A 3-Part Type Inference Engine

François Pottier

INRIA Rocquencourt, BP 105, 78153 Le Chesnay Cedex, France.
Francois.Pottier@inria.fr

Abstract. Extending a *subtyping-constraint-based type inference* framework with *conditional constraints* and *rows* yields a powerful type inference engine. We illustrate this claim by proposing solutions to three delicate type inference problems: "accurate" pattern matchings, record concatenation, and "dynamic" messages. Until now, known solutions required significantly different techniques; our theoretical contribution is in using only a single (and simple) set of tools. On the practical side, this allows all three problems to benefit from a common set of constraint simplification techniques, leading to efficient solutions.

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.

This paper presents a common solution to several seemingly unrelated type inference problems, by unifying in a single type inference system several previously proposed techniques, namely: a simple framework for *subtyping-constraint-based type inference* [15], *conditional constraints* inspired by Aiken, Wimmers and Lakshman [2], and *rows* à la Rémy [18].

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, 15], Trifonov and Smith [22], as well as an abstract framework by Odersky, Sulzmann and Wehr [12]. Related systems include set-based analysis [8, 6] and type inference systems based on feature constraints [9, 10].

Conditional Constraints

In most constraint-based systems, the expression `if` e_0 `then` e_1 `else` e_2 may, at best, be 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 "e_1's (resp. e_2's) value may become the whole expression's value", regardless of the test's outcome. A more precise description – "if e_0 may evaluate to `true` (resp. `false`), then e_1's (resp e_2's) value may become the whole expression's value" – may be given using natural *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 the program's *control* flow – that is, mirroring the way evaluation is affected by a test's outcome, at the level of types.

Conditional set expressions were introduced by Reynolds [21] as a means of solving set constraints involving strict type constructors and destructors. Heintze [8] 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 *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 a programming language allows manipulating structured data, then its type system shall likely require labelled sums, where labels are names of data constructors.

Rémy [18] elegantly deals 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 shall not syntactically distinguish regular 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 informal explanations are made precise via an equational theory:

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

For more details, we refer the reader to [18].

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 what happens at the level of a single label – written using types – can also be read as a description of the whole 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 "dynamic" 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, 15]. 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. Theoretical aspects of constraint solving are discussed in [15, 17]. Furthermore, a robust prototype implementation is publicly available [14]. 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 brief technical overview of the type system, focusing on the notion of constrained type scheme, which should be enough to gain an understanding of the paper. Sections 3, 4, and 5 discuss type inference for "accurate" pattern matchings, record concatenation, and "dynamic" messages, respectively, within our system. Section 6 sums up our contribution, then briefly discusses future research topics. Appendix A gives some more technical details, including the system's type inference rules. Lastly, Appendix B gives several examples, which show what inferred types look like in practice.

2 System's Overview

The programming language considered throughout the paper is a call-by-value λ-calculus with `let`-polymorphism, i.e. essentially core ML.

$$e ::= x, y, \ldots \mid \lambda x.e \mid (e\,e) \mid X, Y, \ldots \mid \texttt{let } X = e \texttt{ in } e$$

The type algebra needed to deal with such a core language is simple. The set of *ground terms* contains all regular trees built over $\bot$, $\top$ (with arity 0) and $\rightarrow$ (with arity 2). It is equipped with a straightforward *subtyping* relationship [15], denoted $\leq$, which makes it a lattice. It is the logical model in which subtyping constraints are interpreted.

Symbols, *type variables*, *types* and *constraints* are defined as follows:

$$\begin{array}{ll} s ::= \bot \mid \rightarrow \mid \top & v ::= \alpha, \beta, \ldots \\ \tau ::= v \mid \bot \mid \tau \rightarrow \tau \mid \top & c ::= \tau \leq \tau \\ & \quad\;\; \mid \;\; s \leq v \,?\, \tau \leq \tau \end{array}$$

A ground substitution ϕ is a map from type variables to ground terms. A constraint of the form $\tau_1 \leq \tau_2$, which reads "τ_1 must be a subtype of τ_2", is satisfied by ϕ if and only if $\phi(\tau_1) \leq \phi(\tau_2)$. A constraint of the form $s \leq \alpha \,?\, \tau_1 \leq \tau_2$, which reads "if α exceeds s, then τ_1 must be a subtype of τ_2", is satisfied by ϕ if and only if $s \leq_{\mathcal{S}} \text{head}(\phi(\alpha))$ implies $\phi(\tau_1) \leq \phi(\tau_2)$, where head maps a ground term to its head constructor, and $\leq_{\mathcal{S}}$ is the expected ordering over symbols. A constraint set C is satisfied by ϕ if and only if all of its elements are.

A *type scheme* is of the form

$$\sigma ::= \forall C.\, \tau$$

where τ is a type and C is a constraint set, which restricts the set of σ's ground instances. Indeed, the latter, which we call σ's *denotation*, is defined as

$$\{\tau' \,;\, \exists \phi \quad \phi \text{ satisfies } C \wedge \phi(\tau) \leq \tau'\}$$

Because *all* of a type scheme's variables are universally quantified, we will usually omit the $\forall$ quantifier and simply write "τ where C".

Of course, the type algebra given above is very much simplified. In general, the system allows defining more type constructors, separating symbols (and terms) into *kinds*, and making use of rows. (A full definition – without rows – appears in [17].) However, for presentation's sake, we will introduce these features only step by step.

The core programming language described above is also limited. To extend it, we will define new primitive operations, equipped with an operational semantics and an appropriate *type scheme*. However, no extension to the *type system* – e.g. in the form of new typing rules – will be made. This explains why we do not further describe the system itself. (Some details are given in Appendix A.) Really, all this paper is about is *writing expressive constrained type schemes*.

3 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 [8, 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.

3.1 The Basic Case

We assume the language allows building and accessing tagged values.

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

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 "regular types," written τ, and "field types," written ϕ.

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

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:

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

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 $\mathtt{Pre}^{-1}$ is described as follows:

$$\begin{aligned} &\mathtt{Pre}^{-1} : (\alpha \to \beta) \to [\,\varphi\,] \to \gamma \\ &\text{where} \quad \varphi \leq \mathtt{Pre}\,\alpha \\ &\qquad\qquad \mathtt{Pre} \leq \varphi\,?\,\beta \leq \gamma \end{aligned}$$

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

3.2 The General Case

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

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

(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:

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

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 3.1, 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.

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

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

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

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 shall 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: k-ary *case* constructs may be written using k nested destructor applications and `close`, and receive the desired, accurate type. Thus, no specific language construct or type inference rule is needed to deal with them.

4 Record Concatenation

Static typing for record operations is a widely studied problem [4, 13]. 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 [7]. An inference algorithm for asymmetric concatenation was suggested by Wand [23]. He uses *disjunctions* of constraints, however, which gives his system exponential complexity. Rémy [19] 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 [20], Rémy suggests a direct, constraint-based algorithm, which involves a special form of constraints. Our approach is inspired from this work, but re-formulated in terms of conditional constraints, thus showing that no ad hoc construct is 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.

4.1 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.

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

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

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

The semantics of asymmetric record concatenation is given as follows:

$$\begin{aligned} v_1\,@\,\mathtt{Abs} &\quad \text{reduces to} \quad v_1 \\ v_1\,@\,(\mathtt{Pre}\,v_2) &\quad \text{reduces to} \quad \mathtt{Pre}\,v_2 \end{aligned}$$

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

$$\begin{aligned} \mathtt{Abs}\,@@\,v_2 &\quad \text{reduces to} \quad v_2 \\ v_1\,@@\,\mathtt{Abs} &\quad \text{reduces to} \quad v_1 \end{aligned}$$

(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 3.1. We introduce a (unary) record type constructor, as well as a distinction between regular types and field types:

$$\begin{aligned} \tau &::= \alpha, \beta, \gamma, \ldots \mid \bot \mid \top \mid \tau \rightarrow \tau \mid \{\phi\} \\ \phi &::= \varphi, \psi, \ldots \mid \mathtt{Bot} \mid \mathtt{Abs} \mid \mathtt{Pre}\;\tau \mid \mathtt{Either}\;\tau \mid \mathtt{Any} \end{aligned}$$

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 $\texttt{Abs} \sqcup (\texttt{Pre}\ \tau)$ equals $\texttt{Either}\ \tau$. This optional refinement allows us to keep track of a field's type, even when its presence is not ascertained. The lattice of field types is shown in figure 1.

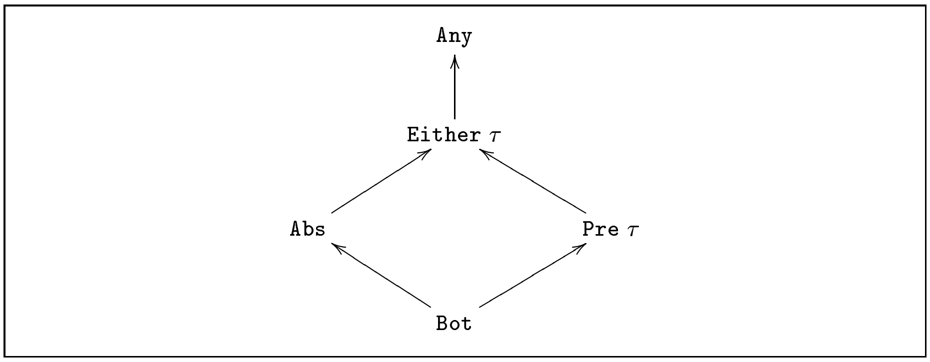

Fig. 1. The lattice of record field types

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

$$\begin{aligned} \texttt{Abs} &: \{\texttt{Abs}\} \\ \texttt{Pre} &: \alpha \rightarrow \{\texttt{Pre}\ \alpha\} \\ \texttt{Pre}^{-1} &: \{\texttt{Pre}\ \alpha\} \rightarrow \alpha \end{aligned}$$

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 3 has shown how to accurately describe such a function. Let us begin, then, with asymmetric concatenation:

$$\begin{aligned} @ &: \{\varphi_1\} \rightarrow \{\varphi_2\} \rightarrow \{\varphi_3\} \\ \text{where} \quad & \varphi_2 \leq \texttt{Either}\ \alpha_2 \\ & \texttt{Abs} \leq \varphi_2\ ?\ \varphi_1 \leq \varphi_3 \\ & \texttt{Pre} \leq \varphi_2\ ?\ \texttt{Pre}\ \alpha_2 \leq \varphi_3 \end{aligned}$$

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:

$$\begin{aligned} @@ : \{\varphi_1\} &\to \{\varphi_2\} \to \{\varphi_3\} \\ \text{where}\quad & \mathtt{Abs} \leq \varphi_1 \mathbin{?} \varphi_2 \leq \varphi_3 \\ & \mathtt{Abs} \leq \varphi_2 \mathbin{?} \varphi_1 \leq \varphi_3 \\ & \mathtt{Pre} \leq \varphi_1 \mathbin{?} \varphi_2 \leq \mathtt{Abs} \\ & \mathtt{Pre} \leq \varphi_2 \mathbin{?} \varphi_1 \leq \mathtt{Abs} \end{aligned}$$

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.

4.2 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; we shall dispense with them.

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

We do not give the language's semantics, 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 regular types, denoted by ϱ. Lastly, we lift the five field type constructors to the level of rows.

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

This allows writing complex constraints between rows, such as $\varphi \leq \mathtt{Pre}\ \alpha$, where φ and α are row variables. A constraint between rows stands for an infinite family of constraints between types, obtained component-wise. That is,

$$(l : \varphi';\ \varphi'') \leq \mathtt{Pre}\ (l : \alpha';\ \alpha'') \quad \text{stands for} \quad (\varphi' \leq \mathtt{Pre}\ \alpha') \wedge (\varphi'' \leq \mathtt{Pre}\ \alpha'')$$

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

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

Interestingly, the types of both concatenation operations are *unchanged* from the previous section – at least, syntactically. (For space reasons, 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.

5 Dynamic Messages

So-called "dynamic" messages have recently received new attention in the static typing community. Bugliesi and Crafa [3] propose a higher-order type system which accounts for first-class messages. Nishimura [11] tackles the issue of type inference and suggests a second-order system à la Ohori [13]. Müller and Nishimura [10] propose a simplified approach, based on an extended feature logic.

The problem consists in performing type inference for an object-oriented language where messages are first-class values, made up of a *label* and a *parameter*. Here, we view objects as records of functions, and messages as tagged values. (Better ways of modeling objects exist, but that is an independent issue.) Thus, we consider a language with records and data constructors, as described in Sections 3.2 and 4.2. Furthermore, we let record labels and data constructors range over a single name space, that of message labels. (To save space, we choose to deal directly with the case of multiple message labels; however, our usual, two-step presentation would still be possible.) Lastly, we define a primitive message-send operation, written #, whose semantics is as follows:

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

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.

In a language with "static" messages, a message-send operation may only involve a constant message label. So, instead of a single message-send operation, a family thereof, indexed by message labels, is provided. In fact, in our simple model, these operations are definable within the language. The operation $\#m$, which allows sending the message m to some object o with parameter p, may be defined as $\lambda o.\lambda p.(o.m\ p)$. Then, type inference yields

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

Because the message label, m, is statically known, it may be explicitly mentioned in the type scheme, making it easy to require the receiver object to carry an appropriate method. In a language with "dynamic" messages, on the other hand, m is no longer known. The problem thus appears more complex; it has, in fact, sparked the development of special-purpose constraint languages [10]. Yet, the machinery introduced so far in this paper suffices to solve it.

Consider the partial application of the message send primitive # to some record r. It is a function which accepts some tagged value $(m\, v)$, then invokes an appropriate piece of code, selected according to the label m. This should ring a bell – it is merely a form of pattern matching, which this paper has extensively discussed already. Therefore, we propose

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

(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 message parameter's type. The conditional constraint, which involves two row terms, should again be understood as a family, indexed by message labels, of conditional constraints between record field types. The conditional constraint associated with some label m shall 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 whole operation's result type must be (at least) the join of all potentially invoked methods' return types.

Our proposal shows that type inference for "dynamic" messages requires no dedicated theoretical machinery. It also shows that "dynamic" messages are naturally compatible with all operations on records, including concatenation – a question which was left unanswered by Nishimura [11].

6 Conclusion

In this paper, we have advocated enriching an existing constraint-based type inference framework [15] with rows [18] and conditional constraints [2]. This provides a single (and simple) solution to several difficult type inference problems, each of which seemed to require, until now, special forms of constraints. From a practical point of view, it allows them to benefit from known constraint simplification techniques [17], leading to an efficient inference algorithm [14].

We believe our system subsumes Rémy's proposal for record concatenation [20], as well as Müller and Nishimura's view of "dynamic" messages [10]. 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. In fact, the ideas developed in this paper could have been presented in the setting of Bane [5], or, more generally, of any system which allows writing sufficiently expressive constrained type schemes.

References

[1] Alexander S. Aiken and Edward L. Wimmers. Type inclusion constraints and type inference. In *Functional Programming & Computer Architecture*, pages 31–41. ACM Press, June 1993. URL: `http://http.cs.berkeley.edu/~aiken/ftp/fpca93.ps`.

[2] Alexander S. Aiken, Edward L. Wimmers, and T. K. Lakshman. Soft typing with conditional types. In *Principles of Programming Languages*, pages 163–173, January 1994. URL: `http://http.cs.berkeley.edu/~aiken/ftp/popl94.ps`.

[3] Michele Bugliesi and Silvia Crafa. Object calculi for dynamic messages. In *The Sixth International Workshop on Foundations of Object-Oriented Languages, FOOL 6, San Antonio, Texas*, January 1999. URL: `ftp://ftp.cs.williams.edu/pub/kim/FOOL6/bugliesi.ps`.

[4] Luca Cardelli and John Mitchell. Operations on records. *Mathematical Structures in Computer Science*, 1:3–48, 1991. Also in Carl A. Gunter and John C. Mitchell, editors, *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design*, MIT Press, 1994. URL: `http://research.microsoft.com/Users/luca/Papers/Records.ps`.

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

[6] Cormac Flanagan and Matthias Felleisen. Componential set-based analysis. In *Proceedings of the ACM SIGPLAN '97 Conference on Programming Language Design and Implementation*, pages 235–248, Las Vegas, Nevada, June 1997. URL: `http://www.cs.rice.edu/CS/PLT/Publications/pldi97-ff.ps.gz`.

[7] Robert Harper and Benjamin Pierce. A record calculus based on symmetric concatenation. In *Conference Record of the 18th Annual ACM Symposium on Principles of Programming Languages (POPL '91)*, pages 131–142, Orlando, Florida, January 1991. ACM Press. URL: `http://www.cis.upenn.edu/~bcpierce/papers/merge.ps.gz`.

[8] Nevin Heintze. Set based analysis of ML programs. Technical Report CMU-CS-93-193, Carnegie Mellon University, School of Computer Science, July 1993. URL: `ftp://reports.adm.cs.cmu.edu/usr/anon/1993/CMU-CS-93-193.ps`.

[9] Martin Müller, Joachim Niehren, and Andreas Podelski. Ordering constraints over feature trees. *Constraints, an International Journal, Special Issue on CP'97, Linz, Austria*, 1999. URL: `ftp://ftp.ps.uni-sb.de/pub/papers/ProgrammingSysLab/ftsub-constraints-99.ps.gz`.

[10] Martin Müller and Susumu Nishimura. Type inference for first-class messages with feature constraints. In Jieh Hsiang and Atsushi Ohori, editors, *Asian Computer Science Conference (ASIAN 98)*, volume 1538 of *LNCS*, pages 169–187, Manila, The Philippines, December 1998. Springer-Verlag. URL: `ftp://ftp.ps.uni-sb.de/pub/papers/ProgrammingSysLab/FirstClass98.ps.gz`.

[11] Susumu Nishimura. Static typing for dynamic messages. In *Conference Record of POPL '98: The 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 266–278, San Diego, California, January 1998. URL: `ftp://ftp.kurims.kyoto-u.ac.jp/pub/paper/member/nisimura/dmesg-popl98.ps.gz`.

[12] Martin Odersky, Martin Sulzmann, and Martin Wehr. Type inference with constrained types. *Theory and Practice of Object Systems*, 5(1), 1999. URL: `http://www.cs.yale.edu/~sulzmann-martin/publications/tapos.ps`.

[13] Atsushi Ohori. A polymorphic record calculus and its compilation. *ACM Transactions on Programming Languages and Systems*, 17(6):844–895, November 1995.
[14] François Pottier. `Wallace`: an efficient implementation of type inference with subtyping. URL: `http://pauillac.inria.fr/~fpottier/wallace/`.
[15] François Pottier. Simplifying subtyping constraints: a theory. Submitted for publication, December 1998. URL: `http://pauillac.inria.fr/~fpottier/publis/fpottier-journal-98.ps.gz`.
[16] François Pottier. Type inference in the presence of subtyping: from theory to practice. Technical Report 3483, INRIA, September 1998. URL: `ftp://ftp.inria.fr/INRIA/publication/RR/RR-3483.ps.gz`.
[17] François Pottier. Subtyping-constraint-based type inference with conditional constraints: algorithms and proofs. Unpublished draft, July 1999. URL: `http://pauillac.inria.fr/~fpottier/publis/fpottier-conditional.ps.gz`.
[18] Didier Rémy. Projective ML. In *1992 ACM Conference on Lisp and Functional Programming*, pages 66–75, New-York, 1992. ACM Press. URL: `ftp://ftp.inria.fr/INRIA/Projects/cristal/Didier.Remy/lfp92.ps.gz`.
[19] Didier Rémy. Typing record concatenation for free. In Carl A. Gunter and John C. Mitchell, editors, *Theoretical Aspects Of Object-Oriented Programming. Types, Semantics and Language Design*. MIT Press, 1993. URL: `ftp://ftp.inria.fr/INRIA/Projects/cristal/Didier.Remy/taoop2.ps.gz`.
[20] Didier Rémy. 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, August 1995. URL: `http://cristal.inria.fr/~remy/work/sub-concat.dvi.gz`.
[21] John C. Reynolds. Automatic computation of data set definitions. In A. J. H. Morrell, editor, *Information Processing 68*, volume 1, pages 456–461, Amsterdam, 1969. North-Holland.
[22] Valery Trifonov and Scott Smith. Subtyping constrained types. In *Proceedings of the Third International Static Analysis Symposium*, volume 1145 of *LNCS*, pages 349–365. SV, September 1996. URL: `http://www.cs.jhu.edu/~trifonov/papers/subcon.ps.gz`.
[23] Mitchell Wand. Type inference for record concatenation and multiple inheritance. *Information and Computation*, pages 1–15, 1993. A preliminary version appeared in *Proc. 4th IEEE Symposium on Logic in Computer Science* (1989), pp. 92–97. URL: `ftp://ftp.ccs.neu.edu/pub/people/wand/papers/ic-91.dvi`.

A Rules

This appendix gives a short description of the system's type inference rules (Figure 2). Even though only the core language is explicitly treated, these rules are sufficient to deal with a full-featured programming language. Indeed, any extra language construct may be viewed either as syntactic sugar, or as a new primitive operation, which can be bound in an initial typing environment Γ_0. Also, note that these type inference rules use neither conditional constraints, nor rows; these will come only from Γ_0.

For simplicity, we distinguish identifiers bound by λ, denoted $x, y, \ldots$ from those bound by `let`, denoted $X, Y, \ldots$ Furthermore, we expect λ-identifiers to be unique; that is, each λ-identifier must be bound at most once in a given

$$\frac{\alpha \text{ fresh}}{\Gamma \vdash_{\mathrm{I}} x : \forall\varnothing.\, \langle x : \alpha\rangle \Rightarrow \alpha} \quad (\textsc{Var}_{\mathrm{I}})$$

$$\frac{\Gamma \vdash_{\mathrm{I}} e : \forall C.\, A \Rightarrow \tau' \qquad A(x) = \tau}{\Gamma \vdash_{\mathrm{I}} \lambda x.e : \forall C.\, (A \setminus x) \Rightarrow \tau \to \tau'} \quad (\textsc{Abs}_{\mathrm{I}})$$

$$\frac{\begin{array}{c}\Gamma \vdash_{\mathrm{I}} e_1 : \forall C_1.\, A_1 \Rightarrow \tau_1 \qquad \Gamma \vdash_{\mathrm{I}} e_2 : \forall C_2.\, A_2 \Rightarrow \tau_2 \\ \alpha \text{ fresh} \qquad C = C_1 \cup C_2 \cup \{\tau_1 \leq \tau_2 \to \alpha\}\end{array}}{\Gamma \vdash_{\mathrm{I}} e_1\, e_2 : \forall C.\, (A_1 \sqcap A_2) \Rightarrow \alpha} \quad (\textsc{App}_{\mathrm{I}})$$

$$\frac{\Gamma(X) = \sigma \qquad \rho \text{ fresh renaming of } \sigma}{\Gamma \vdash_{\mathrm{I}} X : \rho(\sigma)} \quad (\textsc{LetVar}_{\mathrm{I}})$$

$$\frac{\Gamma \vdash_{\mathrm{I}} e_1 : \sigma_1 \qquad \Gamma + [X \mapsto \sigma_1] \vdash_{\mathrm{I}} e_2 : \sigma_2}{\Gamma \vdash_{\mathrm{I}} \texttt{let } X = e_1 \texttt{ in } e_2 : \sigma_2} \quad (\textsc{Let}_{\mathrm{I}})$$

Fig. 2. Type inference rules

program. Lastly, in every expression of the form `let` $X = e_1$ `in` e_2, we require X to appear free within e_2. It would be easy to overcome these restrictions, at the expense of heavier notation.

The rules are fairly straightforward. The main point of interest is the way each application node produces a subtyping constraint. The only peculiarity is in the way type environments are dealt with. The *environment* Γ, which appears on the left of the turnstile, is a list of bindings of the form $X : \sigma$. Type schemes are slightly more complex than initially shown in Section 2. They are, in fact, of the form $\sigma ::= \forall C.\, A \Rightarrow \tau$, where the *context* A is a set of bindings of the form $x : \tau$. The point of such a formulation is to obtain a system where no type scheme has free type variables. This allows a simpler theoretical description of constraint simplification.

As far as notation is concerned, $\langle x : \alpha\rangle$ represents a context consisting of a single entry, which binds x to α. $A \setminus x$ is the context obtained by removing x's binding from A, if it exists. For the sake of readability, we have abused notation slightly. In rule ($\textsc{Abs}_{\mathrm{I}}$), $A(x)$ stands for the type associated with x in A, if A contains a binding for x; it stands for $\top$ otherwise. In rule ($\textsc{App}_{\mathrm{I}}$), $A_1 \sqcap A_2$ represents the point-wise intersection of A_1 and A_2. That is, whenever x has a binding in A_1 or A_2, its binding in $A_1 \sqcap A_2$ is $A_1(x) \sqcap A_2(x)$. Because we do not have intersection types, this expression should in fact be understood as a fresh type variable, accompanied by an appropriate conjunction of subtyping constraints.

The rules implicitly require every constraint set to admit at least one solution. Constraint solving and simplification are described in [15, 17].

B Examples

Example 1. We define a function which reads field l out of a record r, returning a default value d if r has no such field, by setting $\texttt{extract} = \lambda d.\lambda r.(\{l = d\} \mathbin{@} r).l$. In our system, `extract`'s inferred type is

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

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 implementing slightly more aggressive simplification algorithms.

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

Example 2. We assume given an object o, of the following type:

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

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:

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

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`.