Simple Type Inference for Structural Polymorphism†

Jacques Garrigue
Research Institute for Mathematical Sciences
Kyoto University, 606-8502 Kyoto, JAPAN
garrigue@kurims.kyoto-u.ac.jp

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.

Even though our system is notably weaker than general constraint-based type systems, making it unable to handle subtyping, for instance, it is powerful enough to accommodate many features from simple polymorphic records à la Obori 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 [14, 17] to polymorphic variants [16, 5] and objects with first-class (dynamic) messages [12]. All these systems were proposed as extensions of the Hindley-Milner type system [10], with growing degrees of complexity.

A recent trend is to simplify these type systems by moving to a constraint-based framework [13, 11, 18, 15]. 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 [7]; 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 accommodate 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 [3]. 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 [9] 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 ungradable 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 Obori’s polymorphic records [14], 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. Section 8 studies in more detail the differences

†A preliminary version of this paper has been presented at the 9th FOOL Workshop, Portland, OR, January 2002.
with HM(X). 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 grand scale of increasing complexity. In order to uniformize the presentation, we write K→→ for types, where K is a kinding environment, containing constraints for individual type variables, under which the open type γ is to be understood. We use a few built-in functions, to obtain interesting types: integer addition +, string concatenation ↑, conversions string_of_int: string → int and float: int → float.

2.1 Records à la O'Hori

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

```plaintext
{ name = "Jacques", age = 30 } : { name : string, age : int }

fun x → x.age + 1 : a → int
fun x → x.name ↑ " " ↑ string_of_int x.age
   : a → { name : string, age : int } v a → string
```

Intuitively, the kinding a: {age: int} means that a 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 O'Hori's. Fitting it inside the local constraint framework will require a small change in presentation, without changing expressive power.

2.2 Records and variants with masking

O'Hori'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 parts 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 [16], but here we use a more intuitive formalism [5], which is also closer to O'Hori's.

```plaintext
{ name = "Jacques", age = 30 } : a → { name : string, age : int, weight = 13 }

let l1 = [{ name = "Jacques", age = 30 },
          { name = "Serge", weight = 13 }]

l1 : a → [ { name : string, age : int, weight : int }, 0 ]
```

Kinds are now represented by a triple (T, L, U). Along with the type of each field, we have two sets of labels. Required labels (union of all field accesses, a lower bound) form a subset of available labels (intersection of the possible record values, an upper bound). 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 a: ( { name : string, age : int, { age, name, age } } ) is an acceptable description for both { name = "Jacques", age = 30 }, which makes name and age available, and fun x → x.age + 1, which only requires age. The need for two distinct sets of labels stems from the natural appearance 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.

```plaintext
Number(5) : a : { { Number : int }, { Number }, L } v a
let l2 = [Number(5), Face("E63")]

I2 : a : { { Number : int, Face : string }, { Number }, L } v a
```

Our two sets of labels have new 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 f1, 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.

```plaintext
let f2 = fun g → h → function (A|B) a x → g x
     | (C|D) a x → h x

f2 : a1 : [(A : A1, B : a3), { A, B, L }],
   a2 : [(C : ac, D : ad), { C, D, L }],
   a3 : [(A : A1, B : a3, C : ac, D : ad), 0, { A, B, C, D }]
   v (a1 → a) → (a2 → a) → a3 → a
```

Our choice of keeping all constraints local (constraining only one variable) makes our system slightly weaker than Rémy's, which has rows and presence variables. For instance we have no way to relate the set of required cases in a1 and a3, which forces us to make the safe assumption { A, B } in a1. Rémy's system handles requirements 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.

2.3 Discarding masked types

One may wonder about why one should keep all field types in the type of l1, 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:

```plaintext
let l3 = [{ name = "Jacques", age = 30 },
          { name = "Rachel", age = 6.5 }]
```

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 \texttt{age} in \texttt{l1}, we would be able to type \texttt{i3: \{name = "pachel", age = 6.5\}}, which contains the untypeable \texttt{l3} as a sub-list.

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 \texttt{l3}.

\begin{align*}
\texttt{l3 : \alpha ::= (\{name : string, age : int\& float\}, \emptyset, \{name, age\})}
\end{align*}

Since the two types for the field \texttt{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 300. This allows principal type inference of \texttt{f3} for the following case, solving the long lasting problem of “masked but not discarded” types.

\begin{align*}
\texttt{let f3 = function Number(n) \rightarrow n \mid Face() \rightarrow 15} \\
\texttt{f3 : \alpha ::= (\{Number : int, Face : unit\}, \emptyset, \{Number, Face\})}
\end{align*}

\begin{align*}
\texttt{\triangleright \alpha \rightarrow int}
\end{align*}

\begin{align*}
\texttt{let f3 = function Number(n) \rightarrow n/2} \\
\texttt{f3 : \alpha ::= (\{Number : int\}, \emptyset, \{Number\})}
\end{align*}

\begin{align*}
\texttt{\triangleright \alpha \rightarrow int}
\end{align*}

\begin{align*}
\texttt{let f3 = fun x \rightarrow (f1(x), f3(x), f4(x))} \\
\texttt{f3 : \alpha ::= (\{Number : int\}, \emptyset, \{Number\})}
\end{align*}

\begin{align*}
\texttt{\triangleright \alpha \rightarrow (\text{string} \times \text{int} \times \text{int})}
\end{align*}

2.4 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 [1]. Potter reformulated it as an instance of the HM(X) framework, calling it accurate analysis of pattern matching [15]. 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 \texttt{cases} construct, which has the same syntax as function, but allows each case to return a value of different type. Here is an example of the expected behaviour:

\begin{align*}
\texttt{let f4 = cases Number() \rightarrow 10 \mid Face() \rightarrow 15} \\
\texttt{f4(Number()) : int \rightarrow 10} \\
\texttt{f4(Face()) : string \rightarrow 15}
\end{align*}

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 \texttt{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{align*}
\texttt{f5 : \alpha ::= (\{Number : int, Face : unit\},} \\
\texttt{\{Number : int, Face : string, return : \alpha_1\},} \\
\texttt{\emptyset, \{Number, Face\})}
\end{align*}

\begin{align*}
\texttt{\triangleright \alpha \rightarrow \alpha_1}
\end{align*}

\begin{align*}
\texttt{let f5 = fun x : \alpha \rightarrow (f1(x), f3(x), f4(x))} \\
\texttt{f5 : \alpha ::= (\{Number : int, Face : unit\},} \\
\texttt{\{Number : int, Face : string, return : \alpha_2\},} \\
\texttt{\emptyset, \{Number, Face\})}
\end{align*}

\begin{align*}
\texttt{\triangleright \alpha \rightarrow \alpha_2}
\end{align*}

\begin{align*}
\texttt{let decomp = fun x \rightarrow (tag x, arg x)} \\
\texttt{decomp : \alpha ::= (\{Number : int, Face : string\},} \\
\texttt{\{Number : int, Face : int, return : \alpha_1\},} \\
\texttt{\emptyset, \{Number, Face\})}
\end{align*}

\begin{align*}
\texttt{\triangleright \alpha \rightarrow \alpha_1 \times \alpha_1}
\end{align*}

\begin{align*}
\texttt{decomp \text{ (Face "king") : int \times string} \rightarrow (\text{int}, "king")} \\
\texttt{decomp \text{ (Number 10) : int \times int} \rightarrow (1, 10)}
\end{align*}

In a more object-oriented context, the same kind of mechanism was called dynamic messages by Nishimura [12]; later it was reformulated using different instances of HM(X) [11, 15]. 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 \{[...]\} are case-analyzing functions, and message passing \texttt{m is
the application o m.

\[
\begin{align*}
\text{let } o_1 & \equiv \{ \text{Number}(x) = \text{float } x; \text{Face}(s \text{ : string}) = s \} \\
o_2 & \equiv \{ \{ \text{Number} : \text{int}, \text{Face} : \text{string} \}, \\
& \quad \{ \text{Number} : \text{float}, \text{Face} : \text{string}, \text{return} : o_1 \}, \\
& \quad \{ \text{Number}, \text{Face} \} \}
\end{align*}
\]

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 \(o_1\) cannot be typed: constraints involving both \(\text{Number}\) and \(\text{Face}\) 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 \([2]\) is available to solve the problem. This only requires to write object types when they appear as function parameters.

3. CONCEPTS DOMAINS

Contrary to usual constraint-based type systems, our local constraints are gathered on top of the 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 class 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 \(C\) is composed of the following items.

1. A theory \(T_C\) with an entailment relation \(\models\) satisfying the following properties
   
   \(a\) There is a constant \(\perp\), such that for any \(C\) we have \(\perp \models C\).
   
   \(b\) A constraint \(C\) such that \(C \models \perp\) is invalid. Validity is decidable.

2. An observation relation \(\vdash\) checking some atomic properties of a constraint: \(C \vdash p(\alpha, \tau)\) where \(p\) and \(\alpha\) are respectively a predicate and a symbol for the domain. Observation should be compatible with entailment:

   \(\text{If } C \models C' \text{ and } C' \vdash p(\alpha) \text{ then } C \vdash p(\alpha)\).

3. A set of relating predicates of the form \(r(\alpha, \tau)\), which relate symbols and types. Some of these predicates are said to be generative, that is they have an extra index \(\epsilon\), which can be existentially quantified: \(\exists \epsilon. r'(\alpha, \tau)\).

4. A set of propagation rules \(E_C\), of the form (depending on whether \(r\) is generative or not)

   \(\forall x.(r(x, \alpha_1) \land r(\alpha, \alpha_2) \land p(x) \Rightarrow \alpha_1 = \alpha_2)\)

   \(\forall x\forall \epsilon.(r'(x, \alpha_1) \land r'(\alpha, \alpha_2) \land p(x) \Rightarrow \alpha_1 = \alpha_2)\)

   where \(\alpha\) is either the same variable \(x\) or a symbol.

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

The observation relation \(\vdash\) is a consequence of the representation independency of our constraints: we need an explicit way to relate them to the rest of the world. It interfaces constraints, which are semantic, with propagation rules, which are syntactic.

Relating predicates are similar in flavor to features in feature algebras \([2]\). Each constraint will be coupled with a set \(R\) of instances of these predicates (e.g. \(R = \{r(a, \text{int}), r(b, \text{int}), r(c, a \text{ list})\}\), in order to relate its semantics with types. \(R\) need not always describe functions: coherence will be enforced by the propagation rules. Generative predicates are used for type information depending on the constraint, but which should not be identified throughout all its occurrences. The idea is that merging two quantified sets of predicates will not mix them: \(\exists \epsilon. \{r'(a, \tau)\} \cup \exists \epsilon. \{r'(a, \tau')\} = \exists \epsilon. \{r'(a, \tau) \lor r'(a, \tau')\}\). Among our examples, only the typing of accurate matching uses generative predicates. Other cases can be handled in a system without \(\epsilon\)'s.

Propagation rules specify how \(C\) and \(R\) constrain other types. They come in two forms: either \(A\) is \(x\), and \(r\) only has to be functional for every symbol \(a\) such that \(C \vdash p(a)\); or \(A\) is some symbol \(b\), and additionally the image of \(a\) by \(r\) should be the same as the image of \(b\) (i.e. the second case includes the first one by transitivity). There is an additional variation for generative predicates, expressing that the functionality condition only applies when identical \(\epsilon\)'s are involved.

The syntax may seem restrictive, 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 \(a_1 = a_2\) 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: \((L, x) \entails (L', x)\) iff \(L \supseteq L'\), which makes set union the conjunction operation. For exact types entailment is only reflexive. All such constraints are valid, so we must add a \( \perp \) element. We do not need to observe anything particular, so we just use \(C \vdash \text{true}(l)\) 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:

\[
\begin{align*}
    r(l, a_1) \land r(l, a_2) \land \text{true}(l) & \Rightarrow a_1 = a_2. \\

\end{align*}
\]

**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 \(L\), or a finite set of available labels. Entailment is defined by \((L, U) \entails (L', U')\) iff \(L \supseteq L'\) and \(U \subset U'\). We can choose \((L, \emptyset)\) as \(\perp\). The validity check is \((L, U) \entails \perp\) when \(L \not\subseteq U\). One can observe required labels: \((L, U) \vdash \text{req}(l)\) 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.

\[
\begin{align*}
    r(l, a_1) \land r(l, a_2) \land \text{req}(l) & \Rightarrow a_1 = a_2. \\

\end{align*}
\]

**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 \(v'(l, \tau)\), and a second propagation rule, acting on result types:

\[
\begin{align*}
    v'(l, a_1) \land v'(\text{return}, a_2) \land \text{req}(l) & \Rightarrow a_1 = a_2. \\

\end{align*}
\]

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 \(\tau\) are defined as usual. Polytypes \(\sigma\) are extended with a kinding environment \(K\) which restricts possible instances for constrained variables.

\[
\begin{align*}
    \tau & ::= a & \text{type variable} \\
    & | u & \text{base type} \\
    & | \tau @ \tau & \text{function type} \\
    K & ::= [K, a :: (C; \exists \forall R)] & \text{kinding environment} \\
    \sigma & ::= \tau | V \ldots . A, K @ \tau & \text{polytypes} \\

\end{align*}
\]

\(K\) is a set of bindings \(a :: (C; \exists \forall R)\), \(C\) a constraint and \(R\) a set of relations to types, describing the possible values admitted for the type \(a\). 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 \(\exists \forall R\) is to be understood modulo \(\alpha\)-conversion of the \(\epsilon\)'s, and polytypes modulo \(\alpha\)-conversion of quantified type variables.

**Definition 2.** A kind \(k = (C; \exists \forall R)\) is well formed if

1. the constraint \(C\) is satisfiable.

2. for each \(t(x, a_1) \land t(A, a_2) \land p(x) \Rightarrow a_1 = a_2\) in \(E\) (where \(t = \tau\) or \(\tau'\)) and each \(t(a, \tau)\) and \(t([a/x], \tau)\) 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 \(\text{FV}_K(\sigma)\) of a polytype \(\sigma\) under a kinding environment \(K\) are defined as the minimum set satisfying the following equations. We write \(\text{FV}(\sigma)\) when \(K\) is clear from the context.

\[
\begin{align*}
    \text{FV}_K(\forall a_1 \ldots a_n, K' @ \tau) & = \text{FV}_K(\tau) \setminus \{a_1, \ldots, a_n\} \\
    \text{FV}_K(a :: (C; \exists \forall R)) & = \{a\} \cup \text{FV}_K(R) \\
    \text{FV}_K([a]) & = \emptyset \\
    \text{FV}_K(\tau_1 @ \tau_2) & = \text{FV}_K(\tau_1) \cup \text{FV}_K(\tau_2) \\

\end{align*}
\]

Our notion of free variables is an important difference with systems like HM(X). We explicitly 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 Hindley-Milner type systems, whereas in HM(X) constraints may include unrelated variables.

**Definition 3.** A type substitution \(\theta\), extended as usual on monotypes and polytypes, is admissible between the kinding environments \(K\) and \(K'\), written \(K \vdash \theta : K'\), if for all \(a :: (C; \exists \forall R)\) in \(K\), \(\theta(a)\) is a type variable \(a'\) and it satisfies the following properties.

1. \(a' :: (C'; \exists \forall R')\in K'\)

2. \(C' \models C\)

3. there is an index substitution \(\eta : \exists \rightarrow \exists\) 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 $K_\theta$ (resp. $\theta_\mid_D$) for their restriction to variables in $D$, and $K_\theta |_{\Gamma}$ (resp. $\theta_\mid_{\Gamma}$) 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 $t = \tau$ or $\tau'$. If for all propagation rule $t(x, \alpha_1) \land t(A, \alpha_2) \land p(x) \Rightarrow \alpha_1 \tau_2$ in $E$, we have $C \vdash p(x)$, 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 $E$ and constraint $C$. Such an inert atom can be safely replaced by a dummy $r_{x}(\tau)$, which just makes sure that free variables are kept. Moreover, $r_{x}(\tau)$ can be replaced by $\{r_{a}(\alpha) \in FV(\tau)\}$ at any point. For this reason we can simplify $R$ through the reflectivity and transversely closed relation $\approx_{E}:

\begin{align*}
R \cup \{t(a, \tau)\} & \approx_{E} \cup R \cup \{r_{x}(\tau) \mid \tau \in T\} \\
when \ t(a, \tau) \ \text{inert} \ \text{and} \ \text{FV}_{x}(\tau) = \text{FV}_{y}(T) & \\
R & \approx_{E} R' \Rightarrow (C, \exists E, R) \approx_{E} (C, \exists E, R')
\end{align*}

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

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

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.

1. Here $\neg p$ is some observable predicate in the complement of $p$, i.e. $\forall C(C \vdash \neg p(x) \land C \vdash p(x)) \Rightarrow C \models \bot$. As an observable, it is preserved by entailment. Since constraints are to be understood under the open world hypothesis, $C \vdash \neg p(x)$ is not equivalent to $\neg C \vdash p(x)$.

5. **TERMS AND TYPING RULES**

Expressions are the standard ones,

$$
e ::= x \mid \text{fun } x \to e \mid e \ e \mid \text{core lambda} \mid \text{let } x = e \text{ in } e \mid \text{ constants and let}
$$

Type judgments are extended with a kinding environment,

$$K; \Gamma \vdash e : \tau$$

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

TYPING rules appear in figure 1. They are in the syntactically directed style, instantiating and generalizing in one step, which simplifies the handling of eventual mutual recursion in the kinding environment. GENERALIZE2 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 $K \vdash \sigma$ is to be understood as the polytype $\forall \forall V_{K}(\sigma)$.

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

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

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

$$A[e_1, \ldots, e_n] := c \mid x_1 \mid \{A[e_1, \ldots, e_n] A[x_1, \ldots, x_n]\}$$

Each $\delta$-rule should be type safe: if $K_0 \vdash A[x_1, \ldots, x_n] : \tau$, then $K_0; \Gamma \vdash A'[x_1, \ldots, x_n] : \tau$ must be provable.

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

**Theorem 4 (Subject reduction).** If $K, \Gamma \vdash E[\text{let } c = e : \sigma$ and $c' = e' \mid \text{ by a } \delta$-rule $c \vdash E[\text{let } c = e : e'[\text{let } c = e'] : \sigma]$, or LET $let x = e_1 \in e_2 \to e_3 x_1 x_2$ then $K; \Gamma \vdash E[\text{let }] : \sigma$.
Records à la Okori

\[
\begin{align*}
\text{TConst} & (\text{record}_{i_1...i_n}) = \alpha_0 : ((\{l_1, ..., l_n\}, 1), \{r(l_1, a_1), ..., r(l_n, a_n)\}) \\
\quad \triangleright a_1 \mapsto \cdots \mapsto a_n \mapsto \alpha_0 \\
\text{TConst} & (\text{get}_i) = \alpha_0 : ((\{l\}, 0), \{r(l, a_1)\}) \triangleright \alpha_0 \mapsto a_1
\end{align*}
\]

Records with maskable fields

\[
\begin{align*}
\text{TConst} & (\text{record}_{i_1...i_n}) = \alpha_0 : ((\emptyset, \{l_1, ..., l_n\}), \{r(l_1, a_1), ..., r(l_n, a_n)\}) \\
\quad \triangleright a_1 \mapsto \cdots \mapsto a_n \mapsto \alpha_0 \\
\text{TConst} & (\text{get}_i) = \alpha_0 : ((\{l\}, \{l\}), \{r(l, a_1)\}) \triangleright \alpha_0 \mapsto a_1
\end{align*}
\]

Variants with maskable constructors

\[
\begin{align*}
\text{TConst} & (\text{tag}_{i}) = \alpha_0 : ((\{l\}, \{r(l, a_1)\}) \triangleright a_1 \mapsto \alpha_0 \\
\text{TConst} & (\text{match}_{i_1...i_n}) = \alpha_0 : ((\emptyset, \{l_1, ..., l_n\}), \{r(l_1, a_1), ..., r(l_n, a_n)\}) \\
\quad \triangleright (a_1 \mapsto \alpha) \mapsto \cdots \mapsto (a_n \mapsto \alpha) \mapsto \alpha_0 \mapsto \alpha \\
\text{TConst} & (\text{split}_{i_1...i_n,i_k+1...i_n}) = \alpha' : ((\{l_1, ..., l_n\}, \{L\}, \{r(l_1, a_1), ..., r(l_k, a_k)\}), \\
\quad \alpha'' : ((\{l_{k+1}, ..., l_n\}, \{L\}, \{r(l_{k+1}, a_{k+1}), ..., r(l_n, a_n)\})), \\
\quad \alpha_0 : ((\emptyset, \{l_1, ..., l_n\}), \{r(l_1, a_1), ..., r(l_n, a_n)\}) \\
\quad \triangleright (a' \mapsto \alpha) \mapsto (a'' \mapsto \alpha) \mapsto \alpha_0 \mapsto \alpha
\end{align*}
\]

Accurate matching

\[
\begin{align*}
\text{TConst} & (\text{cases}_{i_1...i_n}) = \alpha_0 : ((\emptyset, \{l_1, ..., l_n\}), \exists. \{r(l_1, a_1), ..., r(l_n, a_n)\}, \\
\quad \{v'(l_1, a_1), ..., v'(l_n, a_n)\}, \{v'(\text{return}, a)\}) \\
\quad \triangleright (a_1 \mapsto a') \mapsto \cdots \mapsto (a_n \mapsto a_n') \mapsto \alpha_0 \mapsto \alpha
\end{align*}
\]

Figure 2: Examples of constructs with their associated constraints

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 [20]. 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 \(\delta\)-rule is

\[
\text{get}_{i_1} (\text{record}_{i_1...i_n} e_1 ... e_n) \rightarrow e_i.
\]

For variants, which share the same constraint as records with maskable fields, we have three \(\delta\)-rules. The first 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:

\[
\begin{align*}
\text{match}_{i_1...i_n} f_1 ... f_n (\text{tag}_{i_1} e) & \rightarrow f_i e \\
\text{split}_{i_1...i_n,i_k+1...i_n} f_1 ... f_n (\text{tag}_{i_1} e) & \rightarrow f_i (\text{tag}_{i_1} e) \quad \text{if } i \leq k \\
\text{split}_{i_1...i_n,i_k+1...i_n} f_1 ... f_n (\text{tag}_{i_1} e) & \rightarrow f_i (\text{tag}_{i_1} e) \quad \text{if } i > k
\end{align*}
\]

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 \(\delta\)-rule is the same as for match, only the type differs.

\[
\text{cases}_{i_1...i_n} f_1 ... f_n (\text{tag}_{i_1} e) \rightarrow f_i e
\]

6. UNIFICATION

We define unification on monotypes through rules of the form

\[
\varphi \vdash \varphi'
\]

where \(\varphi\) are preconditions and \(\varphi'\) conclusions. \(\varphi\) is a unification problem composed both of kind constraints \(\alpha : (C, \exists \varphi, R)\) and equality constraints \(\tau_1 = \tau_2\) connected by \(\wedge\). Both \(\wedge\) and \(\Rightarrow\) 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 \(\varphi\) if it is of the form \(\alpha = \tau\) and \(\alpha\) only appears as strict subterm in other constraints of \(\varphi\). A kinding constraint is solved if its kind is well-formed. A unification problem \(\varphi\) is solved if all its constraints are solved.

A solution to a unification problem \(\varphi\), whose kinding constraints form a kinding environment \(K\) (its base), is a kinding environment \(K'\) and an admissible substitution \(\theta : K\) such that for all equality constraint \(\tau_1 = \tau_2\) in \(\varphi\), we have \(\theta(\tau_1) = \theta(\tau_2)\).

Rules for unification are in figure 3. \(\text{sort}_{\varphi}\) is a partial function from types to \(\{u_1, ..., u_n \rightarrow k\}\), left undefined on non-kindned variables:

\[
\begin{align*}
\text{sort}_{\varphi}(u) & = u \quad \text{u a base type} \\
\text{sort}_{\varphi}(\tau_1 = \tau_2) & = \rightarrow \\
\text{sort}_{\varphi}(\alpha) & = k \quad \alpha : (C, \exists \varphi, R) \in \varphi
\end{align*}
\]

Proposition 5. Rewriting a unification problem leads either to \(\perp\) or to a solved problem.

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

A substitution \(K \vdash \theta_1 : K_1\) is said to be more general than \(K \vdash \theta_2 : K_2\) if there is a substitution \(K_1 \vdash \theta : K_2\) such that \(\theta_2 = \theta \circ \theta_1\).

Proposition 6. The solution \(K' \vdash \theta : K'\) read from the solved form obtained by rewriting a unification problem \(\varphi\) of
Figure 3: Rewriting rules for unification

\[
K; \Gamma \vdash e : \tau = (\Gamma \vdash e : \tau) \land K
\]

\[
\Gamma, x : (\forall B.K \vdash \tau_1) \vdash x : \tau = \tau \land \theta(\tau_1) \land \bigwedge_{a_i : (C_i, \exists \xi, R_i) \in K} \beta_i :: (C_i, \exists \xi, \theta(R_i))
\]

where \( B = \{a_1, \ldots, a_n\} \) and \( \theta = \{a_1 \mapsto \beta_1, \ldots, a_n \mapsto \beta_n\} \)

\[
\Gamma \vdash \text{fun } x \to e : \tau = (\Gamma, x : a_1 \to e : \alpha_2) \land \tau = a_1 \to a_2
\]

\[
\Gamma \vdash e_1 e_2 : \tau = (\Gamma \vdash e_1 : \alpha_1 \to \tau) \land (\Gamma \vdash e_2 : \alpha)
\]

\[
\Gamma \vdash \text{let } x = e_1 \text{ in } e_2 : \tau = (\forall \tau_0 \land \bigwedge_{a_i : (c_i, 3 \xi, R_i) \in K_0} \beta_i :: (C_i, \exists \xi, \theta(R_i))
\]

where \( \varphi = (\Gamma \vdash e_1 : \alpha) \) and \( K \vdash \theta : K = \text{solve}(\varphi) \)

\[
\text{and } B = \text{FV}_K(\theta(\alpha)) \land \text{FV}_K(\varphi)
\]

\[
\Gamma \vdash c : \tau = \tau \land \bigwedge_{a_i : (c_i, 3 \xi, R_i) \in K_0} \beta_i :: (C_i, \exists \xi, \theta(R_i))
\]

where \( T\text{const}(c) = K_0 \vdash \tau_0, \text{FV}_K(\tau_0) = \{a_1, \ldots, a_n\} \)

and \( \theta = \{a_1 \mapsto \beta_1, \ldots, a_n \mapsto \beta_n\} \)

Figure 4: Reconstruction algorithm
basis K is also K ⊢ θ : K', and it is the most general unifier for this problem.

7. TYPE RECONSTRUCTION

Type reconstruction is done by translating a typing problem K, Γ : e : τ into a unification problem ϕ, and solving it to obtain a substitution K ⊢ θ : 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 K, Γ : e : τ is a substitution K ⊢ θ : K' such that K'; θ(Γ) ⊢ e : θ(τ) is derivable.

Theorem 7. If K, Γ : e : τ can be reduced to K ⊢ θ : K' by the type reconstruction algorithm, K'; θ(Γ) ⊢ e : θ(τ) is derivable, and θ is the most general solution; otherwise it reduces to ⊥ and there is no solution.

8. RELATION TO HM(X)

While we have hinted at subtle differences with HM(X) in previous sections, one may still wonder why we cannot present this system as a subframework of HM(X). The subframework approach not only would relieve us from part of the proof burden (which, luckily, was not so big), it would also allow for a direct comparison with other systems developed inside HM(X).

In this section we explain in detail why this system does not fit inside the HM(X) framework, and why attempting to make it fit into would lose some theoretical and practical properties. In order to do so, we will first define an HM(X) version of our framework, and show on an example how it differs from our version. Then we will formalize some distinctive properties.

8.1 Generalization

Obtaining an HM(X) version of our framework is very easy. One just needs to change the definition of the generalize typing rule.

\[
\text{GENERALIZE-HM(X)}
\]

\[
\begin{align*}
K_1, K_2; \Gamma & : e : \tau \quad B \cap (\text{FV}_B(K_1) \cup \text{FV}_B(\Gamma)) = \emptyset \\
K_1, \exists B K_2; \Gamma & : e : \forall B K_2 \triangleright \tau
\end{align*}
\]

To make the comparison easier, we consider a syntax-directed version of this rule.

\[
\text{GENERALIZE-HM(X)}
\]

\[
\begin{align*}
K; \Gamma & : e : \tau \quad B = (\text{FV}_B(\tau) \cup \text{FV}_B(K_B)) \setminus \text{FV}_B(\Gamma) \\
\exists B K; \Gamma & : e : \forall B K_B \triangleright \tau
\end{align*}
\]

where K_B is the subset of K containing all kinds related to B (either as name of the kind, or inside R). We solve the mutual recursion between B and K_B by choosing the smallest B satisfying the equation.

Here is a concrete example where the HM(X) version would infer a different type. Let’s consider the function

\[
\text{fun } x \rightarrow \text{let } y = (\text{function } A(z) \rightarrow z \mid B(z) \rightarrow z) \text{ in } y
\]

If we write K for the kinding environment α :: (θ, {A, B}, {A : β, B : β}), the typing derivation for the function application becomes (independently of the framework used)

\[
K; x : \alpha \vdash (\text{function } A(z) \rightarrow z \mid B(z) \rightarrow z) : \beta
\]

According to our framework, β is a free variable of α, so we cannot generalize this type. However, using rather GENERA-
LIZE-HM(X), one can reach the following conclusion.

\[
K; x : \alpha \vdash y : \forall \beta \alpha :: (\theta, \{A, B\}, \{A : \beta, B : \beta\}) \triangleright \beta
\]

You can see here the peculiar form of the polymorphism obtained: the kinding part in \forall B K \triangleright \tau is no longer restricted to variables of B (as requires our type reconstruction algorithm). And a polymorphic variable (β) may actually depend on a non-polymorphic one (α).

In the above example, the difference in typing is only superficial, making the extra polymorphism spurious: actually one gets exactly the same polymorphism when considering the whole term. While there exist examples where GENERA-
LIZE-HM(X) provides really more polymorphism, they are rare, and there are workarounds.

8.2 Relevant variables

A formal way to distinguish the two systems is by considering relevant type variables. Intuitively, a variable is relevant if the meaning of a type depends on it proper (i.e. its whole meaning is included in the type).

More specifically, we consider the meaning of a type as the set of terms typable by one of its instances, and we say that a type variable α is relevant to a type K ⇒ R, if for some instance K' ⇒ R' of this type, there is a term e with a derivation K', θ : θ(Γ) ⊢ e : R', such that any substitution K ⊢ θ : K'', verifying K'' : θ : e : θ(τ), has either to substitute α or have it bound in K'' to a kind with a different constraint (a kind with different related types does not imply relevance).

Our system has the following property: all relevant variables of K ⇒ R are included in F V K (τ). This is not the case with HM(X) based systems. For instance, in the above example, β is clearly relevant to K ⇒ α (e.g. A(1) binds β to int), but it is not in F V K (α) = {α}.

This property gives a clean way to print a type, with all relevant type information, by inserting a kind where its variable occurs. For instance, in Objective Caml, we would write

\[
[< A of \beta | B of \beta ] \rightarrow \beta
\]

for the above term’s type.

This property is also needed for tracing sharing in types, as we do for semi-explicit first-class polymorphism [7]. Sharing is easily obtained by making each explicit polymorphic kind α :: [τ] and using α for each shared occurrence of this polymorphic kind, following the idea in the section 2.6 of [7]. Since HM(X) fails to capture relevance in the type system, our first-class polymorphism would not have principal inference for it.

A possibly simpler way to characterize the relation of relevance is at the type inference level. In our system, if α ∈ F V K (τ₁) ∪ F V K (τ₂), then solving K ⇒ τ₁ = τ₂ will not bind α. Of course, this is not true for HM(X).

This has technical consequences. For instance this property is needed to do “in place generalization”, that is marking directly generic variables in a type. This is only possible when
generizable variables are guaranteed not to be shared with types in the environment. While "in place generalization" is not a goal in itself, detecting the extent of the enclosing type which should be copied, as expected by \( \text{GENERALIZE-HM}(X) \), would require a fair amount of extra machinery.

This is also an important property for programmers, as they can easily identify the consequences of a type constraint: to prove that a constraint does not affect a type, one only needs to check that they do not share free variables. With \( \text{HM}(X) \), the baseline is to consider the transitive closure of all constraints sharing some variables, which potentially contains lots of irrelevant variables.

9. 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 unsperscious 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 [8]. 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 [19], correctly showing that, while this is not immediately explicit, all these systems require some form of "subtyping" or multi-directional constraints.

Ironically the last counter-example standing against a characterization of structural polymorphism by local constraints seems to be the labelled selective lambda-calculus and its type system [6]. Function types are of the form \( \lambda T \rightarrow T \), modulo the congruence \( l_1:T_1 \rightarrow l_2:T_2 \rightarrow T = l_2:T_2 \rightarrow l_1:T_1 \rightarrow T \).

Clearly, this bears no relation to subtyping, as shown by the existence of a typing algorithm without constraints, yet one has to separate \( l_1:T_1 \rightarrow T \) from \( T \), 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 \( l_1 \neq l_2 \land \sigma(l_1, \alpha_1) \land \sigma(l_2, \alpha_2) \Rightarrow \alpha_1 = \alpha' \alpha_2 \iff \langle \{l_2\} \cdot \sigma(l_2, \alpha_2) \rangle \), 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 labelled 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 [4], that the return type variable after an arbitrary number of arrows should differ.

\[\text{This requirement was missing in [6].}\]

10. REFERENCES


APPENDIX

A. PROOFS OF THEOREMS

PROPOSITION 1 (DISCARDING POSTCONDITION). If $K$, $\alpha \vdash k : \theta$ and $K \vdash \theta(k) \equiv k$, then there is a kind $k_0$ such that $k_0 \equiv \varepsilon k$ and $K, \alpha \vdash k_0 : \theta'$.

Proof. Suppose $k_0 = (C, \exists \varepsilon R \cup R_0)$ and $k = (C, \exists \varepsilon R \cup R_1)$, with $R_0$ and $R_1$ minimal. By definition of $\equiv \varepsilon$, all $\tau(k, \alpha)$ in $R_0$ are inert for $C$. By definition of admissibility, $k' = (C', \exists \varepsilon R' \cup \theta(R_0)) \equiv \varepsilon (C', \exists \varepsilon R' \cup \theta(R_1)) = k'$. $\blacksquare$

LEMMA 2 (TYPE SUBSTITUTION). If $K, \Gamma \vdash \epsilon : \tau$ and $K \vdash \theta : \theta'$, then $K', \theta' \vdash \epsilon : \theta'$. $\blacksquare$

Proof. Induction on the derivation of $K, \Gamma \vdash \epsilon : \tau$. The two important steps are VARIABLE and GENERALIZE. For VARIABLE, we have $K, \epsilon \vdash \theta : K$ and $\text{Dom}(\theta) \subseteq B$; we can choose $B$ outside of $\theta$ and $K'$: $\forall \alpha \in B, \theta(\alpha) = \alpha$, and $FV(K') = \emptyset$. By composition, we construct $\theta_1 = (\sigma \circ \theta_0)|_B$, such that $K', \theta_1(K_0) \vdash \epsilon : K'$: conditions (1) and (2) are satisfied by transitivity. We check condition (3) for kind variables.

- If $\alpha \not\in B$, then $\theta_1(\alpha) = \alpha$, $\alpha \equiv (C, \exists \varepsilon R) \in K'$ and $FV_k(R') \cup \emptyset = \emptyset$, so that $\theta_1(R) \equiv \emptyset$.
- If $\alpha \in B$, then $\theta_1(\alpha) = \theta_0(\alpha)$, and $\alpha \equiv (C, \exists \varepsilon R) \in K_0$. For any variable $\alpha' \in FV_k(R)$, we have $\theta_1(\alpha'(\alpha')) = \theta_0(\alpha'(\alpha'))$: either $\alpha' \not\in B$, and $\alpha' = \alpha'$, so that $\theta_1(\alpha'(\alpha')) = \theta_1(\alpha') = \theta_0(\alpha'(\alpha'))$, or $\alpha' \in B$, and $\theta_0(\alpha'(\alpha')) = \alpha'$, giving $\theta_1(\alpha'(\alpha')) = \theta_0(\alpha'(\alpha'))$.

Combined with $\theta_1(\alpha) \equiv (C', \exists \varepsilon R') \in K'$ implying $\theta_0(R(R)) \in K'$, we obtain $\theta_1(\theta_0(R)) \in K'$, giving condition (3) since $\alpha \equiv (C, \exists \varepsilon R) \in K_0$.

Note also that $\theta_1 \circ \theta'_{\varepsilon} = \theta \circ \theta_0$. Then we can deduce $K', \theta'_{\varepsilon} : \theta(R), x : \forall \beta \in B, \theta_0(\beta) \vdash \epsilon : \theta_0(\epsilon)(\alpha')$. $\blacksquare$

FOR GENERALIZE, again we choose $B$ outside of $\theta$ and $K'$. Then $K_0, \theta(\theta_0(B)) \vdash \epsilon : K', \theta_0(K_0)$. By induction hypothesis we have $K', \theta(\theta_0(B)); \Gamma, \theta(\theta_0(B)) \vdash \epsilon : \theta(\epsilon)$. Then we can deduce $K', \theta(\theta_0(B)); \Gamma \vdash \epsilon : \theta(\epsilon)$. $\blacksquare$

LEMMA 3 (TERM SUBSTITUTION). If $K, \Gamma \vdash \epsilon : \sigma$ and $K, \Gamma, \sigma \vdash \epsilon' : \tau'$, then $K, \Gamma \vdash \epsilon'[\sigma/x] = \tau'$. $\blacksquare$

Proof. Easy induction on the derivation tree. When grafting GENERALIZE somewhere, use a type substitution $K \vdash \theta : K'$. When substituting under GENERALIZE, $B$ may grow. In such case, preserve $K_{\theta}$ by duplicating the kind under a different name, and instantiate extra variables with VARIABLE $\blacksquare$

THEOREM 4 (SUBJECT REDUCTION). If $K, \Gamma \vdash E[x] : \sigma$ and $e \equiv e'$ by a $\delta$-rule, Betta $\vdash (\text{fix} x : e_1 \rightarrow e_2) \rightarrow [x_1/x_2] E[x_1]$, or let $e : e_1 \rightarrow e_2 \vdash [\lambda x_1 \rightarrow e_2/x_{1}, \text{then } K, \Gamma \vdash E[e'] : \sigma'$. $\blacksquare$

Proof. First we consider the case when $E[x] = e$. If $e \equiv e'$ is the instance of the $\delta$-rule $A[x_1, \ldots, x_n] \rightarrow A'[x_1, \ldots, x_n]$, then, supposing $e_1 : \tau_1, \ldots, e_n : \tau_n$, we have already a derivation of $K_0; x_1 : \tau_1, \ldots, x_n : \tau_n \vdash e'[x_1, \ldots, x_n] : \tau$. We conclude by term substitution. Similarly, $\beta$-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. $\blacksquare$

PROPOSITION 5. Rewriting a unification problem leads to either $\perp$ or a solved problem. $\blacksquare$

Proof. First, we verify that unification terminates. One is allowed to apply rules in any order, except that (1) failure is easier (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 unvalued 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 $\tau_1$ equal to $\tau_2$) 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 unvalued 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 $\epsilon = \epsilon$ and cyclic and all applicable. One can write examples to check this in detail. Note that this is a kind of constraint if it causes a constraint unification, so no new propagation will occur until one of these happen. Substitution reduces the number of unvalued variables. Function reduces the number of arrows. Redundancy reduces the number of equations.

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

$\blacksquare$
PROOF. For each rewriting rule, we verify that it is sound and complete. Together with soundness, we also check changes in bindings were necessary.

Rules INCOMPATIBLE, CYCLIC, REDUNDANCY and FUNCTION are immediate: the upper and lower constraints accept the same solutions.

For SUBSTITUTION, soundness is clear: \( \theta(a) = \theta(r) \) then if \( \theta \) is a solution of \( \phi(x)/a \), it is also a solution of \( \phi \). Looking at bindings, let \( K_1 \) be the basis for \( \phi \), and \( K_2 \) the basis for \( \phi(x)/a \). We have \( K_1 \vdash \{ a \mapsto \tau \} : K_2 \), and \( K_2 \vdash \theta : K' \), so \( \theta \) is a solution of \( \phi \). For completeness, the side-conditions guarantee that \( \alpha \) has no kind (reduction rules only introduce kinds for fresh variables). Then it is safe to replace \( \alpha \) by \( \tau \) in \( \phi \), and the same solutions are accepted.

For BAD CONSTRAINT, no substitution can apply to \( \alpha \), so that the constraint is equivalent to \( \bot \).

For CONSTRAINT, if we have a solution \( K \vdash \theta : K' \) of the upper side, then it has to map both \( \alpha_1 \) and \( \alpha_2 \) to \( a' : (C_1, \exists \exists \exists R') \), with \( C \vdash C_1 \) and \( C' \vdash C_2 \). Since \( C_1 \cap C_2 \) is a subset of \( C_1 \cap C_2 \), we can extend \( \theta \) into \( K \vdash \{ \alpha_1 \mapsto \alpha_2 \} : K' \), and it is a solution of \( \phi \). Completeness is immediate: any solution \( K \vdash \theta : K' \) of the upper side has \( \theta \) as a solution of \( \phi \).

For PROPAGATION, soundness is immediate (the lower side subsumes the upper side). Any solution of \( K \vdash \theta : K' \) of the upper side must have \( \theta(\tau_1) = \theta(\tau_2) \) as a solution of \( \phi \). Completeness is immediate: any solution \( K \vdash \theta : K' \) of the upper side has \( \theta \) as a solution of \( \phi \).

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

THEOREM 7. If \( K, \Gamma \vdash e : \tau \) can be reduced to \( K \vdash \theta : K' \) by the type reconstruction algorithm, \( K', \theta(\Gamma) \vdash e : \theta(\tau) \) is derivable, and \( \theta \) 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 splits a complete problem in its kinded environment and typing problem.

The second case handles the VARIABLE rule. Kinded \( \beta \)'s ensure soundness: if \( K' = (\beta : (C_1, \exists \exists \exists R_i)_{\alpha_i}(K)) \), then \( K \vdash K' \), so that \( K', K \vdash \theta : K' \) (since \( \beta(\theta_{\downarrow\alpha_i}(K)) = \beta \)), and any solution \( K' \vdash \theta : K' \) of the right hand side is such that \( K', K \vdash \theta : K' \) (i.e., \( \beta(\theta_{\downarrow\alpha_i}(K)) = \beta \)), proves \( K', \theta(\Gamma), x \vdash K', K \vdash \theta(\tau_1) \), and \( \theta(\tau_1) = \theta(\tau_1) \). For completeness,