The ALF proof editor and its proof engine *

Lena Magnusson and Bengt Nordström

University of Göteborg/Chalmers, S-412 96 Göteborg, Sweden

Abstract. Alf is an interactive proof editor. It is based on the idea that to prove a mathematical theorem is to build a proof object for the theorem. The proof object is directly manipulated on the screen, different manipulations correspond to different steps in the proof. The language we use is Martin-Löf's monomorphic type theory. This is a small functional programming language with dependent types. The language is open in the sense that it is easy to introduce new inductively defined sets. A proof is represented as a mathematical object and a proposition is identified with the set of its proof objects. The basic part of the proof editor can be seen as a proof engine with two basic commands, one which builds an object by replacing a placeholder in an object by a new object, and another one which deletes a part of an object by replacing a sub-object by a placeholder. We show that the validity of the incomplete object is preserved by admissible insertions and deletions.

1 Background

During the years we have learned that there is no such thing as "the logic of programming". Different kinds of programs require different kind of reasoning. Programs are manipulating different kinds of objects, and it would be very awkward to code these objects into a fixed set of objects. Objects have their own logic, it is for instance very different to reason about ongoing processes and fixed objects like natural numbers and lists. We also need a different kind of logic when we are interested in computational aspects of a program (like complexity and storage requirements). We don't think we will ever find *the* logic of programming.

The idea behind a logical framework is to have a flexible formal logic, in which it is possible to introduce new kinds of objects, including objects for proofs. The logical framework we are using is Martin-Löf's monomorphic type theory, which can be seen as a small functional language with dependent types. We express problems as types and solutions (proofs) as programs.

The fundamental notion of proof is a process which leads to a conviction of something to hold (an assertion, or equivalently, a judgement). You have a series of steps, in each step you make an assertion which holds because earlier assertions have been made. A proof object should be a mathematical object which represents this proof process. The proof object must be derivable from

* This research has been done within the ESPRIT Basic Research Action "Types for Proofs and Programs". It has been paid by NUTEK, Chalmers and the University of Göteborg.

the proof process. But we need something more: If a proof object represents a proof then it must be possible to compute a proof process from the proof object.

The traditional way of using a computer for interactive proof checking is to formalize the proof process and then letting the computer check each step. The user types in commands in some imperative language and the effect of executing a command is to update some internal data base which represents assertions being made. We call this indirect editing. It is like we cannot see the objects being built, instead we have to make experiments on it to see what we have.

To directly build something with a computer is to have an impression that the objects which are built (and changed) are directly manipulated on the screen using the keyboard and the mouse. It is like we have a hand (represented by the cursor) on the screen to select parts and to grasp for different tools which can manipulate the object. A change of the object is immediately shown on the screen.

The idea we use is to use a proof object as a true representative of a proof. The process of proving the proposition A is represented by the process of building a proof object of A. There is a close connection between the individual steps in proving A and the steps to build a proof object of A. For instance the act of applying a rule is done by building an application of a constant, to assume that a proposition A holds is to make an abstraction of a variable of the type A and to refer to an assumption is to use the corresponding variable. To delete a part of a proof object corresponds to regretting some eaarlier steps in the proof. Notice that the deleted steps don't have to be the last steps in the derivation, by moving the pointer around in the proof object it is possible to undo any of the preceeding steps without altering the effect of following steps. However, the deletion of a sub-object may cause the deletion of other parts which are depending on this.

We are interested in an interactive direct proof checker. So if we represent the proof process by the process of building a proof object it must be possible to deal with *incomplete* proof objects, i.e. proof objects which represents incomplete proofs.

The proof editor we are using can be seen as an interactive structure-oriented editor for Martin-Löf's monomorphic type theory. An object which has been created by the editor is always meaningful (well typed), which means that the object really represents a proof. Before we explain how a partial proof is represented, we will explain how a complete proof is represented. To do this we need to explain Martin-Löf's monomorphic type theory.

2 Martin-Löf's type theory

There are four judgement forms in type theory:

- A *type*. We know that A is a type when we know what it means to be an object in A.
- $A = B$. Two types are equal when they have the same objects, so an object in A must be an object of B and conversely. Identical objects in A must also be identical in B and vice versa.

– $a \in A$. a is an object in A.
– $a = b \in A$. a and b are identical objects in A.

These judgements are decidable.

In general, a judgement may depend on a context, i.e. a list of assumptions. For instance, the general form of the third judgement is therefore:

$$a \in A \quad [x_1 \in A_1, \ldots, x_n \in A_n]$$

where a and A may depend on $x_1, \ldots, x_n$ and for $j \leq n$, A_j may depend on $x_1, \ldots, x_{j-1}$. Notice that the order of the assumptions is in general important, since the type of one assumption may depend on earlier assumptions.

2.1 How to form types

The type structure is very simple, there are two ways of forming ground types and one way of forming function types. We will use the notation $b\{x{:=}a\}$ for the expression obtained by substituting the expression a for all free occurrences of the variable x in the expression b.

– Set is a type. This is the type whose objects are (inductively defined) sets.

  Set formation

$$\mathsf{Set}\ \mathit{type}$$

– If $A \in \mathsf{Set}$, i.e. if A is a set, then $El(A)$ is a type. The objects in this type are the elements of the set A. We will write A instead of $El(A)$, since it will always be clear from the context whether we mean A as a set (i.e. as an object in Set) or as a type.

  El-formation

$$\frac{A \in \mathsf{Set}}{El(A)\ \mathit{type}}$$

– If A is a type and B is a family of types for $x \in A$ then $(x \in A)B$ is the type which contains functions from A to B as objects. All free occurrences of x in B become bound in $(x \in A)B$.

  Fun formation

$$\frac{A\ \mathit{type} \qquad B\ \mathit{type}\ \ [x \in A]}{(x \in A)B\ \mathit{type}}$$

  To know that an object c is in the type $(x \in A)B$ means that we know that when we apply it to an object a in A we get an object $c(a)$ in $B\{x{:=}a\}$ and that we get identical objects in $B\{x{:=}a_1\}$ when we apply it to identical objects a_1 and a_2 in A.

2.2 How to form objects in a type

Objects in a type are formed from constants and variables using application and abstraction. We already mentioned how to apply a function to an object:

Application

$$\frac{c \in (x \in A)B \qquad a \in A}{c(a) \in B\{x{:=}a\}}$$

Functions can be formed by abstraction, if $b \in B$ under the assumption that $x \in A$ then $[x]b$ is an object in $(x \in A)B$. All free ocurrences of x in b become bound in $[x]b$.

Abstraction

$$\frac{b \in B \quad [x \in A]}{[x]b \in (x \in A)B}$$

The abstraction is explained by the ordinary β-rule which defines what it means to apply an abstraction to an object in A.

β – rule

$$\frac{a \in A \qquad b \in B \quad [x \in A]}{([x]b)(a) = b\{x{:=}a\} \in B\{x{:=}a\}}$$

The traditional η-, α- and ξ-rules can be justified.

We will sometimes use the notation $(A)B$ or $A \to B$ when B does not contain any free occurrences of x. We will write $(x_1 \in A_1, \ldots, x_n \in A_n)B$ instead of $(x_1 \in A_1) \ldots (x_n \in A_n)B$ and $b(a_1, \ldots, a_n)$ instead of $b(a_1) \ldots (a_n)$ in order to increase the readability. Similarly, we will write $[x_1] \ldots [x_n]e$ as $[x_1, \ldots, x_n]e$.

An object is *saturated* if it is not a function, i.e. if its type is Set or $El(A)$, for $A \in \mathsf{Set}$. The *arity* of an object is the number of arguments it can be applied to in order for the result to be saturated. It is an important property that a well-typed object has a unique arity.

2.3 Experimental additions to the language

In order to make type theory more similar to a standard functional language we have extended it in various ways:

explicit substitution. This has been suggested by Martin-Löf in various lectures in Göteborg, Båstad and Leiden during 1993. The syntax we use is

$$d\{x{:=}e\}$$

where d and e are well-typed expressions and x is a variable. The expression can be read as d where x is e. A detailed description can be found in Alvaro Tasistro's licentiate thesis [**?**].

case-expressions We have implemented a version of case-expressions. The system generates an exhaustive, non-overlapping list of patterns following ideas from Thierry Coquand [**?**]. In this implementation, case-expressions may only occur on the outer level of a definiens, they are thus not allowed inside abstractions or applications.

named contexts In order to have a weak notion of abstract data type, we have given the user a possibility to declare constants in an explicit (possibly named) context. The licenciate thesis by Gustavo Betarte [**?**] reports on an experiment with this facility.

2.4 Definitions

Most of the generality and strength of the language comes from the possibilities of introducing new constants. It is in this way that we can introduce the usual mathematical objects like natural numbers, integers, functions, tuples etc. It is also possible to introduce more complicated inductive sets like sets for proof objects.

A distinction is made between primitive and defined constants. The value of a *primitive constant* is the constant itself. So the constant has only a type, it doesn't have a definition. It gets its meaning in other ways (outside the theory). Such a constant is also called a constructor. Examples of primitive constants are N, s and $\mathsf{0}$, they can be introduced by the following declarations:

$$\begin{aligned} \mathsf{N} &\in \mathsf{Set} \\ \mathsf{s} &\in \mathsf{N} \to \mathsf{N} \\ \mathsf{0} &\in \mathsf{N} \end{aligned}$$

A *defined constant* is defined in terms of other objects. When we apply a defined constant to all its arguments in an empty context, e.g. $c(e_1, \ldots, e_n)$, then we get an expression which is a definiendum, i.e. an expression which computes in one step to its definiens (which is a well-typed object).

A defined constant can either be explicitly or implicitly defined. We declare an *explicitly defined constant* c by giving it as an abbreviation of a well-typed object:

$$c = a \in A$$

The constant c is a definiendum in itself, not only when it is applied to its arguments. For instance we can make the following explicit definitions:

$$\begin{aligned} 1 &= \mathsf{s}(\mathsf{0})) \in \mathsf{N} \\ I_N &= [x]x \in \mathsf{N} \to \mathsf{N} \\ I &= [A][x]x \in (A \in \mathsf{Set}, A)A \end{aligned}$$

The last example is the monomorphic identity function which when applied to an arbitrary set A yields the identity function on A.

It is easy to check whether an explicit definition is correct, you just check that the definiens is an object in the correct type.

We declare an *implicitly defined constant* by showing what definiens it has when we apply it to its arguments. This is done by pattern-matching and the

definition is sometimes recursive. Here are some examples:

$$
\begin{aligned}
+ &\in \mathsf{N} \to \mathsf{N} \to \mathsf{N} \\
&\quad +(\mathsf{0}, y) = y \\
&\quad +(\mathsf{s}(x), y) = \mathsf{s}(+(x, y)) \\
\mathsf{natrec} &\in \mathsf{N} \to (\mathsf{N} \to \mathsf{N} \to \mathsf{N}) \to \mathsf{N} \to \mathsf{N} \\
&\quad \mathsf{natrec}(d, e, \mathsf{0}) = d \\
&\quad \mathsf{natrec}(d, e, \mathsf{s}(a)) = e(a, \mathsf{natrec}(d, e, a))
\end{aligned}
$$

The last example is a specialized version of the primitive recursion operator.

Whether a definition of this kind is meaningful can in general only be checked outside the theory. We must be sure that all well-typed expressions of the form $c(e_1, \ldots, e_n)$ is a definiendum with a unique well-typed definiens. The system now generates a complete set of nonoverlapping patterns, but there is no check that the definition is well-founded. We could syntactically restrict the definiens to guarantee well-foundedness, but we know that these kind of restrictions will be too limited. However, this is only a poor excuse that some limited check has not been implemented.

3 The representation of proofs, theories, theorems, derived rules etc.

We are representing proofs as mathematical objects, the type of a proof object represents the proposition which is the conclusion of the proof. Variables are used as names of assumptions and constants are used as rules. To apply a rule to a number of subproofs is done by applying a constant to the corresponding subproof objects.

A theory is presented by a list of typings and definitions of constants. When we read the constant as a name of a rule, then a primitive constant is usually a formation or introduction rule, an implicitly defined constant is an elimination rule (with the contraction rule expressed as the step from the definiendum to the definiens) and finally, an explicitly defined constant is a lemma or derived rule. As an example of this, consider the definition of conjunction.

The formation rule for conjunction expresses that $A \,\&\, B$ is a proposition if A and B are propositions:

$$\frac{A \ prop \qquad B \ prop}{A \,\&\, B \ prop}$$

We express this by introducing the primitive constant $\&$ by the following typing:

$$\& \ \in (\mathsf{Set},\ \mathsf{Set})\mathsf{Set}$$

We use the type of sets to represent the type of propositions. A canonical proof of the problem $A \,\&\, B$ is on the form $\&\mathbf{I}(a, b)$, where a is a proof of A and b a proof of B. This reflects the explanation that a proof of $A \,\&\, B$ consists of a proof

of A and a proof of B. If we are in a context where A and B are propositions we can define $A \,\&\, B$ by introducing the primitive constant

$$\&\mathbf{I} \in (A, B)A \,\&\, B$$

Another notation for this is:

$$\frac{A \qquad B}{A \,\&\, B}$$

This is the introduction rule for conjunction. If we are in an empty context we must declare the parameters A and B explicitly:

$$\&\mathbf{I} \in (A{\in}\mathsf{Set},\ B{\in}\mathsf{Set},\ A,\ B)A \,\&\, B$$

The elimination rules for conjunction

$$\frac{A \,\&\, B}{A} \qquad \frac{A \,\&\, B}{B}$$

are expressed by introducing the implicitly defined constants $\&\mathbf{E}_l$ and $\&\mathbf{E}_l$ by the following declarations:

$$\begin{aligned}
\&\mathbf{E}_l &\in (A{\in}\mathsf{Set},\ B{\in}\mathsf{Set},\ A \,\&\, B)A \\
&\quad \&\mathbf{E}_l(A, B, \&\mathbf{I}(a, b)) = a \\
\&\mathbf{E}_r &\in (A{\in}\mathsf{Set},\ B{\in}\mathsf{Set},\ A \,\&\, B)B \\
&\quad \&\mathbf{E}_r(A, B, \&\mathbf{I}(a, b)) = b
\end{aligned}$$

The equalities are the contraction rules for $\&$ and these are essential for the correctness of the elimination rule. Since all proofs of $A \,\&\, B$ is equal to a proof on the form $\&\mathbf{I}(a, b)$, where a is a proof of A and b a proof of B, we know from the contraction rule that we get a proof of A if we apply $\&\mathbf{E}_r$ to an arbitrary proof of $A \,\&\, B$, and similarly for $\&\mathbf{E}_l$.

Notice the difference with the Edinburgh LF encoding of logic, where the elimination rules (like all rules) are expressed as *primitive* constants. This means that in LF the logical constants are not inductively defined by their introduction rules and hence does not reflect Heyting's explanation. He defines a logical constant by giving their introduction rules and the elimination rules are just consequences of this definition.

To summarize, we have the following declarations and definitions for conjunction. We use the symbol $\downarrow$ in front of arguments which are not printed, e.g. the term $\&\mathbf{I}(A,\ B,\ a,\ b)$ will be written $\&\mathbf{I}(a,\ b)$.

$$\begin{aligned}
\& &\in (\mathsf{Set},\ \mathsf{Set})\mathsf{Set} \\
&\quad \&\mathbf{I} \in (\downarrow A{\in}\mathsf{Set},\ \downarrow B{\in}\mathsf{Set},\ A,\ B)A \,\&\, B \\
\&\mathbf{E}_l &\in (\downarrow A{\in}\mathsf{Set},\ \downarrow B{\in}\mathsf{Set},\ A \,\&\, B)A \\
&\quad \&\mathbf{E}_l(\&\mathbf{I}(a, b)) = a \\
\&\mathbf{E}_r &\in (\downarrow A{\in}\mathsf{Set},\ \downarrow B{\in}\mathsf{Set},\ A \,\&\, B)B \\
&\quad \&\mathbf{E}_r(\&\mathbf{I}(a, b)) = b
\end{aligned}$$

4 Representation of incomplete objects

When we are proving a proposition A in a theory then we are building a proof object of type A in an environment consisting of a list of declaration of constants. This is presented on the screen by having two windows, a theory window containing declarations of constants and a scratch area containing objects being edited.

The objects which are being built in the scratch area are always correct relative to the current theory.

4.1 Editing objects

When we are making a top-down proof of a proposition A, then we try to reduce the problem A to some subproblems $B_1, \ldots, B_n$ by using a rule c which takes proofs of $B_1, \ldots, B_n$ to a proof of A. Then we continue by proving $B_1, \ldots, B_n$. For instance, we can reduce the problem A to the two problems $C \supset A$ and C by using modus ponens. In this way we can continue until we have only axioms and assumptions left. This process corresponds exactly to how we can build a mathematical object from the outside and in. Suppose that we want to build an object like

$$f(g_1(a_1, a_2), g_2(b)).$$

Then we start from its outer form to build $f(?_1, ?_2)$, where $?_1$ and $?_1$ are placeholders for not yet filled-in objects, then continue to fill in g_1 or g_2 etc. This means that the type or the problem to solve is constant while the solution to it is edited. It is an important property of the formal system that it is possible to compute the expected type of the placeholders. It is because of this that we can look at the editing operations as a way of decomposing a problem into subproblems.

Let's see what kind of structure we need to represent incomplete objects. We will first introduce placeholders $?_1, \ldots ?_n$ to be used for parts of the objects which are to be filled in. The expression

$$? \in A$$

expresses a state of an ongoing process of finding an object in the type A. We say that the expected type of ? is A. Objects are built up from variables and constants using application and abstraction. Therefore there are four ways of refining a placeholder:

- The placeholder is replaced by a constant c. This is correct if the type of c is equal to A.
- The placeholder is replaced by a variable x. The type of x must be equal to A. But we cannot replace a placeholder with any variable of the correct type, the variable must have been abstracted earlier. We keep track of this by associating not only a type but also a *local context* to each place holder. The placeholders stand for open expressions and their local context expresses what variables the substituted expression may depend on. The general form of the local context is $x_1 \in A_1, \ldots, x_n \in A_n$, where A_i is a type which may depend on the variables $x_1, \ldots x_{i-1}$.
- The placeholder is replaced by an abstraction $[x]?_1$. We must have that

  $$[x]?_1 \in A$$

  which holds if A is equal to a functional type $(y \in B)C$. The type of the variable x must be B and we must keep track of the fact that $?_1$ may be substituted by an expression which may depend on the variable x. So the local context of $?_1$ must be the local context of ? extended with the typing $x \in B$. So after this refinement we have that

  $$?_1 \in C\{y := x\}$$

  and $?_1$ has a local context which contains $x \in B$. This corresponds to making a new assumption, when we are constructing a proof. We reduce the general problem $(y \in B)C$ to the problem $C\{y := x\}$ under the assumption that $x \in B$. The assumed object x can be used to construct a solution to C, i.e. we may use the knowledge that we have a solution to the problem B when we are constructing a solution to the problem C.
  Notice that the placeholder will in general be replaced by an open term, this is a motivation for having open terms as first-class objects.

– Finally, the placeholder can be replaced by an application $c(?_1, \ldots ?_n)$ where c is a constant, or $x(?_1, \ldots ?_n)$, where x is a variable. In the case that we have a constant, we must have that $c(?_1, \ldots ?_n) \in A$, which holds if the type of the constant c is equal to $(x_1 \in A_1, \ldots, x_n \in A_n)B$ and $?_1 \in A_1, ?_2 \in A_2\{x_1 := ?_1\}, \ldots, x_n \in A_n\{x_1 :=?_1, \ldots, x_{n-1} :=?_{n-1}\}$ and

$$B\{x_1 :=?_1, \ldots, x_{n-1} :=?_{n-1}\} = A$$

So, we have reduced the problem A to the subproblems A_1, $A_2\{x_1 :=?_1\}, \ldots,$ $A_n\{x_1 :=?_1, \ldots, x_{n-1}\}$ and further refinements must satisfy the constraint $B\{x_1 :=?_1, \ldots, x_{n-1} :=?_{n-1}\} = A$. The number n of new placeholders can be computed from the arity of the constant c and the expected arity of the placeholder. As an example, if we start with $? \in A$ and A is not a function type and if we apply the constant c of type $(x \in B)C$, then the new term will be

$$c(?_1) \in A$$

where the new placeholder $?_1$ must have the type B (since all arguments to c must have that type) and furthermore the type of $c(?_1)$ must be equal to A, i.e. the following equality must hold:

$$C\{x :=?_1\} \equiv A.$$

As will be described later, these kind of constraints will in general be simplified by the system. So, the editing step from $? \in A$ to $c(?_1) \in A$ is correct if $?_1 \in B$ and $C\{x :=?_1\} \equiv A$. This operation corresponds to applying a rule when we are constructing a proof. The rule c reduces the problem A to the problem B.

To summarize, an ongoing proof process (or an ongoing construction of a mathematical object) is represented by:

– A list of definitions, in which the definiens is a partial object.
– A list of placeholders $?_1, \ldots, ?_n$, each with an expected type and a local context (i.e. a list of typed variables).
– A set of constraints, which is a set of definitional equalities containing the placeholders. All refinements of the placeholders must satisfy the constraints.

The proof engine, which is the abstract machine representing an ongoing proof process (or an ongoing construction of a mathematical object) has two parts, the theory (which is a list of constant declarations) and the scratch area. Objects are built up in the scratch area and moved to the theory part when they are completed. There are two basic operations which are used to manipulate the scratch area. The *insertion* command replaces a placeholder by a new (possible incomplete) object and the *deletion* command replaces a sub-object by a placeholder. In this paper we will concentrate on describing these operations. But before we do this we will go through a small example how the system actually works.

An example in ALF

As a small example of a proof we prove one half of the distributivity of conjunction over disjunction in propositional logic. A variant of this example was used already by Gentzen [**?**] to motivate natural deduction. We have already defined the constants for conjunction; we repeat them here together with the corresponding definitions for disjunction as given to ALF. At present, ALF does not support infix operators, so we will write propositions with the logical operations in prefix form as in $\mathsf{And}(A, \mathsf{Or}(B, C))$.

This picture and the ones to follow in this section are screen dumps from a session with ALF. The vertical arrows in front of some arguments indicate that these arguments should be suppressed, as is done in the contraction rules. These arguments can typically be inferred by the system, so the user may completely ignore them.

The theorem we want to prove is

$$(A, B, C \in \mathsf{Set}; \mathsf{And}(A, \mathsf{Or}(B, C)))\mathsf{Or}(\mathsf{And}(A, B), \mathsf{And}(A, C))$$

This is a function type. An object of this type expects four arguments: three sets A, B and C and a proof of $\mathsf{And}(A, \mathsf{Or}(B, C))$. Given this input, the function must produce an element of (a proof of) the set (the proposition)

$$\mathsf{Or}(\mathsf{And}(A, B), \mathsf{And}(A, C))$$

. We shall do this in two different ways:

1. By defining a function *distr1* as an abbreviation, or in ALF terminology, an explicit constant. This will be done by using the elimination rules to analyze the last argument and then the introduction rules to build up the result.

2. By defining a function *distr2* as an implicit constant, where the argument is analyzed directly by pattern matching and the elimination rules are not needed.

Using the first method, we instruct the system that we intend to define an explicit constant with the name *distr1*. The system responds with a template for the definition:

The definition has two placeholders, one for the definiens and one for its type. We proceed by filling in the type:

The definiens is built top-down. The first step is to form an abstraction

where the $?_e$ should have type $\mathsf{Or}(\mathsf{And}(A,B),\mathsf{And}(A,C))$. To construct an element in this type we must analyze the second projection of h using the *OrE* rule:

The three first arguments are inferred by the system and not displayed. We have to fill in the remaining two arguments. The type of the fourth argument is $(B)\mathsf{Or}(\mathsf{And}(A,B),\mathsf{And}(A,C))$. An object in this type is again a function, which gets a proof of B as input. Using this input, it is easy to construct the result using the left introduction rule for Or. Similarly, the second argument can be completed using the right Or-introduction rule, giving the complete proof

The omitted steps in this proof all consist of simple direct manipulations with the mouse; nothing is typed on the keyboard (except the name *distr1*).

In the second method, using pattern matching instead of elimination rules, the initial situation in building the proof is the following:

Here we can now ask the system to analyze the possible forms of an argument. We perform this pattern matching on the last argument, giving

There is only one constructor for And, so h must evaluate to $\mathsf{AndI}(h_1, h_2)$, where $h_1 \in A$ and $h_2 \in \mathsf{Or}(B, C)$. We proceed by analyzing h_2, which must reduce to one of two possible forms:

So, the pattern matching separates the task of defining *distr2* into two mutually exhaustive cases. In both cases it is immediate to construct the right hand side using constructors and the variables on the left hand side:

The proof is complete.

The pattern matching approach is the outcome of experiments with the ALF system [**?**] and is not present in type theory proper as presented by Martin-Löf. In fact, the two proofs presented here are just instances of two different disciplines. In the first, one defines for each set once and for all an elimination rule, capturing proof by structural induction over elements of the set.[2] These elimination rules are defined as implicit constants and are justified by reflection on the definition of the set. Having done this, all proofs involving elements of this set are defined as explicit constants. In the second discipline, only the set with its introduction rules is defined at the outset and later proofs are done by pattern-matching, involving a reflection on the arguments specially adapted to the particular proposition one wants to prove. Interestingly, the two disciplines are not proof-theoretically equivalent. One can exhibit propositions that can be

[2] The elimination rules for And and Or given here can be slightly generalized, using dependent sets. In this generalization, the two And-elimination rules are replaced by just one rule.

proved by pattern-matching but are false in certain models of type theory with the standard elimination rules [**?**]. It is presently a research topic to gain a better understanding of this phenomenon.

5 The scratch area

We will now present how partial objects are represented and manipulated by insertion and deletion in the scratch area. The representation of partial objects is designed in such a way that the two operations insert and delete becomes mainly a matter of type checking the partial objects. The type checking algorithm produces a list of equations, which in the case of complete objects becomes a decision procedure since the convertibility of the equations is decidable. In the case of an incomplete object, the problem of solving the list of equations becomes a unification problem, since place holders may occur in the equations. Therefore, the *same* algorithm is used for type checking complete objects as well as incomplete ones.

The direct manipulation of insertion in an object can be achieved by using a general user interface for theorem provers ([**?**]) which has been adopted to proof editors such as Coq ([**?**]) and Isabelle ([**?**]). The insertion operation corresponds to the refinement command in these provers which can be called by an interface instead of a user. However, since the delete operation we refer to may resume in a completely *new state* of the proof engine, it must be supported by the proof engine and is not simply a matter of interface. The desire for this operation is expressed in [**?**] where it is referred to as local undo.

We will define a valid scratch area to be such that *the partial objects in the scratch area are instantiated to complete, type correct objects exactly when all remaining placeholders have type correct instantiations which satisfies the constraints*. Then we will show that the operations on the scratch area preserves the validity. This idea of inserting and deleting subobjects can be applied to other formalisms with explicit proof objects.

To summarize, what we need to be able to do in the scratch area are

1. Represent partial objects.
2. Replace a placeholder by a partial object (*insertion*).
3. Replace a (partial) object by a placeholder (*deletion*).

which requires that we can do the following

1. Type check partial objects. We use a type checking algorithm which reduces the problem of checking if an object has a given type, to the problem of checking if a set of equations holds. This idea was first used in Automath [**?**]. If the equations only contain complete objects, we can check the equalities by convertibility, and we have a decision procedure. If the objects are partial (i.e. contain placeholders) we have instead a unification problem, since we want to find instantiations of all placeholders in the equations, such that

the equations hold. This unification problem can not always be solved, [3] since we may have higher order placeholders and placeholders as arguments to functions defined by pattern matching, which requires an instantiation of the placeholder before the matching can be performed. Therefore, the unification may leave unsolved equations as constraints restricting further instantiations.

2. Check if a replacement of a placeholder by an object is type correct relative to the expected type and the constraints, and reject the replacement if it leads to an (detectably) inconsistent set of constraints.
3. Automatic instantiations caused by the content of the deleted subobject must be withdrawn. This requires that we separate instantiations caused by unification from the user instantiations. If we only had the insertion operation we could perform the unification instantiations on the object directly.

We will start by giving an example which illustrates why we cannot define the validity of a scratch area as an "absolute property", rather as a property which depends on future instantiations. The example shows a partial object, in which either the constraint is satisfied but then it is not possible to give instantiations to the remaining placeholders, or the placeholders are instantiated but then the constraint is not satisfied.

Ex. Assume we define a set $\mathsf{Seq}(A, n)$ (denoting a sequence of type A and length n) with the two constructors atom and cons

$\mathsf{Seq} \in (\mathsf{Set}; \mathsf{N})\mathsf{Set}$

$\mathsf{atom} \in (A \in \mathsf{Set}; a \in A)\mathsf{Seq}(A, 1)$

$\mathsf{cons} \in (A \in \mathsf{Set}; a \in A; n \in \mathsf{N}; l \in \mathsf{Seq}(A, n))\mathsf{Seq}(A, s(n))$

and an append function with the type

$\mathsf{append} \in (A \in \mathsf{Set}; n, m \in \mathsf{N}; l_1 \in \mathsf{Seq}(A, n); l_2 \in \mathsf{Seq}(A, m))\mathsf{Seq}(A, n + m)$.

Assume we want to find an element in $\mathsf{Seq}(\mathsf{N}, 2)$ by using the append function:

$\mathsf{append}(\mathsf{N}, ?n, ?m, ?s_1, ?s_2) \in \mathsf{Seq}(\mathsf{N}, 2)$

where $?n, ?m, ?s_1$ and $?s_2$ are placeholders, with the typings

$?n, ?m \in \mathsf{N}$,

$?s_1 \in \mathsf{Seq}(\mathsf{N}, ?n)$ and

$?s_2 \in \mathsf{Seq}(\mathsf{N}, ?m)$.

Now, we will do a refinement which leads to a scratch area which is impossible to complete, but which can not be detected by the computation of the constraints. If $?s_1$ is refined by cons, we get the object

$\mathsf{append}(\mathsf{N}, ?n, ?m, \mathsf{cons}(\mathsf{N}, ?a, ?n_1, ?s), ?s_2) \in \mathsf{Seq}(\mathsf{N}, 2)$

and the type checking will produce the equalities

$?n = s(?n_1)$ and $?n + ?m = 2$.

[3] In [?] and [?] it is shown that higher order unification [?] can be generalized to dependent types. However, in our situation it seems difficult even to ensure that flexible-flexible pairs are always unifiable, since a flexible term may be defined by pattern matching.

The first equation will instantiate $?n$, leaving us with the constraint
$s(?n_1)+?m = 2$.
Now, we can see that since a sequence always has positive length, it is impossible to both instantiate the remaining placeholders and satisfy the constraint.

So we have here an example where a user refinement leads to a "dead end", which cannot be detected by the constraints. Of course, if the placeholders are further instantiated the constraint *will* be violated, but it was impossible to detect at the point when the erroneous step was made. This is also an argument for why the deleting operation is important.

Now, we will define what a scratch area is, but first we must define exactly what we mean by a partial object. We are forced to restrict all objects in definitions to be β-normal, since we can only type check β-normal objects. The reason is that given an object $([x]b)a$ and a type B, we can not uniquely infer the type family $(x \in A)B'$, which when applied to a gives the type B, due to dependent function types (see [**?**]).

Definition A *partial object* is a β-normal object which may contain placeholders everywhere except as the head of an application.

The reason we can not allow a placeholder as the head of an application is that we must know the type of the head to compute the argument types.

Definition A *scratch area* is represented by a tuple $\langle \mathcal{D}, \langle \mathcal{P}, \mathcal{C} \rangle \rangle$, where
- $\mathcal{D}$ is a set of definitions, in which the definiens are partial objects,
- $\mathcal{P}$ is a list of placeholders (instantiated or not) together with their types and contexts.
- $\mathcal{C}$ is a list of constraints.

We require all placeholders in $\mathcal{D}$ to be distinct and typed in $\mathcal{P}$, and all placeholders in $\mathcal{C}$ to be in $\mathcal{P}$.

The intuitive meaning of this representation is that the partial objects in $\mathcal{D}$ corresponds to the user instantiations, the instantiated placeholders in $\mathcal{P}$ is instantiations caused by unification, and $\mathcal{C}$ is the unsolved constraints. The uninstantiated placeholders in $\mathcal{P}$ is what is left to instantiate. When the partial object is presented to the user, then the instantiated placeholders are expanded.

Definition Let $\langle \mathcal{D}, \langle \mathcal{P}, \mathcal{C} \rangle \rangle$ be a scratch area. If every placeholder in $\mathcal{P}$ is instantiated such that the constraints in $\mathcal{C}$ are satisfied, then the scratch area is *complete*. A *solution* to $\langle \mathcal{P}, \mathcal{C} \rangle$ is an instantiation of the placeholders which makes the scratch area complete.

Note that if $\mathcal{P}$ is empty, then the definitions in $\mathcal{D}$ can not contain any placeholders and are therefore complete.

Since we allow several definitions in the scratch area, which can be manipulated simultaneously, we must ensure that the set of definitions are not circular in an illegal way, i.e. there must be a non-circular dependency order between the

definitions. Mutually recursive definitions will be considered as one combined definition. Also, explicit constant definitions may not be recursive, whereas implicit definitions may.

Definition A set of definitions is *well-ordered* if the dependency graph of the definitions becomes acyclic when recursive dependencies of implicit constants are removed from the graph.

Definition A scratch area $\langle \mathcal{D}, \langle \mathcal{P}, \mathcal{C} \rangle \rangle$ is *valid* if
- The definitions in $\mathcal{D}$ are well-ordered.
- $\langle \mathcal{P}, \mathcal{C} \rangle$ has the same set of solutions as the union of the unification problems we get from type checking each definition in $\mathcal{D}$

The point is that we can type check instantiations locally, i.e. it is enough to check that an instantiation of a placeholder has the expected type, and we must not type check the entire definition every time a placeholder is instantiated. As a consequence of the definition of a valid scratch area, we get the following property;

Corollary 1. *All definitions in a complete scratch area are type correct.*

Proof. If the scratch area is complete, then we have a solution to all unification problems of type checking the definitions in $\mathcal{D}$, and since placeholders are distinct, we can divide this solution into solutions for each definition separately, which means that the instantiated definitions are complete and type correct.

6 Operations on incomplete proofterms

As mentioned, the two main operations on the scratch area is to replace a placeholder by a partial object and the converse, to replace a partial object by a placeholder. Placeholders (together with their types and local contexts) are generated when the user *refines* a placeholder. Consider again the distributivity example, where we had the definition

$\mathsf{distr} = ?d \in (A, B, C \in \mathsf{Set}; \mathsf{And}(A, \mathsf{Or}(B, C)))\mathsf{Or}(\mathsf{And}(A, B), \mathsf{And}(A, C))$.

The first step was to form an abstraction, yielding the new definition

$$\mathsf{distr} = [A, B, C, h]?e \in (A, B, C \in \mathsf{Set}; \mathsf{And}(A, \mathsf{Or}(B, C))) \\ \mathsf{Or}(\mathsf{And}(A, B), \mathsf{And}(A, C))$$

where the new placeholder $?e$ of type $\mathsf{Or}(\mathsf{And}(A, B), \mathsf{And}(A, C))$ in the local context $[A, B, C \in \mathsf{Set}; h \in \mathsf{And}(A, \mathsf{Or}(B, C))]$ is generated. The type and the context can be computed from the original type. Then the new placeholder is added to the scratch area and the old is replaced by the partial object $[A, B, C, h]?e$. In the next step $?e$ is refined with the constant OrE. Here, the number of required arguments and their types can be computed from the types of OrE and $?e$, yielding a list of new placeholders

$?A, ?B, ?C \in \mathsf{Set}$

$?h1 \in (?A)?C$

$?h2 \in (?B)?C$
$?h3 \in \mathsf{Or}(?A, ?B)$

and $?e$ is replaced by the partial object $\mathsf{OrE}(?A, ?B, ?C, ?h1, ?h2, ?h3)$. When $\mathsf{OrE}(?A, ?B, ?C, ?h1, ?h2, ?h3)$ is checked to be of proper type, the first three arguments are instantiated by unification.

When a part of an object is deleted, a new placeholder is created and its type can be computed from its position in the object. However, to ensure that all placeholders have their most general type after deletion, we may have to change the type of other placeholders as well. For example, given the definition of append in section 5, we can have a object

$\mathsf{append}(\mathsf{N}, \mathsf{s}(?n), ?m, ?s_1, ?s_2) \in \mathsf{Seq}(\mathsf{N}, 2)$,

where the placeholder $?s_1$ must be of type $\mathsf{Seq}(\mathsf{N}, \mathsf{s}(?n))$. If we replace $\mathsf{s}(?n)$ by a new placeholder $?n_1$, we must recompute the type of $?s_1$ to get the most general type $\mathsf{Seq}(\mathsf{N}, ?n_1)$. Actually, if we did not change the type of s_1, unification would again instantiate $?n_1$ to $\mathsf{s}(?n)$, and it would be impossible to delete any dependent argument. Another way to solve this problem of implicit sharing is to give an internal name to each dependent argument, and use the name instead in the object ([**?**]). However, the second approach becomes rather inefficient when objects grow large.

Before we describe the operations, we will explain the type checking algorithm briefly.

6.1 Typechecking

The algorithm takes as input a context, a type and a β-normal object and returns a list of constraints which must be satisfied for type correctness, or a failure. If the list is empty, the object is type correct. The algorithm starts by checking if the context is valid, then checks if the type is a valid type in that context, before calling the actual type checking algorithm. This algorithm consists of mainly three parts; type checking *TC*, type conversion and conversion *Conv*. The first part computes a list of *type equations*. These type equations are simplified one by one with the type conversion algorithm, which reduces a type equation to a list of object equations or reports a failure. The object equations are simplified one by one with the conversion algorithm, which reduces an object equation to simpler equations and finally removes trivial equations.

The algorithm will be computed in an *environment*, where constants are declared. Since the environment stays constant during type checking, we will assume a valid environment throughout this presentation. The environment is denoted by Σ below.

When an object is on β-normal form, we know that the head of an application is always a constant or a variable, and its type can be looked up in the environment or context, respectively. Therefore, we have an algorithm *CT* (for *C*ompute *T*ype), which is called by *TC* in the case of an application, and it returns the computed type and the equations that must hold for the arguments to be type correct. @ denotes concatenation of two lists.

$TC(a, A, \Gamma)$ checks if a is of type A in context Γ:

$$\frac{}{TC(x, A, \Gamma) \Rightarrow [A = A' \;\; \Gamma]} \text{ TC-Var } (x \in A' \in \Gamma) \qquad \frac{}{TC(c, A, \Gamma) \Rightarrow [A = A' \;\; \Gamma]} \text{ TC-Const } (c \in A' \in \Sigma)$$

$$\frac{TC(b, B\{y{:=}x\}, [\Gamma, x \in A]) \Rightarrow \xi}{TC([x]b, (y \in A)B, \Gamma) \Rightarrow \xi} \text{ TC-Abs}$$

$$\frac{CT(f, \Gamma) \Rightarrow \langle \xi_1, (x \in A')B \rangle \quad TC(e, A', \Gamma) \Rightarrow \xi_2}{TC(fe, A, \Gamma) \Rightarrow \xi_1 @ \xi_2 @ [B\{x{:=}e\} = A \;\; \Gamma]} \text{ TC-App}$$

$$\frac{}{CT(x, \Gamma) \Rightarrow \langle [\,], A \rangle} \text{ CT-Var } (x \in A \in \Gamma) \qquad \frac{}{CT(c, \Gamma) \Rightarrow \langle [\,], A \rangle} \text{ CT-Const } (c \in A \in \Sigma)$$

$$\frac{CT(f, \Gamma) \Rightarrow \langle \xi_1, (x \in A)B \rangle \quad TC(e, A, \Gamma) \Rightarrow \xi_2}{CT(fe, \Gamma) \Rightarrow \langle \xi_1 @ \xi_2, B\{x{:=}e\} \rangle} \text{ CT-App}$$

Type conversion is computed in the following way, first the types are reduced to (outermost) constructor form, which pushes substitutions inside the type constructors. Then, if the types have the same form, the parts of the types are checked recursively. If they have not the same form type conversion returns a failure (constructors are assumed to be one-to-one).

Conversion The conversion algorithm is similar to the one presented in [**?**]. The algorithm proceeds as follows;

1. If the objects are syntactically equal - we are done.
2. If the objects have a function type, then both objects are applied to a fresh variable, and conversion is called recursively.
3. Try to reduce both objects to *head normal form.*
4. If both objects are rigid (i.e the head is a variable or a constructor and can not be changed by any instantiation), the simpler head conversion is invoked.
5. Otherwise, (i.e. at least one object is flexible), we leave the equation as a constraint.

(2) guarantees that the objects are *saturated*, and (4) implies that the object is of the form $b(a_1, \ldots, a_n)$ where b is either a variable or a constructor. (Note that this is a stronger restriction on b than β-normal, where b may be any constant, and since constants may have reduction rules associated with them, the object may be further reduced). The only possibility for two saturated objects on head normal form to be equal, is that the heads are identical and the arguments are pairwise convertible. Therefore, this is exactly what the head conversion algorithm checks. There are two advantages of performing the rule (2), first η-conversion need not be checked separately ($[x](fx)$ will be equal to f when applied to a variable) and second, we know that a function defined by pattern matching will be applied to all its arguments and this simplifies the matching.

$Conv(a, b, A, \Gamma)$ checks if a and b are convertible. The head-normal-form reduction ($\xrightarrow{hnf}$) returns a labeled object, where the labels are either *Rigid* or *Flex*. The reason is that we can not determine if an object is rigid or flexible just by looking at the form, since an application of an implicit constant can be either rigid ($+(x, y)$, where x, y are variables) or flexible ($+(?x, y)$). This is decided during reduction.

$$\frac{}{Conv\,(a, a, A, \Gamma) \Rightarrow [\,]}\ \text{Conv-Id} \qquad \frac{Conv\,(az, bz, B\{x{:=}z\}, [\Gamma, z{\in}\ A]) \Rightarrow \xi}{Conv\,(a, b, (x \in A)B, \Gamma) \Rightarrow \xi}\ \begin{array}{l}\text{Conv-fun}\\ (z \notin Dom(\Gamma))\end{array}$$

$$\frac{a \xrightarrow{hnf} Rigid(a') \quad b \xrightarrow{hnf} Rigid(b') \quad HConv\,(a', b', \Gamma) \Rightarrow \langle \xi, A\rangle}{Conv\,(a, b, A, \Gamma) \Rightarrow \xi}\ \text{Conv-rigid}$$

$$\frac{a \xrightarrow{hnf} Flex(a') \text{ or } b \xrightarrow{hnf} Flex(b')}{Conv\,(a, b, A, \Gamma) \Rightarrow [a' = b' \in A\ \ \Gamma]}\ \text{Conv-flexible}$$

$$\frac{}{HConv\,(b, b, \Gamma) \Rightarrow \langle [\,], A\rangle}\ \begin{array}{l}\text{HConv-head}\\ (\ b \in \{x, c\}\ )\end{array}$$

$$\frac{HConv\,(f, g, \Gamma) \Rightarrow \langle \xi_1, (x \in A)B\rangle \quad Conv\,(a, b, A, \Gamma) \Rightarrow \xi_2}{HConv\,(fa, gb, \Gamma) \Rightarrow \langle \xi_1 @\, \xi_2, B\{x{:=}a\}\rangle}\ \text{HConv-app}$$

The conversion algorithm correspond exactly to the rigid-rigid transformation in Elliot's unification algorithm for dependent function types. It computes a well-formed list of equations, so if all equations hold then all equations are well-typed.

Assuming that the substitution calculus in [**?**] is normalizing, we can show the following properties:

Proposition 1 *The type checking algorithm is sound and complete relative to Martin-Löf's substitution calculus, for β-normal, complete objects and types.*

This result will appear in the forthcoming Ph.D thesis of the first author.

Lemma 2. *If the input to the conversion algorithm is well-typed, then the resulting list of equations have the same set of solutions as the input equation.*

Proof. The Conv-Id rule is trivial and the Conv-fun rules is justified by the extensionality rule of the calculus. Due to normalization, objects are equal if and only if their (head) normal forms are equal, so the Conv-rigid rule preserves the set of solutions. Finally, since constructors are assumed to be one-to-one, and free variables are unique, the equation $b(a_1, \ldots, a_n) = b(a'_1, \ldots, a'_n)$ holds iff $a_1 = a'_1 \wedge \cdots \wedge a_n = a'_n$ hold, and if the heads are distinct, there is no solution (and the conversion algorithm returns fail).

6.2 Insertion

We will define when a placeholder can be replaced by an object, and how the list of constraints is simplified.

Definition Let $\langle \mathcal{D}, \langle \mathcal{P}, \mathcal{C} \rangle \rangle$ be a valid scratch area and let $?u$ be a placeholder occurring in a definition c, and $?u$ is not instantiated in $\mathcal{C}$. Then the insertion of object e in place of $?u$ is *admissible* if the following holds

1. Type checking e with type A_u in Γ_u produces a list of constraints $\mathcal{C}_u$, where A_u and Γ_u is the type and context of $?u$, respectively.
2. e contains no constants which depends on c, unless c is an implicit constant then c and other mutually defined constants may occur in e.
3. Simplifying the new list of constraints, where $?u$ is replaced by e everywhere and $\mathcal{C}_u$ is added, does not produce a detectable inconsistency.

If the conditions hold, then the placeholder is removed and $?u$ is replaced by e everywhere.

Definition A solved constraint is an equation $?u = e \in A\ \Gamma$ (or $e = ?u \in A\ \Gamma$), where $?u$ does not occur in e.

Simplification of constraints: Simplification of the constraints proceeds by applying the following rule of transformation, until the rule is no longer applicable.

1. Pick a solved constraint $?u = e \in A\ \Gamma$ in $\mathcal{C}$
2. Type check e with the expected type $A_u\ \Gamma_u$, yielding a list of constraints $\mathcal{C}_u$
3. Replace $?u$ by e everywhere in $\mathcal{C}$ and $\mathcal{P}$, and add $\mathcal{C}_u$ to $\mathcal{C}$.
4. Update $?u$ by e in $\mathcal{P}$.
5. For each constraint in $\mathcal{C}$, replace the constraint by the result of applying the conversion algorithm. If the conversion returns fail, the list of constraints is inconsistent.

We will illustrate why we need to type check the instantiation from the solved constraint with the following example. Suppose we want to prove the proposition $\exists x.\forall y.R(x,y)$ for some reflexive relation R, so we suppose we have a constant $\mathsf{refl} \in (x \in A)R(x,x)$. After using the exist- and forall- introduction rules, we are in the situation

$$c = ?c \in R(?x, y) \quad [y \in A]$$

and want to fill in the placeholders $?c$ and $?x$, where $?x$ is of type A and has an empty local context. Refining $?c$ with the constant refl creates a new placeholder $?u$ of type A in context $[y \in A]$. Type checking the insertion of $\mathsf{refl}(?u) with expected type R(?x, y)$ produces the equation

$$R(?u, ?u) = R(?x, y) \in Set \quad [y \in A]$$

which is simplified to

$$?x = y \in A \quad [y \in A] \text{ and } ?u = y \in A \quad [y \in A].$$

Since $?x$ is defined in the empty context, the instantiation is not valid, it violates the scoping rules. We conjecture that it is enough to check that the free variables

of a solved constraint is included in the placeholders local context, rather than type checking the instantiation. This is enough if we can show that the type in a solved constraint is always the same as the expected type of the placeholder.

Lemma 3. *The simplification of a list of constraints preserves the set of solutions.*

Proof. We can only guarantee termination if the constraints are consistent, since otherwise the objects in the equations may not be well-typed, and ill-typed objects may be reduced in the conversion algorithm. However, if the constraints are consistent, then the transformation will terminate since for each transformation the number of un-instantiated placeholders decreases, and the conversion algorithm produces a result which is either empty, identical to the original constraint or replaced with a list of constraints corresponding to the arguments compared pairwise which are all strictly smaller in complexity then the original constraint.

Finally, since $?u = e$ must hold in any solution, we can replace $?u$ by e without changing the set of solutions.

Proposition 2 *The validity of a scratch area is invariant under admissible insertions.*

Proof. Let $\langle \mathcal{D}, \langle \mathcal{P}, \mathcal{C} \rangle \rangle$ be a valid scratch area and let $\langle \mathcal{D}', \langle \mathcal{P}', \mathcal{C}' \rangle \rangle$ be the scratch area after the admissible insertion of e for $?u$ in definition c. By assumption $\mathcal{D}$ is well-ordered and since e can not contain any constants that depend on c in an illegal way, the insertion can not create a cycle. Hence, $\mathcal{D}'$ is well-ordered.

Let $\mathcal{U}$ ($\mathcal{U}'$) denote the union of the unification problem we get from type checking each definition in $\mathcal{D}$ ($\mathcal{D}'$), respectively. Let us call a list of constraints *updated by* $(?u, e)$ if $?u$ is replaced by e in the list, and the constraints from type checking e with the type of $?u$ is added to the list. Then we know that the solutions of $\langle \mathcal{P}, \mathcal{C} \rangle$ updated by $(?u, e)$ are the same as the solutions of $\langle \mathcal{P}', \mathcal{C}' \rangle$ by lemma 3, and since $\mathcal{U}'$ is the same set of equations as $\mathcal{U}$ updated by $(?u, e)$, they have the same solutions. By assumption, we have that $\mathcal{U}$ and $\langle \mathcal{P}, \mathcal{C} \rangle$ have the same solutions, and hence, $\mathcal{U}'$ and $\langle \mathcal{P}', \mathcal{C}' \rangle$ do to.

6.3 Deletion

To delete the object e in definition c, in a scratch area $\langle \mathcal{D}, \mathcal{P}, \mathcal{C} \rangle$ we only have to do the following steps

1. Create a new placeholder and replace e by it.
2. Delete the placeholders in e from $\mathcal{P}$, and remove all instantiations in $\mathcal{P}$.
3. Delete $\mathcal{C}$.
4. Recompute the types of all placeholders in c.
5. Retype check all definitions in $\mathcal{D}$, yielding a list of constraints $\mathcal{C}_{new}$.
6. Simplify $\mathcal{C}_{new}$.

Proposition 3 *The validity of a scratch area is invariant under deletion.*

Proof. Since the definitions in are noncyclic by assumption, we can not create a cycle by deleting a subobject. We have that the new list of constraints has the same set of solutions as the simplified list, by lemma 3.

In the implementation of ALF, the operations on the scratch area is optimized by having one visible part of the scratch area, in which *all* instantiations are updated in the definitions, and one invisible part where only the user instantiations are updated. This speeds up the insertion operation, and does not effect the deletion operation.

7 Proof Experiments and Experiences

The experiments done so far are relatively small. Nora Szasz has given a formal proof [?] that Ackermann's function is not primitive recursive. Veronica Gaspes showed [?] functional completeness of combinatorial logic, i.e. that every function can be compiled into an expression only involving the basic combinators **S**, **K** and **I**. Björn von Sydow [?] showed the fundamental theorem of arithmetic, i.e. that every integer is a product of a unique multiset of primes. Karlis Cerans and K.V.S. Prasad has made experiments with expressing proofs in various process calculii.

The proof engine is implemented in Standard ML and the window interface is written in C++ using the Interviews package. The next version is being written in Haskell and is substantially smaller. The current system is still incomplete, but people who are interested to try it are welcome to get a copy by anonymous ftp from ftp.cs.chalmers.se. A user manual can also be obtained in that way.

8 Comparision with other work

This editor is in a tradition of editors which started with Stanford and Edinburgh LCF [?]. Later descendants to this system are Coq and Lego. Coq [?] was developed in INRIA-Rocquencourt by Gerard Huet and his group and is based on Coquand's and Huet's Calculus of Constructions [?]. Lego [?] started as an offspring of Coq, but is now a completely new system. None of these systems use direct manipulation of the proof object, instead a proof is built up by giving commands to a proof engine.

The first implementation of Martin-Löf's type theory was made in 1982 by Kent Petersson [?] who replaced PPλ, the logic of the Edinburgh LCF with the polymorphic type theory as described in [?]. A group in Cornell under Robert Constable implemented the NuPrl system, a proof editor for a version of Martin-Löf's type theory with a modern user interface. This system is also based on indirect building of a proof [?].

9 Acknowledgements

This is joint work with Thierry Coquand. We want to thank him for many discussions. The first version of ALF was designed by Thierry Coquand and Bengt

Nordström in 1991, Thierry implemented the first proof engine and Lennart Augustsson implemented the user interface. The current proof engine is implemented by Lena. Johan Nordlander has programmed the user interface. We also want to thank the members of the programming logic group at Chalmers for many discussions on these topics. In particular we want to thank Björn von Sydow for writing the section describing the distributivity example.