Automatically Deriving Schematic Theorems for Dynamic Contexts

Olivier Savary Bélanger
McGill University
osasvary@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 (λTS) [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 that 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 αβγ-conversion. For example, consider the following signature (in λProlog [10]) specifying a data structure for simply typed λ-terms.

\[
\begin{align*}
\text{kind } & \text{ ty } \rightarrow \text{ type. } \\
\text{type } & \text{ i } \rightarrow \text{ ty. } \\
\text{type } & \text{ arr ty ty ty ty. } \\
\text{kind } & \text{ tm ty. } \\
\text{type } & \text{ app tm tm tm tm. } \\
\text{type } & \text{ abs ty tm tm tm. } \\
\end{align*}
\]

The term \(\lambda f::i \rightarrow i. \lambda x:: i. f(fx)\) would be represented as follows.

\[
\begin{align*}
\text{abs } & \text{ (arr i i) (ax. app f (app f x))}\end{align*}
\]

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 (HHOH) [10, sec. 5.2.2] that forms the basis of λProlog 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 \(\Gamma\) 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 λProlog we write it using a relation \(\beta\) with these program clauses.

\[
\begin{align*}
\text{type } & \text{ of } \text{ tm ty ty o. } \\
\text{of } & \text{ (app M N) B } \\
& \text{ of M (arr A B), of N A. } \\
& \text{of (abs A R) (arr A B) } \\
& \beta i \text{ (ax. app u x), for example, then (R x) would be equal to abs i (ax. app u x), which avoids the capture of x. } \\
& \text{Let } \Phi \text{ stand for this pair of defining clauses for } \beta. \text{ The typing judgment } \lambda f::i \rightarrow i. \lambda x:: i. f(fx) : (i \rightarrow i) \rightarrow i \rightarrow i \text{ amounts to showing that the following sequent is derivable: } \Phi \mid \text{ of (abs (arr i i) (ax. app f (app f x))) (arr (arr i i) (arr i i)). } \\
\end{align*}
\]

1 Note: all relations have target type \(\beta\), the type of λProlog formulas.
To prove this sequent, we would need to backchain the second clause in \( \Phi \), which will produce the subgoal:

\[
\forall i. \Phi, \text{ of } f \ (\text{arr i i}) \vdash \\
\text{of } (\text{abs i (Ax. app f (app f x))) (arr i i)}
\]

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

\[
\forall i. \Phi, \text{ of } f \ (\text{arr i i}) \vdash \\
\text{of } (\text{app f (app f x))) i
\]

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 \( \text{of} \) relation has the structure:

\[
\forall i. \Phi, \text{ of } x_i t_i, \ldots, n \text{.}
\]

where for \( i \in 1..n \), the variable \( x_i \) is fresh for \( \Phi \) and for \( (x_j, t_j) \) for every \( j \in 1..i-1 \). The program \( \Phi \) 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 \( \forall \) quantifier [11]), the form of this dynamic portion of the context can be specified as an inductively defined predicate \( \text{ctx} \) as follows.

\[
\text{Define } \text{ctx} : \text{olist} \rightarrow \text{prop} \text{ by }\]

\[
\text{ctx nil ; nabla } x, \text{ctx (of x A :: G) } \triangleq \text{ctx G.}
\]

The type \( \text{olist} \) denotes a list of \( \text{HOHH} \) formulas (built as usual using \( \text{nil} \) and \( ; \)) that represents the context, while the type \( \text{prop} \) 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 \( \text{LP} \) convention of universally closing every clause of the definition over its capitalized free variables. The \( \text{nabla} \) 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 \( \text{ctx} \) in proofs requires a number of essentially administrative inductive theorems for reasoning about the dynamic context (i.e., lists of type \( \text{olist} \)) specified by it. These lemmas amount to unfolding the \( \text{ctx} \) definition to observe the structural properties of its argument, such as that the head of the list is of the form \( \forall x, A \), 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 \( \text{of} \) 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 tactics 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 \( \alpha \beta \gamma \) and a systematic permutation of their nominal constants are equated;\(^3\) and

One particular inductive definition for a focused sequent calculus for \( \text{HOHH} \) 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 \( \text{LP} \) language, so Abella can be used to reason about \( \text{LP} \) 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, \text{stlc.sig}), and the clauses for the declared relations are placed in a corresponding module (here \text{stlc.mod}). The pair of \text{.sig} and \text{.mod} files can be directly executed in \text{LP}, such as using the \text{tjcc} compiler and \text{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 \text{.thm} file (here, \text{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 \( \lambda \)-terms are uniquely determined by the \( \text{of} \) predicate.

The theorem unique is proved by induction on the structure of the first \( \text{HOHH} \) derivation, viz. \( \vdash \{ G \vdash A \} \). This is achieved in Abella by means of the induction tactic that produces this subgoal:

\[
\text{IH : forall G M A B, ctx G \rightarrow (G \vdash A B) } \rightarrow \ (G \vdash M A) \rightarrow A = B.
\]

\[
\text{forall G M A B, ctx G \rightarrow (G \vdash A) } \rightarrow \ (G \vdash M B) \rightarrow A = B.
\]

\(^2\)http://abella-prover.org

\(^3\) This is related to a similar notion from nominal logic [18], but we retain the HOAS representation of terms.
Figure 1. Simply typed \(\lambda\)-calculus in Abella
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 \( \texttt{ctx} \), each producing its own \( \texttt{ctx} \) definition and their associated administrative lemmas. Indeed, Abella even allows for context relations, which are inductive definitions such as \( \texttt{ctx} \) 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 \( \texttt{ctx} \)-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 \( \texttt{case} \) (for case-analysis) and \( \texttt{search} \) (for depth-bounded automated search) are implemented using these core tactics. However, Abella 2.0 lacks a mechanism for defining new tactics like \( \texttt{case} \) and \( \texttt{search} \); 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 top-level command or tactic added by a plugin named \( \texttt{Plug} \) must have the form

\[
\texttt{Plug} \! \langle \texttt{text} \rangle \! \!.\!
\]

where the \( \langle \texttt{text} \rangle \) is arbitrary text that must not contain the token ‘\( \ldots \)’. Abella will scan its list of known plugins for a plugin named \( \texttt{Plug} \), which will then be asked to elaborate the \( \langle \texttt{text} \rangle \) into either top-level 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:

\[
\begin{align*}
\text{module type } \texttt{PLUGIN} & = \ \
\text{sig} & \\
\text{val } \texttt{process_tactic} : \\
& \quad \text{core} : (\text{string } \rightarrow \text{Prover.semant}) \\
& \quad \rightarrow \text{string} \\
& \quad \rightarrow \text{Prover.semant} \\
& \quad \rightarrow \text{unit}
\end{align*}
\]

\(^4\) 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 \( \texttt{PLUGIN} \) has to implement two functions, \( \texttt{process_tactic} \) and \( \texttt{process_top} \), defining its behavior on tactics and top-level commands, respectively. Each function takes a named parameter \( \texttt{core} \), a shallow wrapper around the core Abella functionality which processes the elaborated string produced by the plugin. In particular, this string argument to \( \texttt{core} \) 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 \( \texttt{core} \) function abstractly. The \( \texttt{process_tactic} \) function can additionally reflect on the state of the proper—i.e., the current subgoal that has the type \( \text{Prover.semant} \)—at the point where the corresponding plugin tactical was invoked. However, this function cannot construct new sequents and must instead drive the \( \texttt{core} \) 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 \( \texttt{core} \) function.

To add a new plugin to Abella, it is necessary to add the module implementing \( \texttt{PLUGIN} \) to a global plugins table. This table is stored in the file \( \text{abella.ml} \) 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 \( \text{abella.ml} \) and recompile Abella.

\[
\text{Hashtbl.add plugins "Ctx" (module Schemas : PLUGIN)}::
\]

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 \( \text{ctx} \) 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 \( \text{list} \);
- relates \( n \) \text{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 top-level declaration, Schemas, for declaring such regular context relations. This declaration has the following general form

\[
\text{Schemas } \langle \text{name} \rangle \triangleq \langle \text{clause} \rangle ; \ldots ; \langle \text{clause} \rangle
\]

where each \( \langle \text{clause} \rangle \) has the form:

\[
\exists \ A_1 \ldots A_m \ \text{nabra} \ x_1 \ldots x_n \ (F_1, \ldots, F_k)
\]

where the \( F_i \) are either arbitrary \( \text{HOH} \) formulas built using the variables \( A_1, \ldots, A_m, x_1, \ldots, x_n \) 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 \( \text{exists} \) and \( \text{nabla} \) is fixed and guarantees that every \( x \) is fresh for each \( A \).

As a simple example, here is how the \( \text{ctx} \) definition of Fig. 1 can be written as a schema.

**Schema** \( \text{ctx} \triangleq \text{exists} \ A, \text{nabla} \ x, \ (\text{of} \ x \ A) \).

Using the \( \text{Ctx} \) plugin, we would in fact write it as follows:

\[
\text{Ctx! Schema} \quad \text{ctx} \triangleq \text{exists} \ A, \text{nabla} \ x, \ (\text{of} \ x \ A). \quad !.
\]

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

A more complex example comes from the \( \text{cr.thm} \) file from \( \text{Abella} \)'s example suite that shows how to partition \( \lambda \)-terms into normal and neutral (aka. atomic) forms:

\[
\text{Define} \quad \text{ctxs} : \text{olist} \rightarrow \text{olist} \rightarrow \text{prop} \ \\
\text{ctxs nil nil} ; \text{nabla} \ x, \text{ctxs} \ (\text{term} \ x :: \ L) \ \\
\ (\text{neutral} \ x :: \ K) \triangleq \text{ctxs} \ L \ K.
\]

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

**Schema** \( \text{ctxs} \triangleq \text{nabla} \ x, \ (\text{term} \ x, \text{neutral} \ x) \).

Note that if any of the \( \text{exists} \) or \( \text{nabla} \) bound variables list is empty, the corresponding \( \text{exists} \) or \( \text{nabla} \) 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 \( \text{ctx2} \) definition from \( \text{breduce.thm} \) [19].

\[
\text{Define} \quad \text{ctx2} : \text{olist} \rightarrow \text{olist} \rightarrow \text{prop} \ \\
\text{ctx2 nil nil} ; \text{nabla} \ x, \text{ctx2} \ (\text{bred} \ x \ x :: \ G) \ \\
\ (\text{path} \ x :: \ P :: \ D) \triangleq \text{ctx2} \ G \ D \ \\
\ (\text{ctx2} \ ((\text{pi} \ \lambda u. \ \text{bred} \ N \ u \Rightarrow \text{bred} \ x \ u) :: \ G) \ \\
\ ((\text{pi} \ \lambda q. \ \text{path} \ N \ q \Rightarrow \text{path} \ x \ q) :: \ D) \triangleq \text{ctx2} \ G \ D.
\]

Here is its depiction as a context relation schema.

**Schema** \( \text{ctx2} \triangleq \text{nabla} \ x, \ (\text{bred} \ x \ x, \text{path} \ x \ P) \ \\
\ (\text{exists} \ x, \text{nabla} \ x, \ \\
\ (\text{pi} \ \lambda u. \ \text{bred} \ N \ u \Rightarrow \text{bred} \ x \ u), \ \\
\ (\text{pi} \ \lambda q. \ \text{path} \ N \ q \Rightarrow \text{path} \ x \ q)).
\]

The second clause above has three kinds of variables: existential (\( \theta \)), 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 \( \text{ctx3} \) definition from \( \text{cr.thm} \):\(^7\)

\[
\text{Define} \quad \text{ctx3} : \text{olist} \rightarrow \text{olist} \rightarrow \text{prop} \ \\
\text{ctx3 nil nil nil} ; \text{nabla} \ x, \text{ctx3} \ (\text{term} \ x :: \ L) \ \\
\ (\text{pr1} \ x :: \ K) \ \\
\ (\text{cd1} \ x :: \ \text{notabs} \ x :: \ J) \triangleq \text{ctx3} \ L \ K \ J.
\]

\(^7\)In: examples/lambda-calculus/term-structure/normal.thm

The third argument of the second clause adds two elements to the head. We use the conjunction operator (\( \land \)) of \( \lambda \text{Prolog} \) in the corresponding schema.

**Schema** \( \text{ctxs} = \text{nabla} \ x, \ (\text{trm} \ x, \text{pr1} \ x \ x, \text{cd1} \ x \ x \ & \text{notabs} \ x) \).

It should be noted that the support for reasoning about \( \& \) is currently rather primitive in \( \text{Abella} \). While the above declaration is accepted by the plugin, the automatically derived theorems currently are not accepted by \( \text{Abella} \). (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 \( \text{Abella} \):

- Multiple schemas may not be mutually recursive.
- Schemas can only relate dynamic contexts (\( \text{olist} \)), 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 \( \text{Abella} \) 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 \( \text{Schemas} \) 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 \( \text{ctx} \) predicate of Fig. 1.

**Schema** \( \text{ctx} \triangleq \text{exists} \ A, \text{nabla} \ x, \ (\text{of} \ x \ A) \).

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 \( \text{Abella} \), these properties are easily defined, but because \( \text{Abella} \) 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 \( \text{ty} \) and \( x \) has type \( \text{tm} \). Thus, we would need the following instances of the \( \text{name} \) and \( \text{fresh} \) predicates:

\[
\text{Define} \quad \text{name.tm} : \text{tm} \rightarrow \text{prop} \ \\
\text{name.x} : \text{x} \rightarrow \text{prop} \ \\
\text{Define} \quad \text{fresh.tm.in_ty} : \text{tm} \rightarrow \text{ty} \rightarrow \text{prop} \ \\
\text{fresh.x.in_A} : \text{x} \rightarrow \text{A} \rightarrow \text{prop}.
\]

As a side-effect of processing the \( \text{Schema} \) declaration above, the \( \text{Schemas} \) plugin automatically adds these definitions. Precisely, a \( \text{name} \) predicate is generated for each type of nominally quantified variable in any clause of a schema, and a \( \text{fresh} \) 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 \( \text{Schemas} \) plugin keeps track of all such administrative definitions to prevent duplicates, but it may add definitions that are not ever used. This is because \( \text{Abella} \) only allows inductive definitions at the top-level, not during a proof, and no plugin is allowed to "rewind" the state of \( \text{Abella} \) to retroactively add definitions.

For each type of a nominally quantified variable, the \( \text{Schemas} \) plugin also generates a \( \text{prune} \) lemma, as explained in Section 2. Here is the version for \( \text{tm} \):

\[
\text{In: examples/lambda-calculus/term-structure/normal.thm}
\]
Theorem member_prune_tm : 
  for all G E, nabla (x:tm), 
  member G (E x) \rightarrow exists F, E = \lambda x. 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 
respective schemas 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 G₁ ... Gₙ, where ctx is 
produced by a Schema declaration, and the other is of the form 
member E G, for some i ∈ 1..n. The result of the tactical is that E 
must be one of the formulas that occur in the i-th 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)  
  \land member (of X A) G  
  \land fresh_tm_in_ty X A)  
\lor (exists A V X, (E = eval X V)  
  \land member (of X A) G  
  \land fresh_tm_in_ty X A  
  \land fresh_tm_in_ty 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 assert⁵ 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

⁵The 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 G logic.

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 n₁ that is raised over both n₁ and n₂. The first 
clause of the schema does not match because bred n₁ n₂ does not 
equivalently unify with bred n₁ n₂. In this case, the additional 
assumption H₄ would suffice to show that n₁ n₂ does not actually 
contain n₁, i.e., that it has a vacuous \lambda-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:
In this case, if we are in a subgoal with:

\[ H_1 : \text{ctx} \triangleq \exists A, \text{nabla} x, (\text{of} \ x \ A). \]

then it must be that \( A \) and \( B \) are equal, since there is only one clause of \( \text{ctx} \) that could have introduced any member of the form \( x \in C \) into \( G \). This is achieved by the tactical application unique \( H_1 \text{ H2 H3} \), which has the side effect of unifying the terms \( A \) and \( B \).

While easily explained, this tactic 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 \( \in \) \( A \) and \( \in \) \( B \) 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 \( \sigma \) would use a unary context schema about \( \sigma \), while those about evaluation using \( \text{eval} \) would use a unary schema for \( \text{eval} \). 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 \texttt{projas} tactical applies to an assumption:

\[ H_1 : \text{rel1} G_1 \ldots G_n \]

where \( \text{rel1} \) is a schematic context relation. The tactical application

\texttt{projas (rel2 D_1 \ldots D_m) H1}

where each \( D_j \) is either one of the \( G_i \), or is a new evengiablear, has the effect of adding the assumption

\[ H_2 : \text{rel2} D_1 \ldots D_m \]

to the goal, when justified.

This tactical application is interpreted into a general *projection lemma* that has the following form. Let \( D_{0(1)}, \ldots, D_{0(k)} \) be the evengiablear s that are distinct from all the \( G_i \). Then, the following lemma is proved by induction:

\[ \text{forall} G_1 \ldots G_n, \text{rel1} G_1 \ldots G_n \Rightarrow \exists D_{0(1)} \ldots D_{0(k)}, \text{rel2} D_1 \ldots D_m. \]

This proof proceeds by induction on the definition of \texttt{rel1}, but is rather straightforward. Of course, if all the \( D_j \) are distinct from the \( G_i \), then this tactic is useless. Like the other tacticals, \texttt{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:

\[ \text{Schema \ rel1} \triangleq (1, 1). \]
\[ \text{Schema \ rel2} \triangleq (1, 1); (1, 1). \]

where \( i \) is an atomic \texttt{HODH} formula of type \( o \). Clearly,

\[ \text{forall \ G, rel1 G \rightarrow rel2 G.} \]

is provable. However, as no single clause of \texttt{rel2} matches the non-trivial clause of \texttt{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. Our initial experiments are promising. For instance, using the Schemas plugin to rewrite the \texttt{breduce} example from [19] removes over 40% of the lines of code from the file \texttt{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.

<table>
<thead>
<tr>
<th>File</th>
<th># schemas</th>
<th># lemmas derived</th>
<th>LOC removed</th>
<th>% removed</th>
</tr>
</thead>
<tbody>
<tr>
<td>breduce</td>
<td>3</td>
<td>11</td>
<td>124</td>
<td>42</td>
</tr>
<tr>
<td>copy</td>
<td>1</td>
<td>4</td>
<td>28</td>
<td>43</td>
</tr>
<tr>
<td>cr</td>
<td>1</td>
<td>3</td>
<td>32</td>
<td>19</td>
</tr>
<tr>
<td>type_uniq</td>
<td>1</td>
<td>3</td>
<td>27</td>
<td>63</td>
</tr>
</tbody>
</table>

**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 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

---

\footnote{See: \url{http://abella-prover.org/schemas}}
We have described an extension to Abella and Hybrid Coq \cite{1} of LTac numbers—but still currently require manual proofs. Subordination—the type of the dynamic context. Such lemmas are an easy consequence of commutative even when there are assumptions about $\lambda$-terms. Nevertheless, since we can tailor the context relations to fit the theorems, instead of using a common global context for all theorems, strongly on its logical foundations in the in (trusted) feature of both context strengthening and inversion requires a sequence of applications of inversion. Other theorems such as context membership \cite[Theorems 15, 20]{4} correspond to relational strengthening, while \cite[Theorem 10]{4} re-quires 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 $G$ 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 $\lambda$-terms in the dynamic context. Such lemmas are an easy consequence of subordination—the type of $\lambda$-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

\begin{thebibliography}{1}
\end{thebibliography}