Simple Type Inference for Structural Polymorphism

Jacques Garrigue
garrigue@kurims.kyoto-u.ac.jp

Research Institute for Mathematical Sciences
Kyoto University, 606-8502 Kyoto, JAPAN

Abstract

We propose a new way to mix constrained types and type inference, where the interaction between the two is minimal. By using local constraints embedded in types, rather than the other way round, we obtain a system which keeps the usual structure of an Hindley-Milner type system. In practice, this means that it is easy to introduce local constraints in existing type inference algorithms.

Eventhough our system is notably weaker than general constraint-based type systems, making it unable to handle subtyping for instance, it is powerful enough to accomodate many features, from simple polymorphic records *à la* Ohori to Objective Caml's polymorphic variants, and accurate typing of pattern matching (*i.e.* polymorphic message dispatch), all these through tiny variations in the constraint part of the system.

1 Introduction

Type inference for *structural polymorphism* has been a long standing area of research. By structural polymorphism, we mean a form of parametric polymorphism in which some types may have a partially known structure, often denoted by a notion of row variable or kinded variable. Applications range from polymorphic records [Oho95, Rém93] to polymorphic variants [Rém89, Gar98] and objects with first-class (dynamic) messages [Nis98]. All these systems were proposed as extensions of the Hindley-Milner type system [Mil78], with growing degrees of complexity.

A recent trend is to simplify these type systems by moving to a constraint-based framework [OSW99, MN00, Sul00, Pot00]. This indeed greatly improved the understanding of the various systems, as they can all be described as particular instances of the HM(X) framework, on various constraint domains. However, the switch to a constraint system also means that previous work on Hindley-Milner does not directly apply. This has practical consequences: it is hard to extend an existing algorithm to work with constraints, as all types must now be embedded in constraints. This has also theoretical consequences: take for instance Hindley-Milner extended with first-class polymorphism [GR99]; the extension intimately depends on the way polymorphism is handled in the original type system, and as HM(X) handles it in a radically different way (a type variable can be both constrained and polymorphic), it is unclear how it could handle such an extension.

This is all the more disturbing as HM(X) is somehow "too powerful" for structural polymorphism. It was designed to accomodate subtyping, which is carefully avoided in structural polymorphism.

A natural way to go is then to search for a weaker system, not handling subtyping, but closer to the original Damas-Milner formulation [DM82]. This is what we present here. We do it by restricting the role of constraints to individual type nodes: each constraint lives inside its type node, and can only influence outside types by requiring two children of the node to be equal. Since the embedding is done the other way round, existing type inference algorithms can easily be extended with such constrained types. For instance, the handling of polymorphic variants in Objective Caml 3.00[LDG+00] can be seen as an instance of this framework. We keep the original definition of free variable, making this system compatible with first-class polymorphism for instance. Locality is also beneficial for producing readable type errors: rather than producing an unsolvable set of arbitrary constraints, we can localize the problem, and translate it back to a meaningful form.

Contributions of this paper can be found at the application level: to our knowledge, both the problem of discarding irrelevant type information in shrinking types, and of accurately typing pattern matching, were still open for type systems without subtyping-like mechanisms. The solutions we propose here are simple instances of our framework.

Our system is very close to Ohori's polymorphic records [Oho95], reusing the concept of kind as constraint. It is made modular by completely separating the descriptions of types and constraints. New typed constructs can be added by merely specifying a new constraint domain, and new typed constants, without needing to introduce new type inference rules.

First, we will present by examples various forms of structural polymorphism. Then we introduce the notion of *constraint domain*, which is used as parameter to our framework. All forms of structural polymorphism we present here appear to have very similar constraint domains. In a 4th section we introduce the type system itself, with its specific notion of substitution. Section 5 introduces terms and typing rules. Section 6 and 7 are devoted to unification and type reconstruction. We conclude discussing the expressive power of this system. Proofs are given in appendix.

2 Structural polymorphism

We present here various forms of structural polymorphism, on a gradual scale of increasing complexity. In order to uniformize the presentation, we write $K \triangleright \tau$ for types, where K is a kinding environment, containing constraints for individual type variables, under which the open type τ is to be understood. We use a few built-in functions, to obtain interesting types: integer addition $+$, string concatenation $\uparrow$, conversions $string\_of\_int : string \to int$ and $float : int \to float$.

Records *à la* Ohori Arguably this is the simplest form of structural polymorphism considered in the literature. Record values have monomorphic types, and polymoprhism is only used for typing field access.

$$
\begin{array}{ll}
\{name = \texttt{"Jacques"}, age = 30\} & : \{name : string, age : int\} \\
\mathsf{fun}\ x \to x.age + 1 & : \alpha :: \{age : int\} \triangleright \alpha \to int \\
\multicolumn{2}{l}{\mathsf{fun}\ x \to x.name \uparrow \texttt{" is "} \uparrow string\_of\_int\ x.age} \\
 & : \alpha :: \{name : string, age : int\} \triangleright \alpha \to string
\end{array}
$$

Intuitively, the kinding $\alpha :: \{age : int\}$ means that α should at least have a field age, and this field should have type int. Accessing several fields result in a kind containing all of them. Two kinds are compatible if they agree on the types of their common fields. A type satisfies a kind if all the required fields are provided, with the correct types.

Note that the above formulation is a direct adaptation of Ohori's. Fitting it inside the local constraint framework will require a small change in presentation, without changing expressive power.

Records and variants with masking Ohori's records are weak in that record values have only structurally monomorphic types. We might want to allow building lists of records containing different fields, with only part of them common to all members. This requires giving polymorphic types not only to field access, but also to record values. A type system allowing it was first proposed by Rémy [Rém89], but here we use a more intuitive formalism [Gar98], which is also closer to Ohori's.

$$
\begin{array}{l}
\{name = \texttt{"Jacques"}, age = 30\} \\
\qquad : \alpha :: (\{name : string, age : int\}, \emptyset, \{name, age\}) \triangleright \alpha \\
\mathsf{fun}\ x \to x.age + 1 \quad : \alpha :: (\{age : int\}, \{age\}, \mathcal{L}) \triangleright \alpha \to int \\
\mathsf{let}\ l_1 = [\{name = \texttt{"Jacques"}, age = 30\}, \\
\qquad\quad \{name = \texttt{"Serge"}, weight = 13\}] \\
l_1\ :\ \alpha :: (\{name : string, age : int, weight : int\}, \emptyset, \{name\}) \\
\qquad \triangleright \alpha\ list
\end{array}
$$

Along with the type of each field, we have two sets of labels. Required labels (union of all field accesses) form a subset of available labels (intersection of the possible record values). $\mathcal{L}$ is the set of all labels. You can see how fields are masked in the third example: while we have types for the *name, age* and *weight* fields, only the *name* field is accessible.

The combined type $\alpha :: (\{name : string, age : int\}, \{age\}, \{name, age\})$ is an acceptable description for both $\{name = \texttt{"Jacques"}, age = 30\}$, which makes *name* and *age* available, and $\mathsf{fun}\ x \to x.age + 1$, which only requires *age*. The need for two distinct sets of labels stems from the natural appearence of such combined types during type inference. Since attempting to access an unavailable label would be a type error, this also explains why, for a constraint kind to be meaningful (to have a solution), the set of required labels should be included in the set of available labels.

By duality, the same types can be used to describe polymorphic variants. The basic idea is that case-analysis of a variant can receive the same type as a record, while the variant itself would get the type of a field accessor.

$$
\begin{array}{l}
Number(5) \qquad : \alpha :: (\{Number : int\}, \{Number\}, \mathcal{L}) \triangleright \alpha \\
\mathsf{let}\ l_2 = [Number(5), Face(\texttt{"King"})] \\
l_2\ :\ \alpha :: (\{Number : int, Face : string\}, \{Number, Face\}, \mathcal{L}) \\
\qquad \triangleright \alpha\ list \\
\mathsf{let}\ f_1 = \mathsf{function}\ Number(n) \to string\_of\_int\ n \\
\qquad\qquad\qquad\quad \mid Face(name) \to name \\
f_1\ :\ \alpha :: (\{Number : int, Face : string\}, \emptyset, \{Number, Face\}) \\
\qquad \triangleright \alpha \to string
\end{array}
$$

Our two sets of labels have now different meanings. The first one is the set of present constructors, or required cases, which must be handled by case-analysis; the second one is the set of handled constructors, or available cases, which is accepted by all case-analyses.

An advantage of not using predefined sum types, is that we can make case analysis modular. Consider the following function f_2, which uses a special syntax for dispatch. g (respectively h) will only receive A or B (respectively C or D). This can be reflected at the type level by requiring them to handle only relevant cases.

$$
\begin{array}{l}
\mathsf{let}\ f_2 = \mathsf{fun}\ g \to \mathsf{fun}\ h \to \mathsf{function}\ (A|B)\ as\ x \to g\ x \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \mid (C|D)\ as\ x \to h\ x \\
f_2\ :\ \alpha_1 :: (\{A : \alpha_A, B : \alpha_B\}, \{A, B\}, \mathcal{L}), \\
\qquad \alpha_2 :: (\{C : \alpha_C, D : \alpha_D\}, \{C, D\}, \mathcal{L}), \\
\qquad \alpha_3 :: (\{A : \alpha_A, B : \alpha_B, C : \alpha_C, D : \alpha_D\}, \emptyset, \{A, B, C, D\}) \\
\qquad \triangleright (\alpha_1 \to \alpha) \to (\alpha_2 \to \alpha) \to \alpha_3 \to \alpha
\end{array}
$$

Our choice of keeping all constraints local (constraining only one variable) makes our system slightly weaker than Rémy's, which has row and presence variables. For instance we have no way to relate the set of required cases in α_1 and α_3, which forces us to make the safe assumption $\{A, B\}$ in α_1. Rémy's system handles requirement for each constructor as an independent presence variable, which may be shared between two different variant types. This allows more precise typing, but at the cost of harder to understand types, with lots of variables, of different sorts. Experience suggests that kinded variables, by reducing the number of variables to one by record or variant type, and keeping only one sort of type variables, make reading types much easier.

Discarding masked types One may wonder about why one should keep all field types in the type of l_1, a list of records with some masked fields. If fields *age* and *weight* are actually unavailable, why should their types matter? Clearly, this is not the case with subtyping, which would allow to discard not only the fields, but also their types. However, our requirement of principal type inference makes impossible to simply forget the type.

For instance, let us consider the list:

$$
\begin{array}{l}
\mathsf{let}\ l_3 = [\{name = \texttt{"Jacques"}, age = 30\}, \\
\qquad\quad \{name = \texttt{"Rachel"}, age = 6.5\}]
\end{array}
$$

The two records disagree on the type for *age*, either *int* or *float*. This should trigger a type error. Yet, if we choose to discard the type for *age* in l_1, we would be able to type $l_1 @ [\{name = \texttt{"Rachel"}, age = 6.5\}]$, which contains the untypable l_3 as a sublist.

A solution to this problem is to only trigger an error when a field with conflicting types is used. This amounts to allowing conjunctive types in the types of non-required fields, only forcing them to be equal when they are required. Then we can give the following type to l_3.

$$l_3 \;:\; \alpha :: (\{name : string, age : int \wedge float\}, \emptyset, \{name, age\}) \triangleright \alpha$$

Since the two types for the field *age* are incompatible, it cannot be accessed. And if it disappears from the list of available fields, we can just drop these types as useless information.

While the absence of error may seem strange in the above case (even though it is necessary for coherence), it is a natural way to allow the construction of heterogeneous collections of objects in a system without subtyping.

The same mechanism also applies to polymorphic variants, and is actually used in Objective Caml 3.00. This allows principal type inference of f_5 for the following case, solving the long lasting problem of "masked but not discarded" types.

$$\begin{array}{l}
\mathsf{let}\ f_3 = \mathsf{function}\ Number(n) \to n \mid Face() \to 15 \\
f_3 \;:\; \alpha :: (\{Number : int, Face : unit\}, \emptyset, \{Number, Face\}) \\
\qquad \triangleright\, \alpha \to int \\
\mathsf{let}\ f_4 = \mathsf{function}\ Number(n) \to n/2 \\
f_4 \;:\; \alpha :: (\{Number : int\}, \emptyset, \{Number\}) \\
\qquad \triangleright\, \alpha \to int \\
\mathsf{let}\ f_5 = \mathsf{fun}\ x \to (f_1(x), f_3(x), f_4(x)) \\
f_5 \;:\; \alpha :: (\{Number : int\}, \emptyset, \{Number\}) \\
\qquad \triangleright\, \alpha \to (string \times int \times int)
\end{array}$$

Accurate matching and dynamic messages Polymorphic variants provide already more precise typing than classical predefined variant types, since a specific variant type is computed for every case-analysis, even if the same constructors are partly reused.

We can go further, and allow the type of the result to depend on which case was used; this happens for instance when a message is dispatched between methods whose result types differ. Such a mechanism was first proposed by Aiken *et al.* in the context of soft typing for dynamically typed languages [AWL94]. Pottier reformulated it as an instance of the HM(X) framework, calling it *accurate analysis of pattern matching* [Pot00]. In both cases, the type of the result is described by a conditional type, depending on the input. Moreover, they use global inclusion constraints to deal with the problem, while we use only local ones.

Concretely, we add a new cases construct, which has the same syntax as $\mathsf{function}$, but allows each case to return a value of different type. Here is an example of the expected behaviour:

$$\begin{array}{l}
\mathsf{let}\ f_6 = \mathsf{cases}\ Number() \to 10 \mid Face() \to \text{"}King\text{"} \\
f_6\ (Number()) \;:\; int \;\hookrightarrow 10 \\
f_6\ (Face()) \;:\; string \;\hookrightarrow \text{"}King\text{"}
\end{array}$$

Our basic idea to accommodate different return types, is to add another set of type bindings, corresponding to the results of each branch of the pattern matching. The special label return is used to indicate the result type of the whole pattern matching. If a constructor is required, then its associated type is unified with the type of the result.

$$\begin{array}{l}
f_6 \;:\; \alpha :: (\{Number : unit, Face : unit\}, \\
\qquad\qquad \{Number : int, Face : string, \mathsf{return} : \alpha_1\}, \\
\qquad\qquad \emptyset, \{Number, Face\}) \\
\qquad \triangleright\, \alpha \to \alpha_1
\end{array}$$

$$\begin{array}{l}
\mathsf{let}\ f_7 = \mathsf{fun}\ (x : \alpha_2 :: (\emptyset, \emptyset, \{Number\}, \mathcal{L}) \triangleright \alpha_2) \to f_6\ x \\
f_7 \;:\; \alpha :: (\{Number : unit, Face : unit\}, \\
\qquad\qquad \{Number : int, Face : string, \mathsf{return} : int\}, \\
\qquad\qquad \{Number\}, \{Number, Face\}) \\
\qquad \triangleright\, \alpha \to int
\end{array}$$

You can see how inference proceeds in f_7. The type annotation on x is only there to simulate application on a monomorphic constructor. Note how the type of the variant itself ends up including the type of the result, avoiding the need for a global constraint.

This mechanism is then extended to deal with the case when the same variant is passed to several pattern-matching functions, with different return types: $\mathsf{fun}\ x \to (f\ x, g\ x)$. If we have only one return type set in our variant type, then the monomorphism of x would force return types in f and g to be identical. But we provide several sets, one by function, by a mechanism we call *generativity*. This way the type of x may contain different return types for different functions, without interference between them. We leave details for the formal development.

A simple and useful application of this extension is to provide polymorphic decomposing functions for variants, without overloading.

$$\begin{array}{l}
\mathsf{let}\ arg = \mathsf{cases}\ Number(n : int) \to n \mid Face(s : string) \to s \\
arg \;:\; \alpha :: (\{Number : int, Face : string\}, \\
\qquad\qquad \{Number : int, Face : string, \mathsf{return} : \alpha_1\}, \\
\qquad\qquad \emptyset, \{Number, Face\}) \\
\qquad \triangleright\, \alpha \to \alpha_1 \\
\mathsf{let}\ tag = \mathsf{cases}\ Number(n : int) \to 1 \mid Face(s : string) \to 2 \\
tag \;:\; \alpha :: (\{Number : int, Face : string\}, \\
\qquad\qquad \{Number : int, Face : int, \mathsf{return} : \alpha_1\}, \\
\qquad\qquad \emptyset, \{Number, Face\}) \\
\qquad \triangleright\, \alpha \to \alpha_1 \\
\mathsf{let}\ decomp = \mathsf{fun}\ x \to (tag\ x, arg\ x) \\
decomp \;:\; \alpha :: (\{Number : int, Face : string\}, \\
\qquad\qquad \{\{Number : int, Face : int, \mathsf{return} : \alpha_1\}, \\
\qquad\qquad \{Number : int, Face : string, \mathsf{return} : \alpha_2\}\}, \\
\qquad\qquad \emptyset, \{Number, Face\}) \\
\qquad \triangleright\, \alpha \to \alpha_1 \times \alpha_2 \\
decomp\ (Face\ \text{"}\mathtt{King}\text{"}) \;:\; int \times string \;\hookrightarrow (2, \text{"}\mathtt{King}\text{"}) \\
decomp\ (Number\ 10) \;:\; int \times int \;\hookrightarrow (1, 10)
\end{array}$$

In a more object-oriented context, the same kind of mechanism was called dynamic messages by Nishimura [Nis98]; later it was reformulated using different instances of HM(X) [MN00, Pot00]. Dynamic messages can be encoded using a small amount of syntactic sugar, but the way we rely on polymorphism in pattern-matching functions makes our system weaker.

This can be seen on the following example. Objects $\{|\ldots|\}$ are case-analyzing functions, and message passing $o \leftarrow m$ is the application $o\ m$.

$$\begin{array}{l}
\mathsf{let}\ o_1 = \{|Number(x) = float\ x; Face(s : string) = s|\} \\
o_1 \;:\; \alpha :: (\{Number : int, Face : string\}, \\
\qquad\qquad \{Number : float, Face : string, \mathsf{return} : \alpha_1\}, \\
\qquad\qquad \emptyset, \{Number, Face\}) \\
\qquad \triangleright\, \alpha \to \alpha_1 \\
o_1 \leftarrow Number\ 6 \;:\; float \;\hookrightarrow 6.0 \\
o_1 \leftarrow Face\ \text{"}\mathtt{King}\text{"} \;:\; string \;\hookrightarrow \text{"}\mathtt{King}\text{"} \\
\mathsf{let}\ f_{11} = \mathsf{fun}\ o \to (o \leftarrow Number\ 6, o \leftarrow Face\ \text{"}\mathtt{King}\text{"}) \\
f_8 \;:\; \alpha_1 :: (\{Number : int, Face : string\}, \emptyset, \\
\qquad\qquad \{Number, Face\}, \mathcal{L}) \\
\qquad \triangleright\, (\alpha_1 \to \alpha) \to \alpha \times \alpha \\
f_8\ o_1 \;:\; untypable
\end{array}$$

As you can see, the encoding is sound, and different methods can have different result types. However, if we apply the same monomorphic object to two different messages, they are merged in one, and later we are unable to distinguish their result types. For this reason, the application $f_8\ o_1$ cannot be typed: constraints involving both *Number* and *Face* cannot be solved simultaneously. Nishimura's system did not require polymorphism, since message and object types were not merged, rather a global subtyping constraint was generated.

While complex applications of dynamic messages cannot be encoded directly in our system, simpler cases where objects are only used as second class values can be handled. Moreover, if one needs real first class objects, first class polymorphism [GR99] is available to solve the problem. This only requires to write object types when they appear as function parameters.

3 Constraints domains

Contrary to usual constraint-based type systems, our local constraints are grafted on top of an Hindley-Milner type system, rather than mixing constraints and types at the same level. This means that we can reduce requirements on constraints to a minimum: their interaction with the rest of the type system will be minimal anyway. In particular, we do not introduce any syntax for constraints: they are black boxes, and need just be able to answer some questions. While not essential, we think this freedom is important, as the choice of how to represent constraints is relevant to their understanding. Except for this extra freedom, nothing would prevent one to formalize our constraint domains as special kinds of cylindric algebras, which can be plugged into HM(X).

A constraint domain describes a type of constraints, and how they interact with the type system. A particular instance of the type system may contain several constraint domains, as long as all their operations and values are clearly distinguished. For simplicity, we will only consider type systems operating on a single constraint domain.

Definition 1 *A constraint domain $\mathcal{C}$ is composed of the following items.*

1. *A theory $\mathcal{T}_\mathcal{C}$ with an entailment relation $\models$ satisfying the following properties*
   - (a) *There is a constraint $\perp$, such that for any C we have $\perp \models C$.*
   - (b) *A constraint C such that $C \models \perp$ is* invalid. *Validity is decidable.*
   - (c) *Entailment is reflexive and transitive: $C \models C$; if $C \models C'$ and $C' \models C''$ then $C \models C''$.*
   - (d) *For any two constraints C and C', there is a constraint $C \wedge C'$ such that $C \wedge C' \models C$, $C \wedge C' \models C'$, and for all C'' such that $C'' \models C$ and $C'' \models C'$, we have $C'' \models C \wedge C'$.*
2. *An observation relation $\vdash$ checking some atomic properties of a constraint: $C \vdash p(a)$ where p and a are respectively a predicate and a symbol for the domain. Observation should be compatible with entailment:*

   $$\text{If } C \models C' \text{ and } C' \vdash p(a) \text{ then } C \vdash p(a).$$
3. *A set of relating predicates of the form $r(a, \tau)$, which relate symbols and types. Some of these predicates are said to be* generative*, that is they have an extra index $r^\epsilon(a, \tau)$, which can be existentially quantified.*
4. *A set of propagation rules $\mathcal{E}_\mathcal{C}$, of the form (depending on whether r is generative or not)*

   $$\forall x.(r(x, \alpha_1) \wedge r(A, \alpha_2) \wedge p(x) \Rightarrow \alpha_1 = \alpha_2)$$
   $$\forall x \forall \epsilon.(r^\epsilon(x, \alpha_1) \wedge r^\epsilon(A, \alpha_2) \wedge p(x) \Rightarrow \alpha_1 = \alpha_2)$$

   where A is either the variable x or a symbol.

Our requirements on the entailment relation are as free as one can get. Basically, we only need a way to distinguish valid constraints from invalid one, and build the intersection of two constraints.

Observation predicates are a consequence of the representation independency of our constraints: we need an explicit way to relate them to the rest of the world. They are interfacing constraints, which are semantic, with propagation rules, which are syntactic.

Relating predicates are similar in flavor to features in feature algebras [AKPS94]. Each constraint will be coupled with a set R of instances of these predicates, in order to relate its semantics with types. Generative predicates are used for types the constraint may need to control, but which do not need to be identified in all objects holding this constraint. The idea is that merging two quantified sets of predicates will not mix them: $\exists \epsilon.\{r^\epsilon(a, \tau)\} \cup \exists \epsilon.\{r^\epsilon(a, \tau')\} = \exists \epsilon \epsilon'.\{r^\epsilon(a, \tau), r^{\epsilon'}(a, \tau')\}$. Among our examples, only the typing of accurate matching uses generative constraints. Other cases can be handled in a system without ϵ's.

Propagation rules specify how C and R constrain other types. They may seem ad-hoc, but relaxing the definition would not give more expressive power: arbitrarily complex conditions can be encoded in the constraint itself. Keeping the syntax of the rules simple makes checking the applicability of rules easy: one just has to iterate on the (finitary) relating predicates, and check observation for ground atoms. On the other hand, the restriction to the form $\alpha_1 = \alpha_2$ on the right hand side is crucial: this is what allows our unification algorithm to terminate.

We say that a constraint is *exact* if it cannot be further refined, *i.e.* if it is only entailed by itself and $\perp$. This is the equivalent of a ground type in the world of constraints. Whether a constraint is exact or not does not impact anything in the theory, but knowing it may be helpful when reading types.

We now consider the constraint domains associated to examples in the previous sections.

Records *à la* Ohori The original formulation did mix required fields and their types in a single kind. We have to distinguish the two, to adapt to our framework. Moreover, since we do not extend monomorphic types themselves, monomorphic records should also be described by exact constraints.

A constraint is a pair (L, x) of a finite set of labels L, together with a mark x distinguishing exact types (1) from refinable ones (0). Entailment on refinable types is containment: $(L, x) \models (L', 0)$ iff $L \supset L'$, which makes set union the conjunction operation. For exact types entailment is only reflexive. All constraints are valid, so we just add a $\perp$ element. We do not need to observe anything particular, so we just use $C \vdash true(l)$ for any l. The type associated to each label is described by a predicate $r(l,\tau)$, and the fact a label has only one type is captured by the unique propagation rule

$$r(l,\alpha_1) \wedge r(l,\alpha_2) \wedge true(l) \Rightarrow \alpha_1 = \alpha_2.$$

Records with maskable fields Constraints are represented by a pair of sets (L,U), L a finite set of accessed labels, and U either the set of all labels $\mathcal{L}$, or a finite set of available labels. Entailment is defined by $(L,U) \models (L',U')$ iff $L \supset L'$ and $U \subset U'$. We can chose $(\mathcal{L},\emptyset)$ as $\perp$. The validity check is $(L,U) \models \perp$ when $L \not\subset U$. One can observe required labels: $(L,U) \vdash req(l)$ iff $l \in L$. We can either keep the previous propagation rule, and obtain records without discarding, or switch to the following new rule, which allows conjunctive typing, by delaying the equality constraint until the field is accessed.

$$r(l,\alpha_1) \wedge r(l,\alpha_2) \wedge req(l) \Rightarrow \alpha_1 = \alpha_2.$$

Accurate matching As for records with maskable fields, constraints are pairs (L,U), with the same entailment relation. We reuse also the conditional propagation rule, but we also add a second generative relation predicate $v^\epsilon(l,\tau)$, and a second propagation rule, acting on result types:

$$v^\epsilon(l,\alpha_1) \wedge v^\epsilon(\mathsf{return},\alpha_2) \wedge req(l) \Rightarrow \alpha_1 = \alpha_2.$$

Both argument and return types will only be unified if the case corresponding to l is required, that is if the method l is called.

4 Types and kinds

Simple types τ are defined as usual. Polytypes σ are extended with a kinding environment K which restricts possible instances for constrained variables.

τ	$::=$	α	type variable
	$\mid$	u	base type
	$\mid$	$\tau \to \tau$	function type
K	$::=$	$\emptyset \mid \mathrm{K}, \alpha :: (C, \exists\bar{\epsilon}.R)$	kinding environment
σ	$::=$	$\tau \mid \forall\alpha \ldots \alpha.\mathrm{K} \triangleright \tau$	polytypes

K is a set of bindings $\alpha :: (C, \exists\bar{\epsilon}.R)$, C a constraint and R a set of relations to types, describing the possible values admitted for the type α. Note that recursive types can be defined using a mutually recursive kinding environment, *i.e.* where kinds are related to each other. The set of relations $\exists\bar{\epsilon}.R$ is to be understood modulo α-conversion of the ϵ's, and polytypes modulo α-conversion of quantified type variables.

Definition 2 *A kind $k = (C, \exists\bar{\epsilon}.R)$ is well formed if*

1. *the constraint C is satisfiable.*
2. *for each $t(x,\alpha_1) \wedge t(A,\alpha_2) \wedge p(x) \Rightarrow \alpha_1 = \alpha_2$ in $\mathcal{E}$ (where $t = r$ or r^ϵ) and each $t(a,\tau_1)$ and $t([a/x]A,\tau_2)$ in R such that $C \vdash p(a)$, we have $\tau_1 = \tau_2$.*

We define kinding environments as containing only well formed kinds. Notice that all type variables do not necessarily have a kind, only those that represent constrained types do.

Free variables $\mathrm{FV}_\mathrm{K}(\sigma)$ of a polytype σ under a kinding environment K are defined as the minimum set satisfying the following equations. We write $\mathrm{FV}(\sigma)$ when K is clear from the context.

$$\begin{aligned}
\mathrm{FV}_\mathrm{K}(\forall\alpha_1\ldots\alpha_n.\mathrm{K}' \triangleright \tau) &= \mathrm{FV}_{\mathrm{K},\mathrm{K}'}(\tau) \setminus \{\alpha_1,\ldots,\alpha_n\} \\
\mathrm{FV}_{\mathrm{K},\alpha::(C,\exists\bar{\epsilon}.R)}(\alpha) &= \{\alpha\} \cup \mathrm{FV}_\mathrm{K}(R) \\
\mathrm{FV}_\mathrm{K}(u) &= \emptyset \\
\mathrm{FV}_\mathrm{K}(\tau_1 \to \tau_2) &= \mathrm{FV}_\mathrm{K}(\tau_1) \cup \mathrm{FV}_\mathrm{K}(\tau_2)
\end{aligned}$$

Our notion of free variables is an important difference with systems like HM(X). We explicitely force all free variables inside a kind to be free inside any type containing a variable of this kind. We obtain thus the property that possible instances of a type only depend of its free variables, like in traditional Hindler-Milner type systems, whereas in HM(X) constraints may include unrelated variables.

Definition 3 *A type substitution θ, extended as usual on monotypes and polytypes, is* admissible *between the kinding environments* K *and* K′, *written $\mathrm{K} \vdash \theta : \mathrm{K}'$, if for all $\alpha :: (C, \exists\bar{\epsilon}.R)$ in* K, *$\theta(\alpha)$ is a type variable α' and it satisfies the following properties.*

1. *$\alpha' :: (C', \exists\bar{\epsilon}'.R') \in \mathrm{K}'$*
2. *$C' \models C$*
3. *there is an index substitution $\eta : \bar{\epsilon} \to \bar{\epsilon}'$ such that $\eta(\theta(R)) \subseteq R'$.*

Condition 1 ensures that all kinded variables are mapped to kinded variables. Condition 2 ensures that constraints are instantiated correctly (according to entailment). Condition 3 ensures that all type constraints are kept, while making provision for changes in relation indices.

An immediate consequence of this definition is that one cannot substitute a ground type for a kinded variable. This does not mean that kinded variables always stay polymorphic: the constraint may be an exact one, which cannot be refined further. This only reflects our design choice of only changing the constraint domain, without extending the types themselves.

For both kinding environments and substitutions, we will write $\mathrm{K}|_D$ (resp. $\theta|_D$) for their restriction to variables in D, and $\mathrm{K}|_{\overline{D}}$ (resp. $\theta|_{\overline{D}}$) for their restriction to variables outside of D.

Discarding useless types The above definition of substitution requires one to keep eventually useless relational information (that cannot be matched by any propagation rule anymore). In practice, it may be useful to allow forgetting some relations, to lighten kind descriptions. However we must be careful of not dropping free type variables, as they may have an impact on which types can be made polymorphic.

Let us consider the relation $t(a,\tau)$ in R, where $t = r$ or r^ϵ. If for all propagation rule $t(x,\alpha_1) \wedge t(A,\alpha_2) \wedge p(x) \Rightarrow \alpha_1 = \alpha_2$ in $\mathcal{E}$, we have[1] $C \vdash \neg p(x)$, and moreover if $A = a$, we have $\forall y.C \vdash \neg p(y)$, then $t(a,\tau)$ will never cause propagation. We call such a $t(a,\tau)$ *inert* under propagation rules $\mathcal{E}$ and constraint C. Such an inert atom can be safely replaced by

[1] Here $\neg p$ is any observable predicate such that $\forall C.(C \vdash p(x) \wedge C \vdash \neg p(x)) \Rightarrow C \models \perp$. As an observable, it is preserved by entailment. Since constraints are to be understood under the open world hypothesis, $C \vdash \neg p(a)$ is not equivalent to $\neg C \vdash p(a)$.

a dummy $r_{fv}(\tau)$, which just makes sure that free variables are kept. Moreover, $r_{fv}(\tau)$ can be replaced by $\{r_{fv}(\alpha) \mid \alpha \in \mathrm{FV}(\tau)\}$ at any point. For this reason we can simplify R through the reflectively and transitively closed relation $\cong_{\mathcal{E}}$:

$$R \cup \{r(a,\tau)\} \cong_{C\mathcal{E}} R \cup \{r_{fv}(\tau') \mid \tau' \in T\}$$
$$\text{when } r(a,\tau) \text{ inert and } \mathrm{FV}_\emptyset(\tau) = \mathrm{FV}_\emptyset(T)$$

$$R \cong_{C\mathcal{E}} R' \Rightarrow (C, \exists\bar{\epsilon}.R) \cong_{\mathcal{E}} (C, \exists\bar{\epsilon}.R')$$

Note that $\cong_{\mathcal{E}}$ is not symmetric: this is a refinement from left to right. $\cong_{\mathcal{E}}$ commutes with substitution.

Proposition 1 (Discarding postponement) *If* $\mathrm{K}, \alpha :: k \vdash \theta : \mathrm{K}', \theta(\alpha) :: k'$ *and* $k_0 \cong_{\mathcal{E}} k$*, then there is a kind* k_0' *such* $k_0' \cong_{\mathcal{E}} k'$ *and* $\mathrm{K}, \alpha :: k_0 \vdash \theta : \mathrm{K}, \theta(\alpha) :: k_0'$.

The above proposition means that one can arbitrarily postpone discarding of useless types. For this reason we do not consider discarding in the rest of this paper, and assume it will be done before presenting types to the programmer.

5 Terms and typing rules

Expressions are the standard ones,

$$\begin{array}{rcll} e & ::= & x \mid \mathsf{fun}\ x \to e \mid e\ e & \text{core lambda} \\ & \mid & c \mid \mathsf{let}\ x = e\ \mathsf{in}\ e & \text{constants and let} \end{array}$$

Type judgments are extended with a kinding environment,

$$\mathrm{K}; \Gamma \vdash e : \tau$$

where K is a well-formed kinding environment and Γ is a set of bindings $x : \sigma$ from term variables to polytypes.

Typing rules appear in figure 1. They are in the syntax-directed style, instantiating and generalizing in one step, which simplifies the handling of eventual mutual recursion in the kinding environment. GENERALIZE[2] and LET are two rules, but they are always used together.

The extra rule CONSTANT is a variation on VARIABLE, intended to allow easy definition of extra operations through polymorphically typed constants. As in the informal presentation, the pair $\mathrm{K}_0 \triangleright \tau$ is to be understood as the polytype $\forall \mathrm{FV}_{\mathrm{K}_0}(\tau).\mathrm{K}_0 \triangleright \tau$.

The following lemma is paramount in proving the soundness of type inference.

Lemma 2 (Type substitution) *If* $\mathrm{K}, \Gamma \vdash e : \tau$ *and* $\mathrm{K} \vdash \theta : \mathrm{K}'$*, then* $\mathrm{K}', \theta(\Gamma) \vdash e : \theta(\tau)$.

For dynamic semantics, constants providing extra operations are coupled with δ-rules, of the form $A[e_1, \ldots, e_n] \to A'[e_1, \ldots, e_n]$, where A and A' are application contexts.

$$A[x_1, \ldots, x_n] ::= c \mid x_i \mid (A[x_1, \ldots, x_n]\ A[x_1, \ldots, x_n])$$

Each δ-rule should be type safe: if $\mathrm{K}_0; \Gamma \vdash A[x_1, \ldots, x_n] : \tau$, where $\Gamma = \{x_1 : \tau_1, \ldots, x_n : \tau_n\}$, then $\mathrm{K}_0; \Gamma \vdash A'[x_1, \ldots, x_n] : \tau$ must be provable.

If all the δ-rules are type safe, we can prove subject reduction in the usual way. $E[\,]$ denotes an arbitrary single hole context, not restricted to application contexts.

[2] A somehow strange case of GENERALIZE happens if K contains a variable α such that $\mathrm{FV}_{\mathrm{K}}(\alpha) \cap B \neq \emptyset$ but $\alpha \notin B$. The meaning of α in $\mathrm{K}|_{\overline{B}}$ is not the same as in K, but since then α would not be in $\mathrm{FV}_{\mathrm{K}}(\Gamma)$ either (if it were, then we would have $\mathrm{FV}_{\mathrm{K}}(\alpha) \subset \mathrm{FV}_{\mathrm{K}}(\Gamma)$, and $\mathrm{FV}_{\mathrm{K}}(\Gamma) \cap B = \emptyset$), this does not matter anyway.

Theorem 4 (Subject reduction) *If* $\mathrm{K}; \Gamma \vdash E[e] : \sigma$ *and* $e \to e'$ *by a* δ*-rule,* BETA : $((\mathit{fun}\ x \to e_1)\ e_2) \to [e_2/x]e_1$ *or* LET : $\mathit{let}\ x = e_1\ \mathit{in}\ e_2 \to [e_2/x]e_1$*, then* $\mathrm{K}; \Gamma \vdash E[e'] : \sigma$

Subject reduction is only a step on the way to type soundness (*i.e.* well-typed programs do not go wrong), the next one being to show that normal forms of programs typable in the empty environment are values [WF94]. While easy, this part is not modular (reduction rules may interact). For this reason we leave it to specific instances.

Examples Some typed constructs are given in figure 2.

For record calculi the δ-rule is

$$\mathsf{get}_{l_i}\ (\mathsf{record}_{l_1 \ldots l_n}\ e_1\ \ldots\ e_n) \to e_i.$$

For variants, which share the same constraint as records with maskable fields, we have three δ-rules. The first one is the dual of field extraction, while the two others implement polymorphic dispatch. Notice that with dispatch the functions' types are not directly connected to the input type.

$$\begin{array}{lcll} \mathsf{match}_{l_1 \ldots l_n}\ f_1\ \ldots\ f_n\ (\mathsf{tag}_{l_i}\ e) & \to & f_i\ e & \\ \mathsf{split}_{l_1 \ldots l_k, l_{k+1} \ldots l_n}\ f_1\ f_2\ (\mathsf{tag}_{l_i}\ e) & \to & f_1\ (\mathsf{tag}_{l_i}\ e) & \text{if } i \leq k \\ \mathsf{split}_{l_1 \ldots l_k, l_{k+1} \ldots l_n}\ f_1\ f_2\ (\mathsf{tag}_{l_i}\ e) & \to & f_2\ (\mathsf{tag}_{l_i}\ e) & \text{if } i > k \end{array}$$

For variants with accurate matching, we do not need to change the typing of the tag constructor; it is enough to add a new cases constructor for polymorphic functions. The δ-rule is the same as for match, only the type differs.

$$\mathsf{cases}\ _{l_1 \ldots l_n}\ f_1\ \ldots\ f_n\ (\mathsf{tag}_{l_i}\ e) \quad \to \quad f_i\ e$$

6 Unification

We define unification on monotypes through rules of the form

$$\frac{\varphi}{\varphi'}$$

where φ are preconditions and φ' conclusions. φ is a *unification problem* composed both of *kinding constraints* $\alpha :: (C, \exists\bar{\epsilon}.R)$ and *equality constraints* $\tau_1 \doteq \tau_2$ connected by $\wedge$. Both $\wedge$ and $\doteq$ are commutative, and $\wedge$ is associative. A variable may not be kinded by more than one kinding constraint. Note that the "constraints" here, along with $\wedge$, have nothing to do with the previously defined constraint domains; we just reuse a natural terminology in a different context, where constraints are purely syntactic.

An equality constraint is said to be *solved* in φ if it is of the form $\alpha \doteq \tau$ and α only appears as strict subterm in other constraints of φ. A kinding constraint is *solved* if its kind is well-formed. A unification problem φ is solved if all its constraints are solved.

A solution to a unification problem φ, whose kinding constraints form a kinding environment K (its *basis*), is a kinding environment K' and an admissible substitution $\mathrm{K} \vdash \theta : \mathrm{K}'$ such that for all equality constraint $\tau_1 \doteq \tau_2$ in φ, we have $\theta(\tau_1) = \theta(\tau_2)$.

Rules for unification are in figure 3. $sort_\varphi$ is a partial function from types to $\{u_1, \ldots, u_n, \to, k\}$, left undefined on non-kinded variables:

$$\begin{array}{rcll} sort_\varphi(u) & = & u & u \text{ a base type} \\ sort_\varphi(\tau_1 \to \tau_2) & = & \to & \\ sort_\varphi(\alpha) & = & k & \alpha :: (C, \exists\bar{\epsilon}.R) \in \varphi \end{array}$$

$$\textsc{Variable}\quad \frac{\mathrm{K}, \mathrm{K}_0 \vdash \theta : \mathrm{K} \quad \mathrm{Dom}(\theta) \subset B}{\mathrm{K}; \Gamma, x : \forall B.K_0 \triangleright \tau \vdash x : \theta(\tau)}$$

$$\textsc{Generalize}\quad \frac{\mathrm{K}; \Gamma \vdash e : \tau \quad B = \mathrm{FV}_{\mathrm{K}}(\tau) \setminus \mathrm{FV}_{\mathrm{K}}(\Gamma)}{\mathrm{K}|_{\overline{B}}; \Gamma \vdash e : \forall B.\mathrm{K}|_B \triangleright \tau}$$

$$\textsc{Abstraction}\quad \frac{\mathrm{K}; \Gamma, x : \tau \vdash e : \tau'}{\mathrm{K}; \Gamma \vdash \mathsf{fun}\ x \rightarrow e : \tau \rightarrow \tau'}$$

$$\textsc{Let}\quad \frac{\mathrm{K}; \Gamma \vdash e_1 : \sigma \quad \mathrm{K}; \Gamma, x : \sigma \vdash e_2 : \tau}{\mathrm{K}; \Gamma \vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : \tau}$$

$$\textsc{Application}\quad \frac{\mathrm{K}; \Gamma \vdash e_1 : \tau \rightarrow \tau' \quad \mathrm{K}, \Gamma \vdash e_2 : \tau}{\mathrm{K}; \Gamma \vdash e_1\ e_2 : \tau'}$$

$$\textsc{Constant}\quad \frac{\mathrm{K}_0 \vdash \theta : \mathrm{K} \quad \mathrm{Tconst}(c) = \mathrm{K}_0 \triangleright \tau}{\mathrm{K}; \Gamma \vdash c : \theta(\tau)}$$

Figure 1: Typing rules

Records à la *Ohori*

$$\begin{aligned}
\mathrm{Tconst}(\mathsf{record}_{l_1 \ldots l_n}) &= \alpha_0 :: ((\{l_1, ..., l_n\}, 1), \{r(l_1, \alpha_1), ..., r(l_n, \alpha_n)\}) \\
&\quad \triangleright \alpha_1 \rightarrow \ldots \rightarrow \alpha_n \rightarrow \alpha_0 \\
\mathrm{Tconst}(\mathsf{get}_l) &= \alpha_0 :: ((\{l\}, 0), \{r(l, \alpha_1)\}) \triangleright \alpha_0 \rightarrow \alpha_1
\end{aligned}$$

Records with maskable fields

$$\begin{aligned}
\mathrm{Tconst}(\mathsf{record}_{l_1 \ldots l_n}) &= \alpha_0 :: ((\emptyset, \{l_1, ..., l_n\}), \{r(l_1, \alpha_1), ..., r(l_n, \alpha_n)\}) \\
&\quad \triangleright \alpha_1 \rightarrow \ldots \rightarrow \alpha_n \rightarrow \alpha_0 \\
\mathrm{Tconst}(\mathsf{get}_l) &= \alpha_0 :: ((\{l\}, \mathcal{L}), \{r(l, \alpha_1)\}) \triangleright \alpha_0 \rightarrow \alpha_1
\end{aligned}$$

Variants with maskable constructors

$$\begin{aligned}
\mathrm{Tconst}(\mathsf{tag}_l) &= \alpha_0 :: ((\{l\}, \mathcal{L}), \{r(l, \alpha_1)\}) \triangleright \alpha_1 \rightarrow \alpha_0 \\
\mathrm{Tconst}(\mathsf{match}_{l_1 \ldots l_n}) &= \alpha_0 :: ((\emptyset, \{l_1, ..., l_n\}), \{r(l_1, \alpha_1), ..., r(l_n, \alpha_n)\}) \\
&\quad \triangleright (\alpha_1 \rightarrow \alpha) \rightarrow \ldots \rightarrow (\alpha_n \rightarrow \alpha) \rightarrow \alpha_0 \rightarrow \alpha \\
\mathrm{Tconst}(\mathsf{split}_{l_1 \ldots l_k, l_{k+1} \ldots l_n}) &= \alpha' :: ((\{l_1, ..., l_k\}, \mathcal{L}), \{r(l_1, \alpha_1), ..., r(l_k, \alpha_k)\}), \\
&\quad \alpha'' :: ((\{l_{k+1}, ..., l_n\}, \mathcal{L}), \{r(l_{k+1}, \alpha_{k+1}), ..., r(l_n, \alpha_n)\}), \\
&\quad \alpha_0 :: ((\emptyset, \{l_1, ..., l_n\}), \{r(l_1, \alpha_1), ..., r(l_n, \alpha_n)\}) \\
&\quad \triangleright (\alpha' \rightarrow \alpha) \rightarrow (\alpha'' \rightarrow \alpha) \rightarrow \alpha_0 \rightarrow \alpha
\end{aligned}$$

Accurate matching

$$\begin{aligned}
\mathrm{Tconst}(\mathsf{cases}\ _{l_1 \ldots l_n}) &= \alpha_0 :: ((\emptyset, \{l_1, ..., l_n\}), \exists \epsilon.\{r(l_1, \alpha_1), ..., r(l_n, \alpha_n), \\
&\qquad v^{\epsilon}(l_1, \alpha'_1), ..., v^{\epsilon}(l_n, \alpha'_n), v^{\epsilon}(\mathsf{return}, \alpha)\}) \\
&\quad \triangleright (\alpha_1 \rightarrow \alpha'_1) \rightarrow \ldots \rightarrow (\alpha_n \rightarrow \alpha'_n) \rightarrow \alpha_0 \rightarrow \alpha
\end{aligned}$$

Figure 2: Examples of constructs with their associated constraints

$$\textsc{Incompatible}\quad \frac{\varphi \wedge \tau_1 \doteq \tau_2}{\bot}\ \text{when } sort_{\varphi}(\tau_1) \neq sort_{\varphi}(\tau_2)$$

$$\textsc{Redundancy}\quad \frac{\varphi \wedge \tau \doteq \tau}{\varphi}$$

$$\textsc{Cyclic}\quad \frac{\varphi \wedge \alpha \doteq \tau}{\bot}\ \text{when } \alpha \neq \tau \text{ and } \alpha \in \mathrm{FV}_{\emptyset}(\tau)$$

$$\textsc{Function}\quad \frac{\varphi \wedge \tau_1 \rightarrow \tau_2 \doteq \tau'_1 \rightarrow \tau'_2}{\varphi \wedge \tau_1 \doteq \tau'_1 \wedge \tau_2 \doteq \tau'_2}$$

$$\textsc{Substitution}\quad \frac{\varphi \wedge \alpha \doteq \tau}{\varphi[\tau/\alpha] \wedge \alpha \doteq \tau}\ \begin{array}{l}\text{when } \alpha :: (C, \exists \bar{\epsilon}.R) \notin \varphi \text{ and } \alpha \notin \mathrm{FV}_{\emptyset}(\tau) \\ \quad \text{and } \alpha \in \mathrm{FV}(\varphi) \text{ and } \tau \neq \beta \vee \beta \in \mathrm{FV}(\varphi)\end{array}$$

$$\textsc{Bad constraint}\quad \frac{\varphi \wedge \alpha :: (C, \exists \bar{\epsilon}.R)}{\bot}\ \text{when } C \models \bot$$

$$\textsc{Constraint}\quad \frac{\varphi \wedge \alpha_1 :: (C_1, \exists \bar{\epsilon}_1.R_1) \wedge \alpha_2 :: (C_2, \exists \bar{\epsilon}_2.R_2) \wedge \alpha_1 \doteq \alpha_2}{\varphi \wedge \alpha :: (C_1 \wedge C_2, \exists \bar{\epsilon}_1 \bar{\epsilon}_2.R_1 \cup R_2) \wedge \alpha_1 \doteq \alpha \wedge \alpha_2 \doteq \alpha}\ \alpha \text{ fresh},\ \bar{\epsilon}_1 \cap \bar{\epsilon}_2 = \emptyset$$

$$\textsc{Propagation}\quad \begin{array}{l} t(x, \alpha_1) \wedge t(A, \alpha_2) \wedge p(x) \Rightarrow \alpha_1 = \alpha_2 \in \mathcal{E} \qquad t = r \text{ or } r^{\epsilon} \\ \dfrac{\varphi \wedge \alpha :: (C, \exists \bar{\epsilon}.R) \quad t(a, \tau_1) \in R \quad r([a/x]A, \tau_2) \in R \quad C \vdash p(a)}{\varphi \wedge \alpha :: (C, \exists \bar{\epsilon}.R) \wedge \tau_1 \doteq \tau_2}\ \text{when } \tau_1 \neq \tau_2 \end{array}$$

Figure 3: Rewriting rules for unification

$$
\begin{array}{rcl}
\mathrm{K};\Gamma \triangleright e : \tau & = & (\Gamma \triangleright e : \tau) \wedge \mathrm{K} \\
\Gamma, x : (\forall B.\mathrm{K} \triangleright \tau_1) \triangleright x : \tau & = & \tau \doteq \theta(\tau_1) \wedge \bigwedge_{\alpha_i :: (C_i, \exists \bar{\epsilon}_i . R_i) \in \mathrm{K}} \beta_i :: (C_i, \exists \bar{\epsilon}_i . \theta(R_i)) \\
 & & \text{where } B = \{\alpha_1, \ldots, \alpha_n\} \text{ and } \theta = \{\alpha_1 \mapsto \beta_1, \ldots, \alpha_n \mapsto \beta_n\} \\
\Gamma \triangleright \mathsf{fun}\ x \rightarrow e : \tau & = & (\Gamma, x : \alpha_1 \triangleright e : \alpha_2) \wedge \tau \doteq \alpha_1 \rightarrow \alpha_2 \\
\Gamma \triangleright e_1\ e_2 : \tau & = & (\Gamma \triangleright e_1 : \alpha \rightarrow \tau) \wedge (\Gamma \triangleright e_2 : \alpha) \\
\Gamma \triangleright \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : \tau & = & \varphi|_{\overline{B}} \wedge (\Gamma, x : (\forall B.\mathrm{K}|_B \triangleright \theta(\alpha)) \triangleright e_2 : \tau) \\
 & & \text{where } \varphi = (\Gamma \triangleright e_1 : \alpha) \text{ and } K \vdash \theta : K = \mathrm{solve}(\varphi) \\
 & & \quad \text{and } B = \mathrm{FV}_{\mathrm{K}}(\theta(\alpha)) \setminus \mathrm{FV}_{\mathrm{K}}(\theta(\Gamma)) \\
\Gamma \triangleright c : \tau & = & \tau \doteq \theta(\tau_0) \wedge \bigwedge_{\alpha_i :: (C_i, \exists \bar{\epsilon}_i . R_i) \in \mathrm{K}_0} \beta_i :: (C_i, \exists \bar{\epsilon}_i . \theta(R_i)) \\
 & & \text{where } \mathrm{Tconst}(c) = \mathrm{K}_0 \triangleright \tau_0,\ \mathrm{FV}_{\mathrm{K}_0}(\tau_0) = \{\alpha_1, \ldots, \alpha_n\} \\
 & & \quad \text{and } \theta = \{\alpha_1 \mapsto \beta_1, \ldots, \alpha_n \mapsto \beta_n\}
\end{array}
$$

Figure 4: Reconstruction algorithm

Proposition 5 *Rewriting a unification problem leads either to $\perp$ or to a solved problem.*

From a solved unification problem φ one can directly read a pair of a kinding environment K (the kinding constraints) and a substitution θ (the solved equality constraints), such that $\mathrm{K} \vdash \theta : \mathrm{K}$.

A substitution $\mathrm{K} \vdash \theta_1 : \mathrm{K}_1$ is said to be more general than $\mathrm{K} \vdash \theta_2 : \mathrm{K}_2$ if there is a substitution $\mathrm{K}_1 \vdash \theta : \mathrm{K}_2$ such that $\theta_2 = \theta \circ \theta_1$.

Proposition 6 *The solution $\mathrm{K}' \vdash \theta : \mathrm{K}'$ read from the solved form obtained by rewriting a unification problem φ of basis K is also $\mathrm{K} \vdash \theta : \mathrm{K}'$, and it is the most general unifier for this problem.*

7 Type reconstruction

Type reconstruction is done by translating a typing problem $\mathrm{K};\Gamma \triangleright e : \tau$ into a unification problem φ, and solving it to obtain a substitution $\mathrm{K} \vdash \theta : \mathrm{K}'$.

The algorithm in figure 4 does not depend on a specific evaluation order, equality and kinding constraints only needing to be solved eagerly for let nodes. The function *solve*(φ) extracts an admissible substitution from a normal form of φ.

A solution to a typing problem $\mathrm{K};\Gamma \triangleright e : \tau$ is a substitution $\mathrm{K} \vdash \theta : \mathrm{K}'$ such that $\mathrm{K}';\theta(\Gamma) \vdash e : \theta(\tau)$ is derivable.

Theorem 7 *If $\mathrm{K};\Gamma \triangleright e : \tau$ can be reduced to $\mathrm{K} \vdash \theta : \mathrm{K}'$ by the type reconstruction algorithm, $\mathrm{K}';\theta(\Gamma) \vdash e : \theta(\tau)$ is derivable, and θ is the most general solution; otherwise it reduces to $\perp$ and there is no solution.*

8 Conclusion

We proposed a type system where constraints are local to type nodes, and showed how it can be applied to many type inference problems, which fit between usual parametric polymorphism and systems requiring the power of subtyping. Its simplicity, and the unperspicuous way in which it extends the original type inference algorithm, make it a good candidate for practical implementations and further extensions.

While we think that this system has a wide range of applicability, as is shown by a new structural encoding for accurate matching, it does not cover the full range of structural polymorphism. For instance, Rémy's row and presence variables provide for slightly more precise typing than the encoding we give. Similarly Kennedy's dimension types use a multi-sorted algebra [Ken94]. Nonetheless, if we restrict sharing in types to one single sort of type variables, as is done in Objective Caml's type system (which is itself based on Rémy's work), then we come back into the expressive area of local constraints.

An interesting conjecture is whether local constraints can handle all one-sorted ML-like type systems without the "power of subtyping". This would confirm the intuition that there is a qualitative jump when going from systems without subtyping to systems which allow it. In particular, local constraints fail to encode simply typed dynamic messages or record concatenation [Wan91], correctly showing that, while this is not immediately explicit, all these systems require some form of "subtyping" or multi-parameter constraints.

Ironically the last counter-example standing against a characterization of structural polymorphism by local constraints seems to be the label-selective lambda-calculus and its type system [GAK94]. Function types are of the form $l{:}\tau \rightarrow \tau$, modulo the congruence $l_1{:}\tau_1 \rightarrow l_2{:}\tau_2 \rightarrow \tau = l_2{:}\tau_2 \rightarrow l_1{:}\tau_1 \rightarrow \tau$. Clearly, this bears no relation to subtyping, as shown by the existence of a typing algorithm without constraints, yet one has to separate $l_1{:}\tau_1 \rightarrow \tau$ from τ, which is not possible with the local constraints presented in this paper. The intuitive encoding would be to describe function types as kinds containing all applicable labels, and add a propagation rule of the form $l_1 \neq l_2 \wedge r(l_1, \alpha_1) \wedge r(l_2, \alpha_2) \Rightarrow \alpha_1 = \alpha' \wedge \alpha' :: (\{l_2\}, \{r(l_2, \alpha_2)\})$, introducing equalities with newly built types. Not only does it break our proof of termination for unification, but in general this form of propagation rules could be used to encode subtyping. This suggests that label-selective lambda-calculus requires some form of non-local relations, which is coherent with the presence of a fundamentally non-local requirement in the COMPLETION rule of its unification algorithm [FG95], that the return type variable after an arbitrary number of arrows should differ (this requirement was missing in [GAK94]). Actually finding what are these non-local relations may be an interesting topic for further work, as the same phenomenon appears in systems with explicit row variables [Rém89, Pot00].

References

[AKPS94] Hassan Aït-Kaci, Andreas Podelski, and Gert Smolka. A feature-based constraint system for logic programming with entailment. *Theoretical Computer Science*, 122(1–2):263–283, January 1994.

[AWL94] Alexander Aiken, Edward L. Wimmers, and T. K. Lakshman. Soft typing with conditional types. In *Proc. ACM Symposium on Principles of Programming Languages*, pages 163–173, Portland, Oregon, January 1994.

[DM82] L. Damas and R. Milner. Principal type-schemes for functional programs. In *Proc. ACM Symposium on Principles of Programming Languages*, pages 207–212, 1982.

[FG95] Jun P. Furuse and Jacques Garrigue. A label-selective lambda-calculus with optional arguments and its compilation method. RIMS Preprint 1041, Research Institute for Mathematical Sciences, Kyoto University, October 1995.

[GAK94] Jacques Garrigue and Hassan Aït-Kaci. The typed polymorphic label-selective λ-calculus. In *Proc. ACM Symposium on Principles of Programming Languages*, pages 35–47, 1994.

[Gar98] Jacques Garrigue. Programming with polymorphic variants. In *ML Workshop*, Baltimore, September 1998.

[GR99] Jacques Garrigue and Didier Rémy. Extending ML with semi-explicit higher order polymorphism. *Information and Computation*, 155:134–171, December 1999.

[Ken94] A. J. Kennedy. Dimension types. In *Proceedings of the 5th European Symposium on Programming*, volume 788 of *Lecture Notes in Computer Science*, pages 348–362. Springer-Verlag, 1994.

[LDG+00] Xavier Leroy, Damien Doligez, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. *The Objective Caml system release 3.00, Documentation and user's manual*. Projet Cristal, INRIA, April 2000.

[Mil78] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

[MN00] Martin Müller and Susumu Nishimura. Type inference for first-class messages with feature constraints. *International Journal of Foundations of Computer Science*, 11(1):29–63, 2000.

[Nis98] Susumu Nishimura. Static typing for dynamic messages. In *Proc. ACM Symposium on Principles of Programming Languages*, San Diego, California, January 1998.

[Oho95] Atsushi Ohori. A polymorphic record calculus and its compilation. *ACM Transactions on Programming Languages and Systems*, 17(6):844–895, November 1995. Long version of a POPL'92 paper.

[OSW99] Martin Odersky, Martin Sulzmann, and Martin Wehr. Type inference with constrained types. *Theory and Practice of Object Systems*, 5(1):35–55, 1999.

[Pot00] François Pottier. A versatile constraint-based type inference system. *Nordic Journal of Computing*, 7(4):312–347, November 2000.

[Rém89] Didier Rémy. Typechecking records and variants in a natural extension of ML. In *Proc. ACM Symposium on Principles of Programming Languages*, pages 77–87, 1989.

[Rém93] 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, 1993.

[Sul00] Martin Sulzmann. *A General Framework for Hindley/Milner Type Systems with Constraints*. PhD thesis, Yale University, Department of Computer Science, May 2000.

[Wan91] Mitchell Wand. Type inference for record concatenation and multiple inheritance. *Information and Computation*, 93:1–15, 1991.

[WF94] Andrew Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, November 1994.

A Proofs of theorems

Proposition 1 (Discarding postponement) *If* $\mathrm{K}, \alpha :: k \vdash \theta : \mathrm{K}', \theta(\alpha) :: k'$ *and* $k_0 \cong_\mathcal{E} k$, *then there is a kind* k_0' *such* $k_0' \cong_\mathcal{E} k'$ *and* $\mathrm{K}, \alpha :: k_0 \vdash \theta : \mathrm{K}, \theta(\alpha) :: k_0'$.

Proof Suppose $k_0 = (C, \exists\bar\epsilon.R \cup R_0)$ and $k = (C, \exists\bar\epsilon.R \cup R_1)$, with R_0 and R_1 minimal. By definition of $\cong_\mathcal{E}$, all $r(a, \tau)$ in R_0 are inert for C. By definition of admissibility, $k' = (C', \exists\bar\epsilon.R')$, with $C' \models C$, so that all elements of R_0 are also inert for C'; and $\theta(R_1) \subset R'$, so that $k_0' = (C', \exists\bar\epsilon.R' \cup \theta(R_0)) \cong_\mathcal{E} (C', \exists\bar\epsilon.R' \cup \theta(R_1)) = k'$. □

Lemma 2 (Type substitution) *If* $\mathrm{K}, \Gamma \vdash e : \tau$ *and* $\mathrm{K} \vdash \theta : \mathrm{K}'$, *then* $\mathrm{K}', \theta(\Gamma) \vdash e : \theta(\tau)$.

Proof Induction on the derivation of $\mathrm{K}, \Gamma \vdash e : \tau$.

The two important steps are Variable and Generalize. For Variable, we have $\mathrm{K}, \mathrm{K}_0 \vdash \theta_0 : \mathrm{K}$ and $\mathrm{Dom}(\theta_0) \subset B$; we can choose B outside of θ and K': $\forall \alpha \in B$, $\theta(\alpha) = \alpha$, and $\mathrm{FV}(\mathrm{K}') \cap B = \emptyset$. By composition, we construct $\theta_1 = (\theta \circ \theta_0)|_B$, such that $\mathrm{K}', \theta(\mathrm{K}_0) \vdash \theta_1 : \mathrm{K}'$: conditions (1) and (2) are satisfied by transitivity. We check condition (3) for kinded variables:

- If $\alpha \notin B$, then $\theta_1(\alpha) = \alpha$, $\alpha :: (C, \exists\bar\epsilon.R) \in \mathrm{K}'$ and $\mathrm{FV}_\emptyset(R) \cap B = \emptyset$, so that $\theta_1(R) = R$.

- If $\alpha \in B$, then $\theta_1(\alpha) = \theta(\theta_0(\alpha))$, and $\alpha :: (C, \exists\bar{\epsilon}.R) \in \mathrm{K}_0$. For any variable $\alpha' \in \mathrm{FV}_\emptyset(R)$, we have $\theta_1(\theta(\alpha')) = \theta(\theta_0(\alpha'))$: either $\alpha' \in B$, and $\theta(\alpha') = \alpha'$, so that $\theta_1(\theta(\alpha')) = \theta_1(\alpha') = \theta(\theta_0(\alpha'))$, or $\alpha' \notin B$, and $\theta_0(\alpha') = \alpha'$, giving $\theta_1(\theta(\alpha')) = \theta(\alpha') = \theta(\theta_0(\alpha'))$. Combined with $\theta_1(\alpha) :: (C', \exists\bar{\epsilon}.R') \in \mathrm{K}'$ implying $\theta(\theta_0(R)) \subset R'$, we obtain $\theta_1(\theta(R)) \subset R'$, giving condition (3) since $\alpha :: (C, \exists\bar{\epsilon}.\theta(R)) \in \theta(\mathrm{K}_0)$.

Note also that $\theta_1 \circ \theta|_{\overline{B}} = \theta \circ \theta_0$. Then we can deduce $\mathrm{K}'; \theta(\Gamma), x : \forall B, \theta(\mathrm{K}_0) \triangleright \theta(\tau) \vdash x : \theta(\theta_0(\tau))$.

For Generalize, again we choose B outside of θ and K'. Then $\mathrm{K}|_{\overline{B}}, \theta(\mathrm{K}|_B) \vdash \theta : \mathrm{K}', \theta(\mathrm{K}|_B)$. By induction hypothesis we have $\mathrm{K}', \theta(\mathrm{K}|_B); \theta(\Gamma) \vdash e : \theta(\tau)$. Then we can deduce $\mathrm{K}'; \theta(\Gamma) \vdash e : \forall B.\theta(\mathrm{K}|_B) \triangleright \theta(\tau)$. □

Lemma 3 (Term substitution) *If $\mathrm{K}; \Gamma \vdash e : \sigma$ and $\mathrm{K}; \Gamma, x : \sigma \vdash e' : \tau'$, then $\mathrm{K}; \Gamma \vdash e'[e/x] : \tau'$.*

Proof Easy induction on the derivation tree. When grafting Generalize somewhere, use a type substitution $\mathrm{K} \vdash \theta : \mathrm{K}|_{\overline{B}}$. When substituting under Generalize, B may grow. In such case, preserve $\mathrm{K}|_{\overline{B}}$ by duplicating the kind under a different name, and instantiate extra variables with Variable. □

Theorem 4 (Subject reduction) *If $\mathrm{K}; \Gamma \vdash E[e] : \sigma$ and $e \to e'$ by a δ-rule,* Beta : $((\mathit{fun}\ x \to e_1)\ e_2) \to [e_2/x]e_1$ *or* Let : $\mathit{let}\ x = e_1\ \mathit{in}\ e_2 \to [e_2/x]e_1$, *then $\mathrm{K}; \Gamma \vdash E[e'] : \sigma$*

Proof First we consider the case when $E[e] = e$. If $e \to e'$ is the instance of the δ-rule $A[x_1, \ldots, x_n] \to A'[x_1, \ldots, x_n]$, then, supposing $e_1 : \tau_2, ..., e_n : \tau_n$, we have already a derivation of $\mathrm{K}_0; x_1 : \tau_1, ..., x_n : \tau_n \vdash e'[x_1, ..., x_n] : \tau$. We conclude by term substitution. Similarly β-reduction and let-reduction are immediate consequences of term substitution.

If the redex occurs inside a deeper context, extend the proof by induction on the depth of the context. □

Proposition 5 *Rewriting a unification problem leads either to $\bot$ or to a solved problem.*

Proof First, we verify that unification terminates. One is allowed to apply rules in any order, except that (1) failure is eager (*i.e.* Incompatible, Cyclic and Bad constraint are given highest priority), and (2) Propagation only applies when no other rule applies (*i.e.* it has lowest priority).

Our measure is a lexicographical ordering on the tuple *(number of kind constraints, number of pending propagations, number of unsolved variables, number of arrows, number of equality constraints).* Constraint reduces the number of kind constraints by 1, while possibly increasing the number of pending propagations. Propagation eventually reduces the number of pending propagations (if substitutions make τ_1 equal to τ_2) or the number of kind constraints (if it causes a constraint unification); no new propagation will occur until one of these happen. Substitution reduces the number of unsolved variables. Function reduces the number of arrows. Redundancy reduces the number of equations.

Then we check that reduction cannot be stuck. If a constraint is stuck, one of its members is not solved. If this is an equality constraint, either Substitution applies, or it is $\alpha \doteq \tau[\alpha]$ and Cyclic applies, or one member is a kinded variable and either Constraint or Incompatible applies, or both are not type variables and one of Incompatible, Redundancy or Function applies. If this is a kinding constraint, then either the constraint is not satisfiable and Bad constraint applies, or a propagation equation is not satisfied, and since other rules do not apply, Propagation applies. □

Proposition 6 *The solution $\mathrm{K}' \vdash \theta : \mathrm{K}'$ read from the solved form obtained by rewriting a unification problem φ of basis K is also $\mathrm{K} \vdash \theta : \mathrm{K}'$, and it is the most general unifier for this problem.*

Proof For each rewriting rule, we verify that it is sound and complete. Together with soundness, we also check changes in kindings were necessary.

Rules Incompatible, Cyclic, Redundancy and Function are immediate: the upper and lower constraints accept the same solutions.

For Substitution, soundness is clear: $\theta(\alpha) = \theta(\tau)$ then if θ is a solution of $\varphi[\tau/\alpha]$, it is also a solution of φ. Looking at kindings, let K_1 be the basis for φ, and K_2 the basis for $\varphi[\tau/\alpha]$. We have $\mathrm{K}_1 \vdash \{\alpha \mapsto \tau\} : \mathrm{K}_2$, and $\mathrm{K}_2 \vdash \theta : \mathrm{K}'$, since $\theta(\alpha) = \theta(\tau)$, $\theta \circ \{\alpha \mapsto \tau\} = \theta$, and $\mathrm{K}_1 \vdash \theta : \tau$. For completeness, the side-conditions guarantee that α has no kind (reduction rules only introduce kinds for fresh variables). Then it is safe to replace α by τ in φ, and the same solutions are accepted.

For Bad constraint, no substitution can apply to α, so that the constraint is equivalent to $\bot$.

For Constraint, if we have a solution $\mathrm{K} \vdash \theta : \mathrm{K}'$ of the upper side, then it has to map both α_1 and α_2 to $\alpha' :: (C', \exists\bar{\epsilon}'.R')$, with $C' \models C_1$ and $C' \models C_2$. Since $C_1 \wedge C_2$ is the weakest constraint implying both C_1 and C_2, we can extend θ into $\mathrm{K}, \alpha :: (C_1 \wedge C_2, \exists\bar{\epsilon}_1\bar{\epsilon}_2.R_1 \cup R_2) \vdash \theta\{\alpha \mapsto \alpha'\} : \mathrm{K}'$, and it is a solution to the lower side ($\theta(\alpha) = \theta(\alpha_1) = \theta(\alpha_2) = \alpha'$), showing completeness. Reciprocally, with K_1 and K_2 bases for the upper and lower sides, if we have a solution $\mathrm{K}_2 \vdash \theta : \mathrm{K}'$ of the lower side, then $\mathrm{K}_1 \vdash \{\alpha_1 \mapsto \alpha, \alpha_2 \mapsto \alpha\} : \mathrm{K}_2$, and since $\theta(\alpha) = \theta(\alpha_1) = \theta(\alpha_2)$, $\theta \circ \{\alpha_1 \mapsto \alpha, \alpha_2 \mapsto \alpha\} = \theta$, and $\mathrm{K}_1 \vdash \theta : \mathrm{K}'$ is a solution to the upper side.

For Propagation, soundness is immediate (the lower side subsumes the upper side). Any solution $\mathrm{K} \vdash \theta : \mathrm{K}'$ of the upper side must have $\theta(\tau_1) = \theta(\tau_2)$ to be admissible, so it is also complete.

Soundness shows that the solution $\mathrm{K}' \vdash \theta : \mathrm{K}'$ we read from a solved unification problem, is also a solution $\mathrm{K} \vdash \theta : \mathrm{K}'$ to the original problem. Thanks to completeness, we know that any other solution $\mathrm{K} \vdash \theta_1 : \mathrm{K}''$ of the original problem can be extended in a solution $\mathrm{K}' \vdash \theta_2 : \mathrm{K}''$ of the solved problem. Since θ_2 satisfies all the solved constraints, there is a $\mathrm{K}' \vdash \theta_3 : \mathrm{K}''$ such that $\theta_2 = \theta_3 \circ \theta$, and, when restricted to the original domain, $\theta_1 = (\theta_3 \circ \theta)|_{\mathrm{FV}(\varphi)}$. So θ is the most general unifier. □

Theorem 7 *If $\mathrm{K}; \Gamma \triangleright e : \tau$ can be reduced to $\mathrm{K} \vdash \theta : \mathrm{K}'$ by the type reconstruction algorithm, $\mathrm{K}'; \theta(\Gamma) \vdash e : \theta(\tau)$ is derivable, and θ is the most general solution; otherwise it reduces to $\bot$ and there is no solution.*

Proof We verify by induction on the structure of e that an inference problem is translated into an equivalent unification problem.

The first case only splits a complete problem in its kinding environment and typing problem.

The second case handles the VARIABLE rule. Kinded β_i's ensure soundness: if $\mathrm{K}' = \{\beta_i :: (C_i, \exists\bar{\epsilon}_i.\theta(R_i))\}_{\alpha_i::(C_i,\exists\bar{\epsilon}_i.R_i)\in\mathrm{K}}$, then $\mathrm{K} \vdash \theta : \mathrm{K}'$, so that $\mathrm{K}', \mathrm{K} \vdash \theta : \mathrm{K}'$ (since $\theta|_{\mathrm{Dom}(\mathrm{K}')} = id$), and any solution $\mathrm{K}' \vdash \theta' : \mathrm{K}''$ of the right hand side is such that $\mathrm{K}'', \mathrm{K} \vdash (\theta' \circ \theta)|_B : \mathrm{K}''$; this proves $\mathrm{K}''; \theta'(\Gamma), x : \forall B.\theta'(\mathrm{K}) \triangleright \theta'(\tau_1) \vdash x : (\theta' \circ \theta)|_B(\theta'(\tau_1))$, and $(\theta' \circ \theta)|_B(\theta'(\tau_1)) = \theta'(\theta(\tau_1)) = \theta'(\tau)$ gives the left hand side. For completeness, suppose θ' is a solution to the left hand side. Then $\theta'|_B$ is the instantiation substitution. We can extend θ' in $\theta_1 = \theta' \circ \theta^{-1}$, since β_i's are fresh, and θ_1 is still a solution of the left hand side. θ_1 is also a solution of the right hand side.

ABSTRACTION and APPLICATION are as in ML.

The LET rule requires to solve constraints on e_1 before proceeding. This corresponds to generating a derivation, and adding the GENERALIZE step. The result is then used to type e_2. Information related to variables of B is discarded from the global constraint as superfluous. This is sound as constraints were already solved once, and cannot be refined as these variables are not accessible through the environment. Completeness requires considering the case when we do not use the GENERALIZE step on e_1. Generalizable variables are kept as free variables. Then they cannot be generalized when typing e_2, since they are in the environment. Since this is equivalent to choosing $\theta = id$ in the corresponding VARIABLE rules, and the VARIABLE case is complete, LET is complete.

CONSTANT is similar to VARIABLE.

Since we already proved rewriting on unification problems to terminate, type inference terminates with either a solution or $\perp$. □