Term inference for Dedukti

Gaëtan Gilbert, supervised by Arnaud Spiwack,
Équipe Deducteam at Inria Paris Rocquencourt

September 11, 2015

The general context

Dedukti is an experimental language designed to write proof checkers for various logics. It is used to implement independent proof checkers for proof assistants such as Coq and Isabelle and trace checkers for automated deduction tools such as Zenon and iprover modulo.

The logic underlying Dedukti is called $\lambda\Pi$-modulo: it is an extension of $\lambda\Pi$, the simplest form of dependently typed λ-calculus (also called λP), with user-defined rewriting rules for type equality (a.k.a. conversion) [5].

The research problem

Dedukti proofs require information to be made explicit when it could be deduced from the rest of the proof. This makes writing and reading real world proofs difficult.

Proof assistants such as Coq solve this problem by inserting a term inference phase before type checking. Given its small code base and low requirements for backward compatibility, Dedukti provides an opportunity to define a well-understood type inference implementation.

Your contribution

I designed and implemented the various parts necessary for term inference, notably extensions to the syntax and, self-contained with respect to each other, a safe unification kernel, an extensible unification engine making use of complex heuristics, and a constraint generating elaboration procedure, along with a specification of their semantics. The entire term inference system is expressed through a monadic interface to abstract certain operations.

Summary and future work

My implementation has been seen to behave well on small examples including code directly translated from Coq, which infers many non-trivial type annotations. Its code is modular and writing it was fairly painless, which is promising for future extensions.

Dedukti has two kinds of statements, declarations and patterns. Term inference currently only works for the first, patterns having subtly different semantics which involve unification. These semantics need to be clarified with regards to term inference before it can be expended to cover all Dedukti statements.

The proof assistant Matita provides implicit arguments as first-class elements of the syntax through a feature of its term inference [1]. Stéphane Lengrand has also used these co-variables in his unification algorithm [4]. It should be possible to extend Dedukti's term inference to cover that use case as well.

Finally the presentation of the unification heuristics as logical rules ordered by priority leaves open the possibility of allowing user-provided extensions to the unification through a system such as unification hints. That system, most famously implemented in Matita, is known to be able to emulate high level Coq features such as canonical structures and type classes.

Contents

1 Lambda Pi Calculus Modulo / Dedukti

Lambda Pi ($\lambda\Pi$) Calculus is a dependently typed $\lambda-$calculus, i.e. a calculus where types may contain values.

Dependent type calculi use computation inside types to shorten and clarify proofs, and makes proof verification more efficient. The computation appears in a conversion rule, usually $\beta-$conversion or $\beta\eta-$conversion. In $\lambda\Pi-$calculus modulo we may add rewrite rules to the conversion.

Consider a proof of $2+2=4$. In first order logic with the axioms of Peano arithmetic, it requires multiple uses of transitivity and of the $(S\ x)+y = S(x+y)$ axiom. In $\lambda\Pi-$modulo the latter axiom is replaced by a rewrite rule, making $2+2$ and 4 convertible. Then $2+2=4$ has a one-step proof which is efficient to check.

Let x, y, z be variable symbols and c, d constant symbols. We use a presentation where types are a subset of terms. The terms (including the types) of $\lambda\Pi$ calculus modulo are produced by the following grammar:

$$t, u, A ::= x \mid c \mid Kind \mid Type \mid t\ u \mid \lambda x : A.t \mid \Pi x : A.t$$

The terms $Kind$ and $Type$ are called sorts and classify types.

This grammar produces ground terms, which we will later extend with other constructors to support implicit terms and term inference.

Typing contexts associate types with variables:

$$\Gamma ::= [] \mid \Gamma, x : A$$

Alpha conversion, capture avoiding substitution $t[x \leftarrow u]$ and $\beta-$reduction are defined as usual for $\lambda-$calculus. Parallel substitutions are partial functions from variables to terms:

$$\sigma ::= \emptyset \mid \sigma, x \leftarrow t$$

They may be applied to a term as $\sigma(t)$ and to a context as $\sigma(\Gamma)$ by applying it to each term within the context.

Typing requires a signature Σ which associates types to constants via an operation denoted $c : A \in \Sigma$. The signature also provides term reduction $\triangleright$ and its equivalence closure the conversion $\equiv$. Since it is not modified throughout term inference, we assume a signature Σ verifying certain properties throughout this report.

We will define several properties by inference rules such as $\dfrac{\mathcal{A} \qquad \mathcal{B}}{\mathcal{C}}$, read as "from $\mathcal{A}$ and $\mathcal{B}$ we may deduce $\mathcal{C}$". The properties defining typing are the following:

- $\Gamma \vdash$ typing context Γ is well-formed.
- $\Gamma \vdash t = *$ term t is a sort under typing context Γ (for ground terms this is equivalent to $t \in \{Kind, Type\}$. In section 3 we will consider extended terms for which this judgement will be less trivial.)
- $\Gamma \vdash t : T$ term t has type T under typing context Γ.
- $\Gamma \vdash \sigma : \Delta$ substitution σ is a well-formed substitution between typing contexts Γ and Δ.

The following are defined in terms of the previous properties:

- $\Gamma \vdash t : *$ term t is a well-formed type under typing context Γ, defined as $\Gamma \vdash t = *$ or $\exists T, \Gamma \vdash T = *$ and $\Gamma \vdash t : T$. (For extended terms this judgement will also be modified.)
- $\Gamma \vdash t$ term t is well-formed under typing context Γ, defined as $\Gamma \vdash t = *$ or $\Gamma \vdash t : *$ or $\exists T, \Gamma \vdash t : T$.

Σ is required to verify properties such that:

- If $\Gamma \vdash$ and $\Gamma \vdash t$ then the reduction $\triangleright$ is confluent and strongly normalising on t. Then we can decide conversion.
- If $\Gamma \vdash$ any terms in Γ may be reduced without invalidating the judgement. If $\Gamma \vdash$ and $\Gamma \vdash t : T$ or $\Gamma \vdash t : *$ or $\Gamma \vdash t = *$ any terms in Γ, t and T may be reduced without invalidating the judgements.
- If Γ and $\Gamma \vdash \sigma : \Delta$ and $\Delta \vdash t : T$ then $\Gamma \vdash \sigma(t) : \sigma(T)$, with analogous properties for $\Delta \vdash t : *$ and $\Delta \vdash t = *$.
- If $\Gamma \vdash$ and $\Gamma \vdash t : T$ then $\Gamma \vdash T : *$.

Finally the definitions:

Well-formed typing context

$$\frac{}{[] \vdash}$$

$$\frac{\Gamma \vdash \qquad \Gamma \vdash A : Type}{\Gamma, x : A \vdash}$$

Variables must have a type sorted by $Type$, but constants may have a type sorted by $Kind$.

Sorts

$$\frac{t \in \{Kind, Type\}}{\Gamma \vdash t = *}$$

Typed substitutions

$$\frac{}{\Gamma \vdash \emptyset : []}$$

$$\frac{\Gamma \vdash \sigma : \Delta \qquad \Gamma \vdash t : A}{\Gamma \vdash (\sigma, x \leftarrow t) : (\Delta, x : A)}$$

Typing The typing rules are normal for dependent $\lambda-$calculus, resembling the following one. The complete list is found in appendix A.

$$\frac{\Gamma \vdash f : T \qquad T \rhd \Pi x : A.B \qquad \Gamma \vdash u : A}{\Gamma \vdash f\ u : B[x \leftarrow u]}$$

2 Monads

2.1 State monad

A function to count the nodes in a tree is easy to define in imperative style, and that definition can be translated to a functional language:

```
let count t = let rec count acc = function
  | Node(left,right) -> let acc1 = acc+1 in
    let acc_left = count acc1 left in
    count acc_left right
  | Leaf -> acc
  in count 0 t
```

The variable *acc* is a counter incremented at each $Node$. We need to pass it explicitly at each explicit call, and a simple typing error could result in passing an outdated counter, for instance *acc* instead of *acc*1 when computing acc_left.

We can automatise this operation by providing the following type and functions:

```
type 'a t = int -> 'a*int

let return (x:'a) : 'a t = fun s -> x,s
let (>>=) : (m:'a t) -> (f:'a -> 'b t) : 'b t =
```

```
  fun s -> let x,s' = m s in f x s'

let get : int t = fun s -> s,s
let set : (n:int) : unit t = fun _ -> (),n
```

Then the count function becomes

```
let count t = let rec count : tree -> unit t = function
  | Node(left,right) -> get >>= fun acc ->
    set (acc+1) >>= fun () ->
    count left >>= fun () ->
    count right
  | Leaf -> return ()
  in let (),n = count 0 in
    n
```

The counter is now passed implicitly and we ensure that it is always up to date.

Of course, we can provide the same functions by using an arbitrary type *state* instead of *int*. Other implementations are also possible if the type is made abstract, although we then need a function $run : \alpha\ t \to state \to \alpha * state$ to use the result.

2.2 Definition

The previous construct can be generalised to functionalities other than providing a state in a functional setting.

A monad is a structure representing computations along with ways to combine them. In Ocaml, a monad is the combination of the following elements:

```
type 'a t
val return : 'a -> 'a t
val (>>=) : 'a t -> ('a -> 'b t) -> 'b t
```

Values of type $\alpha\ t$ represent computations producing a value of type α. Such computations can be produced from a value of type α by the function *return*, and computations are combined by the function >>= (also called *bind*, as it binds the result of the first computation inside the second).

The functions should verify the monad identities:

```
bind (return x) f = f x
bind m return = m
bind m (fun x -> bind (f x) g) = bind (bind m f) g
```

2.3 Backtracking monad

A backtracking monad provides a way to raise errors of type *err* and backtrack depending on them with the following functions:

```
val zero : err -> 'a t
val plus : 'a t -> (err -> 'a t) -> 'a t
```

zero raises an error and *plus* inserts a backtracking point: with *plus* m f, if computation fails with an error e in m or while using the output of m, we compute using f e instead.

They verify equations such as

```
plus (zero e) f = f e
plus (plus a b) c = plus a (fun e -> plus (b e) c)
zero e >>= f = zero e
(plus a b) >>= f = plus (a >>= f) (fun e -> b e >>= f)
```

Values are extracted with

```
type 'a out = | Nil of err | Cons of 'a*(err -> 'a out)
val run : 'a t -> 'a out
```

If we don't consider errors, a value of type α *out* is a list of values of type α such as we could obtain from a nondeterministic computation.

Backtracking monads are defined in depth in [3].

2.4 Monad transformers

To provide term inference we will use a monad with both state effects and backtracking. Although we could simply define such a monad, monad transformers are a relatively generic way of combining monad definitions. Note that composing monads $t1$ and $t2$ by taking *type* α $t = \alpha$ $t1$ $t2$ does not work: it does not have a monad structure in general.

A monad transformer is an operation taking a monad with type α m and producing a monad α t with the ability to produce computations in the later from those in the former, i.e. a function

```
val lift : 'a m -> 'a t
```

It should verify

```
lift (return x) = return x
lift (m >>= k) = lift m >>= fun x -> lift (k x)
```

The identity monad, where computations of type $'a$ are values of type α, serves as a basic monad from which others are constructed with monad transformers.

It is possible to define a monad transformer adding a state to a monad, as well as a transformer adding backtracking. However, when the state transformer is applied to a monad which has backtracking operations, those operations are not immediately available to the resulting monad. We need to write specific code depending on the implementation of the state transformer to lift the backtracking operations through it. Since this work needs to be done for each combination of effect and monad transformer except when the effects are simple enough to be passed through the $lift$ function, monad transformers may not scale to a large number of effects.

For our purposes, two effects (state and backtracking) are sufficient, so monad transformers are an acceptable solution.

The order in which monad transformers are applied matters: applying the backtracking transformer to a state monad produces a monad where backtracking does not reset the state, whereas applying the state transformer to a backtracking monad produces a monad where backtracking does reset the state. We will use the latter.

3 Safe unification

During type inference, we will at times have a term t which we know has type A while we need a term with type B (for instance when inferring the type of $f\ x$, with $f : B \to C$ and $x : A$ inferred). In that case, making the terms A and B convertible solves the problem: we call this unification.

In this section, we describe what information is contained in a unification problem and what properties it should verify. We introduce the constructs of guards and of metavariables with explicit substitutions and describe their purpose. Finally we describe how to manipulate unification problems while preserving their semantics.

3.1 Unification problem

Ground terms are extended into "extended terms" by two new constructors and typing rules are added to cover them. Typing rules for extended terms involve an implicit unification problem Θ (to be inferred from context, or we may say for example "$\Gamma \vdash t : T$ under Θ". In the implementation it is accessed through the monadic interface.) whose operations are defined below, and use an extended reduction and conversion.

3.1.1 Metavariables

Definition 3.1 (Partial term)**.** *Partial terms are ground terms extended by a constructor to support implicit terms:*

$$t ::= ... \mid ?$$

The placeholder term ? is intended to be replaced by a different ground term, possibly capturing variables, for each appearance in a partial term.

Partial terms cannot be typed.

Definition 3.2 (Metavariable)**.** *A metavariable is an extended term $?_i[\sigma]$ where $?_i$ is a name and σ a substitution of extended terms.*

Metavariable terms contain a substitution to express dependencies, i.e. the variables which may be used when replacing the metavariable by a term.

Consider a partial term $(\lambda x : A.?)\, y$ which we unify with y. We may replace the placeholder with x (which is captured by the abstraction) or with y. However if we reduce to ? that information is lost and only the later solution can be found.

By adding the substitution, before reduction we have $(\lambda x : A.?[x \leftarrow x, y \leftarrow y])\, y$ and after we have $?[x \leftarrow y, y \leftarrow y]$, retaining the capture information.

In formal terms, we will in a moment define functions θ which we call meta-substitutions. Finding a solution to a unification problem is equivalent to finding the right meta-substitution. Having explicit substitution attached to metavariables makes it so that meta-substitutions are compatible with reduction: if $t \rhd t'$ then $\theta(t) \rhd \theta(t')$. See for instance [2] for a detailed analysis of explicit substitutions.

Definition 3.3 (Metavariable typing declaration)**.** *A metavariable typing declaration is a statement of the form $\Gamma \vdash ?_i : T$ or $\Gamma \vdash ?_i : *$ or $\Gamma \vdash ?_i = *$ where Γ is a typing context, $?_i$ is a metavariable name and T is a term.*

Unification problems contain at most one metavariable typing declaration for each metavariable name. We write for instance $(\Gamma \vdash ?_i : *) \in \Theta$ when $?_i$ is a type under context Γ and unification problem Θ.

When a unification problem Θ contains a typing declaration for a metavariable $?_i$, $?_i$ is said to be declared in Θ.

Definition 3.4 (Metavariable typing)**.** *The typing rules are extended with rules of the form*

$$\frac{(\Delta \vdash ?_i : T) \in \Theta \qquad \Gamma \vdash \sigma : \Delta}{\Gamma \vdash ?_i[\sigma] : \sigma(T)}$$

There is one for each kind of declaration, as detailed in the appendix A.

3.1.2 Guards

Definition 3.5 (Unification constraint)**.** *A unification constraint is a statement* $\Gamma \vdash A \equiv B$ *where* Γ *is a typing context and* A *and* B *are terms.*

It is read as "A *and* B *need to be unified under context* Γ*".*

During inference, we may have a term t which we know verifies $x : X \vdash t : A$ when we need a term verifying $x : X \vdash t : B$. This creates a unification constraint $x : X \vdash A \equiv B$.

Let $\Omega = (\lambda x :?.x\ x)\ (\lambda x :?.x\ x)$. There is no way to give it a type and reduction on it does not terminate. When typing the term $\lambda y : \Omega : y\ y$ we will need to check that the type of y converts to a product type. Without specific precautions, we must make sure that we have detected that Ω should not be reduced before doing that check. Since Ω cannot be typed due to the lack of solutions to the equation $? \equiv \Pi z :?.?'$ which appears when checking either half of Ω, we have to check constraints in a certain order and they cannot be delayed.

With guards, we create a "guarded constant" p of type $X \to A \to B$ to which we associate the constraint $x : X \vdash A \equiv B$, then we can use $x : X \vdash p\ x\ t : B$.

If we successfully unify A and B, we can then safely replace p by $\lambda\ x\ y.y$ so that $p\ x\ t$ reduces to t.

For compactness, we internalise the context $x : X$ as a substitution (used only for typing) and reduce $p[\sigma]\ t$ to t in one step.

This is of course generalisable to contexts with more than one variable, as we will do in section 4.

Guards make it so that solving separate constraints can be interleaved or even parallelized (if they involve separate metavariables). However this is future work: the current algorithm does not make use of that capability to solve more cases or to execute faster but only to make the code slightly cleaner.

Definition 3.6 (Guarded term)**.** *A guarded term is an extended term* $p[\sigma]\ t$ *where* p *is a guard name,* σ *a substitution and* t *an extended term.*

Definition 3.7 (Guard typing declaration)**.** *A guard typing declaration is a statement of the form* $\Gamma \vdash p : A \to B$ *where* Γ *is a typing context,* p *is a guard name and* A *and* B *are terms.*

Unification problems contain at most one typing declaration for each guard name, which is then said to be declared for that problem.

Definition 3.8 (Guard typing)**.** *The typing rules are extended with a rule*

$$\frac{(\Delta \vdash p : A \to B) \in \Theta \qquad \Gamma \vdash \sigma : \Delta \qquad \Gamma \vdash t : \sigma(A)}{\Gamma \vdash p[\sigma]\ t : \sigma(B)}$$

3.1.3 Meta-substitution

Definition 3.9. *A metavariable definition is a statement $?_i \coloneqq t$ where $?_i$ is a metavariable name and t a term. $?_i$ is said to depend on every metavariable appearing in t.*

Definition 3.10 (Meta-substitution)**.** *A meta-substitution θ is a collection of metavariable definitions $?_i \coloneqq t_i$ and of pass-through guards p_j.*

Definition 3.11 (Terminating meta-substitution)**.** *A meta-substitution θ is terminating if for each metavariable defined in θ it does not depend directly or through other metavariables on itself.*

In other words, the directed graph whose nodes are the metavariables defined in θ and with arcs from each metavariable to those it depends on has no cycles.

Definition 3.12 (Applying a meta-substitution)**.** *A meta-substitution θ induces a (partial) function on terms also denoted θ defined inductively:*

- *for the ground term constructors simply apply θ to each subterm, e.g. $\theta(f\ u) \coloneqq \theta(f)\ \theta(u)$.*
- *$\theta(?_i[\sigma]) \coloneqq \theta(\sigma(t_i))$ if $(?_i \coloneqq t_i) \in \theta$. (This is not always valid, e.g. if $t_i = ?_i[]$.)*
- *$\theta(?_i[\sigma]) \coloneqq ?_i[\theta \circ \sigma]$ otherwise.*
- *$\theta(p[\sigma]\ t) \coloneqq \theta(t)$ if $p \in \theta$.*
- *$\theta(p[\sigma]\ t) \coloneqq p[\theta \circ \sigma]\ \theta(t)$ otherwise.*

Lemma 3.13. *If a meta-substitution θ is terminating then the induced function is total.*

3.1.4 Unification problems

Definition 3.14 (Unification problem)**.** *A unification problem is a list of metavariable declarations, metavariable definitions and guard declarations each associated with a list of unification constraints.*

Definition 3.15 (Induced substitution)**.** *A unification problem Θ induces a meta-substitution θ formed by the collection of the metavariable definitions and the guards associated with no constraints in Θ.*

The reduction used when type checking extended terms is modified so that $t \rhd \theta(t)$ where θ is the induced meta-substitution of the implicit unification problem. The conversion is extended to match.

Definition 3.16 (Valid metavariable declaration)**.** *A metavariable declaration $\Gamma \vdash ?_i : *$ (resp. $\Gamma \vdash ?_i = *$, resp. $\Gamma \vdash ?_i : T$) is valid for a unification problem Θ when $?_i$ is not declared in Θ and Γ is a valid typing context under Θ (resp. idem, resp. idem and $\Gamma \vdash T : *$ under Θ).*

Definition 3.17 (Valid metavariable definition)**.** *A metavariable definition $?_i \coloneqq t$ is valid for a unification problem Θ when one of the following is true:*

- $(\Gamma \vdash ?_i : T) \in \Theta$ *and* $\Gamma \vdash t : T$
- $(\Gamma \vdash ?_i : *) \in \Theta$ *and* $\Gamma \vdash t : *$
- $(\Gamma \vdash ?_i = *) \in \Theta$ *and* $\Gamma \vdash t = *$

Additionally $?_i$ must not depend on itself when considering the definitions in Θ as well as the one being considered.

Definition 3.18 (Valid guard declaration)**.** *A guard declaration $\Gamma \vdash p : A \to B$ associated with a list of unification constraints $(\Gamma_i \vdash u_i \equiv v_i)_{i<n}$ is valid for a unification problem Θ when all the following are true:*

- *p is not declared in Θ*
- *For each $i < n$ and Θ' formed from Θ by adding valid metavariable definitions, if for every $j < i$ Θ' unifies u_j and v_j, then under Θ' Γ_i is a valid typing context such that $\Gamma_i \vdash u_i$ and $\Gamma_i \vdash v_i$.*
- *For each Θ' formed by adding valid metavariable definitions to Θ, if for each $i < n$ Θ' unifies u_i and v_i then Θ' unifies A and B.*

Definition 3.19 (Valid unification problem)**.** *A unification problem is valid when each metavariable declaration and definition and each guard declaration added to the prefix Θ_0 is valid under Θ_0.*

Lemma 3.20. *A valid unification problem induces a terminating meta-substitution.*

Theorem 3.21. *If Θ is a valid unification problem inducing meta-substitution θ and under which $\Gamma \vdash t : T$ (resp. $\Gamma \vdash t : *$, resp. $\Gamma \vdash t = *$), then $\theta(\Gamma) \vdash \theta(t) : \theta(T)$ (resp. $\theta(\Gamma) \vdash \theta(t) : *$, resp. $\theta(\Gamma) \vdash \theta(t) = *$).*

Theorem 3.22. *If Θ is a valid unification problem under which $\Gamma \vdash t : T$ (resp. $\Gamma \vdash t : *$, resp. $\Gamma \vdash t = *$) and Γ, t and T contain no metavariables and no guards, then $\Gamma \vdash t : T$ (resp. $\Gamma \vdash t : *$, resp. $\Gamma \vdash t = *$) with the standard $\lambda\Pi$ calculus modulo rules.*

Definition 3.23 (Restriction of a problem)**.** *If Θ_1 and Θ_2 are valid unification problems, Θ_2 is a restriction of Θ_1 when for all Γ, t and T such that $\Gamma \vdash t : T$ (resp. $\Gamma \vdash t : *$, resp. $\Gamma \vdash t = *$) under Θ_1, it is also true under Θ_2.*

Remark 3.24. *The order of the components of a unification problem has no effect on the semantics, we only care that there exists a valid order.*

The overall inference algorithm can now be described: from a partial term t_0 and the empty unification problem, we produce extended terms t and T and a valid unification problem Θ_0 such that $[] \vdash t : T$ under Θ_0. We compute a restriction Θ of Θ_0 by solving constraints, with induced meta-substitution θ removing all metavariables and guards. Then $[] \vdash \theta(t) : \theta(T)$ under Θ.

We will obtain ground terms $t^* \coloneqq \theta(t)$ and $T^* \coloneqq \theta(T)$ such that $[] \vdash t^* : T^*$ by the rules of $\lambda\Pi$ calculus modulo.

3.2 Safe operations

Given a valid problem, how can we produce a valid restriction of it?

3.2.1 Simple operations

Lemma 3.25. *A valid unification problem is a restriction of itself.*

Lemma 3.26. *Adding valid declarations and definitions to a unification problem produces a restriction of it.*

Adding a valid metavariable definition is also called metavariable refinement.

Corollary 3.27 (Metavariable narrowing)**.** *If $?_i$ is a metavariable with declaration $\Gamma \vdash ?_i : T$ (resp. $\Gamma \vdash ?_i : *$, resp. $\Gamma \vdash ?_i = *$), $?_j$ a fresh metavariable name, Δ a valid context formed by selecting only certain variables from Γ and such that all free variables of T appear in Δ, we may add a declaration $\Delta \vdash ?_j : T$ (resp. $\Delta \vdash ?_j : *$, resp. $\Delta \vdash ?_j = *$) and a definition $?_i \coloneqq ?_j[id_\Delta]$.*

Metavariable narrowing prevents the use of some variables in valid definitions of a given metavariable.

3.2.2 Forcibly typed metavariable

Lemma 3.28. *If $?_i$ is declared and not defined in Θ, we can form a restriction of Θ where $?_i$ is typed:*

- *If $(\Gamma \vdash ?_i : T) \in \Theta$ no change is necessary.*
- *If $(\Gamma \vdash ?_i : *) \in \Theta$, let $?_k$ a fresh metavariable name. We replace the declaration for $?_i$ with the 2 following declarations:*

  $\Gamma \vdash ?_k = *$ *and* $\Gamma \vdash ?_i : ?_k[id]$
- *If $(\Gamma \vdash ?_i = *) \in \Theta$, we replace the declaration for $?_i$ with the 2 following elements:*

  $\Gamma \vdash ?_i : Kind$ *and* $?_i \coloneqq Type$

If $?_i$ is declared and defined, a similar operation may be possible. For instance, if $?_i$ declared as a type is defined by $?_j$ also declared as a type and not itself defined, we force $?_j$ to be typed and then use its type for $?_i$. How often this problem occurs in practice has not been investigated.

3.2.3 Constraint simplification

Lemma 3.29. *If p is a guard associated with a list of constraints beginning by $\Gamma \vdash u \equiv v$, we may replace that first constraint:*

- *if u and v are unified, by no constraint.*
- *by $\Gamma \vdash u' \equiv v'$ with $u \rhd^* u'$ and $v \rhd^* v'$.*
- *by a decomposition if u and v have the same head shape, e.g. both are applications or both are abstractions.*

  In the application case $\Gamma \vdash f\ u \equiv g\ v$ is replaced by $\Gamma \vdash f \equiv g$ and $\Gamma \vdash u \equiv v$.

  See appendix C for all cases.

Remark 3.30. *Some of these transformations are invertible.*

4 Elaboration

We define several functions to transform partial terms into extended terms. A valid unification problem Θ is threaded throughout the definitions and is safely modified.

The following functions are wrappers around safe operations:

Definition 4.1 (New $\Gamma \vdash ?_i : T$)**.** *This function declares a new metavariable.*

Inputs Γ *and* T*, outputs* $?_i$*.*

Precondition: $\Gamma \vdash T : *$*.*

Postconditions: $\Gamma \vdash ?_i : T$*,* $?_i$ *fresh metavariable.*

It is implemented by adding a new metavariable declaration. The rule corresponding to type metavariable declarations is also available.

Definition 4.2 ($\Gamma \vdash u : A \overset{\mathcal{C}}{\rightsquigarrow} v : B$)**.** *This function ensures that a term has a certain type.*

Inputs Γ, u, A *and* B*, outputs* v*.*

Preconditions: $\Gamma \vdash u : A$ *and* $\Gamma \vdash B : *$*.*

Postcondition: $\Gamma \vdash v : B$*.*

It is implemented by adding a new guard declaration and applying the guard to u*.*

Definition 4.3 ($\Gamma \vdash u : A \overset{\mathcal{C}}{\rightsquigarrow}{}^{*} v : s$)**.** *This function ensures that a term is typed by a sort.*

Inputs Γ, u *and* A*, outputs* v *and* s*.*

Precondition: $\Gamma \vdash u : A$*.*

Postconditions: $\Gamma \vdash v : s$ *and* $\Gamma \vdash s = *$*.*

It is implemented by declaring a new sort metavariable, then a guard with the metavariable as the target type.

Definition 4.4 ($\Gamma \vdash t \neq Kind$)**.** *This function ensures that a term will not be unified with* $Kind$*.*

Inputs: Γ *and* t*, outputs only the implicit unification problem.*

Precondition: $\Gamma \vdash t$*.*

Postcondition: there exists a term T *such that* $\Gamma \vdash t : T$*.*

t *being well-formed, if it is a metavariable we can force it to be typed, otherwise either it is* $Kind$ *and typing fails, or it cannot be unified with* $Kind$*.*

The following functions are mutually recursive:

Definition 4.5 ($\Gamma \vdash t \Uparrow u : T$)**.** *Type inference.*
Inputs Γ *and* t*, outputs* u *and* T*.*
Precondition: Γ *valid typing context.*
Postcondition: $\Gamma \vdash u : T$*.*

Definition 4.6 ($\Gamma \vdash t : T \Downarrow u$)**.** *Type forcing.*
Inputs Γ, t *and* T*, outputs* u*.*
Precondition: $\Gamma \vdash T : *$*.*
Postcondition: $\Gamma \vdash u : T$*.*

Type forcing inserts a guard:

$$\frac{\Gamma \vdash t \Uparrow u : A \qquad \Gamma \vdash u : A \overset{\mathcal{C}}{\rightsquigarrow} v : B}{\Gamma \vdash t : B \Downarrow v}$$

Type inference

Some rules are trivial, such as the one for variables (the full list can be found in appendix B):

$$\frac{x : A \in \Gamma}{\Gamma \vdash x \Uparrow x : A}$$

This leaves simple rules for products and abstractions:

$$\frac{\Gamma \vdash a : Type \Downarrow A \qquad \Gamma, x : A \vdash b \Uparrow B_0 : s_0 \qquad \Gamma, x : A \vdash B_0 : s_0 \overset{\mathcal{C}}{\rightsquigarrow}{}^{*} B : s}{\Gamma \vdash \Pi x : a.b \Uparrow \Pi x : A.B : s}$$

$$\frac{\Gamma \vdash a : Type \Downarrow A \qquad \Gamma, x : A \vdash t \Uparrow u : B \qquad \Gamma, x : A \vdash B \neq Kind}{\Gamma \vdash \lambda x : a.t \Uparrow \lambda x : A.u : \Pi x : A.B}$$

Holes are replaced by metavariables:

$$\frac{\text{New } \Gamma \vdash ?_k : * \qquad \text{New } \Gamma \vdash ?_j : ?_k[id]}{\Gamma \vdash ? \Uparrow ?_j[id] : ?_k[id]}$$

This rule is the main source of type metavariables.

Applications use one of two rules depending on the type of the application head:

$$\frac{\Gamma \vdash f \Uparrow g : T \qquad T \rhd_{whnf} \Pi x : A.B \qquad \Gamma \vdash u : A \Downarrow v}{\Gamma \vdash f\ u \Uparrow g\ v : B[x \leftarrow v]}$$

$$\frac{\begin{array}{ccc}\Gamma \vdash f \Uparrow g_0 : T & T \rhd_{whnf} ?_i[\sigma]\ t_1 ... t_n & \Gamma \vdash u \Uparrow v_0 : A_0 \\ \Gamma \vdash A_0 : s & \Gamma \vdash A_0 : s \overset{\mathcal{C}}{\rightsquigarrow} A : Type & \Gamma \vdash v_0 : A_0 \overset{\mathcal{C}}{\rightsquigarrow} v : A \\ \text{New } \Gamma, x : A \vdash ?_s = * & \text{New } \Gamma, x : A \vdash ?_k : ?_s[id] & \Gamma \vdash g_0 : T \overset{\mathcal{C}}{\rightsquigarrow} g : \Pi x : A.\, ?_k[id]\end{array}}{\Gamma \vdash f\ u \Uparrow g\ v : ?_k[x \leftarrow v]}$$

If the type of the head of the application is a metavariable, we first infer the type of the argument. As the type of the argument of a function, it must be of type $Type$. Once this is assured, we unify the type of the head with a Π whose first component if the type of the argument and the second component is left indeterminate, i.e. a metavariable typed by a sort.

Remark 4.7. *Since the pre- and post-conditions do not relate the input and output terms, we could modify the terms arbitrarily in the* $\overset{\mathcal{C}}{\rightsquigarrow}$ *function. For instance, it could be replaced by a coercion system which could default with inserting a unification guard.*

5 Unification resolution

We make progress in solving a unification problem by applying one of the constraint simplification operations. In order to be able to eliminate constraints, we usually need to narrow or refine metavariables, and introducing new guards may be needed to do the later.

The first decision to make before modifying a unification problem is which constraint should be considered. My current implementation in Dedukti is naive, which results in the entire algorithm only making use of the constraint delaying capability which guards provide in the inversion operation presented below.

Once a constraint is selected, we attempt to apply certain heuristics to it, backtracking if we fail. The heuristics are heavily inspired by [6].

Essentially, we try to reduce constraints to matching problems, which are decidable. This is achieved through decomposing constraints, reducing in constraints and narrowing metavariables to remove dependencies.

The heuristics are presented as inference rules, transforming a constraint in the conclusion into none or some constraints in the premisses. For instance in the following rule we use metavariable narrowing to solve a constraint:

$$\frac{\begin{array}{ccc}(\Gamma \vdash ?_i : T) \in \Theta & \Delta = \sigma_1 \cap \sigma_2 & FV(T) \subseteq \Delta \\ \text{Narrow meta } ?_i \text{ to } \Delta \vdash ?_j : T & & \overrightarrow{\Gamma \vdash t_n \equiv u_n}\end{array}}{\Gamma \vdash ?_i[\sigma_1]\ \overrightarrow{t_n} \equiv ?_i[\sigma_2]\ \overrightarrow{u_n}}\ \text{meta-same}$$

Specifically, when we encounter a constraint $?_i[\sigma_1]\ \overrightarrow{t_n} \equiv ?_i[\sigma_2]\ \overrightarrow{u_n}$, we forbid the use of the variables in Δ which are given different values in σ_1 and σ_2 to define $?_i$. Then $?_i[\sigma_1] \equiv ?_j[\sigma'] \equiv ?_i[\sigma_2]$ where σ' is the common part of the other substitutions, and we only have to unify the arguments t_i and u_i respectively.

The other rules are described in appendix D.

Certain operations, such as removing solved constraints or replacing a constraint equating two abstractions by the decomposed constraints, can be done without loss of generality. They are applied eagerly and without inserting backtracking points.

6 Implementation notes

This section describes two ways in which we reuse code, as well as some high level information on code organisation.

6.1 Typing

Type inference is implemented as an extension of type checking. Dedukti uses type checking for three purposes:

- for partial terms, to produce extended terms and a unification problem which needs to be solved, as described in section 4.
- for extended terms under an arbitrary unification problem, to make metavariable refinement safe.
- for ground terms, to ensure that soundness errors in the unification implementation are caught.

All can be implemented as elaboration parametrised by the non-recursive functions, the presence or absence of a unification problem being hidden by the monadic presentation. For instance, type checking for ground terms replaces the "guard term" operation $\Gamma \vdash t : A \overset{\mathcal{C}}{\rightsquigarrow} u : B$ by a conversion check between terms A and B.

6.2 Polymorphic term type

Parametric elaboration is made easier by using an open type to represent terms:

```
type 'a term = Kind | Type | ...
  | Extra of 'a tkind * 'a
```

The type *tkind* is a GADT which allows us to do generic operations over open terms: without it functions like printing would need an extra argument to print *Extra* values.

Then the extra types are:

```
type ground = { exfalso : 'r. 'r }
type partial = unit
type extended =
  | Meta of meta_name*extended substitution
  | Guard of guard_name*extended substitution*extended term
```

They describe which extensions need to be made over ground terms: ground terms have no extensions over themselves, so we use the empty type encoded in Ocaml. Partial terms have one extension, the placeholder term ?, encoded as the unit type. Extended terms have two extensions, metavariables which consist of a name and a substitution, and guarded terms consisting of a name, a substitution and an extended term.

For instance, the term $S\ ?_i[\emptyset]$ where S is a constant is encoded as

$$App\ (Const\ S)\ (Extra\ Partial\ Meta(i, \emptyset))$$

(give or take a few details).

Open terms allow us to only deal with the extra values relevant to the kind of type inference we are doing. For instance, when typing ground terms, we have something like

```
match t:ground term with
  | Kind -> ...
  ...
  | Extra (Ground,ex) -> ex.exfalso
```

as opposed to using *assert false* on the metavariable constructor.

6.3 Code organisation

The important changes are divided across six implementation/interface file pairs, five of which are new, although a larger proportion of the code had to be superficially touched to deal with extensions to the syntax and changes to the term type.

- typing.ml (329 LOC to 447 LOC, of which 66 are an unrelated experiment not used by the program) and typing.mli (63 LOC to 88 LOC): the original type checking code for ground terms was adapted into the shared part of type checking as well as the functions specific to ground terms.
- refine.ml (100 LOC) and refine.mli (12 LOC): contains the functions specific to elaborating partial terms to extended terms.
- msubst.ml (128 LOC) and msubst.mli (37 LOC): implements meta-substitutions.
- monads.ml (207 LOC) and monads.mli (109 LOC): implements the monad transformers adding state and backtracking effects to a monad, as well as basic monad infrastructure.
- unif_core.ml (632 LOC) and unif_core.mli (88 LOC): defines unification problems and safe transformations upon them in monadic style. Only the safe operations are exposed to the rest of the program.

  It implements the functions specific to type checking for extended terms, as they are necessary to make the refinement operation safe. It also implements the transformations which we eagerly apply so as to make them transparent to the rest of the program.
- unifier.ml (214 LOC) and unifier.mli (15 LOC): implements the high-level unification heuristics.

Conclusion

Please refer to page 2 for a summary of results and future prospects.

Appendices

A Typing rules

Given a signature Σ and a unification problem Θ, typing rules define the following judgements:

- $\Gamma \vdash s = *$ read "term s is a valid sort under typing context Γ"
- $\Gamma \vdash T : *$ read "term T is a valid type under typing context Γ"
- $\Gamma \vdash t : T$ read "term t has type T under typing context Γ"
- $\Gamma \vdash \sigma : \Delta$ read "substitution σ is between typing contexts Γ and Δ"

Ground terms

Valid sort:

$$\frac{}{\Gamma \vdash Type = *}$$

$$\frac{}{\Gamma \vdash Kind = *}$$

Valid type:

$$\frac{\Gamma \vdash s = *}{\Gamma \vdash s : *}$$

$$\frac{\Gamma \vdash T : s \qquad \Gamma \vdash s = *}{\Gamma \vdash T : *}$$

Typed:

$$\frac{x : A \in \Gamma}{\Gamma \vdash x : A}$$

$$\frac{c : A \in \Sigma}{\Gamma \vdash c : A}$$

$$\frac{}{\Gamma \vdash Type : Kind}$$

$$\frac{\Gamma \vdash A : Type \qquad \Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x : A.t : \Pi x : A.B}$$

$$\frac{\Gamma \vdash A : Type \qquad \Gamma, x : A \vdash B : s \qquad \Gamma \vdash s = *}{\Gamma \vdash \Pi x : A.B : s}$$

Substitutions

$$\frac{}{\Gamma \vdash \emptyset : []}$$

$$\frac{\Gamma \vdash \sigma : \Delta \qquad \Gamma \vdash u : \sigma(A)}{\Gamma \vdash (\sigma, x \leftarrow u) : \Delta, x : A}$$

Guards

$$\frac{(\Delta \vdash p : A \to B) \in \Theta \qquad \Gamma \vdash \sigma : \Delta \qquad \Gamma \vdash t : \sigma(A)}{\Gamma \vdash p[\sigma]\ t : \sigma(B)}$$

Metavariables

$$\frac{(\Delta \vdash ?_i = *) \in \Theta \qquad \Gamma \vdash \sigma : \Delta}{\Gamma \vdash ?_i[\sigma] = *}$$

$$\frac{(\Delta \vdash ?_i : *) \in \Theta \qquad \Gamma \vdash \sigma : \Delta}{\Gamma \vdash ?_i[\sigma] : *}$$

$$\frac{(\Delta \vdash ?_i : A) \in \Theta \qquad \Gamma \vdash \sigma : \Delta}{\Gamma \vdash ?_i[\sigma] : \sigma(A)}$$

B Elaboration rules

$$\frac{\Gamma \vdash t \Uparrow u : A \qquad \Gamma \vdash u : A \overset{\mathcal{C}}{\rightsquigarrow} v : B}{\Gamma \vdash t : B \Downarrow v}$$

$$\frac{T \rhd_{whnf} \Pi x : A.B \qquad \Gamma, x : A \vdash t : B \Downarrow u}{\Gamma \vdash \lambda x :?.t : T \Downarrow \lambda x : A.u}$$

$$\frac{x : A \in \Gamma}{\Gamma \vdash x \Uparrow x : A}$$

$$\frac{c : A \in \Sigma}{\Gamma \vdash c \Uparrow c : A}$$

$$\frac{}{\Gamma \vdash Type \Uparrow Type : Kind}$$

$$\frac{\Gamma \vdash a : Type \Downarrow A \qquad \Gamma, x : A \vdash b \Uparrow B_0 : s_0 \qquad \Gamma, x : A \vdash B_0 : s_0 \overset{\mathcal{C}}{\rightsquigarrow}{}^{*} B : s}{\Gamma \vdash \Pi x : a.b \Uparrow \Pi x : A.B : s}$$

$$\frac{\Gamma \vdash a : Type \Downarrow A \qquad \Gamma, x : A \vdash t \Uparrow u : B \qquad \Gamma, x : A \vdash B \neq Kind}{\Gamma \vdash \lambda x : a.t \Uparrow \lambda x : A.u : \Pi x : A.B}$$

$$\frac{\text{New } \Gamma \vdash ?_k : * \qquad \text{New } \Gamma \vdash ?_j : ?_k[id]}{\Gamma \vdash ? \Uparrow ?_j[id] : ?_k[id]}$$

$$\frac{\Gamma \vdash f \Uparrow g : T \qquad T \rhd_{whnf} \Pi x : A.B \qquad \Gamma \vdash u : A \Downarrow v}{\Gamma \vdash f\ u \Uparrow g\ v : B[x \leftarrow v]}$$

$$\frac{\begin{array}{ccc} \Gamma \vdash f \Uparrow g_0 : T & T \rhd_{whnf} ?_i[\sigma]\ t_1 ... t_n & \Gamma \vdash u \Uparrow v_0 : A_0 \\ \Gamma \vdash A_0 : s & \Gamma \vdash A_0 : s \overset{\mathcal{C}}{\rightsquigarrow} A : Type & \Gamma \vdash v_0 : A_0 \overset{\mathcal{C}}{\rightsquigarrow} v : A \\ \text{New } \Gamma, x : A \vdash ?_s = * & \text{New } \Gamma, x : A \vdash ?_k : ?_s[id] & \Gamma \vdash g_0 : T \overset{\mathcal{C}}{\rightsquigarrow} g : \Pi x : A.\, ?_k[id] \end{array}}{\Gamma \vdash f\ u \Uparrow g\ v : ?_k[x \leftarrow v]}$$

C Constraint decomposition

Original constraint	Resulting constraint(s)	Invertible
$\Gamma \vdash f\ u \equiv g\ v$	$\Gamma \vdash f \equiv g$ and $\Gamma \vdash u \equiv v$	If f and g have rigid heads.
$\Gamma \vdash \lambda x : A.u \equiv \lambda y : B.v$	$\Gamma \vdash A \equiv B$ and $\Gamma, x : A \vdash u \equiv v[y \leftarrow x]$	Yes.
$\Gamma \vdash \Pi x : A.u \equiv \lambda y : B.v$	$\Gamma \vdash A \equiv B$ and $\Gamma, x : A \vdash u \equiv v[y \leftarrow x]$	Yes.

D Unification heuristics

A statement $\overrightarrow{S_n}$ is read as the list of statements S_1 ... S_n. For instance $t\ \overrightarrow{t_n}$ is the term t applied to the terms t_1 ... t_n in that order.

The following rule is read as "we may replace a constraint $\Gamma \vdash A \equiv B$ by a list of constraints $\overrightarrow{\Delta_n \vdash C_n \equiv D_n}$ when the property $\mathcal{P}$ is verified". There may be any number of resulting constraints and of premisses.

$$\frac{\mathcal{P} \qquad \overrightarrow{\Delta_n \vdash C_n \equiv D_n}}{\Gamma \vdash A \equiv B}$$

During unification, the various rules are tried in a certain order, backtracking when no rule can be applied. Backtracking is restricted in an ad-hoc manner.

The rules used at this time are the following:

$$\frac{A \equiv B}{\Gamma \vdash A \equiv B}\ \text{pair-conv}$$

$$\frac{\overrightarrow{\Gamma \vdash t_n \equiv u_n}}{\Gamma \vdash ?_i[\sigma]\ \overrightarrow{t_n} \equiv ?_i[\sigma]\ \overrightarrow{u_n}}\ \text{meta-same-same}$$

$$\frac{(\Gamma \vdash ?_i : T) \in \Theta \qquad \Delta = \sigma_1 \cap \sigma_2 \qquad FV(T) \subseteq \Delta \qquad \text{Narrow meta } ?_i \text{ to } \Delta \vdash ?_j : T \qquad \overrightarrow{\Gamma \vdash t_n \equiv u_n}}{\Gamma \vdash ?_i[\sigma_1]\ \overrightarrow{t_n} \equiv ?_i[\sigma_2]\ \overrightarrow{u_n}}\ \text{meta-same}$$

Here $\sigma_1 \cap \sigma_2$ is the greatest subset of Γ their common domain where the images of σ_1 and σ_2 are syntactically equal.

$$\frac{(\Delta \vdash ?_i : T) \in \Theta \qquad t' = t^{-\xi_1, \xi_2; ?_i} \qquad \Gamma \vdash t' : T' \qquad \Gamma \vdash t' : T' \overset{\mathcal{C}}{\rightsquigarrow} u : T \qquad \text{Refine } ?_i \text{ with } u}{\Gamma \vdash ?_i[\xi_1]\ \xi_2 \equiv t}\ \text{meta-inst}$$

ξ_2 is a list of variables, and ξ_1 is a substitution where the non-variable images are ignored. The operation $t^{-\xi_1, \xi_2; ?_i}$ is a complex inversion which performs the occurs check for $?_i$ and produces a term unifying the goal constraint if it succeeds. It is more fully described in [6].

$$\frac{0 < n \qquad \Gamma \vdash ?_i[\sigma] \equiv u\ \overrightarrow{u'_m} \qquad \overrightarrow{\Gamma \vdash t_n \equiv u''_n}}{\Gamma \vdash ?_i[\sigma]\ \overrightarrow{t_n} \equiv u\ \overrightarrow{u'_m}\ \overrightarrow{u''_n}}\ \text{meta-fo}$$

$$\frac{(\Delta \vdash ?_i : T) \in \Theta \qquad \sigma' \text{ unique variables from } \sigma \qquad \Delta' = \Delta \cap \sigma' \qquad \Delta' \vdash \qquad FV(T) \subseteq \Delta' \qquad \text{Narrow } ?_i \text{ to } \Delta' \vdash ?_j : T \qquad \Gamma \vdash ?_j[\sigma']\ \overrightarrow{t_n} \equiv u}{\Gamma \vdash ?_i[\sigma]\ \overrightarrow{t_n} \equiv u}\ \text{meta-deldeps}$$

In this rule we remove the dependencies on all variables which make inversion harder. Then $meta - inst$ is more likely to succeed.

The constraint decompositions and step by step reduction are also rules.

E References

[1] Andrea Asperti, Wilmer Ricciotti, Claudio Sacerdoti Coen, and Enrico Tassi. A bi-directional refinement algorithm for the calculus of (co)inductive constructions. *Logical Methods in Computer Science*, 8(1), 2012.

[2] Gilles Dowek, Thérèse Hardin, and Claude Kirchner. Higher order unification via explicit substitutions. *Information and Computation*, 157(1–2):183 – 235, 2000.

[3] Oleg Kiselyov, Chung-chieh Shan, Daniel P. Friedman, and Amr Sabry. Backtracking, interleaving, and terminating monad transformers: (functional pearl). *SIGPLAN Not.*, 40(9):192–203, September 2005.

[4] Stéphane Lengrand. *Normalisation & Equivalence in Proof Theory & Type Theory.* Theses, Université Paris-Diderot - Paris VII ; University of St Andrews, December 2006. Commencée en Septembre 2003.

[5] Ronan Saillard. Towards explicit rewrite rules in the λΠ-calculus modulo. In *IWIL - 10th International Workshop on the Implementation of Logics*, Stellenbosch, South Africa, December 2013.

[6] Beta Ziliani and Mathieu Sozeau. A predictable unification algorithm for coq featuring universe polymorphism and overloading. *ICFP*, 2015.