Automatically Deriving Schematic Theorems for Dynamic Contexts

Olivier Savary Bélanger
McGill University
osavary@cs.mcgill.ca

Kaustuv Chaudhuri
INRIA, France
kaustuv.chaudhuri@inria.fr

Abstract

Hypothetical judgments go hand-in-hand with higher-order abstract syntax for meta-theoretic reasoning. Such judgments have two kinds of assumptions: those that are statically known from the specification, and the *dynamic assumptions* that result from building derivations out of the specification clauses. These dynamic assumptions often have a simple regular structure of repetitions of *blocks* of related assumptions, with each block generally involving one or several variables and their properties, that are added to the context in a single backchaining step. Reflecting on this regular structure can let us derive a number of structural properties about the elements of the context.

We present an extension of the Abella theorem prover, which is based on a simply typed intuitionistic reasoning logic supporting (co-)inductive definitions and generic quantification. Dynamic contexts are represented in Abella using lists of formulas for the assumptions and quantifier nesting for the variables, together with an inductively defined *context relation* that specifies their structure. We add a new mechanism for defining particular kinds of regular context relations, called *schemas*, and *tacticals* to derive theorems from these schemas as needed. Importantly, our extension leaves the trusted kernel of Abella unchanged. We show that these tacticals can eliminate many commonly encountered kinds of administrative lemmas that would otherwise have to be proven manually, which is a common source of complaints from Abella users.

Categories and Subject Descriptors F.4.1 [*Mathematical Logic and Formal Languages*]: Mathematical Logic—Computational logic, lambda calculus and related systems, proof theory

General Terms Theory

Keywords dynamic contexts; context relations; context schemas; tactics and tacticals

1. Introduction

Higher-order abstract syntax (*HOAS*) [13], also known as λ-tree syntax (λ*TS*) [9], is the popular name for a representational scheme where data structures with binding constructs are represented using λ-terms in a logical framework in such a way the binding structure of the λ-terms reflects that of the represented data. In this paper we use the term *HOAS* in the narrow sense when the logical framework guarantees that all λ-terms are built out of variables, λ-abstractions, and applications, and that the equational theory of λ-terms identifies terms up to $\alpha\beta\eta$-conversion. For example, consider the following signature (in λ*Prolog* [10]) specifying a data structure for simply typed λ-terms.

```
kind ty   type.
type i    ty.
type arr  ty → ty → ty.

kind tm   type.
type app  tm → tm → tm.
type abs  ty → (tm → tm) → tm.
```

The term $\lambda f{:}i{\to}i.\, \lambda x{:}i.\, f(fx)$ would be represented as follows.

```
abs (arr i i)
  (λf. abs i (λx. app f (app f x)))
```

Reasoning about such representations requires a logic that can support arbitrarily nested implications and universal quantification, such as the logic of higher-order hereditary Harrop formulas (*HOHH*) [10, sec. 5.2.2] that forms the basis of λ*Prolog* and the *Abella* interactive theorem prover [5, 19]. Such logics are generally presented in terms of sequents (or hypothetical judgments) of the form $\Gamma \vdash C$ where C is a formula and Γ is a *context* of formulas. As an illustration, suppose we wish to represent the following type-checking judgment that relates a λ-term to its type. In λ*Prolog* we write it using a relation `of`[1] with these program clauses.

```
type of   tm → ty → o.

of (app M N) B ⇐
   of M (arr A B), of N A.
of (abs A R) (arr A B) ⇐
   pi λx. (of x A ⇒ of (R x) B).
```

The universal quantifier **pi** and implication $\Rightarrow$ occur in the body of the clause for abstractions and determines a scoped assumption for a *fresh* variable `x` that is used to reason about the body of the λ-term `R`. Note that `(R x)` stands for application in λ*Prolog*; if `R` were `λu. abs i (λx. app u x)`, for example, then `(R x)` would be equal to `abs i (λz. app x z)`, which avoids the capture of `x`.

Let Φ stand for this pair of defining clauses for `of`. The typing judgment $\lambda f{:}i{\to}i.\, \lambda x{:}i.\, f(fx) : (i \to i) \to i \to i$ amounts to showing that the following sequent is derivable:

```
Φ ⊢ of (abs (arr i i) (λf.
        abs i (λx. app f (app f x))))
      (arr (arr i i) (arr i i)).
```

[1] Note: all relations have target type **o**, the type of λ*Prolog* formulas.

LFMTP '14, July 17, 2014, Vienna, Austria.

ACM 978-1-4503-2817-3/14/07…$15.00.
http://dx.doi.org/10.1145/2631172.2631181

To prove this sequent, we would need to *backchain* the second clause in Φ, which will produce the subgoal:

```
Φ, of f (arr i i) ⊢
  of (abs i (λx. app f (app f x))) (arr i i)
```

where f is a *fresh* variable, *i.e.*, it does not occur free in the original goal. Backchaining once more would produce the subgoal:

```
Φ, of f (arr i i), of x i ⊢
  of (app f (app f x)) i
```

where x is a fresh variable, *i.e.*, it does not occur in the first subgoal and is different from f.

As should be obvious from this example, every context that occurs in derivations involving the `of` relation has the structure:

```
Φ, of x₁ t₁, ..., of xₙ tₙ
```

where for $i \in 1..n$, the variable x_i is fresh for Φ and for (x_j, t_j) for every $j \in 1..i-1$. The program Φ is a *static* participant in these contexts, while the rest of the context is *dynamic*, determined entirely by the original goal. In Abella, which can support inductive definitions and generic quantification (using the ∇ quantifier [11]), the form of this dynamic portion of the context can be specified as an inductively defined predicate `ctx` as follows.

```
Define ctx : olist → prop by
  ctx nil
; nabla x, ctx (of x A :: G) ≜ ctx G.
```

The type **olist** denotes a list of *HOHH* formulas (built as usual using **nil** and `::`) that represents the context, while the type **prop** denotes formulas of the meta-logic. The definition consists of a sequence of clauses separated by semi-colons; each clause contains a head and an optional body (specified using $\triangleq$). We follow the *λProlog* convention of universally closing every clause of the definition over its capitalized free variables. The **nabla** at the head of the second clause of this inductive definition asserts that `x` is fresh for—*i.e.*, does not occur free in—`A` and `G`.

Using definitions such as `ctx` in proofs requires a number of essentially *administrative* inductive theorems for reasoning about the dynamic context (*i.e.*, lists of type **olist**) specified by it.. These lemmas amount to unfolding the `ctx` definition to observe the structural properties of its argument, such as that the head of the list is of the form `of x A`, that `x` is a nominal constant, and that it does not occur in `A` or in the tail of the list. In Abella 2.0, such theorems have to be proved manually. This is a common source of frustration because of the generally uninteresting nature of these theorems and their proofs. The problem is worse than it appears on the surface because in a large development there may be several relations like `of` above that may even be mutually recursive. Moreover, we often need to reason about multiple contexts at the same time using inductively defined *context relations*, which causes an exponential proliferation of administrative lemmas.

In this work, we add a small amount of automation to Abella that simplifies this kind of administrative overhead in the case where the contexts being specified are *regular*. We add a mechanism to Abella to define such regular contexts in terms of *context relation schemas*, which is an explicit representation of the context relation as a weak form of regular expression. This notion is a variant of regular worlds from Twelf [15] and schemas from Beluga [16], but generalized to context relations of arbitrary arity. We then add *tacticals* to Abella that reflect on both the proof state and the declared schemas to derive a number of administrative lemmas (along with their proofs) automatically and on demand. Our automation is certifying: we leave the core language and tactics of Abella unchanged, but add a shallow surface layer of syntax that is compiled—as needed—into that core. This is achieved by adding a *plugin* architecture to Abella that allows for well-delimited extensions to the grammar of Abella; these plugins in turn produce textual output that is then re-parsed by the core (unmodified) Abella parser. Indeed, these plugins can be used in an *elaboration* mode to remove all uses of the plugin from an Abella development. Therefore, we do not rely on extensions of the trusted computing base of Abella, not even its parser.

We begin with a quick overview of the Abella system (Section 2) followed by a discussion of its new plugin architecture extension (Section 3). We then give the specifics of the main *Schemas* plugin that implements a mechanism for declaring schematic context relations (Section 4). The particular administrative lemmas that are derived automatically by this plugin are explained in detail in Section 5. We end with a some quantitative evaluation of the plugin (Section 6) and summary of related work (Section 7).

The implementation of this version of Abella can be found in:

`http://abella-prover.org/schemas`

2. Abella: an Overview

The Abella system has been documented in a sequence of papers [5, 19] and has a web-site[2] with a sequence of tutorials, a user manual, and an annotated suite of examples. We will only sketch the use of Abella in this paper, eliding all details of its proof language.

Fundamentally, Abella consists of a *reasoning logic* that is ordinary first-order intuitionistic logic extended with:

- inductive and co-inductive definitions of predicates;
- a simply typed higher-order term language endowed with an intensional equality predicate at all types whose semantics is given by *unification*;
- nominal constants and *equivariant* equality—*i.e.*, two terms that may be rewritten to each other by $\alpha\beta\eta$ and a systematic permutation of their nominal constants are equated;[3] and
- the nabla (∇) quantifier [11] and nominal abstraction [7] for reasoning about nominal constants.

One particular inductive definition for a focused sequent calculus for *HOHH* is treated specially, with a notation using `{ }` and tactics designed to leverage certain meta-theoretic properties of this definition [19]. This inner *specification logic* is a fragment of the *λProlog* language, so Abella can be used to reason about *λProlog* specifications of object logics. Thus, Abella is an instance of the *two-level logic approach* to specification and reasoning [8].

As a concrete illustration of the use of Abella, let us take the typing example from the previous section. The type and kind declarations are placed in a *signature* (here, `stlc.sig`), and the clauses for the declared relations are placed in a corresponding *module* (here `stlc.mod`). The pair of `.sig` and `.mod` files can be directly executed in *λProlog*, such as using the `tjcc` compiler and `tjsim` interactive toplevel of the Teyjus implementation [17]. Reasoning about a given specification (a signature/module pair) is done either interactively at the Abella toplevel or in a batch form using a `.thm` file (here, `type_unique.thm`). Figure 1 lists the contents of the signature, the module, and an initial portion of the reasoning file for a theorem stating that the types of λ-terms are uniquely determined by the `of` predicate.

The theorem `unique` is proved by induction on the structure of the first *HOHH* derivation, *viz.* `{G ⊢ of M A}`. This is achieved in Abella by means of the **induction** tactic that produces this subgoal:

```
IH : forall G M A B, ctx G → {G ⊢ of M A}* →
       {G ⊢ of M B} → A = B.
====================
forall G M A B, ctx G → {G ⊢ of M A}@ →
  {G ⊢ of M B} → A = B.
```

[2] `http://abella-prover.org`

[3] This is related to a similar notion from nominal logic [18], but we retain the *HOAS* representation of terms.

```
sig stlc.
  kind ty, tm  type.
  type i    ty.
  type arr  ty → ty → ty.
  type app  tm → tm → tm.
  type abs  ty → (tm → tm) → tm.

  type of   tm → ty → o.
end.
```

stlc.sig

```
module stlc.
  of (app M N) B :-
    of M (arr A B), of N A.
  of (abs A R) (arr A B) :-
    pi λx. of x A ⇒ of (R x) B.
end.
```

stlc.mod

```
Specification "stlc".

Define ctx : olist → prop by
  ctx nil
; nabla x, ctx (of x A :: G) ≜ ctx G.

Theorem unique : forall G M A B, ctx G →
  {G ⊢ of M A} → {G ⊢ of M B} →
  A = B.
```

type_uniq.thm

Figure 1. Simply typed λ-calculus in Abella

(Note: *Abella* adopts the *Coq* style of displaying subgoals: hypotheses and conclusions are separated by a line of '='s.) The *inductive hypothesis*, called `IH`, has the same form as the theorem except the assumption that must be strictly smaller is marked with `*`. This assumption in the conclusion is modified to have annotation `@`. This annotation changes to $*$ after at least one application of a backchaining step, *i.e.*, at least one *unfolding* of the inductive definition of the *HOHH* sequent calculus. In *Abella*, this is achieved using the **case** tactic that considers every possible way to backchain on clauses in the program or in the dynamic context `G` to derive the conclusion `of M A`. As expected, there are exactly three possibilities.

P1. Backchaining on the first program clause produces:

```
Variables: G M A B M1 N1 A1 B1
H1 : ctx G
H3 : {G ⊢ of (app M1 N1) B}
H4 : {G ⊢ of M1 (arr A1 B1)}*
H5 : {G ⊢ of N1 A1}*
====================
B1 = B
```

The *eigenvariables* `M` and `A` are *unified* with the terms `app M1 N1` and `B1` respectively.

P2. Backchaining on the second program clauses produces:

```
Variables: G M A B R A1 B1,
H1 : ctx G
H3 : {G ⊢ of (abs A1 R) B}
H4 : {G, of n1 A1 ⊢ of (R n1) B1}*
====================
arr A1 B1 = B
```

Once again, some eigenvariables get unified with other terms. Moreover, the new assumption `H4` has a larger dynamic context, with a fresh assumption `of n1 A1` where `n1` is a nominal constant. *Abella* uses the convention that all identifiers beginning with '`n`' and followed by numbers are nominal constants.

P3. Finally, backchaining on an element of `G` itself produces:

```
Variables: G M A B F
H1 : ctx G
H3 : {G ⊢ of M B}
H4 : {G, [F] ⊢ of M A}*
H5 : member F G
====================
A = B
```

The hypothesis `H4` stands for the assertion that we are backchaining on `F`, which must be a member of `G` by `H5`. Note that we don't necessarily know what `G` is, but we do have an inductive characterization of its structure by `H1`.

The proof follows the technique of unfolding `H3`, and then appealing to the `IH` on the results. For example, for P1, one of the cases would be:

```
Variables: G M A B M1 N1 A1 B1 A2 B2
H1 : ctx G
H4 : {G ⊢ of M1 (arr A1 B1)}*
H5 : {G ⊢ of N1 A1}*
H6 : {G ⊢ of M1 (arr A2 B2)}
H7 : {G ⊢ of N2 A2}
====================
B1 = B2
```

Here, invoking the `IH` on `H1, H5, H7` will unite `A1` and `A2`, so `H1, H4, H6` will then unite `arr A1 B1` with `arr A1 B2`, which will make the conclusion true. The other case is when `H3` is itself proved by backchaining on a clause in `G`:

```
Variables: G M A B M1 N1 A1 B1
H1 : ctx G
H4 : {G ⊢ of M1 (arr A1 B1)}*
H5 : {G ⊢ of N1 A1}*
H6 : {G, [F] ⊢ of (app M1 N1) B}
H7 : member F G
====================
B1 = B
```

This case is impossible, since (by `H1`) `G` contains only assumptions of the form `of n C` where `n` is a nominal constant. Nominal constants can only be united with other nominal constants up to equivariance, so `{G, [of n C] ⊢ of (app M1 N1) B}` has no proof, since `n` and `app M1 N1` do not unify. In *Abella* this can be stated as a lemma.

```
forall G E, ctx G → member E G →
  exists X A, (E = of X A) ∧ name X.
```

where `name` asserts that its argument is a nominal constant of type `tm`, definable in *Abella* as:

```
Define name : tm → prop by nabla x, name x.
```

This is an administrative lemma that can almost entirely be derived from the `ctx` definition.

Another example of an administrative lemma comes from case P3, where from the above lemma we know that `F = of n1 C` and `member (of n1 C)(G n1)`. Note that `G` is *raised* over the new nominal constant `n1`. This changes `H4` to `{G n1, [of n1 C] ⊢ of (M n1) (A n1)}`, which in turn gives the solution `M n1 = n1` (*i.e.*, `M = λx. x`) and `A n1 = C` (*i.e.*, `A = λx. C`). Now, if we apply the same reasoning to the hypothesis `H3`, we would deduce that `member (of n1 D) (G n1)` and `of n1 D`. The conclusion will require us to show that `C = D`. This requires the following lemma.

```
forall G X A B, ctx G → member (of X A) G →
  member (of X B) G → A = B.
```

This is a *uniqueness* lemma that guarantees that every every variable is assigned a unique type in `G` by the `ctx` definition. As before, this lemma has an uninteresting inductive proof that follows almost entirely from the definition of `ctx`. Indeed, its proof itself uses another administrative lemma asserting that a nominal constant that does not occur in a list cannot occur in any member of the list.

```
forall G E, nabla (n:tm),
  member (E n) G → exists F, E = λx. F.
```

In other words, if `E n` occurs in `G`, which cannot depend on `n` because of the order of **forall** and **nabla**, it must be the case that `E` cannot depend on `n` either, *i.e.*, `E` begins with a vacuous λ-abstraction.

In all, the administrative lemmas and their proofs constitute about 60% of the lines of code in this reasoning file. Such lemmas occur repeatedly in the examples suite of *Abella*, often with slight variations in their formulation and a wide variance in their names. Larger developments contain a number of specified relations such as `of`, each producing its own `ctx` definition and their associated administrative lemmas. Indeed, Abella even allows for *context relations*, which are inductive definitions such as `ctx` with multiple context arguments, which further causes an exponential proliferation of administrative lemmas. It has been clear for a long time that we require better automation to deal with such lemmas about contexts. Indeed, this is one of the criticisms of *Abella* in the recent survey of *HOAS* reasoning systems by Felty *et al*. [4].

3. A Framework for Plugins

This work proposes to derive a large class of these administrative lemmas automatically when the relevant `ctx`-like definition has a *regular* form. We implement this technique in terms of a *plugin* in an extension of the *Abella* system with a plugin architecture. As the architecture is rather generic, we describe it before the particular plugin for deriving administrative lemmas.

Abella is written in *OCaml* and has a broadly *LCF*-style architecture with a core family of trusted tactics that formalize the inference rules of the logic $\mathcal{G}$ [7]. The basic reasoning tactics **case** (for case-analysis) and **search** (for depth-bounded automated search) are implemented using these core tactics. However, *Abella* 2.0 lacks a mechanism for defining new tactics like **case** and **search**; users of *Abella* must write their proofs using the tactics that already exist. This design allows *Abella* to be compiled—even to act as a compiler itself—but does limit its versatility.

Our approach is to allow users to write *Abella* plugins that can extend both the grammar of *Abella* and its family of tactics. However, we do not allow arbitrary extensions of either. We require all extensions to the grammar to be explicitly delimited, and for all top level commands and proof tactics added in the plugins to function as *elaborators* that produce proof text for the core *Abella* plugins. This not only makes the plugins certifying, keeping the trusted core of *Abella* unaltered, but also allows developments built using plugins to be used even in versions of *Abella* without the plugin architecture.

Each toplevel command or tactic added by a plugin named `Plug` must have the form

```
Plug ! <text> !.
```

where the `<text>` is arbitrary text that must not contain the token '`!`'. *Abella* will scan its list of known plugins for a plugin named `Plug`, which will then be asked to elaborate the `<text>` into either toplevel commands or core tactics, depending on where it was encountered. Plugins can be stateful: they can store and recall all the text that they have encountered in a single run of *Abella*. However, they are not allowed to modify any associated specification or reasoning files, nor the internal data structures of *Abella*'s core.[4]

More precisely, every plugin is an *OCaml* module that ascribes to the following module type:

```
module type PLUGIN = sig
  val process_tactic :
        core:(string → Prover.sequent)
        → string
        → Prover.sequent
        → unit
  val process_top :
        core:(string → unit)
        → string
        → unit
end
```

[4] Since *OCaml* is an impure language, it is not possible to enforce these rules as such; however, since all plugins must be able to produce output that can be re-checked in a version of *Abella* without plugins, no plugin can ultimately break soundness.

Each module of type `PLUGIN` has to implement two functions, `process_tactic` and `process_top`, defining its behavior on tactics and toplevel commands, respectively. Each function takes a named parameter `core`, a shallow wrapper around the core *Abella* functionality which processes the elaborated string produced by the plugin. In particular, this string argument to `core` is parsed by the unmodified *Abella* parser, *i.e.*, the parser from *Abella* 2.0 that does not implement the plugin architecture. These core functions may be called—possibly never or multiple times—by the plugin functions, but a plugin must treat the `core` function abstractly. The `process_tactic` function can additionally reflect on the state of the prover—*i.e.*, the current subgoal that has the type `Prover.sequent`—at the point where the corresponding plugin tactical was invoked. However, this function cannot construct new sequents and must instead drive the `core` function using core *Abella* tactics for every new sequent it wishes to create. The only way for the plugin to alter the state of *Abella* using the `core` function.

To add a new plugin to Abella, it is necessary to add the module implementing `PLUGIN` to a global `plugins` table. This table is stored in the file `abella.ml` that is the entry point for *Abella*, so every added plugin requires recompiling this file and relinking *Abella*. For instance, to add the *Schemas* plugin implemented as the *OCaml* module `Schemas` (described in the next section), we add the following line to `abella.ml` and recompile *Abella*.

```
Hashtbl.add plugins "Ctx"
  (module Schemas : PLUGIN);;
```

Note that `plugins` is a mapping from strings to first-class *OCaml* modules, which were added in *OCaml* 3.12 and significantly improved in 4.0. We require plugin names to be valid upper-cased *Abella* identifiers distinct from all built-in core keywords, and their namespace is flat and global. In future work, we plan to use the dynamic loading features of *OCaml* 4.02+—which is not yet released at the time of writing this paper—to avoid recompilation, and instead have *Abella* dynamically initialize the table of plugins from a configuration file.

4. Regular Context Relations

The `ctx` definition of Fig. 1 is a unary *context relation*. *Abella* allows definitions of context relations of arbitrary arity, and even relations between contexts and other inductively defined structures such as natural numbers. From this zoo of possibilities, we select a class of *regular context relations* for which we can automatically derive the administrative lemmas. A regular context relation of arity $n \geq 1$:

- is an inductively defined predicate on n arguments of type **olist** ;
- relates n **nil** s as the base case; and
- each non-base case clause of the predicate completely specifies the heads of all the argument lists and whose bodies are just recursive invocations on the tails of the lists.

The *Schemas* plugin of *Abella* adds a new toplevel declaration, **Schema**, for declaring such regular context relations. This declaration has the following general form

$$\textbf{Schema}\ \texttt{<name>}\ \triangleq\ \texttt{<clause}_1\texttt{>}\ ;\ \cdots\ ;\ \texttt{<clause}_j\texttt{>}$$

where each `<clause`$_i$`>` has the form:

$$\textbf{exists}\ \texttt{A}_1\ \ldots\ A_m,\ \textbf{nabla}\ \texttt{x}_1\ \ldots\ \texttt{x}_n,\ (\texttt{F}_1,\ \ldots,\ \texttt{F}_k)$$

where the $\texttt{F}_i$ are either arbitrary *HOHH* formulas built using the variables $\texttt{A}_1, \ldots, \texttt{A}_m, \texttt{x}_1, \ldots, \texttt{x}_n$ or left blank, indicating that the clause does not modify this projection of the context relation. The number of F_i determines the arity of the definition; each clause must specify exactly k projections for a relation of arity k. Note that the nesting order of **exists** and **nabla** is fixed and guarantees that every x_i is fresh for each A_j.

As a simple example, here is how the `ctx` definition of Fig. 1 can be written as a schema.

```
Schema ctx ≜
  exists A, nabla x, (of x A).
```

Using the `Ctx` plugin, we would in fact write it as follows:

```
Ctx! Schema ctx ≜ exists A, nabla x, (of x A). !.
```

When the `Ctx` plugin processes this declaration, it instructs *Abella*'s kernel (using `process_top`, *cf.* Section 3) to process exactly the inductive definition of `ctx` in Fig. 1. The `ctx nil nil` case is implicitly added, and is therefore not part of the schema declaration. In the rest of this section, we will elide the `Ctx! !` delimiters.

A more complex example comes from the `normal.thm`[5] file from *Abella*'s example suite that shows how to partition λ-terms into normal and neutral (*aka.* atomic) forms:

```
Define ctxs : olist → olist → prop by
  ctxs nil nil
; nabla x, ctxs (term x :: L)
                (neutral x :: K) ≜ ctxs L K.
```

Here is how it is depicted as a context relation schema.

```
Schema ctxs ≜ nabla x, (term x, neutral x).
```

Note that if any of the **exists** or **nabla** bound variables list is empty, the corresponding **exists** or **nabla** prefix may be dropped. The important feature of this schema is that the nominal variable `x` is shared between the two contexts in the relation.

For a yet more complex example to illustrate that the formulas at the heads of the lists representing the related contexts need not be atomic, take the `ctx2` definition from `breduce.thm`[6] [19].

```
Define ctx2 : olist → olist → prop by
  ctx2 nil nil
; nabla x p, ctx2 (bred x x :: G)
                  (path x p :: D) ≜ ctx2 G D
; nabla x,
    ctx2 ((pi λu. bred N u ⇒ bred x u) :: G)
         ((pi λq. path N q ⇒ path x q) :: D) ≜
      ctx2 G D.
```

Here is its depiction as a context relation schema.

```
Schema ctx2 =
  nabla x p, (bred x x, path x p)
; exists N, nabla x,
    ((pi λu. bred N u ⇒ bred x u),
     (pi λq. path N q ⇒ path x q)).
```

The second clause above has three kinds of variables: existential (`N`), nominal (`x`), and bound (`u` and `q`). Only the existential and nominal variables can be shared between the related contexts.

For a final example, take the `ctxs` definition from `cr.thm`:[7]

```
Define ctxs : olist → olist → olist → prop by
  ctxs nil nil nil
; nabla x, ctxs (trm x :: L)
                (pr1 x x :: K)
                (cd1 x x :: notabs x :: J) ≜
     ctxs L K J.
```

[5] In: `examples/lambda-calculus/term-structure/normal.thm`

[6] In: `examples/higher-order/breduce.thm`

[7] In: `examples/lambda-calculus/cr.thm`

The third argument of the second clause adds two elements to the head. We use the conjunction operator (&) of *λProlog* in the corresponding schema.

```
Schema ctxs =
  nabla x, (trm x, pr1 x x, cd1 x x & notabs x).
```

It should be noted that the support for reasoning about & is currently rather primitive in *Abella*. While the above declaration is accepted by the plugin, the automatically derived theorems currently are not accepted by *Abella*. (The generated definition itself is accepted.)

We end this section by noting a number of ways in which regular context relations given as schemas do not capture the full generality of definable relations in *Abella*:

- Multiple schemas may not be mutually recursive.
- Schemas can only relate dynamic contexts (**olist**), not other inductively defined objects such as natural numbers.

None of these restrictions is significant as there is exactly one instance of each kind in the current *Abella* examples. Moreover, removing these restrictions appears to add considerable complications to the automatic derivation of theorems. We therefore leave them to future work.

5. Derived Administrative Lemmas

In this section we inventory the administrative lemmas that are automatically derived from the schema declarations by the *Schemas* plugin. These lemmas are of two basic kinds: those that arise from the *types* of the existentially and nominally quantified variables in a schema declaration, and those that arise from its logical structure. Lemmas of the first kind are mainly used in the automatically derived proofs of the lemmas of the second kind, but are sometimes also useful in the general toolset.

5.1 Lemmas from Types

Consider again the simple schema below corresponding to the `ctx` predicate of Fig. 1.

```
Schema ctx ≜ exists A, nabla x, (of x A).
```

As we already mentioned, from this declaration (and its induced inductive definition), we intend to derive, automatically, that `x` is a nominal constant and that `x` is fresh for `A`. In *Abella*, these properties are easily defined, but because *Abella* is not polymorphic, these definitions have to be manually monomorphized to the types in question. For instance, in the above schema, type-inference would derive the fact that `A` has type `ty` and `x` has type `tm`. Thus, we would need the following instances of the `name` and `fresh` predicate:

```
Define name_tm : tm → prop by
  nabla x, name x.
Define fresh_tm_in_ty : tm → ty → prop by
  nabla x, fresh x A.
```

As a side-effect of processing the **Schema** declaration above, the *Schemas* plugin automatically adds these definitions. Precisely, a `name` predicate is generated for each type of nominally quantified variable in any clause of a schema, and a `fresh` predicate for every pair of types of nominal variables and existential variables in each clause of the schema. Note that these instances apply to basic types declared in the signature, not to arbitrary types; if the schema uses such types, then no such definitions are generated. The *Schemas* plugin keeps track of all such administrative definitions to prevent duplicates, but it may add definitions that are not ever used. This is because *Abella* only allows inductive definitions at the top-level, not during a proof, and no plugin is allowed to "rewind" the state of *Abella* to retroactively add definitions.

For each type of a nominally quantified variable, the *Schemas* plugin also generates a `prune` lemma, as explained in Section 2. Here is the version for `tm`:

```
Theorem member_prune_tm :
  forall G E, nabla (x:tm),
    member (E x) G → exists F, E = λx. F.
```

Note that no `member_prune_ty` is generated as there is never a nominally quantified variable of type `ty` in the schema. The *Schemas* plugin uses `process_top` and `process_tactic` to both state and prove the `member_prune_tm` theorem.

We note here that much of this administrative boilerplate can be removed if *Abella* were to support type-polymorphism. The definitions of `name` and `fresh`, and the statement and proof of `member_prune`, are identical for every type, and ideally should be part of the standard prelude. However, this does not mean that type-based administrative definitions and lemmas are completely worthless. For instance, *Abella* does not currently allow induction on typing itself, which means that induction on the structure of a term must be mediated by a somewhat redundant inductive definition of the structure of well-typed terms. Such definitions and their corresponding lemmas can be automatically derived by the *Schemas* plugin (or by a different specialized plugin) in the future.

5.2 Lemmas from Logical Structure

The remaining administrative lemmas in the *Schemas* tactic come from the logical structure of the **Schema** declaration. These are implemented in the *Schemas* plugin as new *tacticals* that can be invoked in the process of a proof. Each tactical reflects on the structure of the subgoal being proved and the schema declarations known so far to introduce new assumptions into the context, which are then used using the standard *Abella* tactics to continue the proof.

Inversion. The `inversion` tactical reflects on two assumptions, one of which is a context atom `ctx` $G_1 \ldots G_n$, where `ctx` is produced by a **Schema** declaration, and the other is of the form `member E` G_i for some $i \in 1..n$. The result of the tactical is that E must be one of the formulas that occur in the ith position in the clauses of the **Schema** declaration. For example, given the schema:

```
Schema ctx_ofev ≜
  exists A, nabla x, (of x A, eval x x)
; exists A V, nabla x, (of x A, eval x V).
```

and a subgoal with:

```
H1 : ctx_ofev G D
H2 : member E D
```

the tactical `inversion H1 H2` produces the new assumption

```
H3 : (exists A X, (E = eval X X)
        ∧ member (of X A) G
        ∧ fresh_tm_in_ty X A)
   ∨ (exists A V X, (E = eval X V)
        ∧ member (of X A) G
        ∧ fresh_tm_in_ty X A
        ∧ fresh_tm_in_tm X V)
```

Each disjunct produced by this tactical therefore contains:

- the corresponding member(s) of the other context(s) in the context relation; and
- the necessary assumptions about freshness of the **nabla**-quantified variables in the **Schema** declaration

corresponding to the clause.

Internally, the `process_tactic` function of the plugin is used to first assert[8] and prove the general form of an *inversion lemma*; this asserted lemma is then used for the particular hypotheses indicated in the arguments of the tactical. The proof of the assertion is by a nested induction on the induced inductive definition produced by the relevant **Schema** declaration, with one case each for each clause of the schema and an additional base case for the related contexts all being empty.

This generated statement and proof of the inversion lemma is cached by the *Schemas* plugin. If it is used repeatedly in subproofs of the same proof, then it does not need to be re-checked by *Abella*. However, this is not the case if the `inversion` tactical is used in sibling branches or in other theorems, where it would have to be checked again. This design is currently due to limitations of *Abella*'s design that prevents closed lemmas from being exported out of proofs. Moreover, although *Abella* allows aborting of the current proof, the plugin architecture does not see the whole proof and hence cannot itself replay the whole proof in a suitably modified environment with an additional named lemma. These restrictions are not fundamental and may be lifted in future versions of *Abella* and the *Schemas* plugin.

Synchronize. Related to the `inversion` tactical is `sync`, which uses the form of the term in the member relation to select the relevant disjunct(s) of the inversion lemma. For instance, consider the following simplified form of the schema form of the `ctx2` relation of `breduce.thm`:

```
Schema ctx_bp ≜
  nabla x p, (bred x x, path x p)
; exists N, nabla x, (bred x N, jump x N).
```

Here, if we knew that:

```
H1 : ctx_bp G D
H2 : member (path n1 n2) D
```

then the tactical application `sync H1 H2` produces:

```
H3 : member (bred n1 n2) G
```

as that is the only disjunct of the inversion lemma that is relevant. Note that `n1` and `n2` must be nominal constants by the lexical structure of *Abella*.

A more interesting case is:

```
H1 : ctx_bp G D
H2 : member (bred n1 n2) G
```

In this case, `sync H1 H2` would produce:

```
H3 : member (jump n1 (N n1 n2)) G
H4 : fresh_tm_in_ty n1 (N n1 n2)
```

for a fresh variable `N` that is raised over *both* `n1` and `n2`. The first clause of the schema does not match because `bred n1 n1` does not equivariantly unify with `bred n1 n2`. In this case, the additional assumption `H4` would suffice to show that `N n1 n2` does not actually contain `n1`, *i.e.*, that `N` has a vacuous λ-abstraction.

This tactical is more useful than `inversion` when the form of the member is constrained enough to fit exactly one clause of the schema. If it were applied to unconstrained terms, then the effect would just be a case enumeration identical to the use of `inversion`. The `sync` tactical is implemented in much the same way as `inversion`, except it also prunes obviously impossible cases based on the patterns of the formulas in the schema. Note that this tactic would fail to apply in the case that the unification problems fall outside the pattern fragment, but this is not a limitation of the plugin as proving the equivalent theorem in core *Abella* would require manual intervention anyhow. (Such schemas are rare in practice.)

Uniqueness. A very useful administrative lemma is the fact that each **nabla**-quantified variable has at most one point of introduction in a regular context relation. This is best illustrated with an example: consider again the schema for `ctx` from Fig. 1:

[8] The `assert` tactic of Abella is used to assert and prove a lemma in a subproof and then to continue the proof with the lemma as a hypothesis, *i.e.*, it is an instance of the cut rule of the $\mathcal{G}$ logic.

```
Schema ctx ≜ exists A, nabla x, (of x A).
```

In this case, if we are in a subgoal with:

```
H1 : ctx G
H2 : member (of X A) G
H3 : member (of X B) G
```

then it must be that `A` and `B` are equal, since there is only one clause of `ctx` that could have introduced any member of the form `of X C` into `G`. This is achieved by the tactical application `unique H1 H2 H3`, which has the side effect of uniting the terms `A` and `B`.

While easily explained, this tactical has several subtleties. First, we require that the contexts—the `G` above—be identical in all three arguments to `unique`, and that each member—the `of X A` and `of X B` above—be unifiable with one of the formulas in the contexts related in the **Schema** declaration. If the latter assumption is not true, then we can just use `inversion` to rule out this entire subgoal as impossible. Second, we do not require the term corresponding to the **nabla**-quantified variables—the `X` above—to be a nominal constant; if it is not a nominal constant, then the inversion lemma rules out the subgoal as impossible. Finally, the generated lemma and its proof requires the use of the `member_prune` lemma explained in the previous section: in the inductive argument, the case where one of the members is the first element of the context while the other member is not is impossible, and `member_prune` very succinctly rules it out.

Projection. It is a common design pattern in Abella to prove inductive theorems for the smallest context relations that suffice. Thus, theorems about typing using a specified relation `of` would use a unary context schema about `of`, while those about evaluation using `eval` would use a unary schema for `eval`. However, if a theorem has to relate typing to evaluation, such as in proofs of type-preservation, then it is necessary to state the theorem using a binary schema relating the two contexts. Unfortunately, in *Abella* there is no automatic way to "import" a theorem proved using a unary context relation into one with a binary relation, nor "export" theorems the other way. Such facts must be proved by hand.

A common denominator of such facts is that there exist mappings between two context relations that existentially close over the contexts in the target of the mapping that are not present in the source. We call such mappings *projections*. The `projas` tactical applies to an assumption:

$$\mathtt{H1} : \mathtt{rel1}\ \mathtt{G}_1\ \ldots\ \mathtt{G}_n$$

where `rel1` is a schematic context relation. The tactical application

$$\mathtt{projas}\ (\mathtt{rel2}\ \mathtt{D}_1\ \ldots\ \mathtt{D}_m)\ \mathtt{H1}$$

where each $\mathtt{D}_j$ is either one of the $\mathtt{G}_i$ or is a new eigenvariable, has the effect of adding the assumption

$$\mathtt{H2} : \mathtt{rel2}\ \mathtt{D}_1\ \ldots\ \mathtt{D}_m$$

to the goal, when justified.

This tactical application is interpreted into a general *projection lemma* that has the following form. Let $\mathtt{D}_{\phi(1)}, \ldots, \mathtt{D}_{\phi(k)}$ be the eigenvariables that are distinct from all the $\mathtt{G}_i$. Then, the following lemma is proved by induction:

$$\textbf{forall}\ \mathtt{G}_1\ \ldots\ \mathtt{G}_n,\ \mathtt{rel1}\ \mathtt{G}_1\ \ldots\ \mathtt{G}_n \to \\ \textbf{exists}\ \mathtt{D}_{\phi(1)}\ \ldots\ \mathtt{D}_{\phi(k)},\ \mathtt{rel2}\ \mathtt{D}_1\ \ldots\ \mathtt{D}_m.$$

This proof proceeds by induction on the definition of `rel1`, but is rather straightforward. Of course, if all the $\mathtt{D}_j$ are distinct from the $\mathtt{G}_i$, then this tactic is useless. Like the other tacticals, `projas` detects invocations which are invalid or outside its fragment and only generate proofs which will be accepted by the *Abella* kernel.

To illustrate one of the limitations to its fragment, consider the following pair of schemas:

```
Schema rel1 ≜ (i, i).
Schema rel2 ≜ (i,  ); ( , i).
```

where `i` is an atomic *HOHH* formula of type **o**. Clearly,

```
forall G, rel1 G → rel2 G.
```

is provable. However, as no single clause of `rel2` matches the non-trivial clause of `rel1`, `projas` would not apply to this theorem.

6. Experimental Evaluation

We based our implementation of the plugin architecture on *Abella* version 2.0.1.[9] Our initial experiments are promising. For instance, using the *Schemas* plugin to rewrite the `breduce` example from [19] removes over 40% of the lines of code from the file `breduce.thm`. Table 1 contains a summary of improvements in a few other examples from the *Abella* examples suite. In addition to this quantitative reduction in size, we can also compare the plugin qualitatively: the *Schemas* tacticals free us from the tedium of writing and proving the administrative lemmas that make *Abella* developments both tedious to write and hard to read. Our experience using the plugin has been entirely positive, so we plan to integrate the plugin architecture into the next release (2.1.x) of *Abella*.

File	# schemas	# lemmas derived	LOC removed	% removed
`breduce`	3	11	124	42
`copy`	1	4	28	43
`cr`	1	3	32	19
`type_uniq`	1	3	27	63

Table 1. Quantitative evaluation of the *Schemas* plugin on some examples from the *Abella* examples suite

7. Related Work

The concept of regular context relations, at least in the unary case, is similar to that of *regular worlds* from *Twelf*, introduced in version 1.4 [14, Section 9.1]. A regular world is an arbitrary repetition of a sequence of *blocks*, which are individually named in *Twelf* and correspond to the clauses of our **Schema** declarations. Despite the superficial similarity of *Twelf*'s block and world declarations and our **Schema** declarations, there are some significant differences: first, we use nominal abstraction ("**nabla** at the head") [7] to interpret our **nabla**-quantified variables, rather than universal quantification as in *Twelf*, which allows us to directly use the logical principles of $\mathcal{G}$ to derive pruning, inversion, and uniqueness theorems; second, regular worlds in *Twelf* are tied to a particular inductive type family and cannot be reused as such for different families, nor can a family have different regular world declarations; third, the regularity is at the level of local extensions to the dynamic context rather than to the entire dynamic context as a whole; and finally, because *Twelf* contexts contain both variable declarations and ordinary assumptions, the rigid list structure of regular worlds forces the use of somewhat unnatural placement of quantifiers in the specification, explained in [14, p. 49]. *Twelf* also has a concept of *world-checking*, where the constructors of an inductive type family in a signature are automatically checked (using a trusted checker) to conform to the declared world for that family. This feature is sometimes useful as a sanity check on specifications, but is ultimately orthogonal to formal (logical) reasoning about the specifications.

Regular contexts are given a more principled foundational treatment in the *Beluga* system [16], which is a dependently typed programming language for reasoning about contextual modal *LF* terms [12]. Indeed, we appropriated the term "*schema*" from *Beluga*. Schemas in *Beluga*, like regular worlds in *Twelf*, are treated as

[9] See: `http://abella-prover.org/schemas`

classifiers of individual contexts, which make them similar to unary context relation schemas in our plugin. Schemas are tightly integrated into the *Beluga* type system, and it does not make much sense to ask for its treatment of schemas to be certifying with respect to a system without schemas. However, this does raise the level of trust required in the *Beluga* implementation. For instance, administrative lemmas such as uniqueness and inversion are unnecessary in *Beluga* as they are built into the type-checker, which is therefore necessarily more complex than the rather straightforward implementation of the core tactics of *Abella*, which are themselves direct implementations of the $\mathcal{G}$ inference rules [6, 7].

Our plugin architecture is a restricted form of *tacticals*—functions on tactics—initially designed in the *LCF* family of theorem provers (such as *HOL*) but now pervasive in *Coq*, *Isabelle*, *NuPRL*, *etc.* There is a particular similarity to tactics languages such as *LTac* [1] of *Coq* that allow building tactics libraries that can reflect on the state of the prover and construct proofs from meta-procedures, which are then checked by the *Coq* kernel. However, *Coq* provides no support for reasoning about higher-order abstract syntax, which is our main interest. The *Hybrid* system [2] is one approach for reasoning about *HOAS* in *Coq* by using an intermediate De Bruijn representation; indeed, several of the tactics of the *Schemas* plugin are reminiscent of similar operations in *Hybrid*, but a formal correspondence seems difficult given the difference in nature between *Hybrid* and *Abella*.

Meta-theorems for reasoning about *HOAS*-specified object logics has been the topic of a recent survey article sequence [3, 4] that both identifies a family of essential theorems and compares the performance of *Twelf*, *Beluga*, *Abella*, and *Hybrid*. We believe most of the lemmas and theorems in the associated *ORBI* library can be automatically derived from a simple composition of the tacticals in the *Schemas* plugin. For instance, *admissibility of reflexivity* [4, Theorems 7, 14, 21], *context inversion* [4, Lemma 9], and *completeness* [4, Lemma 22] follow immediately from our `sync` tactic while *relational strengthening* [4, Theorems 15, 20] correspond to `projas`. Other theorems such as *context membership* [4, Lemma 6] require both `inversion` and `unique`, while *transitivity* [4, Theorem 10] requires a sequence of applications of `inversion`. While our plugin does not change the logic of *Abella*, it resolves much of the tediousness of an explicit representations of contexts that was criticized in this survey.

8. Conclusion and Perspectives

We have described an extension to *Abella* with a backwards-compatible and certifying plugin architecture, which we have used to implement regular context relations in a *Schemas* plugin, and have given a preliminary experimental evaluation using existing examples from the *Abella* examples suite.

The main missing feature in this plugin is the ability to reason about *context strengthening* using *subordination*, which is a built-in (trusted) feature of both *Twelf* and *Beluga*. Since *Abella* relies strongly on its logical foundations in the $\mathcal{G}$ logic, the first step would be to give a logical characterization of strengthening and subordination, which is currently an open problem. To an extent strengthening and subordination are not strictly necessary in *Abella* since we can tailor the context relations to fit the theorems, instead of using a common global context for all theorems. Nevertheless, there are instances in the *Abella* examples suite where, for instance, one needs to show that the addition of natural numbers remains commutative even when there are assumptions about λ-terms in the dynamic context. Such lemmas are an easy consequence of subordination—the type of λ-terms is not subordinate to that of numbers—but still currently require manual proofs.

Acknowledgements. This work was partially supported by the INRIA Associated Team grant *RAPT* and by the ERC Advanced Grant *ProofCert*.

References

[1] D. Delahaye. A tactic language for the system Coq. In *Proceedings of the 7th International Conference on Logic for Programming and Automated Reasoning*, volume 1955 of *LNCS*, pages 85–95. Springer, 2000.

[2] A. Felty and A. Momigliano. Hybrid: A definitional two-level approach to reasoning with higher-order abstract syntax. *J. of Automated Reasoning*, 48:43–105, 2012.

[3] A. P. Felty, A. Momigliano, and B. Pientka. The next 700 challenge problems for reasoning with higher-order abstract syntax representations: Part 1—A foundational view, 2014. URL `http://www.site.uottawa.ca/~afelty/dist/FMP-Part1.pdf`.

[4] A. P. Felty, A. Momigliano, and B. Pientka. The next 700 challenge problems for reasoning with higher-order abstract syntax representations: Part 2—A survey, 2014. URL `http://www.site.uottawa.ca/~afelty/dist/FMP-Part2.pdf`.

[5] A. Gacek. The Abella Interactive Theorem Prover (System Description). In *Proceedings of the International Joint Conference on Automated Reasoning*, pages 154–161, 2008.

[6] A. Gacek. *A Framework for Specifying, Prototyping, and Reasoning about Computational Systems*. PhD thesis, University of Minnesota, 2009.

[7] A. Gacek, D. Miller, and G. Nadathur. Nominal abstraction. *Information and Computation*, 209(1):48–73, 2011.

[8] A. Gacek, D. Miller, and G. Nadathur. A two-level logic approach to reasoning about computations. *J. of Automated Reasoning*, 49(2): 241–273, 2012. . URL `http://arxiv.org/abs/0911.2993`.

[9] D. Miller and G. Nadathur. A computational logic approach to syntax and semantics. Presented at the Tenth Symposium of the Mathematical Foundations of Computer Science, IBM Japan, May 1985.

[10] D. Miller and G. Nadathur. *Programming with Higher-Order Logic*. Cambridge University Press, June 2012. .

[11] D. Miller and A. Tiu. A proof theory for generic judgments: An extended abstract. In P. Kolaitis, editor, *18th Symp. on Logic in Computer Science*, pages 118–127. IEEE, June 2003.

[12] A. Nanevski, F. Pfenning, and B. Pientka. Contextual model type theory. *ACM Trans. on Computational Logic*, 9(3):1–49, 2008.

[13] F. Pfenning and C. Elliott. Higher-order abstract syntax. In *Proceedings of the ACM-SIGPLAN Conference on Programming Language Design and Implementation*, pages 199–208. ACM Press, June 1988.

[14] F. Pfenning and C. Schuermann. *Twelf User's Guide*. Carnegie Mellon University, 1.4 edition, 2002.

[15] F. Pfenning and C. Schürmann. System description: Twelf — A meta-logical framework for deductive systems. In H. Ganzinger, editor, *Proceedings of the 16th Conf. on Automated Deduction*, number 1632 in LNAI, pages 202–206, Trento, 1999. Springer.

[16] B. Pientka and J. Dunfield. Beluga: A framework for programming and reasoning with deductive systems (system description). In *Proceedings of the 5th International Joint Conference on Automated Reasoning*, number 6173 in LNCS, pages 15–21, 2010.

[17] X. Qi, A. Gacek, S. Holte, G. Nadathur, and Z. Snow. The Teyjus system – version 2, Mar. 2008. http://teyjus.cs.umn.edu/.

[18] C. Urban, A. M. Pitts, and M. Gabbay. Nominal unification. *Theoretical Computer Science*, 323(1-3):473–497, 2004.

[19] Y. Wang, K. Chaudhuri, A. Gacek, and G. Nadathur. Reasoning about higher-order relational specifications. In T. Schrijvers, editor, *Proceedings of the 15th International Symposium on Princples and Practice of Declarative Programming*, pages 157–168, Madrid, Spain, Sept. 2013. .