Elaboration in Dependent Type Theory

Leonardo de Moura[1], Jeremy Avigad[⋆2], Soonho Kong[3], and Cody Roux[4]

[1] Microsoft Research, Redmond
[2] Departments of Philosophy and Mathematical Sciences, Carnegie Mellon University
[3] Department of Computer Science, Carnegie Mellon University
[4] Draper Laboratories

Abstract. We describe the elaboration algorithm that is used in *Lean*, a new interactive theorem prover based on dependent type theory. To be practical, interactive theorem provers must provide mechanisms to resolve ambiguities and infer implicit information, thereby supporting convenient input of expressions and proofs. Lean's elaborator supports higher-order unification, ad-hoc overloading, insertion of coercions, type class inference, the use of tactics, and the computational reduction of terms. The interactions between these components are subtle and complex, and Lean's elaborator has been carefully designed to balance efficiency and usability.

1 Introduction

Just as programming languages run the spectrum from untyped languages like Lisp to strongly-typed functional programming languages like Haskell and ML, foundational systems for mathematics exhibit a range of diversity, from the untyped language of set theory to simple type theory and various versions of dependent type theory. Having a strongly typed language allows the user to convey the intent of an expression more compactly and efficiently, since a good deal of information can be inferred from type constraints. Moreover, a type discipline catches routine errors quickly and flags them in informative ways. But this is a slippery slope: as we increasingly rely on types to serve our needs, the computational support that is needed to make sense of expressions in efficient and predictable ways becomes increasingly subtle and complex.

Our goal here is to describe the elaboration algorithm of an expressive dependent type theory. This allows using a single language to define datatypes, objects, and functions, and also to express assertions and write proofs, in accordance with the propositions-as-types paradigm. Thus, filling in the details of a function definition and ensuring it is type correct is no different from filling in the details of a proof and checking that it establishes the desired conclusion. We have developed these ideas in a fully featured interactive theorem prover, *Lean*[5] [9]. Our main contribution is a novel elaboration algorithm that employs *non-chronological backtracking*, as well as heuristics for handling defined constants that we have shown to be very effective in practice.

[⋆] Work partially supported by AFOSR FA9550-12-1-0370 and FA9550-15-1-0053.

[5] http://leanprover.github.io

2 The elaboration task

What makes dependent type theory "dependent" is that types can depend on elements of other types. Within the language, types themselves are terms, and a function can return a type just as another function may return a natural number. Lean's standard library is based on a version of the *Calculus of Inductive Constructions with Universes* [8, 21, 16], as are formal developments in Coq [5] and Matita [3]. There is an infinite sequence of type universes $\mathtt{Type}_0$, $\mathtt{Type}_1$, $\mathtt{Type}_2$, ..., and any term `t : ` $\mathtt{Type}_i$ is intended to denote a type in the `i`th universe. Each universe is closed under the formation of Π-types `Πx : A, B`, where `A` and `B` are type-valued expressions, and `B` can depend on `x`. The idea is that `Πx : A, B` denotes the type of functions `f` that map any element `a : A` to an element of `B[a/x]`. When `x` does not appear in `B`, `Πx : A, B` is written `A → B` and denotes the usual non-dependent function space. One can also form *inductive families* [10], a mechanism that can be used to define basic types like `nat` and `bool`, and common type-forming operations, like Cartesian products, lists, Σ-types, and so on. Each inductive family declaration generates a recursor (also known as the *eliminator*).

Lean's kernel can be instantiated in different ways. In the standard mode, $\mathtt{Type}_0$, denoted `Prop`, is an impredicative universe of proof-irrelevant types. Lean also provides a homotopy type theory mode without `Prop`, resulting in a version of Martin-Löf type theory [18, 27] similar to the one used in Agda [6].

The task of the elaborator, put simply, is to convert a partially specified expression into a fully specified, type-correct term. For example, in the expression `λx : ℕ, x + x`, a user can omit the type annotation on `x` and leave it to the elaborator to infer that information. In that case, the expression may be ambiguous if, for example, the notation + for both natural numbers and integers is in scope, in which case the elaborator needs to find (or choose) a single type-correct interpretation.

Higher-order unification. Some elaboration problems are easy to solve using the Hindley/Milner approach. However, the elaborator often needs to infer an element of a Π-type, which constitutes a *higher-order unification* problem. For example, if `e : a = b` is a proof of the equality of two terms of some type `A`, and `H : P` is a proof of some expression involving `a`, the term `subst e H` denotes a proof of the result of replacing some or all of the occurrences of `a` in `P` with `b`. Here not just the type `A` is inferred, but also an expression `T : A → Prop` denoting the context for the substitution, that is, the expression with the property that `T a` is convertible to `P`. Such an expression is inherently ambiguous; for example, if `H` has type `R (f a a) a`, then with `subst e H` the user may have in mind `R (f b b) b` or `R (f a b) a` or `R (f a a) a` among other interpretations, and the elaborator has to rely on context and a backtracking search to find an interpretation that fits. Similar issues arise with proofs by induction, which require the system to infer an induction predicate, and with the formation of a dependent pair `⟨a, b⟩ : Σx : A, B`, which requires the elaborator to infer the dependent expression `B`.

Even second-order unification is known to be generally undecidable [12], but the elaborator merely needs to perform well on instances that come up in practice. In Lean, users can import the notation `H ▶ H'` for `subst H H'`, and write:

```
theorem mul_mod_mul_right (x z y : ℕ) :
  (x * z) mod (y * z) = (x mod y) * z :=
!mul.comm ▶ !mul.comm ▶ !mul.comm ▶ !mul_mod_mul_left
```

The proof applies the commutativity of multiplication three times to an appropriate instance of the theorem `mul_mod_mul_left`. (The symbol `!` indicates that all arguments should be synthesized by the elaborator.) The unifier can similarly handle nested inductions and iterated recursion.

Computational behavior. The elaborator should also respect the computational interpretation of terms. It should for instance recognize the equivalence of the terms `(λx, t)s` and `t[s/x]`, as well as `⟨s, t⟩.1` (the first projection) and `s` under the reduction rule for pairs. Elements of inductive types may also have computational behavior; on the natural numbers, `2 + 2` and `4` are both definitionally equal to `succ (succ (succ (succ 0)))`, `x + 0` is definitionally equal to `x`, and `x + 1` is definitionally equal to `succ x`. The elaborator should also support unfolding definitions where necessary: for example, if `x - y` is defined as `x + (-y)`, the elaborator should allow us to use the commutativity of addition to rewrite `x - y` to `-y + x`. Unfolding definitions and reducing projections is especially crucial when working with algebraic structures, where many basic expressions cannot even be seen to be type correct without carrying out such reductions.

It is worth noting that the naive approach of performing *all* such unfoldings leads to unacceptable performance, and it is an important aspect of building a practical elaboration procedure to design heuristics that limit unfolding to situations that require it. In addition, the user often may require different computational behavior of various definitions, e.g. when it is intended as a simple notation rather than a mechanism for abstracting logical content.

Overloading and coercions. Lean supports ad-hoc overloading of constants. For example, users can import notation from "namespaces" for the natural numbers, integers, and algebra, overloading symbols for addition and multiplication. Similarly, if they open all three namespaces, the theorems `nat.mul.assoc`, `int.mul.assoc`, `algebra.mul.assoc`, denoting associativity of multiplication in the various contexts, can all be denoted by the overloaded alias `mul.assoc`. Lean provides the notation `#int a * b` to force an overloaded constant to be interpreted in the `int` namespace, but typically such annotations are unnecessary.

Ad-hoc overloading is more flexible than type class overloading (described below), in that the overloaded constants can denote entirely different kinds of objects. For example in the Lean standard library, we use $^{-1}$ above to denote the inverse function for algebraic structures that support it, as well as for the symmetry operation for equalities, without having to support the symmetry of equality as an element of an axiomatic class.

The treatment of coercions in Lean is as one would expect. One can, for example, coerce a `bool` to a `nat` and a `nat` to an `int`, and Lean will insert coercions in list expressions `[n, i, m, j]` and `[i, n, j, m]` when `n` and `m` have type `nat` and `i` and `j` have type `int`. One can also coerce axiomatic structures, so that the user can provide a group as input anywhere a semigroup is expected. One can also coerce from a suitable family of types to `Type` or to a Π-type.

Type classes. Lean supports the use of Haskell-style *type classes* [14]. For example, we can define a class `has_mul A` of types `A` with an associated multiplication, and a class `semigroup A` of types `A` with semigroup structure, as follows:

```
structure has_mul [class] (A : Type) := (mul : A → A → A)

structure semigroup [class] (A : Type) extends has_mul A :=
(mul_assoc : ∀a b c, mul (mul a b) c = mul a (mul b c))
```

We can then declare appropriate instances of these classes, and instruct the elaborator to synthesize such instances when processing the notation `a * b` or the generic theorem `mul.assoc`.

The `structure` declaration above automatically declares `semigroup` to be an instance of `has_mul`, and also declares a coercion from the former to the latter. The `structure` command supports the construction of an algebraic hierarchy by allowing the user to extend and merge multiple structures:

```
structure group [class] (A : Type) extends monoid A, has_inv A :=
(mul_left_inv : ∀a, mul (inv a) a = one)
```

Users can also rename structure components on the fly.

We mark implicit arguments with square brackets instead of curly brackets, to inform the elaborator that these arguments should be inferred by the type class mechanism. In the following example, type class inference finds the appropriate inverse and instance of the theorem `inv_inv`:

```
theorem eq_inv_of_eq_inv {A : Type} [s : group A] {a b : A}
                         (H : a = b⁻¹) : b = a⁻¹ :=
by rewrite [H, inv_inv]
```

Here, the `rewrite` tactic (see below) replaces `a` by $\texttt{b}^{-1}$ in the goal, and then rewrites $(\texttt{b}^{-1})^{-1}$ to `b`.

In Lean, type classes can be used to infer not only notation and generic facts, but fairly complex data. For example, in the standard library, we define the class of propositions that are decidable:

```
inductive decidable [class] (p : Prop) : Type :=
inl :  p → decidable p,
inr : ¬p → decidable p
```

Logically speaking, having an element `t : decidable p` is more informative than having an element `t : p ∨ ¬p`; it enables us to define values of an arbitrary type depending on the truth value of `p`. The distinction is only useful in constructive mathematics, because classically every proposition is decidable. But this typeclass allows for a smooth transition between constructive and classical logic, allowing classical reasoning in suitable constructive settings as well.

For example, we can prove, constructively, that equality and comparisons on the natural numbers are decidable, and that decidability is preserved under boolean operations and bounded quantification. As a result, we can reason by cases on such statements, and use them, constructively, in an if-then-else expression. We can even use type class inference to prove theorems automatically:

```
example : ∀ x : nat, x < 10 → x ≠ 10 ∧ x < 12 := dec_trivial
```

Here `dec_trivial` is notation for an expression that infers the implicit decision procedure and verifies that it reduces to `true`.

Tactics. Finally, definitions and proofs can invoke *tactics*, that is, user-defined or built-in procedures that construct various subterms. The constraint solver described in this paper invokes user provided tactics to construct terms that cannot be synthesized by solving unification constraints and type class resolution. Lean's tactic language is similar to those found in other LCF-style theorem provers.

Any given definition or theorem in Lean can draw on many of the features just described. Consider the following, which defines the composition of two natural transformations between functors (and establishes that it is, indeed, a natural transformation):

```
variables {C D : Precategory} {F G H : C ⇒ D}
definition nt_compose (η : G ⟹ H) (θ : F ⟹ G) : F ⟹ H :=
natural_transformation.mk
  (take a, η a ∘ θ a)
  (take a b f, calc
    H f ∘ (η a ∘ θ a) = (H f ∘ η a) ∘ θ a : assoc
                  ... = (η b ∘ G f) ∘ θ a : naturality
                  ... = η b ∘ (G f ∘ θ a) : assoc
                  ... = η b ∘ (θ b ∘ F f) : naturality
                  ... = (η b ∘ θ b) ∘ F f : assoc)
```

Here the functors `F`, `G`, and `H` are coerced to their action on morphisms, and the natural transformations η and θ are coerced to their first component. The composition symbol $\circ$ for functions is overloaded to denote composition of morphisms as well, and type class inference infers the category in which the composition takes place. The appropriate substitution contexts in the calculation are inferred, as are the arguments to the theorems that are invoked.

The interactions between the components of the elaboration task are subtle, and the challenge is to deal with them all at the same time. A definition or proof may give rise to hundreds of constraints requiring a mixture of higher-order unification, disambiguation of overloaded symbols, insertion of coercions, type class inference, and computational reduction. The net effect is then a difficult constraint-solving problem with a combinatorial explosion of options. Lean's elaborator manages to solve such problems, and it is fast: Lean's entire standard library compiles in seconds. In the next section, we explain how the elaborator processes the constraints and navigates the search space in an effort to balance completeness and efficiency.

3 The elaboration procedure

The process for getting to a fully elaborated term from a preterm has two main steps: *preprocessing* and *constraint resolution*. Before we explain each phase, we describe the term representation and main data structures used in our elaboration procedure. We assume the term language is a dependent λ-calculus in which terms are described by the following grammar:

$$t, s \ = \ \ell \mid x \mid f \mid ?m \mid \text{Type } u \mid t\ s \mid \lambda x : s, t \mid \Pi x : s, t$$

where

- ℓ a free variable (also called a local constant)
- x is a bound variable
- f is a constant (parametrized by a list of universe terms)
- $?m$ is a metavariable
- u is a universe term

Free variables have a unique identifier and a type, and bound variables are just a number (a de Bruijn index), adopting the *locally nameless* variable binding style. Storing the type with each free variable removes the need to carry around contexts in the type checker and normalizer. As described in [19], this representation style simplifies the implementation considerably, as it minimizes the number of places where explicit calculations with de Bruijn indices must be performed. We use the notation $t[x := s]$ to represent the substitution of x for s in t, where x is a bound variable, free variable, or metavariable. When x is a bound variable, the operation also lowers all bound variables with index greater than x. We use $\boldsymbol{t}$ to denote sequence of terms $t_1 \ldots t_n$, and $\boldsymbol{x} : \boldsymbol{A}$ for the telescope $(x_1 : A_1) \ldots (x_n : A_n)$, where A_i may depend on x_j for $j < i$.

An *environment* stores a sequence of declarations. The Lean kernel supports three different kinds of declarations: *axioms*, *definitions* and *inductive families*. Each has a unique identifier, and can be parametrized by a sequence of universe parameters. Every axiom has a type, and every definition has a type and a value. A constant is just a reference to a declaration.

Users usually provide *partial constructions*, i.e., constructions containing *holes* that must be filled by the system. Internally, each hole is represented by a metavariable. Each metavariable has a unique identifier and a type. The main operation on metavariables is *instantiation*. In our implementation, only closed terms can be assigned to metavariables. This design decision guarantees that operations such as β-reduction and metavariable instantiation commute. Since only closed terms can be assigned to metavariables, on creation a metavariable is applied to the variables in the context where it appears. For example, we encode a hole in the context $(x : A)\ (y : B)$ as $?m\ x\ y$, where $?m$ is a fresh metavariable. The type of $?m$ is $\Pi(x : A)\ (y : B), C$, where C is the expected type for the hole at that position. If the expected type is also unknown at pre-processing time, we create another fresh metavariable $?m_t : \Pi(x : A)\ (y : B), \text{Type}\ ?u$, where $?u$ is a fresh universe metavariable, which gives us $?m : \Pi(x : A)\ (y : B), ?m_t\ x\ y$. We say a term is *fully elaborated* if it does not contain metavariables.

We say a term is β-reducible if it is of the form $(\lambda x : A, s)t$, and ι-reducible if it is of the form $\texttt{C.rec}\ \boldsymbol{s}\ (\texttt{C.mk}_i\ \boldsymbol{r})\ \boldsymbol{t}$, where $\texttt{C.rec}$ is the recursor/eliminator for an inductive datatype $\texttt{C}$; $\boldsymbol{s}$ represents the parameters, minor premises and indices and $(\texttt{C.mk}_i\ \boldsymbol{r})$ is the main premise (where $\texttt{C.mk}_i$ is the i-th constructor of $\texttt{C}$). The function $\texttt{reduce}_{\beta\iota}\ s$ applies head β and ι reduction to s. We say a term t is *stuck* if computation cannot occur without instantiating a metavariable $?m$; where $(?m\ \boldsymbol{s})$ is a sub-term of t, we say $(?m\ \boldsymbol{s})$ is the *reason* for t being stuck. More formally, a term is stuck when the head symbol is a metavariable (i.e., it is of the form $?m\ \boldsymbol{s}$), or it is a recursor application where the main premise is stuck. We say the first case is a *stuck-application*, and the second a *stuck-recursor*.

During the pre-processing step, *unification* and *choice* constraints are generated. Unification constraints are used to enforce typing constraints, and *choice* constraints are for overloading, coercion resolution, and triggering the type class mechanism.

A unification constraint $t \approx s$ is annotated with a *justification*. Justifications are used to assist the generation of error messages when a term fails to be elaborated, and to implement *non-chronological backtracking* [22]. Non-chronological backtracking allows exploring the (possibly infinite) tree of potential solutions more efficiently, by eliminating branches which we know cannot possibly contain an actual solution.

There are three kinds of justifications: *asserted*, *assumption* and *join*. An asserted justification is used to annotate constraints generated during the pre-processing phase. Whenever the solver has to perform a choice (also known as a *case split*), it annotates each choice with a fresh assumption. A join justification $j_1 \bowtie j_2$ represents the "union" of the justifications j_1 and j_2. We use $\langle t \approx s,\ j\rangle$ to denote the unification constraint justified by j. A *substitution* is a finite collection of *assignments* from metavariables to pairs $\langle t, j\rangle$, written $?m \mapsto \langle t, j\rangle$, where t is a closed term and j is a justification for the assignment. Assignments are generated when solving unification constraints. For example, the constraint $\langle ?m \approx t,\ j\rangle$ is solved by adding the assignment $?m \mapsto \langle t, j\rangle$. Whenever we apply a substitution we use a join justification to track its effect. For example, the result of applying the assignment $?m \mapsto \langle t, j_m\rangle$ over the constraint $\langle r \approx s,\ j\rangle$ is the new constraint $\langle r[?m := t] \approx s[?m := t],\ j \bowtie j_m\rangle$. We also use $\langle s \approx t,\ j_1\rangle \bowtie j_2$ to denote the constraint $\langle s \approx t,\ j_1 \bowtie j_2\rangle$. Moreover, if a is a list of constraints $[c_1, \ldots, c_n]$, $a \bowtie j$ is $[c_1 \bowtie j, \ldots, c_n \bowtie j]$.

A *choice constraint* is of the form $\langle ?m\ \boldsymbol{\ell} : t\ \texttt{in}\ f, j\rangle$, where $?m$ is a metavariable, $\boldsymbol{\ell}$ are free variables representing the context where $?m$ was created, t is the type of $?m\ \boldsymbol{\ell}$, and f is a function that, given the term $?m\ \boldsymbol{\ell}$, its type t and a substitution, produces a (possibly unbounded) stream of constraints representing possible ways of synthesizing $?m$, and a justification j. Note that each alternative is itself a list of constraints, and is not necessarily just a *single* unification constraint.

A choice constraint $?m\ \boldsymbol{\ell} : t\ \texttt{in}\ f$ may be marked as *ondemand*. When the flag *ondemand* is set, the constraint solver will *try* to invoke function f only after all metavariables in t have been instantiated. We say a *ondemand* choice constraint is **ready** when t does not contain metavariables, and **postponed** otherwise. We describe further down how this feature is used to implement the type class mechanism and coercions. If a choice constraint is not marked as *ondemand*, we say it is a **regular** choice constraint. We use **regular** choice constraints to specify overloaded symbols. The result of applying the assignment $?m \mapsto \langle s, j_m \rangle$ over the choice constraint $\langle ?n\ \boldsymbol{\ell} : t\ \texttt{in}\ f, j\rangle$ is the new constraint $\langle ?n\ \boldsymbol{\ell} : t[?m := s]\ \texttt{in}\ f, j \bowtie j_m\rangle$. We also use the notation $c \bowtie j$ when c is a choice constraint.

Support functions. Both the preprocessing step and the constraint-solving procedure rely on a constraint-simplification procedure, which we describe below. First, however, we describe some auxiliary functions that are used throughout.

The function `typeof` r returns the inferred type of a term r, where r may contain metavariables. Specifically, it returns a pair $\langle t,\ S\rangle$ where t is the type of r and S is a set of constraints on the metavariables; if r does not contain metavariables, then S is empty. The function `unfold` $(f\ t_1 \ldots t_n)$ applies a δ-reduction, i.e., it unfolds the definition of constant f. In practice, it is not feasible to apply δ-reduction to all constants in a constraint solving problem, but the system would be inconvenient to use if δ-reduction steps were forbidden. To cope with this performance issue, we allow the user to annotate definitions with the following hints: *irreducible*, *semireducible* or *reducible*. A irreducible definition is never unfolded by the constraint solver, while a semireducible or reducible definition may be unfolded or not depending on the constraint being solved. When no annotation is provided, the system assumes the definition is *semireducible*. We remark that when the kernel type checks fully elaborated definitions, these annotations are ignored; they are only relevant during the elaboration process.

The procedure `error` j throws an exception tagged with a justification j. Finally, the function `ensurefun` $s\ j$ ensures that s has a function type. Specifically, it infers the type t of s (using `typeof`) and then reduces t to t' in weak head normal form (*whnf*). If t' is a Π-term, then it returns t' and any new unification constraints. If t' is not a Π-term and is not stuck, then it generates an error with justification j. Otherwise, if $?m\ \boldsymbol{s}$ is the reason that t' is stuck, where $?m : (\Pi \boldsymbol{x} : \boldsymbol{A}, B)$, we create two fresh metavariables: $?m_1 : (\Pi \boldsymbol{x} : \boldsymbol{A},\ \text{Type}\ ?u_1)$ and $?m_2 : (\Pi (\boldsymbol{x} : \boldsymbol{A})\,(y : ?m_1\ \boldsymbol{x}),\ \text{Type}\ ?u_2)$, and the new constraint

$$\langle t \approx (\Pi x : ?m_1\ \boldsymbol{s},\ ?m_2\ \boldsymbol{s}\ x),\ j\rangle.$$

This ensures that s has a function type, and defers the problem of figuring out what that type is.

The contraint simplification procedure , `simp`. To save space, we do not consider symmetric cases such as $\langle(\lambda x : B, t) \approx s,\ j\rangle$. The procedure `mklocal` A creates a fresh free variable with type A. To simplify the presentation, we assume there is a global unique name generator. The function `depth` f returns the *definition depth* of the constant f. It is 0 if f is not a definition, and $1 + \mathtt{max}\{\mathtt{depth}\ g \mid g$ appears in the definition of $f\}$ otherwise.

```
simp $\langle t \approx t,\ j\rangle$                              $= \{\}$
simp $\langle s \approx t,\ j\rangle$ **when** $s$ is $\beta/\iota$-reducible     $=$ simp $\langle \mathtt{reduce}_{\beta\iota}\ s \approx t,\ j\rangle$
simp $\langle \ell\ s_1 \dots s_n \approx \ell\ t_1 \dots t_n,\ j\rangle$          $= \bigcup_{i=1}^{n}$ simp $\langle s_i \approx t_i,\ j\rangle$
simp $\langle f\ s_1 \dots s_n \approx f\ t_1 \dots t_n,\ j\rangle$          $=$
  **if** $s_1 \dots s_n$ and $t_1 \dots t_n$ do not contain metavariables **then**
    simp $(\langle \mathtt{unfold}\ (f\ s_1 \dots s_n) \approx \mathtt{unfold}\ (f\ t_1 \dots t_n),\ j\rangle)$
  **else if** $f$ is not *reducible* **then** $\bigcup_{i=1}^{n}$ simp $\langle s_i \approx t_i,\ j\rangle$
  **else** $\{\langle f\ s_1 \dots s_n \approx f\ t_1 \dots t_n,\ j\rangle\}$
simp $\langle f\ \boldsymbol{s} \approx g\ \boldsymbol{t},\ j\rangle$                         $=$
  **if** depth $f$ ¿ depth $g$ and $f$ is not *irreducible* **then**
    simp $(\langle \mathtt{unfold}\ (f\ \boldsymbol{s}) \approx g\ \boldsymbol{t},\ j\rangle)$
  **else if** depth $f$ ¡ depth $g$ and $g$ is not *irreducible* **then**
    simp $(\langle f\ \boldsymbol{s} \approx \mathtt{unfold}\ (g\ \boldsymbol{t}),\ j\rangle)$
  **else if** depth $f =$ depth $g$ and $f$ and $g$ are not *irreducible* **then**
    simp $(\langle \mathtt{unfold}\ (f\ \boldsymbol{s}) \approx \mathtt{unfold}\ (g\ \boldsymbol{t}),\ j\rangle)$
  **else** error $j$
simp $\langle(\lambda x : A, s) \approx (\lambda y : B, t),\ j\rangle$            $=$
  **let** $\ell =$ mklocal $A$ **in** simp $\langle A \approx B,\ j\rangle \cup$ simp $\langle s[x := \ell] \approx t[y := \ell],\ j\rangle$
simp $\langle(\Pi x : A, s) \approx (\Pi y : B, t),\ j\rangle$            $=$
  **let** $\ell =$ mklocal $A$ **in** simp $\langle A \approx B,\ j\rangle \cup$ simp $\langle s[x := \ell] \approx t[y := \ell],\ j\rangle$
simp $\langle s \approx (\lambda x : B, t),\ j\rangle$                    $=$
  **let** $\langle(\Pi x : A, C), S\rangle =$ ensurefun $s\ j$ **in** simp $\langle(\lambda x : A, s\ x) \approx (\lambda x : B, t),\ j\rangle \cup S$
simp $\langle s \approx t,\ j\rangle$                              $=$
  **if** $s$ or $t$ is *stuck* **then** $\{\langle s \approx t,\ j\rangle\}$ **else** error $j$
```

Given a unification constraint, the `simp` procedure produces a set of (potentially) simpler unification constraints or throws an error. Moreover, if the input constraint does not contain metavariables, then the result is the empty set $\{\}$ or an error. In the actual implementation, we also use a heuristic optimization for the case `simp` $\langle f\ s_1 \dots s_n \approx f\ t_1 \dots t_n,\ j\rangle$, where $s_1 \dots s_n$ and $t_1 \dots t_n$ do not contain metavariables, and f is not a projection. In this case, we first try `simp` $\langle s_1 \approx t_1,\ j\rangle \dots$ `simp` $\langle s_n \approx t_n,\ j\rangle$, and if no error is thrown, we return $\{\}$. Each unification constraint returned by `simp` is in one of the following categories:

- **delta**: $\langle f\ \boldsymbol{s} \approx f\ \boldsymbol{t},\ j\rangle$. Note that, based on the definition of `simp`, f must be a reducible definition.
- **pattern**: $\langle ?m\ \ell_1 \dots \ell_n \approx t,\ j\rangle$, where $\ell_1, \dots, \ell_n$ are pairwise distinct free variables, t only contains free variables in $\{\ell_1, \dots, \ell_n\}$, and $?m$ does not occur in t.

- **quasi-pattern**: $\langle ?m\ \ell_1 \dots \ell_n \approx t,\ j\rangle$, where all $\ell_1, \dots, \ell_n$ are free variables, but are not pairwise distinct.
- **flex-rigid**: $\langle ?m\ s_1 \dots s_n \approx t,\ j\rangle$, where at least one of $s_1, \dots, s_n$ is not a free variable.
- **flex-flex**: $\langle ?m_1\ \boldsymbol{s} \approx ?m_2\ \boldsymbol{t},\ j\rangle$.
- **recursor**: $\langle t \approx s,\ j\rangle$, where t or s is a stuck-recursor.

We remark that, in the literature, **pattern**, **quasi-pattern** and **flex-rigid** are simply called flex-rigid constraints, and the category **pattern** corresponds to Miller patterns [20]. Note that flex-flex constraints are badly underconstrained, and we typically expect that other constraints will do more to limit the interpretation of the metavariables.

Given $(\ell_1 : A_1) \dots (\ell_n : A_n)$, the operation $\mathtt{abstract}_\lambda\ [\ell_1 \dots \ell_n]\ t$ returns

$$\lambda (x_1 : A_1) \dots (x_n : A_n[\ell_1 := x_1, \dots, \ell_{n-1} := x_{n-1}]),\ t[\ell_1 := x_1, \dots, \ell_n := x_n]$$

We also have $\mathtt{abstract}_\Pi$, the equivalent operation for Π-abstraction.

Preprocessing. The preprocessor is a straightforward recursive procedure that given a preterm and a context, returns a term t (potentially containing metavariables), and a set of unification and choice constraints. The basic idea is: if the constraints are solved, their solution should contain an assignment for all metavariables in t. The preprocessor must carry a context, a list of free variables, to be able to create fresh metavariables. This is the only procedure in our implementation that "carries contexts around". The preprocessor only creates *asserted* justification objects.

Applications $(r\ s)$ are the main source of unification constraints. After a preterm p in a context $\boldsymbol{\ell}$ is converted into the application $(r\ s)$, the preprocessor uses `ensurefun` to make sure that the type of r is of the form $\Pi x : A, B$, and `simp` to enforce that the type C of s is convertible to A. If C is not convertible to A, the preprocessor checks the database of available coercions, if there is a coercion c from C to A, it replaces the application $(r\ s)$ with $(r\ (c\ s))$. If A is stuck, but there are coercions $\{c_1, \dots, c_n\}$ from C, the preprocessor creates a fresh metavariable $?m\ : \mathtt{abstract}_\Pi\ \boldsymbol{\ell}\ A$, replaces the application with $(r\ (?m\ \boldsymbol{\ell}))$, and creates a *ondemand* choice constraint $\langle ?m\ \boldsymbol{\ell} : A\ \mathtt{in}\ f, j\rangle$, where the choice function f produces one of the following alternatives s, $c_1\ s$, ..., $c_n\ s$. If possible, the solver will only invoke f after all metavariables in A have been instantiated. In this ideal situation, f returns at most one solution, and no case-analysis is needed. The same process is performed when C is stuck and there are coercions *to* A. We currently do *not* try to inject coercions when both A and C are stuck at preprocessing time. Lean supports parametric coercions, and coercions to sorts and function classes, but due to space constraints we do not describe them here. Ad hoc overloading is also realized using choice constraints. The idea is the same, but we create a *regular* choice constraint where the choice function f produces the different interpretations for the overloaded symbol.

Finally, to handle implicit arguments, when we infer the type t of a term r, if t is of the form $\Pi\{x : A\}, B$, we create a fresh metavariable $?m : \mathtt{abstract}_\Pi\ \boldsymbol{\ell}\ A$, and replace r with the application $(r\ (?m\ \boldsymbol{\ell}))$. If the implicit argument is marked with square brackets to indicate it should be synthesized by the type class mechanism, we also create an *ondemand* choice constraint $\langle ?m\ \boldsymbol{\ell} : A\ \texttt{in}\ f, j\rangle$ where the choice function f invokes the type class resolution procedure. This procedure is essentially a simple λ-Prolog interpreter [20], where the Horn clauses are the user declared instances.

The constraint solving procedure. Given a set of constraints, our solver returns a failure, or a substitution S and set of **flex-flex** constraints of the form $\langle ?m_1\ \boldsymbol{s} \approx ?m_2\ \boldsymbol{t},\ j\rangle$ such that neither $?m_1$ nor $?m_2$ are assigned in S. The solver uses the following data structures: a priority queue Q of constraints, a mapping U of metavariables to constraints, a substitution S, and a case split stack C. To simplify the presentation, we assume Q, U, S and C are global variables. The priorities for the Q are computed using the following total order $\prec$ on constraint categories: **pattern** $\prec$ **ready** (choice constraints) $\prec$ **regular** (choice constraints) $\prec$ **delta** $\prec$ **quasi-pattern** $\prec$ **flex-rigid** $\prec$ **recursor** $\prec$ **postponed** (choice constraints) $\prec$ **flex-flex**. Moreover, if two constraints are in the same category, we use the *first-in-first-out* method. For each metavariable $?m$, $U[?m]$ is the finite subset of the constraints in Q s.t. for each c in $U[?m]$, c is a unification constraint stuck because of $?m$, or c is an *ondemand* choice constraint $\langle ?n\ \boldsymbol{\ell} : t\ \texttt{in}\ f, j\rangle$ and $?m$ occurs in t.

Given a set of constraints s, for each constraint c in s, the procedure `visit` s simply invokes `visiteq` c if c is a unification constraint, and `visitchoice` c otherwise. The procedure `visiteq` $\langle r \approx s,\ j\rangle$ is defined as:

```
**if** $r$ or $s$ is stuck by some $?m$ and $?m \mapsto \langle t, j_m\rangle$ in $S$ **then**
    visit (simp $\langle r[?m := t] \approx s[?m := t],\ j \bowtie j_m\rangle$)
**else if** the constraint is a **pattern** $\langle ?m\ \boldsymbol{\ell} \approx t,\ j\rangle$ **then**
    add the assignment $?m \mapsto \langle(\texttt{abstract}_\lambda\ \boldsymbol{\ell}\ t), j\rangle$ to $S$
**else** update $U$, and insert constraint into $Q$
```

The procedure `visitchoice` $\langle ?n\ \boldsymbol{\ell} : t\ \texttt{in}\ f, j\rangle$ just substitutes any assigned metavariable $?m$ occurring in t, updates U, and inserts the constraint into Q. Note that, we never insert **pattern** constraints into Q.

To implement a backtracking search, we need a mechanism for restoring the state of the solver during a backtrack operation. We use a very simple approach where Q, U and S are implemented using pure data structures (red-black trees) which provide a constant time copy operation. Whenever we need to create a case split, we simply create copies of Q, U and S. An alternative approach is to use a *trail stack* [22] which stores operations that "undo" the destructive updates performed during the search. We remark that our simpler approach for implementing backtracking is not a bottleneck in our implementation. When solving a non-`pattern` constraint c, the solver creates a case split, and stores it on the stack C. Each case split is a tuple of the form $\langle Q_c, U_c, S_c, j_a, j_c, z\rangle$, where Q_c, U_c and S_c store the state of the solver when the case split was created, j_a is a fresh assumption (justification) used to "track" the case split, j_c is the justification for c, and z is a lazy list containing the remaining alternatives, where each alternative is a list of constraints. We use `pull` z to denote the operation that destructively extracts the head of the lazy list z and returns it, or returns `none` when z is empty. The solver catches any `error` j thrown by the `simp` procedure, and uses the error resolution procedure `resolve` j defined as:

```
while $C$ is not empty
  let $\langle Q_c, U_c, S_c, j_a, j_c, z\rangle = \texttt{top}\ C$ in
  if $j$ depends on $j_a$ then
    restore state $Q := Q_c$, $U := U_c$, $S := S_c$
    if $\texttt{pull}\ z = \texttt{some}\ a$ then $\texttt{visit}\ (a \bowtie j_c \bowtie j)$ and return
  pop $C$
failed to solve constraints since $C$ is empty
```

In the procedure above, `visit` $(a \bowtie j_c \bowtie j)$ may throw another `error` j'. If this happens it recursively invokes `resolve` j'.

Processing constraints. We now describe how we process the next constraint in the queue Q. We use an auxiliary procedure `process` $z\ j$, where z is a lazy list of alternatives, and j is a justification. If z is empty, it just invokes `resolve` j. Otherwise, it pulls the head a of z, creates a fresh assumption justification j_a, pushes the new case split $\langle Q, U, S, j_a, j_c, z\rangle$ on the stack C, and invokes `visit` $(a \bowtie j_a \bowtie j)$.

For choice constraints $\langle ?m\ \boldsymbol{\ell} : t\ \texttt{in}\ f, j\rangle$ (**ready**, **regular** or **postponed**), we just invoke `process` $(f\ (?m\ \boldsymbol{\ell})\ t\ S)\ j$. For `delta` constraints $\langle f\ s_1 \ldots s_n \approx f\ t_1 \ldots t_n,\ j\rangle$, we try two alternatives. In the first one, we assume f is opaque, and try to avoid the potentially expensive δ-reduction step by using $a_1 = \bigcup_{i=1}^{n} \texttt{simp}\ \langle s_i \approx t_i,\ j\rangle$. If it fails, as our next alternative, we unfold f and try $a_2 = \texttt{simp}(\langle \texttt{unfold}\ (f\ s_1 \ldots s_n) \approx \texttt{unfold}\ (f\ t_1 \ldots t_n),\ j\rangle)$. We use the operation `tolazy` to convert the list $[a_1, a_2]$ into a lazy list, and process the delta constraint using `process` $(\texttt{tolazy}\ [a_1, a_2])\ j$. This case split is a heuristic optimization and is not necessary for completeness.

The constraint categories **quasi-pattern** and **flex-rigid** are handled in the same way. We use different categories to make sure that "easier" constraints occur first in the priority queue. We (approximately) solve them by using a variation of the flex-rigid case found in Huet's unification algorithm [15]. Given a flex-rigid constraints $\langle ?m\ s_1 \ldots\ s_p \approx t,\ j\rangle$, the main idea in Huet's algorithm is to notice that t must be a term of the form $f\ r_1 \ldots r_n$, where f is a free-variable or a constant. The next idea is to observe that any solution for $?m$ is convertible to one in eta-long normal form, which allows us to consider only solutions for $?m$ that are of the form

$$\lambda x_1 \ldots x_n, h\ (?m_1\ x_1 \ldots x_n) \ldots (?m_p\ x_1 \ldots x_n) \qquad (*)$$

where $?m_i$ are fresh metavariables, and h is a constant or one of the bound variables $x_1 \ldots x_n$. In Huet's algorithm, only opaque constants are considered, thus if h is a constant different from f of the rigid term t, the solution would lead to an unsolvable constraint. Therefore, we say Huet's procedure has two kinds of case splits: *imitation* (when h is the constant f of the rigid term), and *projection* (when h is one of the bound variables $x_1 \ldots x_n$). However, there are two problems in our setting. First, we do not eagerly unfold $f\ r_1 \ldots r_n$ when f is a constant. For example, assume that `sub` $a\ b$ (subtraction for integers) is defined as `add` a (`uminus` b). Then, $\langle ?m\ (\texttt{uminus}\ a) \approx \texttt{sub}\ b\ a,\ j\rangle$ has a solution $?m = \lambda x, \texttt{add}\ b\ x$, but we would miss it if we did not unfold `sub` before trying to imitate. Second, we have recursors in our language, and even if f is an opaque constant, it is not the only constant that can be used for h. For example, given the constraints $\langle ?m\ \texttt{zero} \approx \texttt{true},\ j\rangle, \langle ?m\ (\texttt{succ zero}) \approx \texttt{false},\ j\rangle$, a possible solution is $?m = \lambda x,\ \texttt{nat.rec}\ (\lambda n, \texttt{bool})\ \texttt{true}\ (\lambda n\ r, \texttt{false})\ x$, where `nat.rec` is the recursor for the type `nat` (of the natural numbers). We cope with the first problem using an approach similar to the one used for **delta**-constraints when f is a *reducible* constant. The idea is to have two imitation steps, one where f is not unfolded, and another one where the term $f\ r_1 \ldots r_n$ is put into weak head normal form before performing the imitation. In our implementation, it is currently infeasible to consider the extra imitation step (after *whnf*) for all constants. Even using non-chronological backtracking, the search space becomes too big. The main problem is that the system may spend a huge amount of time traversing the whole search space when the user provides an incorrect partial construction. As to the second issue, we currently simply ignore this possiblity, since the search space would become too big if we considered recursors for h. Moreover, if h is a recursor, the constraint obtained after replacing $?m$ would be a *stuck-recursor*.

As in most higher-order unification procedures, we try first the projection case splits because they generate more general solutions. We also remark that the number of case splits can usually be greatly reduced for **quasi-pattern**s, which is the case the arises most commonly in practice. In this case, if f is a constant (not marked as reducible), then we do not need to consider any projections. Any projection would fail immediately: if we take h to be ℓ_i and substitute (*) for $?m$ in the original constraint, we obtain an unsolvable constraint $\langle \ell_i\ (?m_1\ \boldsymbol{\ell}) \ldots (?m_p\ \boldsymbol{\ell}) \approx f\ r_1 \ldots r_n,\ j'\rangle$. Finally, if f is a free variable ℓ, then we only need to consider the projection where h is x_i if $\ell_i = \ell$. For **flex-rigid** constraints $\langle ?m\ s_1 \ldots s_n \approx t,\ j\rangle$, we only consider the case h is x_i when s_i is a free variable ℓ, or s_i is convertible to t. In the second case, where s_i is convertible to t, we simply assign $\lambda x_1 \ldots x_n, x_i$ to $?m$. This is a heuristic for reducing the size of the state, and minimizing the number of instances the procedure exhibits nonterminating behavior. We remark, that in the second-order case, the solver does not miss solutions by using this heuristic. Finally, our solver has a threshold on the number of steps that can be performed.

We also use an approximate solution for `recursor` constraints $\langle t \approx s,\ j\rangle$. If the head of t and s is the same recursor `C.rec`, then we try to solve the constraint by treating `C.rec` as a regular opaque constant which has no computational behavior associated with it. If t or s is of the form $?m\ \boldsymbol{r}$, then we treat it as a flex-rigid constraint. When the recursor `C.rec` is stuck because of a term $?m\ \boldsymbol{r}$, we previously tried to perform a case split for each constructor $\texttt{C.mk}_i$ of `C`, assigning $?m$ to terms of the form $\lambda \boldsymbol{x}, \texttt{C.mk}_i\ (?m_1\ \boldsymbol{x}) \dots (?m_n\ \boldsymbol{x})$. However, this provides only a minor improvement on the usability of the system: only three theorems in our library broke after we removed this feature, and all of them could be easily fixed by providing implicit arguments explicitly.

4 Related work

We attempt to put our work in the context of recent work on elaboration in dependent type theories. Abel and Pientka present an extension of Miller-style pattern unification [1] which can handle a larger class of problems (in addition to Σ-types) by a method they call *pruning*, which intuitively removes arguments to metavariables which fall outside of the Miller pattern fragment, allowing for more solutions to be found. They also give a bi-directional inference system for a dependently typed λ-calculus, which together with the unification algorithm yields an outline for a practical implementation. They show the soundness of the unification algorithm with respect to this type system. They do not, however, treat the case of defined constants, with or without recursion.

Building upon this is recent work by Ziliani and Sozeau [29] that describes a unification algorithm for the Coq theorem prover which features defined constants and recursively defined functions. They attempt to describe the practicalities of such an algorithm for a realistic dependently typed language, outlining the heuristics and efficiency compromises inherent in this task. In that respect, their motivations are very similar to ours.

In addition to Abel and Pientka's pruning, Ziliani and Sozeau add a more aggressive form of dependency erasure for metavariables, in an attempt to solve more unification problems at the cost of uniqueness of solutions. One example is the problem $\{?t \text{ true} \approx \text{nat},\ ?t \text{ false} \approx \text{nat}\}$. This problem is solved in their framework by dropping the dependency of $?t$ on its argument, and returning the constraint $?t' \approx \text{nat}$ which gives the solution $?t \mapsto \lambda x, \text{nat}$. They also add a resolution rule called *first order approximation*, in which for example the constraint $?f\ ?y \approx S\ 0$ is solved with the assignment $?f \mapsto S, ?y \mapsto 0$

Since we have no qualms about allowing multiple solutions and backtracking search our algorithms can handle both of these problems easily, in the first case by a special case of *projection* and in the second, by an *imitation* step. Our approach to free variables in metavariables is delightfully simple: there are none. In contrast, Ziliani and Sozeau carry around a suspended substitution with every metavariable, that needs to be managed in each resolution step. The heuristics outlined in this paper for constant unfolding are similar to ours: constants are unfolded only after an attempt has been made to apply type-class resolution, and constants are unfolded to a pattern match or fixpoint only in last resort. More study is needed to examine the trade-offs of these various choices. Finally, their system does not allow postponement of constraints, relying on pruning and dependency erasure to treat most cases up-front. They argue that great efficiency gains are obtained in this manner. Again, more study is required to assess the trade-offs of this approach.

Various algebraic developments in Coq make use of type classes [24, 25, 13] and *canonical structures* [23, 11, 17]; see also [2] for the use of *unification hints* in Matita. Many of the features we have described are also implemented in systems based on simple type theory. For example, Isabelle uses axiomatic typeclasses [28] and parameterized contexts (locales) [4] to deal with algebraic structures. It also has mechanisms to insert coercions [26]. The reliance on simple type theory, however, makes the elaboration problem quite different from ours. For example, an algebraic structure that depends on a parameter, such as the integers modulo m, cannot be represented as a type, and so cannot be an instance of an axiomatic type class. In contrast to Lean, Isabelle uses different languages to construct expressions and assertions, build proofs, and express relationships between structures.

In a different vein, recent work by Brady on the dependently typed language Idris [7] describes the elaboration process by analogy with theorem proving (and in the context of pure functional programming). Our work is in stark contrast with his, as our tactic language is completely disjoint from the methods with which we specify the constraint resolution for the unification problems. in Lean, the problems are quite different: in unification, metavariables can be very non-local, appearing in disparate contexts and the solutions can be an infinite stream rather than a simple finite case split.

5 Conclusions

We have described the elaboration procedure used in the new open source interactive theorem prover Lean [9]. Our procedure uses methods found in state-of-the-art constraints solvers, such as non-chronological backtracking, indexing, and justification tracking. We have also described how coercions, type classes and ad-hoc polymorphism can be smoothly integrated in our framework using choice constraints. Our procedure has been tested on more than 25k lines of formalized mathematics, including a standard library with basic datatypes and algebraic structures, a library for homotopy type theory, rudimentary category theory, and elements of non-abelian topology developed in homotopy type theory.

References

1. A. Abel and B. Pientka. Higher-order dynamic pattern unification for dependent types and records. In *TLCA*, volume 6690 of *LNCS*. Springer, 2011.
2. A. Asperti, W. Ricciotti, C. Sacerdoti Coen, and E. Tassi. Hints in unification. In *TPHOLs 2009*, volume 5674 of *LNCS*, pages 84–98. Springer, Berlin, 2009.
3. A. Asperti, W. Ricciotti, C. Sacerdoti Coen, and E. Tassi. The Matita Interactive Theorem Prover. In *CADE-23*, volume 6803 of *LNCS*. Springer, 2011.
4. C. Ballarin. Locales and locale expressions in Isabelle/Isar. In *Types for Proofs and Programs*, pages 34–50. Springer, 2004.
5. B. Barras, S. Boutin, C. Cornes, J. Courant, J.-C. Filliatre, E. Gimenez, H. Herbelin, G. Huet, C. Munoz, C. Murthy, et al. The Coq proof assistant reference manual: Version 6.1. 1997.

6. A. Bove, P. Dybjer, and U. Norell. A brief overview of Agda–a functional language with dependent types. In *TPHOL*, pages 73–78. Springer, 2009.
7. E. Brady. Idris, a general-purpose dependently typed programming language: Design and implementation. *Journal of Functional Programming*, 23:552–593, 9 2013.
8. T. Coquand and G. Huet. The calculus of constructions. *Information and Computation*, 76(2):95–120, 1988.
9. L. de Moura, S. Kong, J. Avigad, F. van Doorn, and J. von Raumer. The Lean Theorem Prover, 2015. submitted.
10. P. Dybjer. Inductive families. *Formal aspects of computing*, 6(4):440–465, 1994.
11. F. Garillot, G. Gonthier, A. Mahboubi, and L. Rideau. Packaging Mathematical Structures. In *TPHOL*, volume 5674 of *LNCS*. Springer, 2009.
12. W. D. Goldfarb. The undecidability of the second-order unification problem. *Theoretical Computer Science*, 13(2):225–230, 1981.
13. J. Gross, A. Chlipala, and D. I. Spivak. Experience implementing a performant category-theory library in Coq. In *ITP*, pages 275–291, 2014.
14. C. V. Hall, K. Hammond, S. L. Peyton Jones, and P. L. Wadler. Type classes in Haskell. *TOPLAS*, 18(2):109–138, 1996.
15. G. Huet. A unification algorithm for typed λ-calculus. *Theoretical Computer Science*, 1:27–57, 1975.
16. Z. Luo. ECC, an extended calculus of constructions. In *LICS*, pages 386–395. IEEE, 1989.
17. A. Mahboubi and E. Tassi. Canonical Structures for the working Coq user. In *ITP*, volume 7998 of *LNCS*, pages 19–34, Rennes, France, July 2013. Springer.
18. P. Martin-Löf. An intuitionistic theory of types. *Twenty-five years of constructive type theory*, 36:127–172, 1998.
19. C. McBride and J. McKinna. Functional pearl: I am not a number–I am a free variable. In *Haskell'04*, pages 1–9. ACM, 2004.
20. D. Miller and G. Nadathur. *Programming with Higher-Order Logic*. Cambridge, 2012.
21. C. Paulin-Mohring. Inductive definitions in the system Coq rules and properties. *Typed Lambda Calculi and Applications*, pages 328–345, 1993.
22. F. Rossi, P. Van Beek, and T. Walsh. *Handbook of constraint programming*. Elsevier, 2006.
23. A. Saïbi. Typing algorithm in type theory with inheritance. In *POPL '97*, pages 292–301. ACM Press, 1997.
24. M. Sozeau and N. Oury. First-Class Type Classes. In *TPHOLs 2008*, volume 5170 of *LNCS*, pages 278–293. Springer, August 2008.
25. B. Spitters and E. van der Weegen. Type classes for mathematics in type theory. *Mathematical Structures in Computer Science*, 21(4):795–825, 2011.
26. D. Traytel, S. Berghofer, and T. Nipkow. Extending Hindley-Milner type inference with coercive structural subtyping. In *APLAS*, pages 89–104, 2011.
27. T. Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. Institute for Advanced Study, 2013.
28. M. Wenzel. Using axiomatic type classes in Isabelle. 2005.
29. B. Ziliani and M. Sozeau. A predictable unification algorithm for Coq featuring universe polymorphism and overloading. submitted.