Hints in unification

Andrea Asperti, Wilmer Ricciotti, Claudio Sacerdoti Coen, and Enrico Tassi

Department of Computer Science, University of Bologna
Mura Anteo Zamboni, 7 — 40127 Bologna, ITALY
{asperti,ricciott,sacerdot,tassi}@cs.unibo.it

Abstract. Several mechanisms such as Canonical Structures [14], Type Classes [16,13], or Pullbacks [10] have been recently introduced with the aim to improve the power and flexibility of the type inference algorithm for interactive theorem provers. We claim that all these mechanisms are particular instances of a simpler and more general technique, just consisting in providing suitable hints to the unification procedure underlying type inference. This allows a simple, modular and not intrusive implementation of all the above mentioned techniques, opening at the same time innovative and unexpected perspectives on its possible applications.

1 Introduction

Mathematical objects commonly have multiple, isomorphic representations or can be seen at different levels of an algebraic hierarchy, according to the kind or amount of information we wish to expose or emphasise. This richness is a major tool in mathematics, allowing to implicitly pass from one representation to another depending on the user needs. This operation is much more difficult for machines, and many works have been devoted to the problem of adding syntactic facilities to mimic the *abus de notation* so typical of the mathematical language. The point is not only to free the user by the need of typing redundant information, but to switch to a more flexible linkage model, by combining, for instance, resolution of overloaded methods, or supporting multiple views of a same component.

All these operations, in systems based on type theories, are traditionally performed during type-inference, by a module that we call "refiner". The refiner is not only responsible for inferring types that have not been explicitly declared: it must synthesise or constrain terms omitted by the user; it must adjust the formula, for instance by inserting functions to pass from one representation to another one; it may help the user in identifying the minimal algebraic structure providing a meaning to the formula.

From the user point of view, the refiner is the primary source of "intelligence" of the system: the more effective it is, the easier becomes the communication with the system. Thus, a natural trend in the development of proof assistants consists in constantly improving the functionalities of this component, and in particular to move towards a tighter integration between the refiner and the modules in charge of proof automation.

Among the mechanisms which have been recently introduced in the literature with the aim to improve the power and flexibility of the refiner, we recall, in Section 2, Canonical Structures [14], Type Classes [13], and Pullbacks [10]. Our claim is that all these mechanisms are particular instances of a simpler and more general technique presented in Section 3, just consisting in providing suitable hints to the unification procedure underlying the type inference algorithm. This simple observation paves the way to a light, modular and not intrusive implementation of all the above mentioned techniques, and looks suitable to interesting generalisations as discussed in Section 4.

In the rest of the paper we shall use the notation $\equiv$ to express the type equivalence relation of the given calculus. A unification problem will be expressed as $A \stackrel{?}{\equiv} B$, resulting in a substitution σ such that $A\sigma \equiv B\sigma$. Metavariables will be denoted with $?_i$, and substitutions are described as lists of assignments of the form $?_i := t$.

2 Type inference heuristics

In this section, we recall some heuristics for type refinement already described in the literature and implemented in interactive provers like Coq, Isabelle and Matita.

2.1 Canonical structures

A canonical structure is declaration of a particular instance of a record to be used by the type checker to solve unification problems. For example, consider the record type of groups, and its particular instance over integers ($\mathbb{Z}$).

$$\mathcal{Z} : \text{Group} := \{\text{gcarr} := \mathbb{Z};\ \text{gunit} := 0;\ \text{gop} := \mathit{Zplus}; \ldots\}$$

The user inputs the following formula, where 0 is of type $\mathbb{Z}$.

$$0 + x = x \tag{1}$$

Suppose that the notation $(x + y)$ is associated with gop ? x y where gop is the projection of the group operation with type:

$$\text{gop} : \forall g : \text{Group.gcarr } g \to \text{gcarr } g \to \text{gcarr } g$$

and gcarr is of type Group $\to$ *Type* (i.e. the projection of the group carrier). After notation expansion equation (1) becomes

$$\text{gop } ?_1\ 0\ x = x$$

where $?_1$ is a metavariable. For (1) to be well typed the arguments of gop have to be of type gcarr g for some group g. In particular, the first user provided argument 0 is of type $\mathbb{Z}$, generating the following unification problem:

$$\text{gcarr } ?_1 \stackrel{?}{\equiv} \mathbb{Z}$$

If the user declared $\mathcal{Z}$ as the canonical group structure over $\mathbb{Z}$, the system finds the solution $?_1 := \mathcal{Z}$. This heuristic is triggered only when the unification problem involves a record projection π_i applied to a metavariable versus a constant c. Canonical structures $S := \{c_1; \ldots; c_n\}$ can be easily indexed using as keys all the pairs of the form $\langle \pi_i, c_i \rangle$.

This device was introduced by A.Saibi in the Coq system [14] and is extensively used in the formalisation of finite group theory by Gonthier et al. [2,6].

2.2 Type classes

Type classes were introduced in the context of programming languages to properly handle symbol overloading in [15,7], and they have been later adopted in interactive provers [16,13].

In a programming language with explicit polymorphism, dispatching an overloaded method amounts to suitably instantiate a type variable. This generalises canonical structures exploiting a Prolog-like mechanism to search for type class instances.

For instance we show how to define the group theoretical construction of the Cartesian product using a simplification of the Coq syntax.

```
Class Group (A : Type) := { unit : A; gop : A → A → A; ...}
Instance Z : Group ℤ := { unit := 0; gop := Zplus; ...}
Instance _ × _ (A,B: Type) (G: Group A) (H: Group B) : Group (A × B) := {
  unit := ⟨unit G, unit H⟩;
  gop ⟨x1,x2⟩ ⟨y1,y2⟩ := ⟨gop G x1 y1, gop H x2 y2⟩;
  ...
}
```

With this device a slightly more complicated formula than (1) can be accepted by the system, such as:

$$\langle 0, 0 \rangle + x = x$$

Unfolding the $\_ + \_$ notation we obtain

$$\text{gop } ?_1 \ ?_2 \ \langle 0, 0 \rangle \ x = x$$

where the type of gop and the type of $?_2$ are:

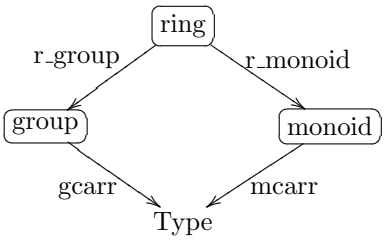
$$\begin{aligned} \text{gop} &: \forall T : \text{Type}. \forall g : \text{Group } T. T \to T \to T \\ ?_2 &: \text{Group } ?_1 \end{aligned}$$

After $?_1$ is instantiated with $\mathbb{Z} \times \mathbb{Z}$ proof automation is used to inhabit $?_2$ whose type has become Group $(\mathbb{Z} \times \mathbb{Z})$. Automation is limited to a Prolog-like search whose clauses are the user declared instances. Notice that the user has not defined a type class instance (i.e. a canonical structure) over the group $\mathcal{Z} \times \mathcal{Z}$.

2.3 Coercions pullback

The coercions pullback device was introduced as part of the manifesting coercions technique by Sacerdoti Coen and Tassi in [10] to ease the encoding of algebraic structures in type theory (see [11] for a formalisation explicating that technique).

This devices comes to play in a setting with a hierarchy of structures, some of which are built combining together simpler structures. The carrier projection is very frequently declared as a coercion [8], allowing the user to type formulas like $\forall g : \text{Group}.\forall x : g.P(x)$ omitting to apply gcarr to g (i.e. the system is able to insert the application of coercions when needed [12]).



The algebraic structure of rings is composed by a multiplicative monoid and an additive group, respectively projected out by the coercions r_group and r_monoid so that a ring can be automatically seen by the system as a monoid or a group. The ring structure can be built when the carriers of the two structures are compatible (that in intensional type theories can require some non trivial efforts, see [10] for a detailed explanation).

When the operations of both structures are used in the same formula, the system has to solve a particular kind of unification problems. For example, consider the usual distributivity law of the ring structure:

$$x * (y + z) = x * y + x * z$$

Expanding the notation we obtain as the left hand side the following

$$\text{mop } ?_1\ x\ (\text{gop } ?_2\ y\ z)$$

The second argument of mop has type gcarr $?_2$ but is expected to have type mcarr $?_1$, corresponding to the unification problem:

$$\text{gcarr } ?_2 \stackrel{?}{\equiv} \text{mcarr } ?_1$$

The system should infer the minimal algebraic structure in which the formula can be interpreted, and the coercions pullback devices amounts to the calculation of the pullback (in categorical sense) of the coercions graph for the arrows gcarr and mcarr. The solution, in our example, is the following substitution:

$$?_2 := \text{r\_group } ?_3 \qquad ?_1 := \text{r\_monoid } ?_3$$

The solution is correct since the carriers of the structures composing the ring structure are compatible w.r.t. equivalence (i.e. the two paths in the coercions graph commute), that corresponds to the following property: for every ring r

$$\text{gcarr (r\_group } r) \equiv \text{mcarr (r\_monoid } r)$$

3 A unifying framework: unification hints

In higher order logic, or also in first order logic modulo sufficiently powerful rewriting, unification $\mathcal{U}$ is undecidable. To avoid divergence and to manage the complexity of the problem, theorem provers usually implement a simplified, decidable unification algorithm $\mathcal{U}_o$, essentially based on first order logic, sometimes extended to cope with reduction (two terms t_1 and t_2 are unifiable if they have reducts t_1' and t_2'' - usually computed w.r.t. a given reduction strategy - which are first order unifiable). Unification hints provide a way to easily extend the system's unification algorithm $\mathcal{U}_o$ (towards $\mathcal{U}$) with heuristics to choose solutions which can be less than most general, but nevertheless constitute a sensible default instantiation according to the user.

The general structure of a hint is

$$\frac{\vec{?_x} := \vec{H}}{P \equiv Q} \text{ myhint}$$

where $P \equiv Q$ is a *linear* pattern with free variable $FV(P,Q) = \vec{?_v}$, $\vec{?_x} \subseteq \vec{?_v}$, all variables in $\vec{?_x}$ are distinct and H_i cannot depend on $?_{x_i}, \ldots, ?_{x_n}$. A hint is *acceptable* if $P[\vec{H} \,/\, \vec{?_x}] \equiv Q[\vec{H} \,/\, \vec{?_x}]$, i.e. if the two terms obtained by telescopic substitution, are *convertible*. Since convertibility is (typically) a decidable relation, the system is able to discriminate acceptable hints.

Hints are supposed to be declared by the user, or automatically generated by the systems in peculiar situation. Formally a unification hint induces a schematic unification rule over the schematic variables $\vec{?_v}$ to reduce unification problems to simpler ones:

$$\frac{\vec{?_x} \stackrel{?}{\equiv} \vec{H}}{P \stackrel{?}{\equiv} Q} \text{ myhint}$$

Since $\vec{?_x}$ are schematic variables, when the rule is instantiated, the unification problems $\vec{?_x} \stackrel{?}{\equiv} \vec{H}$ become non trivial.

When a hint is acceptable, the corresponding schematic rule for unification is sound (proof: a solution to $\vec{?_x} \stackrel{?}{\equiv} \vec{H}$ is a substitution σ such that $\vec{?_x}\sigma \equiv \vec{H}\sigma$ and thus $P\sigma \equiv P[\vec{H} \,/\, \vec{?_x}]\sigma \equiv Q[\vec{H} \,/\, \vec{?_x}]\sigma \equiv Q\sigma$; hence σ is also a solution to $P \stackrel{?}{\equiv} Q$).

From the user perspective, the intuitive reading is that, having a unification problem of the kind $P \stackrel{?}{\equiv} Q$, then the "hinted" solution is $?\overrightarrow{_x}:=\overrightarrow{H}$.

The intended use of hints is upon failure of the basic unification algorithm $\mathcal{U}_o$: the recursive definition `unif` that implements $\mathcal{U}_o$

```
let rec unif m n = body
```

is meant to be simply replaced by

```
let rec unif m n =
  try body
  with failure -> try_hints m n
```

The function try_hints simply matches the two terms m and n against the hints patterns (in a fixed order decided by the user) and returns the first solution found:

```
and try_hints m n =
  match m,n with
  | ...
  | P,Q when unif(x,H) as sigma -> sigma (* myhint *)
  | ...
```

This simple integration excludes the possibility of backtracking on hints, but is already expressive enough to cover, as we shall see in the next Section, all the cases discussed in Section 2.

Due to the lack of backtracking, hints are particularly useful when they are *invertible*, in the sense that the hinted solution is also unique, or at least "canonical" from the user point of view. However, even when hints are not canonical, they provide a strict and sound extension to the basic unification algorithm.

Hints may be easily indexed with efficient data structures investigated in the field of automatic theorem proving, like discrimination trees.

3.1 Implementing Canonical Structures

Every canonical structure declaration that declares T as the canonical solution for a unification problem $\pi_i\ ?_S \stackrel{?}{\equiv} t \mapsto ?_S := T$ can be simply turned in the corresponding unification hint:

$$\frac{?_S := T}{\pi_i\ ?_S \equiv t}$$

3.2 Implementing Type Classes

Like canonical structures, type classes are used to solve problems like $\pi_i\ ? \stackrel{?}{\equiv} t$, where π_i is a projection for a record type R. This kind of unification problem can be seen as inhabitation problems of the form "$? : R$ with $\pi_i := t$". Because

of the lack of the with construction in the Calculus of Inductive Constructions, Sozeau encodes the problem abstracting the record type over t, thus reducing the problem to the inhabitation of the type $R\ t$. Since the the structure of t is explicit in the type, parametric type class instances like the Cartesian product described in Section 2.2 can be effectively used as Prolog-like clauses to solve the inhabitation problem. This approach forces a particular encoding of algebraic structures, where all the fields that are used to drive inhabitation search have to be abstracted out. This practice has a nasty impact on the modularity of non trivial algebraic hierarchies, as already observed in [10,9].

Unification hints can be employed to implement type classes without requiring an ad-hoc representation of algebraic structures. The following hint schema

$$\frac{?_R := \{\pi_1 :=?_1 \ \ldots \pi_i :=?_i \ldots \pi_n :=?_n\}}{\pi_i\ ?_R \equiv ?_i}\ \text{h-struct-i}$$

allows to reduce unification problems of the form $\pi_i\ ? \stackrel{?}{\equiv} t$ to the inhabitation of the fields $?_1 \ldots ?_n$. Moreover, if we dispose of canonical inhabitants for these fields we may already express them in the hint. Note that the user is not required to explicitly declare classes and instances.

Unification hints are flexible enough to also support a different approach that does not rely on inhabitation but reduces the unification problem to simpler problems of the same kind.

For example, the unification problem

$$\text{gcarr}\ ?_1 \stackrel{?}{\equiv} \mathbb{Z} \times \mathbb{Z}$$

can be solved by the following hint:

$$\frac{?_1 := \text{gcarr}\ ?_3 \qquad ?_2 := \text{gcarr}\ ?_4 \qquad ?_0 :=?_3 \times ?_4}{\text{gcarr}\ ?_0 \equiv ?_1 \times ?_2}\ \text{h-prod}$$

Intuitively, the hint says that, if the carrier of a group $?_0$ is a product $?_1 \times ?_2$, where $?_1$ is the carrier of a group $?_3$ and $?_2$ is the carrier of a group $?_4$ then we *may guess* that $?_0$ is the group product of $?_3$ and $?_4$. This is not the only possible solution but, in lack of alternatives, it is a case worth to be explored.

3.3 Implementing Coercions Pullback

Coercions are usually represented as arrows between type schemes in a DAG. A type scheme is a type that can contain metavariables. So, for instance, it is possible to declare a coercion from the type scheme Vect $?_A$ to the type scheme List $?_A$. Since coercions form a DAG, there may exist multiple paths between two nodes, i.e. alternative ways to map inhabitants of one type to inhabitants of another type. Since an arc in the graph is a function, a path corresponds to the functional composition of its arcs. A coercion graph is coherent [8] when every two paths, seen as composed functions p_1 and p_2, are equivalent, i.e. $p_1 \equiv p_2$. In a coherent dag, any pair of cofinal coercions defines a hint pattern, and the corresponding pullback projections (if they exist) are the hinted solution.

Consider again the example given in Sect. 2.3. The generated hint is

$$\frac{?_1 := \text{r\_group } ?_3 \qquad ?_2 := \text{r\_monoid } ?_3}{\text{gcarr } ?_1 \equiv \text{mcarr } ?_2}$$

This hint is enough to solve all the unification problems listed in Table 1, that occur often when formalising algebraic structures (e.g. in [11]).

Table 1. Unification problems solved by coercion hints

Problem	Solution
gcarr $?_1 \stackrel{?}{\equiv}$ mcarr $?_2$	$?_1 :=$ r_group $?_3$, $?_2 :=$ r_monoid $?_3$
gcarr $?_1 \stackrel{?}{\equiv}$ mcarr (r_monoid $?_2$)	$?_1 :=$ r_group $?_2$
gcarr (r_group $?_1$) $\stackrel{?}{\equiv}$ mcarr $?_2$	$?_2 :=$ r_monoid $?_1$
gcarr (r_group $?_1$) $\stackrel{?}{\equiv}$ mcarr (r_monoid $?_2$)	$?_2 :=?_1$

4 Extensions

All the previous examples are essentially based on simple conversions involving records and projections. A natural idea is to extend the approach to more complex cases involving arbitrary, possibly recursive functions.

As we already observed, the natural use of hints is in presence of invertible reductions, where we may infer part of the structure of a term from its reduct.

A couple of typical situations borrowed from arithmetics could be the following, where *plus* and *times* are defined be recursion on the first argument, in the obvious way:

$$\frac{?_1 := 0 \qquad ?_2 := 0}{?_1 + ?_2 \equiv 0}\ \text{plus0} \qquad\qquad \frac{?_1 := 1 \qquad ?_2 := 1}{?_1 * ?_2 \equiv 1}\ \text{times1}$$

To understand the possible use of these hints, suppose for instance to have the goal

$$1 \leq a * b$$

under the assumptions $1 \leq a$ and $1 \leq b$; we may directly apply the monotonicity of times

$$\forall x, y, w, z. x \leq w \rightarrow y \leq z \rightarrow x * y \leq w * z$$

that will succeed unifying (by means of the hint) both x and y with 1, w with a and z with b.

Even when patterns do not admit a unique solution we may nevertheless identify an "intended" hint.

Consider for instance the unification problem

$$?_n + ?_m \stackrel{?}{\equiv} S\ ?_p$$

In this case there are two possible solutions:

$$1)\ ?_n := 0 \quad \text{and} \quad ?_m := S\ ?_p$$
$$2)\ ?_n := S\ ?_q \quad \text{and} \quad ?_p :=?_q+?_m$$

however, the first one can be considered as somewhat degenerate, suggesting to keep the second one as a possible hint.

$$\frac{?_n := S\ ?_q \qquad ?_p :=?_q+?_m}{?_n+?_m \equiv S\ ?_p}\ \text{plus-S}$$

This would for instance allow to apply the lemma le_plus : $\forall x, y : \mathbb{N}.x \leq y+x$ to prove that $m \leq S(n+m)$.
The hint can also be used recursively: the unification problem

$$?_j + ?_i \stackrel{?}{\equiv} S(S(n+m))$$

will result in two subgoals,

$$\frac{?_j \stackrel{?}{\equiv} S\ ?_q \qquad S(n+m) \stackrel{?}{\equiv} ?_q+?_i}{?_j+?_i \stackrel{?}{\equiv} S(S(n+m))}\ \text{plus-S}$$

and the second one will recursively call the hint, resulting in the instantiation $?_j := S(S\ n)$ and $?_i := m$ (other possible solutions, not captured by the hint, would instantiate $?_j$ with 0, 1 and 2).

4.1 Simple reflexive tactics implementation

Reflexive tactics [1,3] are characterised by an initial phase in which the problem to be processed is interpreted in an abstract syntax, that is later fed to a normalisation function on the abstract syntax that is defined inside the logic. This step needs to be performed outside the logic, since there is no way to perform pattern matching on the primitive CIC constructors (i.e. the λ-calculus application).

Let us consider a simple reflexive tactic performing simplification in a semigroup structure (that amounts to eliminating all parentheses thanks to the associativity property).

The abstract syntax that will represent the input of the reflexive tactic is encoded by the following inductive type, where EOp represents the binary semigroup operation and EVar a semi-group expression that is opaque (that will be treated as a black box by the reflexive tactic).

```
inductive Expr (S : semigroup) : Type :=
 | EVar : sgcarr S → Expr S
 | EOp : Expr S → Expr S → Expr S.
```

We call sgcarr the projection extracting the carrier of the semi-group structure, and semigroup the record type representing the algebraic structure under analysis. Associated to that abstract syntax there is an interpretation function $[\![\cdot]\!]_S$ mapping an abstract term of type Expr S to a concrete one of type sgcarr S.

```
let rec ⟦e : Expr S⟧(S:semigroup) : sgcarr S :=
  match e with
  [ EVar x ⇒ x
  | Eop x y ⇒ sgop S ⟦x⟧S ⟦y⟧S
  ].
```

The normalisation function simpl is given the following type and is proved sound:

```
let rec simpl (e: Expr S) : Expr S := ...
lemma soundness:
  ∀S:semigroup.∀P:sgcarr S → Prop.∀x:Expr S. P ⟦simpl x⟧S →P ⟦x⟧S
```

Given the following sample goal, imagine the user applies the soundness lemma (where P is instantiated with $\lambda x.x = d$).

$$a + (b + c) = d$$

yielding the unification problem

$$[\![?_1]\!]_g \stackrel{?}{\equiv} a + (b + c) \tag{2}$$

This is exactly what the extra-logical initial phase of every reflexive tactic has to do: interpret a given concrete term into an abstract syntax.

We now show how the unification problem is solved declaring the two following hints, where h-add is declared with higher precedence.

$$\frac{?_a := \text{Eop } ?_S\ ?_x\ ?_y \qquad ?_m := [\![?_x]\!]_{?_S} \qquad ?_n := [\![?_y]\!]_{?_S}}{[\![?_a]\!]_{?_S} \equiv ?_m + ?_n}\ \text{h-add}$$

$$\frac{?_a := \text{EVar } ?_S\ ?_z}{[\![?_a]\!]_{?_S} \equiv ?_z}\ \text{h-base}$$

Hint h-add can be applied to problem (2), yielding three new recursive unification problems. H-base is the only hint that can be applied to the second problem, while the third one is matched by h-add, yielding three more problems whose last two can be solved by h-base:

$$\dfrac{?_1 \stackrel{?}{\equiv} \text{Eop } g\ ?_x\ ?_y \qquad \dfrac{?_x \stackrel{?}{\equiv} \text{EVar } g\ a}{a \stackrel{?}{\equiv} [\![?_x]\!]_g}\ \text{h-base} \qquad \dfrac{?_y \stackrel{?}{\equiv} \text{Eop } g\ ?_x\ ?_y \qquad \dfrac{\vdots}{b \stackrel{?}{\equiv} [\![?_x]\!]_g} \qquad \dfrac{\vdots}{c \stackrel{?}{\equiv} [\![?_y]\!]_g}}{b + c \stackrel{?}{\equiv} [\![?_y]\!]_g}\ \text{h-add}}{[\![?_1]\!]_g \stackrel{?}{\equiv} a + b + c}$$

The leaves of the tree are all trivial instantiations of metavariables that together form a substitution that instantiates $?_1$ with the following expected term:

$$\text{Eop } g\ (\text{EVar } g\ a)\ (\text{Eop } g\ (\text{EVar } g\ b)\ (\text{EVar } g\ c))$$

4.2 Advanced reflexive tactic implementation

The reflexive tactic to put a semi-group expression in canonical form is made easy by the fact that the mathematical property on which it is based has linear variable occurrences on both sides of the equation:

$$\forall g : \text{semigroup}.\forall a, b, c : \text{sgcarr } g.a + (b + c) = (a + b) + c$$

If we consider a richer structure, like groups, we immediately have properties that are characterised by non linear variable occurrences, for example

$$\forall g : \text{group}.\forall x : \text{gcarr } g.x * x^{-1} = 1$$

To apply the simplification rule above, the data type for abstract terms must support a decidable comparison function. We represent concrete terms external to the group signature by pointers (De Bruijn indexes) to a heap (represented as a context Γ). Thanks to the heap, we can share convertible concrete terms so that the test for equality is reduced to testing equality of pointers.

```
record group : Type :={
  gcarr : Type;
  1 : gcarr;
  _ * _ : gcarr → gcarr → gcarr;
  _ ^-1 : gcarr → gcarr
}.
```

The abstract syntax for expressions is encoded in the following inductive type:

```
inductive Expr : Type :=
  | Eunit : Expr
  | Emult : Expr → Expr → Expr
  | Eopp : Expr → Expr
  | Evar : ℕ → Expr.
```

The interpretation function takes an additional argument that is the heap Γ. Lookup in Γ is written $\Gamma(m)$ and returns a dummy value when m is a dandling pointer.

```
let rec ⟦e : Expr; Γ : list (gcarr g)⟧_(g:group) on e : gcarr g :=
  match e with
  [ Eunit ⇒1
  | Emult x y ⇒⟦x; Γ⟧_g * ⟦y; Γ⟧_g
  | Eopp x ⇒⟦x; Γ⟧_g^-1
  | Evar n ⇒Γ(n) ].
```

For example:

$$[\![\text{Emult (Evar } O) \text{ (Emult (Eopp (Evar } O)) \text{ (Evar } (S\ O)))); [x; y]]\!]_g \equiv x * (x^{-1} * y)$$

The unification problem generated by the application of the reflexive tactic is of the form

$$[\![?_1;\ ?_2]\!]_{?_3} \stackrel{?}{\equiv} x * (x^{-1} * y)$$

and admits multiple solutions (corresponding to permutations of elements in the heap).

To be able to interpret the whole concrete syntax of groups in the abstract syntax described by the Expr type, we need the following hints:

$$\frac{?_a := \text{Emult } ?_x\ ?_y \qquad ?_m := [\![?_x;\ ?_\Gamma]\!]_{?_g} \qquad ?_n := [\![?_y;\ ?_\Gamma]\!]_{?_g}}{[\![?_a;\ ?_\Gamma]\!]_{?_g} \equiv ?_m * ?_n}\ \text{h-times}$$

$$\frac{?_a := \text{Eunit}}{[\![?_a;\ ?_\Gamma]\!]_{?_g} \equiv 1}\ \text{h-unit} \qquad \frac{?_a := \text{Eopp } ?_z \qquad ?_o := [\![?_z;\ ?_\Gamma]\!]_{?_g}}{[\![?_a;\ ?_\Gamma]\!]_{?_g} \equiv ?_o^{-1}}\ \text{h-opp}$$

To identify equal variables, and give them the same abstract representation, we need two hints, implementing the lookup operation in the heap (or better, the generation of a duplicate free heap by means of explicit sharing).

$$\frac{?_a := \text{Evar } 0 \qquad ?_\Gamma := ?_r :: ?_\Theta}{[\![?_a;\ ?_\Gamma]\!]_{?_g} \equiv ?_r}\ \text{h-var-base}$$

$$\frac{?_a := \text{Evar } (S\ ?_p) \qquad ?_\Gamma := ?_s :: ?_\Delta \qquad ?_q := [\![\text{Evar } ?_p;\ ?_\Delta]\!]_{?_g}}{[\![?_a;\ ?_\Gamma]\!]_{?_g} \equiv ?_q}\ \text{h-var-rec}$$

To understand the former rule, consider the following unification problem:

$$[\![\text{Evar } 0;\ ?_t :: ?_\Gamma]\!]_{?_g} \stackrel{?}{\equiv} x$$

Since the first context item is a metavariable, unification (unfolding and computing the definition of $[\![\text{Evar } 0;\ ?_t :: ?_\Gamma]\!]_{?_g}$ to $?_t$) instantiates $?_t$ with x, that amounts to reserving the first heap position for the concrete term x.

In case the first context item has been already reserved for a different variable, unification falls back to hint h-var-rec, skipping that context item, and possibly instantiating the tail of the context $?_\Gamma$ with $x :: ?_\Delta$ for some fresh metavariable $?_\Delta$.

We now go back to our initial example $[\![?_1;\ ?_2]\!]_{?_3} \stackrel{?}{\equiv} x * (x^{-1} * y)$ and follow step by step how unification is able to find a solution for $?_1$ and $?_2$ using hints. The algorithm starts by applying the hint h-times, yielding one trivial and two non trivial recursive unification problems:

$$\frac{?_1 \stackrel{?}{\equiv} \text{Emult } ?_x\ ?_y \qquad x \stackrel{?}{\equiv} [\![?_x;\ ?_2]\!]_{?_g} \qquad x^{-1} * y \stackrel{?}{\equiv} [\![?_y;\ ?_2]\!]_{?_g}}{[\![?_1;\ ?_2]\!]_{?_g} \stackrel{?}{\equiv} x * (x^{-1} * y)}\ \text{h-times}$$

The second recursive unification problem can be solved applying hint h-var-base:

$$\frac{?_x \stackrel{?}{\equiv} \text{Evar } 0 \qquad ?_2 \stackrel{?}{\equiv} x :: ?_\Theta}{x \stackrel{?}{\equiv} [\![?_x;\ ?_2]\!]}\ \text{h-var-base}$$

The application of the hint h-var-base forces the instantiation of $?_2$ with $x :: ?_\Theta$, thus fixing the first entry of the context to x, but still allowing the free instantiation of the following elements.

Under the latter instantiation, the third unification problem to be solved becomes $x^{-1} * y \stackrel{?}{\equiv} [\![?_y;\ x ::?_\Theta]\!]$ that requires another application of hint h-times followed by h-opp on the first premise.

$$\frac{?_y \stackrel{?}{\equiv} \text{Emult (Evar 0) } ?_y \qquad x^{-1} \stackrel{?}{\equiv} [\![?_{x'};\ x ::?_\Theta]\!]_{?_g} \qquad y \stackrel{?}{\equiv} [\![?_{y'};\ x ::?_\Theta]\!]_{?_g}}{x^{-1} * y \stackrel{?}{\equiv} [\![?_y;\ x ::?_\Theta]\!]_{?_g}}\ \text{h-times}$$

The first non-trivial recursive unification problem is $x^{-1} \stackrel{?}{\equiv} [\![?_{x'};\ x ::?_\Theta]\!]_{?_g}$ and can be solved applying hint h-opp first and then h-var-base. The second problem is more interesting, since it requires an application of h-var-rec:

$$\frac{?_{y'} \stackrel{?}{\equiv} \text{Evar } (S\ ?_p) \qquad x ::?_\Theta \stackrel{?}{\equiv} ?_s ::?_\Delta \qquad y \stackrel{?}{\equiv} [\![\text{Evar } ?_p;\ ?_\Delta]\!]_{?_g}}{y \stackrel{?}{\equiv} [\![?_{y'};\ x ::?_\Theta]\!]_{?_g}}\ \text{h-var-rec}$$

The two unification problems on the left are easy to solve and lead to the following instantiation

$$?_{y'} := \text{Evar } (S\ ?_p) \qquad ?_s := x; \qquad ?_\Delta :=?_\Theta$$

The unification problem left is thus $y \stackrel{?}{\equiv} [\![\text{Evar } ?_p;\ ?_\Theta]\!]_{?_g}$ and can be solved using hint h-var-base. It leads to the instantiation

$$?_p := \text{Evar } 0 \qquad ?_\Theta := y ::?_{\Theta'}$$

for a fresh metavariable $?_{\Theta'}$. Note that hint h-var-base was not applicable in place of h-var-rec since it leads to an unsolvable unification problem that requires the first item of the context to be equal to both x and y:

$$\frac{?_{y'} \stackrel{?}{\equiv} \text{Evar } 0 \qquad x ::?_\Theta \stackrel{?}{\equiv} y ::?_\Theta}{y \equiv [\![?_{y'};\ x ::?_\Theta]\!]_{?_g}}\ \text{h-var-base}$$

The solution found for the initial unification problem is thus:

$$?_1 := \text{Emult (Evar } O) \text{ (Emult (Eopp (Evar } O)) \text{ (Evar } (S\ O))))$$
$$?_2 := x :: y ::?_{\Theta'}$$

Note that $?_{\Theta'}$ is still not instantiated, since the solution for $?_1$ is valid for every context that extends $x :: y ::?_{\Theta'}$. The user has to choose one of them, the empty one being the obvious choice.

All problems obtained by the application of the soundness lemma are of the form $[\![?_1;\ ?_2]\!]_{?_3} \stackrel{?}{\equiv} t$. If t contains no metavariables, hints cannot cause

divergence since: h-opp, h-unit and h-times are used a finite number of times since they consume t; every other problem recursively generated has the form $[\![?_1;\ \vec{s}::?_\Gamma]\!]_{?_3} \stackrel{?}{\equiv} r$ where r is outside the group signature. To solve each goal, h-var-rec can be applied at most $\mid \vec{s} \mid + 1$ times and eventually h-var-base will succeed.

5 Conclusions

In a higher order setting, unification problems of the kind $f\ ?_i \stackrel{?}{\equiv} o$ and $?_f\ i \stackrel{?}{\equiv} o$ are extremely complex. In the latter case, one can do little better than using generate-and-test techniques; in the first case, the search can be partially driven by the structure of the function, but still the operation is very expensive. Moreover, higher order unification does not admit most general unifiers, so both problems above usually have several different solutions, and it is hard to guide the procedure towards the intended solution.

On the other side, it is simple to hint solutions to the unification algorithm, since the system has merely to check their correctness. By adding suitable hints in a controlled way, we can restrict to a first order setting keeping interesting higher-order inferences. In particular, we proved that hints are expressive enough to mimic some interesting ad-hoc unification heuristics like canonical structures, type classes and coercion pullbacks. It also seems that system provided unification errors in case of error-free formulae can be used to suggest to the user the need for a missing hint, in the spirit of "productive use of failure" [4].

Unification hints can be efficiently indexed using data structures for first order terms like discrimination trees. Their integration with the general flow of the unification algorithm is less intrusive than the previously cited ad-hoc techniques.

We have also shown an interesting example of application of unification hints to the implementation of reflexive tactics. In particular, we instruct the unification procedure to automatically infer a syntactic representation S of a term t such that $[\![S]\!] \equiv t$, introducing sharing in the process. This operation previously had to be done by writing a small extra-logical program in the programming language used to write the system, or in some ad-hoc language for customisation, like $\mathcal{L}$-tac [5]. Our proposal is superior since the refiner itself becomes able to solve such unification problems, that can be triggered in situations where the external language is not accessible, like during semantic analysis of formulae.

A possible extension consists in adding backtracking to the management of hints. This would require a more intrusive reimplementation of the unification algorithm; moreover it is not clear that this is the right development direction since the point is not to just add expressive power to the unification algorithm, but to get the right balance between expressiveness and effectiveness, especially in case of failure.

Another possible extension is to relax the linearity constraint on patterns with the aim to capture more invertible rules, like in the following cases:

$$\frac{?_x := 0}{?_x + ?_y \equiv ?_y}\ \text{plus-0} \qquad \frac{?_x := S\ ?_z}{?_x + ?_y \equiv S\ (?_z + ?_y)}\ \text{plus-S}$$

It seems natural to enlarge the matching relation allowing the recursive use of hints, at least when they are invertible. For instance, to solve the unification problem $?_1 + (?_2 + x) \stackrel{?}{\equiv} x$ we need to apply hint plus-0 but matching the hint pattern requires a recursive application of hint plus-0 (hence it is not matching in the usual sense, since $?_2$ has to be instantiated with 0). The properties of this "matching" relation need a proper investigation that we leave for future work.

References

1. Gilles Barthe, Mark Ruys, and Henk Barendregt. A two-level approach towards lean proof-checking. In *Types for Proofs and Programs (Types 1995)*, volume 1158 of *LNCS*, pages 16–35. Springer-Verlag, 1995.
2. Yves Bertot, Georges Gonthier, Sidi Ould Biha, and Ioana Pasca. Canonical big operators. In *TPHOLs*, pages 86–101, 2008.
3. Samuel Boutin. Using reflection to build efficient and certified decision procedures. In Martin Abadi and Takahashi Ito editors, editors, *Theoretical Aspect of Computer Software TACS'97, Lecture Notes in Computer Science*, volume 1281, pages 515–529. Springer-Verlag, 1997.
4. Alan Bundy, David Basin, Dieter Hutter, and Andrew Ireland. *Rippling: meta-level guidance for mathematical reasoning*. Cambridge University Press, New York, NY, USA, 2005.
5. David Delahaye. A Tactic Language for the System Coq. In *Proceedings of Logic for Programming and Automated Reasoning (LPAR), Reunion Island (France)*, volume 1955 of *Lecture Notes in Artificial Intelligence*, pages 85–95. Springer-Verlag, November 2000.
6. Georges Gonthier, Assia Mahboubi, Laurence Rideau, Enrico Tassi, and Laurent Thery. A modular formalisation of finite group theory. In *The 20th International Conference on Theorem Proving in Higher Order Logics*, volume 4732, pages 86–101, 2007.
7. Cordelia Hall, Kevin Hammond, Simon Peyton Jones, and Philip Wadler. Type classes in haskell. *ACM Transactions on Programming Languages and Systems*, 18:241–256, 1996.
8. Zhaohui Luo. Coercive subtyping. *J. Logic and Computation*, 9(1):105–130, 1999.
9. Zhaohui Luo. Manifest fields and module mechanisms in intensional type theory. In *TYPES 08*, 2009. To appear.
10. Claudio Sacerdoti Coen and Enrico Tassi. Working with mathematical structures in type theory. In *Proceedins of TYPES 2007*, volume 4941/2008 of *LNCS*, pages 157–172. Springer-Verlag, 2007.
11. Claudio Sacerdoti Coen and Enrico Tassi. A constructive and formal proof of Lebesgue's dominated convergence theorem in the interactive theorem prover Matita. *Journal of Formalized Reasoning*, 1:51–89, 2008.
12. Amokrane Saibi. Typing algorithm in type theory with inheritance. In *The 24th Annual ACM SIGPLAN - SIGACT Symposium on Principle of Programming Language (POPL)*, 1997.
13. Matthieu Sozeau and Nicolas Oury. First-class type classes. In *TPHOLs*, pages 278–293, 2008.

14. The Coq Development Team. The Coq proof assistant reference manual. `http://coq.inria.fr/doc/main.html`, 2005.
15. P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad hoc. In *POPL '89: Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 60–76, New York, NY, USA, 1989. ACM.
16. Markus Wenzel. Type classes and overloading in higher-order logic. In *TPHOLs*, pages 307–322, 1997.