. Indeed, typical field types are Abs (the field does not exist in this record), $\mathsf{Pre}\,\tau$ (the field exists and contains a value of type τ) and $\top_{\mathit{field}}$ (the field may or may not exist). The definition of subtyping typically involves axioms such as $\mathsf{Pre}\,\tau \leq \top_{\mathit{field}}$ and $\mathsf{Abs} \leq \top_{\mathit{field}}$, the first of which is non-structural.

From a type-theoretic perspective, concatenation is the most challenging operation on records, because the type of its result cannot be related to the types of its arguments using subtyping constraints only. Consider, for instance, *symmetric* concatenation, which requires its arguments to have disjoint sets of fields. For every field ℓ, the result type at ℓ is $\mathsf{Pre}\,\tau$ if one argument has type $\mathsf{Pre}\,\tau$ at ℓ and the other has type Abs at ℓ; it is Abs if both arguments have type Abs at ℓ; the operation is ill-typed otherwise. The difficulty is not in the quantification over ℓ, which is dealt with by rows, but in the fact that this definition is *by cases*. In [10], I suggested addressing this issue using *conditional constraints*, a concept whose origin can be traced back to Reynolds [15].

Non-structural subtyping constraints can be solved in cubic time [6]. In fact, McAllester's theorem [4] allows establishing that possibly conditional subtyping constraints can be solved in cubic time as well. However, the combination of such constraints with rows, which is implemented in [9], has never been studied from a complexity-theoretic point of view. Taking advantage of this fact, Palsberg and Zhao [7] recently claimed to have developed the first polynomial time type inference algorithm for symmetric record concatenation. Their algorithm runs in time $O(n^5)$, where n is the size of the program. It is not compositional, that is, it must analyze a whole program at once, which makes it rather impractical.

The contribution of this paper is two-fold. First, we give a new presentation of rows, where row *terms* disappear; instead, we annotate *constraints* with *filters*, which, in the simplest case, are finite or cofinite sets of row labels. We argue that this presentation is simpler and more general. Second, we give a solver for this new constraint language, and establish that it runs in time $O(n^3 m \log m)$, where n is the size of the constraint and m is the number of row labels that appear in it. As an application, we show that type inference for symmetric record concatenation can be reduced to solving a constraint whose size is linear in the program's size. Thus, this approach outperforms Palsberg and Zhao's. It is, furthermore, compositional.

Why a new account of rows? In existing presentations, rows are *terms*. In particular, if ℓ is a row label, τ is a type and ρ is a row, then the term $(\ell : \tau; \rho)$ is a row as well. In type systems *without* subtyping, this allows reducing type inference to first-order unification in an equational theory, whose axioms allow e.g. commuting the order of row labels in a row term. This is crucial, because unification is extremely efficient. One disadvantage, however, is that row unification sometimes allocates fresh type variables, making termination and complexity arguments more delicate. Type systems *with* subtyping, on the other hand, replace unification with a more costly constraint solving procedure. The key insight, then, is that there is no longer a point in using row *terms*. One may instead annotate *constraints* with *filters*, which carry information about row labels. This has two consequences. First, it becomes possible to solve constraints without allocating fresh type variables. This simplifies the complexity analysis, and was our original motivation. Second, even though we initially define filters as finite or cofinite sets of row labels, most of our development (in fact, all of it but the complexity analysis) is independent of the nature of filters, as long as they are preserved by Boolean operations and enjoy a decidable emptiness test. Thus, by varying the nature of filters, we obtain several natural generalizations of rows.

The paper is laid out as follows. Section 2 presents the syntax, as well as the meaning, of types and constraints. Section 3 describes a satisfiability test for constraints, based on a closure computation, and assesses its theoretical complexity. Section 4 explains how type inference for record operations, including symmetric concatenation, reduces to constraint solving. Section 5 discusses a few extensions of the framework. Due to space restrictions, most proofs have been moved to the full version of this paper [12]. The complexity analysis does appear in the present paper.

2 Formal Presentation of the System

In order to achieve a good measure of generality, our presentation is parameterized with respect to a *ground signature*, which specifies a type algebra and a subtype ordering; it is defined in Section 2.1. Given a fixed ground signature, Section 2.2 defines the algebra of ground terms and its ordering. The syntax of types and constraints is given in Section 2.3 and their logical interpretation in Section 2.4.

2.1 Assumptions

Definition 1 *Let a* sort ς *be one of* $\{\mathit{Type}, \mathit{Row}\}$.

A ground signature consists of three components: a family of *symbol* lattices, indexed by *kinds*, a set of *parameter labels*, each of which is co- or contra-variant and has fixed sort and kind, and a description of each symbol's *arity* as a set of parameter labels.

Definition 2 *Let $\mathcal{K}$ be a finite set of* kinds. *For every kind $\kappa \in \mathcal{K}$, let $\mathcal{S}_\kappa$ be a finite lattice of* symbols, *with operations $\bot_\kappa$, $\top_\kappa$, $\leq_\kappa$, $\sqcup_\kappa$ and $\sqcap_\kappa$. We write $\mathcal{S}$ for the set of all symbols. Let $\mathcal{P}^+$ and $\mathcal{P}^-$ be disjoint finite sets of* parameter labels. *We write $\mathcal{P}$ for $\mathcal{P}^+ \uplus \mathcal{P}^-$. With every parameter label $p \in \mathcal{P}$, associate a sort and a kind, written* $\mathrm{sort}(p)$ *and* $\mathrm{kind}(p)$, *respectively. With every symbol $s \in \mathcal{S}$, associate a subset of $\mathcal{P}$, called the* arity *of s and written $a(s)$. We require that, for all $\kappa \in \mathcal{K}$ and $s_0, s_1, s_2 \in \mathcal{S}_\kappa$, $s_0 \leq_\kappa s_1 \leq_\kappa s_2$ imply $a(s_0) \cap a(s_2) \subseteq a(s_1)$.*

The last requirement is used in the proof of Theorem 1 to ensure that the orderings $\leq_\kappa$ do give rise to an ordering on ground terms. The information described in Definition 2 forms a *ground signature*. We now assume that some ground signature is fixed.

Example 1 Let $\mathcal{K}$ be $\{\mathit{type}, \mathit{field}\}$. Let $\mathcal{S}_{\mathit{type}}$ be the flat lattice whose elements other than $\bot_{\mathit{type}}$ and $\top_{\mathit{type}}$ are $\rightarrow$ and $\{\cdot\}$. Let $\mathcal{S}_{\mathit{field}}$ be the flat lattice whose elements other than $\bot_{\mathit{field}}$ and $\top_{\mathit{field}}$ are Abs and Pre. Let $\mathcal{P}^- = \{\mathit{dom}\}$ and $\mathcal{P}^+ = \{\mathit{rng}, \mathit{fields}, \mathit{content}\}$. Let $\mathrm{kind}(\mathit{dom}) = \mathrm{kind}(\mathit{rng}) = \mathrm{kind}(\mathit{content}) = \mathit{type}$ and $\mathrm{kind}(\mathit{fields}) = \mathit{field}$. Let $\mathrm{sort}(\mathit{dom}) = \mathrm{sort}(\mathit{rng}) = \mathrm{sort}(\mathit{content}) = \mathit{Type}$ and $\mathrm{sort}(\mathit{fields}) = \mathit{Row}$. Let $a(\rightarrow) = \{\mathit{dom}, \mathit{rng}\}$ and $a(\{\cdot\}) = \{\mathit{fields}\}$ and $a(\mathsf{Pre}) = \{\mathit{content}\}$. Let the arity of all other symbols be $\varnothing$.

This ground signature defines two kinds of types, plain types and field types. The record type constructor $\{\cdot\}$ forms a plain type out of a row of field types. The constructor Pre forms a field type out of a plain type. Abs and Pre are made incomparable, because that is required to assign a sound type to record concatenation [1, 10], but they do have a common supertype $\top_{\mathit{field}}$, so *width subtyping* is present.

If we take a look ahead at the syntax of terms and at the sorting and kinding restrictions, defined in Section 2.3, we find that this ground signature gives rise to the following grammar of terms, where τ ranges over terms of sort *Type* and kind *type*, ρ ranges over terms of sort *Row* and kind *field*, and φ ranges over terms of sort *Type* and kind *field*:

$$\begin{aligned}
\tau &::= \alpha \mid \bot_{\mathit{type}} \mid \top_{\mathit{type}} \mid \tau \rightarrow \tau \mid \{\rho\} \\
\rho &::= \alpha \mid \partial\varphi \\
\varphi &::= \alpha \mid \bot_{\mathit{field}} \mid \top_{\mathit{field}} \mid \mathsf{Pre}\,\tau \mid \mathsf{Abs}
\end{aligned}$$

This concrete style of definition is standard [14, 10]. Parameterizing our development with respect to a ground signature allows us to accommodate a whole family of definitions in this style.

2.2 Logical Model

We now define the universe of ground terms, which forms the model within which types and constraints are interpreted. As in [11, 7], we allow types to be recursive, that is, we define ground terms as infinite trees. We do not restrict our attention to regular trees, because that would not affect constraint satisfiability or entailment. Since the model contains infinite trees, its definition is naturally coinductive. We cannot devote space to establishing the existence of the fixed points mentioned below; a detailed treatment of a similar construction can be found in [3].

Let $\mathcal{L}$ be a denumerable set of *row labels*. Let ℓ range over row labels.

For the sake of brevity, let us write $s(\bar{t}_p)$ for $s(t_p)_{p \in a(s)}$ and $(\bar{t}_\ell)$ for $(t_\ell)_{\ell \in \mathcal{L}}$. The former is a ground type whose head symbol is s and whose sub-terms are the t_p, where p ranges over the arity of s. The latter is a ground row, that is, a family of ground types indexed by row labels. This is made precise by the following definition:

Definition 3 *The family of models $\mathbb{T}^\varsigma_\kappa$ is the greatest solution to the following equations:*

$$\mathbb{T}^{\mathit{Type}}_\kappa = \{s(\bar{t}_p) \;;\; s \in \mathcal{S}_\kappa \wedge \forall p \in a(s) \quad t_p \in \mathbb{T}^{\mathrm{sort}(p)}_{\mathrm{kind}(p)}\}$$
$$\mathbb{T}^{\mathit{Row}}_\kappa = \{(\bar{t}_\ell) \;;\; \forall \ell \in \mathcal{L} \quad t_\ell \in \mathbb{T}^{\mathit{Type}}_\kappa\}$$

An element t of $\mathbb{T}^\varsigma_\kappa$ is a ground term *of sort ς and kind κ.*

Two ground types are comparable if and only if they have comparable head symbols and their sub-terms at every common parameter label p are comparable as well—in the reverse direction, if p is contravariant. Ground rows are compared point-wise and covariantly.

Definition 4 *Let $t \;{}^{p}\!\leq^\varsigma_\kappa\; t'$ stand for $t \leq^\varsigma_\kappa t'$ if p is a member of $\mathcal{P}^+$, and for $t' \leq^\varsigma_\kappa t$ otherwise. The family of relations $\leq^\varsigma_\kappa$ is the greatest solution to the following equivalences:*

$$\begin{aligned} & s(\bar{t}_p) \leq^{\mathit{Type}}_\kappa s'(\bar{t}'_p) \\ \iff & s \leq_\kappa s' \wedge \forall p \in a(s) \cap a(s') \quad t_p \;{}^{p}\!\leq^{\mathrm{sort}(p)}_{\mathrm{kind}(p)} t'_p \\[1ex] & (\bar{t}_\ell) \leq^{\mathit{Row}}_\kappa (\bar{t}'_\ell) \\ \iff & \forall \ell \in \mathcal{L} \quad t_\ell \leq^{\mathit{Type}}_\kappa t'_\ell \end{aligned}$$

This definition may be made more explicit by viewing $\leq^\varsigma_\kappa$ as the intersection of the denumerable family $({}_k\!\leq^\varsigma_\kappa)_{k \geq 0}$, where every ${}_0\!\leq^\varsigma_\kappa$ is the full binary relation and ${}_{k+1}\!\leq^\varsigma_\kappa$ is given by

$$\begin{aligned} & s(\bar{t}_p) \;{}_{k+1}\!\leq^{\mathit{Type}}_\kappa s'(\bar{t}'_p) \\ \iff & s \leq_\kappa s' \wedge \forall p \in a(s) \cap a(s') \quad t_p \;{}^{p}_{k}\!\leq^{\mathrm{sort}(p)}_{\mathrm{kind}(p)} t'_p \\[1ex] & (\bar{t}_\ell) \;{}_{k}\!\leq^{\mathit{Row}}_\kappa (\bar{t}'_\ell) \\ \iff & \forall \ell \in \mathcal{L} \quad t_\ell \;{}_{k}\!\leq^{\mathit{Type}}_\kappa t'_\ell \end{aligned}$$

Theorem 1 *Every $(\mathbb{T}^\varsigma_\kappa, \leq^\varsigma_\kappa)$ forms a lattice.*

In the following, we write $\leq$ instead of $\leq^\varsigma_\kappa$ when ς and κ are irrelevant or can be inferred from the context. If $t \in \mathbb{T}^{\mathit{Type}}_\kappa$ is of the form $s(\bar{t}_p)$, then we let $t.p$ denote t_p. Similarly, if $t \in \mathbb{T}^{\mathit{Row}}_\kappa$ is of the form $(\bar{t}_\ell)$, then we let $t.\ell$ denote t_ℓ. If $s \in \mathcal{S}_\kappa$ and $t \in \mathbb{T}^{\mathit{Type}}_\kappa$ is of the form $s'(\bar{t}_p)$, then we let $s \leq t$ stand for $s \leq_\kappa s'$.

2.3 Syntax of Terms and Constraints

For every sort ς and every kind κ, let $\mathcal{V}^\varsigma_\kappa$ be a distinct denumerable set of *variables*. Let α range over variables. Let a *filter* L be a finite or cofinite subset of $\mathcal{L}$. Filters are machine representable and are preserved by finitary union, intersection, and complement. The syntax of *terms* and *constraints* is as follows:

$$\begin{aligned} \tau &::= \alpha \mid s(\bar{\tau}_p) \mid \partial\tau \\ C &::= \exists\alpha.C \mid C \wedge C \mid \mathbf{true} \mid \mathbf{false} \\ &\quad\mid \tau \leq \tau \\ &\quad\mid L : \tau \leq \tau \\ &\quad\mid L : s \leq \tau \mathrel{?} \tau \leq \tau \end{aligned}$$

Terms are built out of variables, symbols, and the constructor ∂, which defines constant rows [13]. Constraint forms include existential quantification, conjunction, and *elementary* constraints, namely truth, falsity, subtyping constraints, and conditional subtyping constraints.

Figure 1 (resp. 2) defines the judgements $\vdash \tau : \varsigma$ (resp. $\vdash \tau : \kappa$), which means that the term τ has sort ς (resp. kind κ), and $\vdash^{\mathrm{OK}} C$ (resp. $\vdash_{\mathrm{OK}} C$), which means that the constraint C is well-sorted (resp. well-kinded). We restrict our attention to well-sorted and well-kinded terms and constraints.

$$\frac{\alpha \in \mathcal{V}^\varsigma_\kappa}{\vdash \alpha : \varsigma} \qquad \frac{\forall p \in a(s) \quad \vdash \tau_p : \mathrm{sort}(p)}{\vdash s(\bar{\tau}_p) : \mathit{Type}} \qquad \frac{\vdash \tau : \mathit{Type}}{\vdash \partial\tau : \mathit{Row}}$$

$$\frac{\vdash^{\mathrm{OK}} C}{\vdash^{\mathrm{OK}} \exists\alpha.C} \qquad \frac{\vdash^{\mathrm{OK}} C_1 \qquad \vdash^{\mathrm{OK}} C_2}{\vdash^{\mathrm{OK}} C_1 \wedge C_2} \qquad \begin{array}{l} \vdash^{\mathrm{OK}} \mathbf{true} \\ \vdash^{\mathrm{OK}} \mathbf{false} \end{array}$$

$$\frac{\vdash \tau_1, \tau_2 : \mathit{Type}}{\vdash^{\mathrm{OK}} \tau_1 \leq \tau_2} \qquad \frac{\vdash \tau_0, \tau_1, \tau_2 : \mathit{Row}}{\begin{array}{c} \vdash^{\mathrm{OK}} L : \tau_1 \leq \tau_2 \\ \vdash^{\mathrm{OK}} L : s \leq \tau_0 \mathrel{?} \tau_1 \leq \tau_2 \end{array}}$$

Figure 1. Well-sorted terms and constraints

If τ_1 and τ_2 are *types*, then the (standard) subtyping constraint $\tau_1 \leq \tau_2$ may be understood as a requirement for τ_1 to be a subtype of τ_2. Constraints that involve *rows*, however, must be annotated with a filter. That is, if τ_1 and τ_2

$$\frac{\alpha \in \mathcal{V}^{\varsigma}_{\kappa}}{\vdash \alpha : \kappa} \qquad \frac{\begin{array}{c} s \in \mathcal{S}_\kappa \\ \forall p \in a(s) \quad \vdash \tau_p : \mathrm{kind}(p) \end{array}}{\vdash s(\bar{\tau}_p) : \kappa} \qquad \frac{\vdash \tau : \kappa}{\vdash \partial\tau : \kappa}$$

$$\frac{\vdash_{\text{OK}} C}{\vdash_{\text{OK}} \exists\alpha.C} \qquad \frac{\vdash_{\text{OK}} C_1 \qquad \vdash_{\text{OK}} C_2}{\vdash_{\text{OK}} C_1 \wedge C_2} \qquad \begin{array}{c} \vdash_{\text{OK}} \textbf{true} \\ \vdash_{\text{OK}} \textbf{false} \end{array}$$

$$\frac{\vdash \tau_1, \tau_2 : \kappa}{\begin{array}{c} \vdash_{\text{OK}} \tau_1 \leq \tau_2 \\ \vdash_{\text{OK}} L : \tau_1 \leq \tau_2 \end{array}} \qquad \frac{\begin{array}{c} s \text{ is a } \textit{prime} \text{ element of } \mathcal{S}_\kappa \\ \vdash \tau_0, \tau_1, \tau_2 : \kappa \end{array}}{\vdash_{\text{OK}} L : s \leq \tau_0 \mathrel{?} \tau_1 \leq \tau_2}$$

Figure 2. Well-kinded terms and constraints

are *rows*, then the constraint $L : \tau_1 \leq \tau_2$ may be understood as a requirement for $\tau_1.\ell$ to be a subtype of $\tau_2.\ell$ *for every row label* $\ell \in L$. The introduction of filters compensates the omission of the standard row constructor $(\ell : \cdot\,; \cdot)$ by offering a way of *not* treating all row labels uniformly.

Our conditional constraints are exactly those studied in [10], extended with filters. Conditions must be of the form $s \leq \tau$; this restriction allows our solver to delay inspecting a conditional constraint's conclusion until its condition must be satisfied, and is key to the algorithm's polynomial time complexity.

A symbol $s \in \mathcal{S}_\kappa$ is *prime* if and only if, for any finite subset S of $\mathcal{S}_\kappa$, $s \leq_\kappa (\sqcup_\kappa S)$ implies $\exists s' \in S \quad s \leq_\kappa s'$. The requirement that s be prime in Figure 2 slightly simplifies the constraint solver and leads to more aggressive constraint simplification algorithms, not discussed here; see [10]. It could be relaxed if required by some application.

2.4 Logical Interpretation

Terms and constraints are interpreted within the model under a *ground assignment* that gives meaning to their free variables.

Definition 5 *A* ground assignment ϕ *is a total sort- and kind-preserving mapping from the variables into the model, that is, a total mapping from every* $\mathcal{V}^{\varsigma}_{\kappa}$ *into* $\mathbb{T}^{\varsigma}_{\kappa}$.

Definition 6 *Ground assignments are extended to terms by*

$$\phi(s(\bar{\tau}_p)) = s(\overline{\phi(\tau_p)})$$
$$\phi(\partial\tau) = (\overline{\phi(\tau)})$$

The first clause of this definition is standard. The second clause interprets $\partial\tau$ under ϕ as the row that maps every row label to $\phi(\tau)$. If $\vdash \tau : \varsigma$ and $\vdash \tau : \kappa$ hold, then $\phi(\tau)$ is a member of $\mathbb{T}^{\varsigma}_{\kappa}$.

$$\frac{\phi[\alpha \mapsto t] \vdash C}{\phi \vdash \exists\alpha.C} \qquad \frac{\phi \vdash C_1 \qquad \phi \vdash C_2}{\phi \vdash C_1 \wedge C_2} \qquad \phi \vdash \textbf{true}$$

$$\frac{\phi(\tau_1) \leq \phi(\tau_2)}{\phi \vdash \tau_1 \leq \tau_2} \qquad \frac{\forall \ell \in L \quad \phi(\tau_1).\ell \leq \phi(\tau_2).\ell}{\phi \vdash L : \tau_1 \leq \tau_2}$$

$$\frac{\forall \ell \in L \quad s \leq \phi(\tau_0).\ell \Rightarrow \phi(\tau_1).\ell \leq \phi(\tau_2).\ell}{\phi \vdash L : s \leq \tau_0 \mathrel{?} \tau_1 \leq \tau_2}$$

Figure 3. Constraint satisfaction

Definition 7 *Figure 3 defines the judgement* $\phi \vdash C$ *(read:* ϕ satisfies C*). The assertion* $C_1 \Vdash C_2$ *(read:* C_1 entails C_2*) holds if and only if, for every ground assignment* ϕ*,* $\phi \vdash C_1$ *implies* $\phi \vdash C_2$*. Two constraints are* logically equivalent *if and only if they entail each other.*

The interpretation of constraints is straightforward. A constraint on rows is interpreted point-wise within its filter. A conditional constraint is interpreted as an implication.

The reader may notice that the constraints $\tau_1 \leq \tau_2$ and $L : \partial\tau_1 \leq \partial\tau_2$ are logically equivalent when L is nonempty. So, it would be possible to suppress subtyping constraints on types altogether and to encode them as subtyping constraints on rows. It is convenient, however, to keep both; our constraint solver simplifies the latter into the former. In the case of conditional constraints, only constraints on rows were kept. These choices are somewhat arbitrary; other presentations would be possible.

3 Checking Satisfiability of Constraints

3.1 Representing Constraints

Thanks to α-conversion of existentially bound variables and to scope extrusion, which allows rewriting $(\exists\alpha.C_1) \wedge C_2$ to $\exists\alpha.(C_1 \wedge C_2)$ when α does not appear free within C_2, every constraint can be put in prenex form. Given that $\exists\bar{\alpha}.C$ is satisfiable if and only if C is, it is in fact sufficient to develop a satisfiability test for constraints that do not involve existential quantifiers.

We consider the conjunction operator $\wedge$ commutative, associative and idempotent, which allows viewing every constraint as a set of elementary constraints. This is not quite satisfactory, however, because such a representation could be very redundant. Indeed, given fixed rows τ_1 and τ_2, a constraint set might contain several—in fact, exponentially many—elementary constraints of the form $L : \tau_1 \leq \tau_2$. To remedy this problem, we identify constraints up to

$$
\begin{array}{lrcl}
(\textsc{Trans}) & \tau_1 \leq \alpha \wedge \alpha \leq \tau_2 & \rightarrow & \tau_1 \leq \tau_2 \\
(\textsc{Prop}) & s_1(\bar{\tau}^1_p) \leq s_2(\bar{\tau}^2_p) & \rightarrow & \tau^1_p \;{}^{p}\!\!\leq \tau^2_p \\
 & & & \textit{if } s_1 \leq s_2 \textit{ and } p \in a(s_1) \cap a(s_2) \textit{ and } \mathrm{sort}(p) = \textit{Type} \\
(\textsc{Fail}) & s_1(\bar{\tau}^1_p) \leq s_2(\bar{\tau}^2_p) & \rightarrow & \textbf{false} \\
 & & & \textit{if } s_1 \not\leq s_2 \\
 & & & \\
(\textsc{Trans-Row}) & L_1 : \tau_1 \leq \alpha \wedge L_2 : \alpha \leq \tau_2 & \rightarrow & L_1 \cap L_2 : \tau_1 \leq \tau_2 \\
(\textsc{Prop-Type-Row}) & s_1(\bar{\tau}^1_p) \leq s_2(\bar{\tau}^2_p) & \rightarrow & \mathcal{L} : \tau^1_p \;{}^{p}\!\!\leq \tau^2_p \\
 & & & \textit{if } s_1 \leq s_2 \textit{ and } p \in a(s_1) \cap a(s_2) \textit{ and } \mathrm{sort}(p) = \textit{Row} \\
(\textsc{Prop-Row-Type}) & L : \partial\tau_1 \leq \partial\tau_2 & \rightarrow & \tau_1 \leq \tau_2 \\
 & & & \textit{if } L \neq \varnothing \\
 & & & \\
(\textsc{Trans-Cd-Row}) & L_1 : s \leq \alpha \,?\, \tau_1 \leq \tau_2 \wedge L_2 : \partial\tau \leq \alpha & \rightarrow & L_1 \cap L_2 : s \leq \partial\tau \,?\, \tau_1 \leq \tau_2 \\
(\textsc{Trans-Cd-Type}) & L : s \leq \partial\alpha \,?\, \tau_1 \leq \tau_2 \wedge \tau \leq \alpha & \rightarrow & L : s \leq \partial\tau \,?\, \tau_1 \leq \tau_2 \\
(\textsc{Fire}) & L : s_1 \leq \partial s_2(\bar{\tau}_p) \,?\, \tau_1 \leq \tau_2 & \rightarrow & L : \tau_1 \leq \tau_2 \\
 & & & \textit{if } s_1 \leq s_2
\end{array}
$$

Figure 4. Constraint resolution

the following *fusion* laws:

$$
\begin{array}{cl}
 & (L_1 : \tau_1 \leq \tau_2) \wedge (L_2 : \tau_1 \leq \tau_2) \\
\equiv & (L_1 \cup L_2) : \tau_1 \leq \tau_2 \\
 & (L_1 : s \leq \tau_0 \,?\, \tau_1 \leq \tau_2) \wedge (L_2 : s \leq \tau_0 \,?\, \tau_1 \leq \tau_2) \\
\equiv & (L_1 \cup L_2) : s \leq \tau_0 \,?\, \tau_1 \leq \tau_2
\end{array}
$$

It is clear from Figure 3 that these laws preserve the meaning of constraints, which thus remains well-defined. Under these laws, a constraint C may be viewed as a mapping from *edges* to *weights*, where an edge e is

- one of **false** or $\tau_1 \leq \tau_2$, where $\vdash \tau_1, \tau_2 : \textit{Type}$; then, its weight $C(e)$ must be 0 or 1; or
- one of $\tau_1 \leq \tau_2$ or $s \leq \tau_0 \,?\, \tau_1 \leq \tau_2$, where $\vdash \tau_0, \tau_1, \tau_2 : \textit{Row}$; then, its weight $C(e)$ must be a filter L.

We write $C_1 \subseteq C_2$ if and only if some representatives of the constraints C_1 and C_2, viewed as sets of elementary constraints, are within the subset relation. Equivalently, viewing constraints as mappings from edges to weights, $\subseteq$ is the pointwise ordering on mappings, provided weights are ordered by $0 \leq 1$ and set-theoretic inclusion of filters. If c is an elementary constraint, the notation $c \in C$ stands for $\{c\} \subseteq C$; in particular, if c is of the form, say, $L : \tau_1 \leq \tau_2$, then $c \in C$ is equivalent to $L \subseteq C(\tau_1 \leq \tau_2)$, that is, the edge $\tau_1 \leq \tau_2$ has weight L *at least* in C.

3.2 Closing Constraints

We now define a procedure for determining whether a constraint is satisfiable. Following standard practice, it is presented as a closure computation, defined by the rules in Figure 4.

The first three rules deal with subtyping constraints between types; they implement transitive closure, structural decomposition, and failure, respectively. These rules are standard [6, 11].

The next three rules deal with subtyping constraints between rows. (TRANS-ROW) extends the usual transitivity rule by keeping track of filters: quite naturally, if $\tau_1 \leq \alpha$ holds at every row label $\ell \in L_1$ and if $\alpha \leq \tau_2$ holds at every row label $\ell \in L_2$, then transitivity applies at every row label $\ell \in L_1 \cap L_2$. (Recall that constraints between rows are interpreted pointwise.) (PROP-TYPE-ROW) extends (PROP) to the case where the sub-terms at p of the types being decomposed are rows. The full filter $\mathcal{L}$ is used, because these rows should be comparable at every row label. (PROP-ROW-TYPE) decomposes a constraint between two uniform rows. Through these two rules, constraints between types may give rise to constraints between rows, and vice versa. These three rules are a key novelty of this paper. In particular, (TRANS-ROW) is a very natural generalization of (TRANS). None of the rules requires fresh variables to be allocated, contrary to the standard rules for row unification (see e.g. [14, appendix A]).

The last three rules deal with conditional constraints. (TRANS-CD-ROW) and (TRANS-CD-TYPE) are transitivity rules. They cause the lower bounds of the variable on which the condition bears (namely α) to be examined. (FIRE) releases the conclusion into the pool as soon as it becomes evident that the condition is satisfied. These rules are a generalization of the standard closure rules for conditional constraints [10, appendix A] with filters.

Formally speaking, the closure rules define a monotonic function from constraints to constraints. This is made precise by the following definition.

Definition 8 *Given a constraint C, let* $\operatorname{close}(C)$ *be the least (w.r.t. $\subseteq$) constraint such that:*

- $C \subseteq \operatorname{close}(C)$*; and*
- *if $C_1 \subseteq C$ and $C_1 \to C_2$ is an instance of one of the rules in Figure 4, then $C_2 \subseteq \operatorname{close}(C)$.*

If C is well-sorted and well-kinded, then so is $\operatorname{close}(C)$. Furthermore, the closure rules are compatible with fusion equivalence: if $C_1 \equiv C_2$, then $\operatorname{close}(C_1) \equiv \operatorname{close}(C_2)$. This allows us to keep reasoning up to fusion equivalence.

Let us say that a row label ℓ is *apparent* in a set L iff either L is finite and $\ell \in L$, or L is cofinite and $\ell \notin L$. Let us say that ℓ is apparent in a constraint C iff ℓ is apparent in some L that appears in C. Because the row labels apparent in $L_1 \cup L_2$ are those apparent in L_1 or L_2, this definition is compatible with fusion equivalence. The number of row labels apparent in a constraint C is finite.

Theorem 2 *Given a constraint C, the function* close *has a least fixed point containing C, called the* closure *of C.*

The following theorem implies that a constraint and its closure are logically equivalent.

Theorem 3 *C and $\operatorname{close}(C)$ are logically equivalent.*

A constraint C is *closed* if and only if $C \supseteq \operatorname{close}(C)$. The following theorem shows that a closed constraint is satisfiable if and only if it does not contain **false**.

Theorem 4 *If C is closed and does not contain* **false***, then C is satisfiable.*

Together, these results show that constraint satisfiability is decidable: a constraint is satisfiable if and only if its closure (which may be computed in a finite number of steps) does not contain **false**.

Theorem 3 shows that a constraint and its closure are *logically* equivalent, whereas proving that they are *satisfaction* equivalent (i.e. one is satisfiable if and only if the other is satisfiable) would suffice to establish decidability. The latter approach is followed by Palsberg and Zhao [7, Lemma 5.4]. It is, however, much weaker, because it leads to a *non-compositional* type inference system. Indeed, constraint generation and constraint resolution can be interleaved only if the latter preserves the meaning of constraints, that is, the set of their solutions. If it doesn't, then resolution must be performed *after* generation is complete, that is, after the whole program has been analyzed. A compositional approach, where constraint generation may be freely interleaved with resolution and simplification, is much more desirable, for efficiency and modularity reasons.

Palsberg and Zhao's notion of "esat-closure" eagerly simplifies constraints of the form $V \oplus V' \leq [\ell : U]^{\to}$ into either $V \leq [\ell : U]^{\to}$ or $V' \leq [\ell : U]^{\to}$. When the context does not allow determining which of these two choices is right, an arbitrary decision is made. However, in such a situation, there is no principal choice—that is, each of the two possible choices rules out some solutions—so the solution set is not preserved. Our approach, on the other hand, is to *delay* this choice until it can be safely resolved. We achieve this via conditional constraints; see the type scheme ascribed to symmetric record concatenation in Section 4.

3.3 Description of the Algorithm

We now describe an algorithm that, given a constraint C_0, fails if it is unsatisfiable, and computes its closure otherwise. The algorithm uses an unordered *queue* Q of pending elementary constraints. It also maintains a *current constraint* C, represented as an association table from edges to weights, as suggested in Section 3.1. C is initially **true**, that is, every edge has weight 0 or $\varnothing$, as appropriate.

```
if $c$ is $e$ and $C(e) = 0$
then $C(e) \leftarrow 1$
     $Q \leftarrow Q, c$

else if $c$ is $L : e$
then let $L_1 = L \setminus C(e)$
     if $L_1 \neq \varnothing$
     then $C(e) \leftarrow L \cup C(e)$
          $Q \leftarrow Q, L_1 : e$
```

Figure 5. Algorithm $\textsc{Insert}(c)$

Pseudocode for the procedure $\textsc{Insert}(c)$, which inserts an elementary constraint c into C and Q, is given in Figure 5. In "**if** c is *pattern* **then** *command*", the meta-variables that occur within the pattern are bound in the command. If c is a constraint on types, INSERT adds it to C and Q, unless it has already been processed before. If, on the other hand, c is a constraint on rows, of the form $L : e$, then it is first rewritten to $L_1 : e$, where L_1 is $L \setminus C(e)$, that is, L minus the edge's current weight. The rewritten constraint is then added to C and Q if it is nontrivial. The point of rewriting the constraint in such a way is to avoid any duplication of effort; the complexity analysis (Section 3.5) relies on this optimization.

Using INSERT, we define the procedure $\textsc{Process}(c)$, given in Figure 6, whose purpose is to enumerate and in-

```
if $c$ is $\tau_1 \leq \alpha$                                         (1)
then for each $\tau_2$ such that $C(\alpha \leq \tau_2) = 1$
  do INSERT$(\tau_1 \leq \tau_2)$

if $c$ is $\alpha \leq \tau_2$                                         (2)
then for each $\tau_1$ such that $C(\tau_1 \leq \alpha) = 1$
  do INSERT$(\tau_1 \leq \tau_2)$

if $c$ is $L_1 : \tau_1 \leq \alpha$                                   (3)
then for each $\tau_2$
  do let $L_2 = C(\alpha \leq \tau_2)$
     INSERT$(L_1 \cap L_2 : \tau_1 \leq \tau_2)$

if $c$ is $L_2 : \alpha \leq \tau_2$                                   (4)
then for each $\tau_1$
  do let $L_1 = C(\tau_1 \leq \alpha)$
     INSERT$(L_1 \cap L_2 : \tau_1 \leq \tau_2)$

if $c$ is $s_1(\bar{\tau}_p^1) \leq s_2(\bar{\tau}_p^2)$               (5)
then if $s_1 \not\leq s_2$ then fail
     else for each $p \in a(s_1) \cap a(s_2)$
       do if sort$(p) = \mathit{Type}$
          then INSERT$(\tau_p^1 \overset{p}{\leq} \tau_p^2)$
          else INSERT$(\mathcal{L} : \tau_p^1 \overset{p}{\leq} \tau_p^2)$

if $c$ is $L : \partial\tau_1 \leq \partial\tau_2$                     (6)
then INSERT$(\tau_1 \leq \tau_2)$

if $c$ is $L_1 : s \leq \alpha \mathbin{?} \tau_1 \leq \tau_2$         (7)
then for each $\tau$
  do let $L_2 = C(\partial\tau \leq \alpha)$
     INSERT$(L_1 \cap L_2 : s \leq \partial\tau \mathbin{?} \tau_1 \leq \tau_2)$

if $c$ is $L_2 : \partial\tau \leq \alpha$                             (8)
then for each $s, \tau_1, \tau_2$
  do let $L_1 = C(s \leq \alpha \mathbin{?} \tau_1 \leq \tau_2)$
     INSERT$(L_1 \cap L_2 : s \leq \partial\tau \mathbin{?} \tau_1 \leq \tau_2)$

if $c$ is $L : s \leq \partial\alpha \mathbin{?} \tau_1 \leq \tau_2$    (9)
then for each $\tau$
  do let $L = C(\tau \leq \alpha)$
     INSERT$(L : s \leq \partial\tau \mathbin{?} \tau_1 \leq \tau_2)$

if $c$ is $\tau \leq \alpha$                                           (10)
then for each $s, \tau_1, \tau_2$
  do let $L = C(s \leq \partial\alpha \mathbin{?} \tau_1 \leq \tau_2)$
     INSERT$(L : s \leq \partial\tau \mathbin{?} \tau_1 \leq \tau_2)$

if $c$ is $L : s_1 \leq \partial s_2(\bar{\tau}_p) \mathbin{?} \tau_1 \leq \tau_2$ and $s_1 \leq s_2$   (11)
then INSERT$(L : \tau_1 \leq \tau_2)$
```

Figure 6. Algorithm PROCESS(c)

sert the immediate consequences of a newly discovered constraint c. The procedure explores all new manners in which a closure rule may become applicable due to the addition of c. It is mostly a paraphrase of Figure 4.

```
$C \leftarrow$ true
$Q \leftarrow$ empty
for each $c \in C_0$
  do if $c$ is false then fail else INSERT$(c)$
while $Q$ is nonempty
  do extract $c$ out of $Q$
     PROCESS$(c)$
```

Figure 7. Algorithm CLOSURE(C_0)

Figure 7 defines the main procedure. Starting with an empty current constraint C and an empty queue Q, the function uses INSERT to schedule every element of C_0 for consideration. Then, as long as some constraint remains in the queue, it is processed, causing its consequences to be inserted into the queue, unless they were known already. The algorithm stops when INSERT fails or when the queue becomes empty.

3.4 Properties of the Algorithm

Theorem 5 (Soundness) *The main **while** loop has invariant property* $C_0 \subseteq C \subseteq \text{close}^\infty(C_0)$.

Theorem 6 (Completeness) *The main **while** loop has invariant property* $\text{close}(C \setminus Q) \subseteq C$.

When the algorithm succeeds, Q is empty, so, by Theorem 6, $\text{close}(C) \subseteq C$ holds—that is, C is closed. Together with Theorem 5, this shows that C is then the closure of the initial constraint C_0. Conversely, when the algorithm fails, $\text{close}(C)$ contains **false**, so, by Theorem 5, the closure of C_0 contains **false** as well. Thus, the algorithm succeeds if and only if the closure of C_0 does not contain **false**, that is—according to Theorems 3 and 4—if and only if C_0 is satisfiable.

3.5 Complexity Analysis

We assess the algorithm's time and space complexity in terms of two parameters, namely n, the size of C_0, and m, the number of row labels apparent in C_0. Note that, for n to be properly defined, we must view C_0 as a multiset of elementary constraints; we do *not* apply the fusion laws to it. The parameter m is bounded by n, but is typically much smaller in practice; hence, it is worth distinguishing between them.

By examination of the closure rules (Figure 4), one finds that the number of edges whose weight may become non-null at some point is only $O(n^2)$. The crucial point is that even though new conditional constraints of the form $L : s \leq \tau_0 \mathbin{?} \tau_1 \leq \tau_2$ may be created, no new triples (s, τ_1, τ_2) appear, so the number of distinct such triples remains $O(n)$. Given that the number of choices for τ_0 is also $O(n)$, this leads us to a quadratic number of edges of the form $s \leq \tau_0 \mathbin{?} \tau_1 \leq \tau_2$. Subtyping edges are clearly at most quadratic in number as well.

The association table from edges to weights is implemented as an array, whose space usage is thus $O(n^2)$. Access and update operations have $O(1)$ time complexity. We conduct our analysis based on these assumptions, although it would be desirable, in practice, to use a more space-efficient data structure, possibly at the expense of an additional $O(\log n)$ time penalty. Insertion and removal in Q are performed in constant time.

We assume that $\mathcal{L}$ is totally ordered, so filters may be represented as a pair of a Boolean flag and a balanced binary tree whose keys are row labels. A filter may be checked for emptiness in constant time. Furthermore, membership $\ell \in L$ and insertion $\{\ell\} \cup L$ have $O(\log m)$ time complexity. As a result, union $L_1 \cup L_2$, intersection $L_1 \cap L_2$ and difference $L_1 \setminus L_2$ may be computed in time $O(1 + |L_1| \log m)$, where $|L|$ stands for the cardinal of L if L is finite, and for that of its complement if it is cofinite.

If c is a constraint on types, then INSERT(c) runs in time $O(1)$. If, on the other hand, c is a constraint on rows, of the form $L : e$, then it runs in time $O(1 + |L| \log m)$.

A constraint of the form e can appear at most once in Q, because INSERT checks whether $C(e)$ is 0 before appending e to Q. Furthermore, by definition of INSERT, if constraints of the form $L_1 : e, L_2 : e, \ldots, L_k : e$ successively appear in the queue, then the filters $(L_i)_{1 \leq i \leq k}$ must be non-empty and pairwise disjoint. This implies, in particular, that at most one L_i is cofinite. All others must be finite, and because they are pairwise disjoint, there are at most m of them. As a result, the sum $\Sigma_{1 \leq i \leq k} O(1 + |L_i| \log m)$ is bounded by $O(m \log m)$. This remark will be used shortly.

Let us now examine block 1 in Figure 6, which implements half of (TRANS). The number of distinct constraints of the form $\tau_1 \leq \alpha$ is $O(n^2)$, so this block is executed at most $O(n^2)$ times. There are $O(n)$ distinct terms τ_2. Lastly, looking up C and invoking INSERT takes time $O(1)$ in this case, so the total cost for this block is $O(n^3)$. Block 2, which is symmetric and implements the other half of (TRANS), has the same cost. Indeed, $O(n^3)$ is the cost usually associated with transitive closure.

Let us now examine block 3, which implements half of (TRANS-ROW). What is the cost of its inner **do** statement? If L_1 is finite, then $|L_1 \cap L_2| \leq |L_1|$ holds, so the cost of the call to INSERT is $O(1 + |L_1| \log m)$, and the cost of the **do** statement is $O(1 + |L_1| \log m)$ as well. If, on the other hand, L_1 is cofinite, then we may bound the statement's cost by $O(m \log m)$. According to the remark above, if we now let L_1 vary while τ_1, τ_2 and α remain fixed, the statement's cumulative cost is only $O(m \log m)$. Lastly, letting τ_1, τ_2 and α vary, we find that the total cost for block 3 is $O(n^3 m \log m)$. Block 4 is symmetric.

We now come to block 5, which implements (PROP) and (PROP-TYPE-ROW). Assuming that sort(p) may be looked up in constant time, the cost of the inner **do** statement is $O(1)$. The cardinality of the set $a(s_1) \cap a(s_2)$ is bounded by a constant, because Definition 2 specifies that the set of symbols in the (fixed) ground signature is finite. Assuming that this set can be computed in constant time, the cost of the **for** loop is still $O(1)$. Let us assume that symbols can be compared in constant time. Because the number of distinct constraints of the form $s_1(\bar{\tau}_p^1) \leq s_2(\bar{\tau}_p^2)$ is $O(n^2)$, the total cost for block 5 is $O(n^2)$ as well.

Block 6, which implements (PROP-ROW-TYPE), has cost $O(mn^2)$. This again follows from the fact that, when L varies, its successive values are, on the one hand, at most one cofinite filter, and on the other hand, a sequence of pairwise disjoint, finite filters; so L may only assume $O(m)$ successive values.

The analysis for blocks 7, 8, 9 and 10, which implement (TRANS-CD-ROW) and (TRANS-CD-TYPE), is similar to that for block 3: each of these blocks has total cost $O(n^3 m \log m)$. The analysis for block 11 is analogous to that for block 6: it has total cost $O(n^2 m \log m)$. To obtain these bounds, one must recall that the number of triples (s, τ_1, τ_2) in conditional constraints is bounded by $O(n)$.

So far, we have measured the cost of executing each block, when the **if** statement that governs it is taken. There remains to assess the cumulative cost of going through these **if** statements, which is non-null even when the branch is not taken. It is clear that running one constraint c through all **if** statements costs $O(1)$ time. Because the number of constraints that can be queued is $O(mn^2)$, the cumulative cost of going through these statements is $O(mn^2)$ as well. To sum up, we have established

Theorem 7 *The algorithm runs in time $O(n^3 m \log m)$.*

4 Typing Record Operations

In this section, we use the ground signature defined in Example 1. We instantiate the parametric constraint-based type inference framework HM(X) [5] with our constraint language. Then, we extend it with primitive operations on records, to which we assign *constrained type schemes*, as follows. Let α range over variables of sort *Type* and kind *type*. Let φ range over variables of sort *Row* and kind *field*. The *empty record* has type $\{\partial \mathsf{Abs}\}$. *Access* $(.\ell)$ has type

$\forall \alpha\varphi[\{\ell\} : \varphi \leq \partial(\mathsf{Pre}\,\alpha)].\{\varphi\} \rightarrow \alpha$. Notice how a singleton filter is used to extract information about field ℓ alone. *Non-strict extension* $(+\ell)$, which subsumes extension and update, has type

$$\begin{array}{l}\forall \alpha\varphi_1\varphi_2[\{\ell\} : \partial(\mathsf{Pre}\,\alpha) \leq \varphi_2 \wedge (\mathcal{L} \setminus \{\ell\}) : \varphi_1 \leq \varphi_2]. \\ \qquad\qquad \{\varphi_1\} \rightarrow \alpha \rightarrow \{\varphi_2\}\end{array}$$

Here, the first constraint uses a singleton filter to indicate that the field ℓ is present with type α in the new record, while the second constraint uses a cosingleton filter to indicate that all fields other than ℓ have the same status as in the original record. *Symmetric concatenation* $(+)$ has type

$$\begin{array}{l}\forall \varphi_1\varphi_2\varphi_3[\ \mathcal{L} : \mathsf{Abs} \leq \varphi_1\,?\,\varphi_2 \leq \varphi_3 \\ \qquad\quad \wedge\ \mathcal{L} : \mathsf{Abs} \leq \varphi_2\,?\,\varphi_1 \leq \varphi_3 \\ \qquad\quad \wedge\ \mathcal{L} : \mathsf{Pre} \leq \varphi_1\,?\,\varphi_2 \leq \mathsf{Abs}]. \\ \qquad\qquad \{\varphi_1\} \rightarrow \{\varphi_2\} \rightarrow \{\varphi_3\}\end{array}$$

The first two constraints encode the semantics of record concatenation: if a field is missing from one argument, it is read from the other. The third constraint prevents a field from being present in both arguments at once. All constraints carry the full filter $\mathcal{L}$, because concatenation behaves uniformly with respect to all field labels. Assigning an accurate type to *asymmetric* record concatenation requires a slight extension of our framework; see Section 5.3.

It is straightforward to give a semantics to these operations and to prove that these types are correct with respect to it. We omit this step by lack of space.

Our approach presents no novelty with respect to previous work [14, 10], except in the use of our new constraint language. Because the types above have bounded size, the size of a constraint associated with a monomorphic program is linear in the size of the program. The row labels apparent in the constraint are those apparent in the program text. Thus, we are able to infer types in time $O(n^3 m \log m)$, where n is the program size and m is the number of record labels in the program. The addition of **let**-polymorphism invalidates this result in principle but is known to have little practical impact on performance, provided effective constraint simplification algorithms are available.

5 Extensions

Sections 3.5 and 4 are based on the assumption that a filter is encoded as a pair of a Boolean flag and a finite set of row labels. However, the definition and proof of the constraint solving algorithm only require filters to be preserved by Boolean operations and to enjoy a decidable emptiness test. Sections 5.1 and 5.2 exploit this remark and suggest richer filter languages. Section 5.3 suggests another extension, based on a more liberal sorting discipline.

5.1 Hierarchizing Row Labels

Imagine $\mathcal{L}$ is *finite* and equipped with a partial order $\preceq$. Imagine the programmer specifies $(\mathcal{L}, \preceq)$, in a modular manner, by declaring new elements and edges at the beginning of every program module. Thus, $(\mathcal{L}, \preceq)$ forms a finite but *extensible* hierarchy. Let the *cone* generated by ℓ be the set of ℓ's lower bounds with respect to $\preceq$. Let *filter expressions* λ consist of singletons, cones (represented by their generator), and Boolean combinations thereof:

$$\lambda ::= \{\ell\} \mid \downarrow\ell \mid \lambda \cup \lambda \mid \neg\lambda$$

Every filter expression λ may be interpreted as a filter in the obvious manner. Replace filters with filter expressions in the definition of constraints: the meaning of constraints is now parameterized by $(\mathcal{L}, \preceq)$.

If $(\mathcal{L}, \preceq)$ was known entirely when solving constraints, there would be little point in this extension, because $\downarrow\ell$ would be syntactic sugar for a union of singletons. (It would be more concise, though, which may be interesting in practice.) However, the row label hierarchy can be specified modularly, which means that, when a program module M is being examined in isolation, $(\mathcal{L}, \preceq)$ is only known to be *some* extension of the hierarchy $(\mathcal{L}_M, \preceq_M)$ declared in M. Thus, one must ensure that the constraints associated with M are satisfiable under *every* extension of $(\mathcal{L}_M, \preceq_M)$. To this end, one need not modify our algorithm; it suffices to plug in a test for emptiness that determines whether a given filter expression is empty under every extension of the current hierarchy. For such a test to be decidable, one must likely restrict the way new elements and edges are declared; we leave this issue for future work.

How may such an extension be useful? Consider a type system that gathers information about every object's actual class in a Java-like programming language. Let $(\mathcal{L}, \preceq)$ reflect the class and interface hierarchy. Then, the **new** operator may be described using a singleton filter, because it creates objects of a known class. The **instanceof** operator may be described using a cone filter, because it selects objects whose actual class is *any* subclass of a known class. Using this mechanism, uncaught exception analyses based on rows (see e.g. [8]) could be extended to handle languages where exceptions form a hierarchy, such as Java, while preserving their ability to analyze program modules in isolation. Naturally, this extension is speculative; its details remain to be worked out.

5.2 Structuring Individuals

Let us say that a filter is a set of *individuals*. So far, we have considered individuals to be row labels, that is, atoms. What if instead individuals were structured entities?

For instance, let individuals be k-tuples of row labels, for some fixed positive integer k. Let filters be finite unions of Cartesian products of the form $\mathcal{L}^i \times L \times \mathcal{L}^{k-1-i}$, where $0 \leq i < k$ and L is a finite or cofinite subset of $\mathcal{L}$, as before. Then, filters are preserved by intersection and complement and enjoy a decidable emptiness test. Without modifying the constraint solving procedure, we obtain a constraint language that offers k-*dimensional rows*, a concept previously studied by Rémy. Rémy suggested using a 2-dimensional row (that is, a 2-dimensional array of types), indexed on the one hand by method names and the other hand by class names, to keep track of the type of every method at every class in an object-oriented language. Using filters of the form $L \times \mathcal{L}$ or $\mathcal{L} \times L$, it is possible to update an entire line or an entire column of the array at once, which allows assigning types to the operations that create new methods or new classes.

As another instance, let individuals be *trees*; let filters be some class of *tree automata* that is preserved by Boolean operations and enjoys a decidable emptiness test. We obtain a new flavor of rows, which allows encoding certain functions from trees into types. This may open an avenue towards full type inference for programming languages dedicated to manipulating XML structures, such as XDuce, that have *regular expression types* [2]. Currently, XDuce only has some (local) type inference.

Again, an in-depth exploration of these extensions must be left for future research. Still, we believe it is pleasing and interesting that the algorithm should exhibit such generality.

5.3 More Row Terms

Figure 1 requires types of the form $s(\bar{\tau}_p)$ to have sort *Type*, which means that row terms must be variables or of the form $\partial\tau$. This is indeed sufficient for many applications. However, it is sometimes desirable to give meaning to row terms of the form $s(\bar{\tau}_p)$. Rémy [13] pointed out that every type constructor s of sort $\mathit{Type}^k \Rightarrow \mathit{Type}$ may be viewed, at the same time, as a row constructor of sort $\mathit{Row}^k \Rightarrow \mathit{Row}$. In the logical model, the effect of s on rows is then defined as the point-wise extension of its effect on types. This extension yields extra expressiveness, which is useful to infer types for language features such as *asymmetric* record concatenation and so-called *first-class messages* [10]. We anticipate no difficulty in adapting our results to this more liberal sorting discipline. The algorithm must then be extended with more closure rules; we have avoided this complication for the sake of clarity.

References

[1] Luca Cardelli and John Mitchell. Operations on records. In Carl A. Gunter and John C. Mitchell, editors, *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design*. MIT Press, 1994.

[2] Haruo Hosoya and Benjamin C. Pierce. Regular expression pattern matching for XML. *Journal of Functional Programming*, 2002. To appear. URL: `http://www.kurims.kyoto-u.ac.jp/~hahosoya/papers/tapat-full.ps`.

[3] Dexter Kozen, Jens Palsberg, and Michael I. Schwartzbach. Efficient recursive subtyping. *Mathematical Structures in Computer Science*, 5(1):113–125, 1995.

[4] David McAllester. On the complexity analysis of static analyses. *Journal of the ACM*, 49(4):512–537, July 2002.

[5] Martin Odersky, Martin Sulzmann, and Martin Wehr. Type inference with constrained types. *Theory and Practice of Object Systems*, 5(1):35–55, 1999.

[6] Jens Palsberg, Mitchell Wand, and Patrick M. O'Keefe. Type inference with non-structural subtyping. *Formal Aspects of Computing*, 9:49–67, 1997.

[7] Jens Palsberg and Tian Zhao. Efficient type inference for record concatenation and subtyping. In *IEEE Symposium on Logic in Computer Science (LICS)*, pages 125–136, July 2002.

[8] François Pessaux and Xavier Leroy. Type-based analysis of uncaught exceptions. *ACM Transactions on Programming Languages and Systems*, 22(2):340–377, 2000.

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

[10] François Pottier. A versatile constraint-based type inference system. *Nordic Journal of Computing*, 7(4):312–347, November 2000.

[11] François Pottier. Simplifying subtyping constraints: a theory. *Information and Computation*, 170(2):153–183, November 2001.

[12] François Pottier. A constraint-based presentation and generalization of rows. URL: `http://pauillac.inria.fr/~fpottier/publis/fpottier-lics03-long.ps.gz`, April 2003.

[13] Didier Rémy. Projective ML. In *ACM Symposium on Lisp and Functional Programming (LFP)*, pages 66–75, 1992.

[14] Didier Rémy. Type inference for records in a natural extension of ML. In Carl A. Gunter and John C. Mitchell, editors, *Theoretical Aspects Of Object-Oriented Programming. Types, Semantics and Language Design*. MIT Press, 1994.

[15] John C. Reynolds. Automatic computation of data set definitions. In A. J. H. Morrell, editor, *Information Processing 68*, volume 1, pages 456–461. North Holland, 1969.

[16] Mitchell Wand. Type inference for objects with instance variables and inheritance. In Carl A. Gunter and John C. Mitchell, editors, *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design*, pages 97–120. MIT Press, 1994.