A guided tour through TYPELAB*

Marko Luther and Martin Strecker
Abt. Künstliche Intelligenz
Fakultät für Informatik
Universität Ulm
D-89069 Ulm

{luther,strecker}@ki.informatik.uni-ulm.de

Abstract

This report gives a survey of TYPELAB, a specification and verification environment that integrates interactive proof development and automated proof search. TYPELAB is based on a constructive type theory, the Calculus of Constructions, which can be understood as a combination of a typed λ-calculus and an expressive higher-order logic. Distinctive features of the type system are dependent function types for modeling polymorphism and dependent record types for encoding specifications and mathematical theories. After presenting an extended example which demonstrates how program development by stepwise refinement of specifications can be carried out, the theory underlying the prover component of TYPELAB is described in detail. A calculus with metavariables and explicit substitutions is introduced, and the meta-theoretic properties of this calculus are analyzed. Furthermore, it is shown that this calculus provides an adequate foundation for automated proof search in fragments of the logic.

*This research has partly been supported by the "Deutsche Forschungsgemeinschaft" within the "Schwerpunktprogramm Deduktion". A shorter version of this work appeared in (von Henke et al., 1998).

Contents

1 Introduction

This report gives a survey of TYPELAB, a specification and verification environment that integrates interactive proof development and automated proof search. TYPELAB is based on a constructive type theory, the Calculus of Constructions, which can be understood as a combination of a typed λ-calculus and an expressive higher-order logic. Distinctive features of the type system are dependent function types for modeling polymorphism and dependent sigma types for encoding specifications and mathematical theories.

Type theory provides a homogeneous theoretical framework in which the construction of, say, a function and the construction of a proof can be considered to be essentially the same activity. There is, however, a practical difference in that the development of a function requires more insight and therefore usually has to be performed under human guidance, whereas proof search can, to a large extent, be automated. Internally, TYPELAB exploits the homogeneity provided by type theory, while externally offering an interface to the human user which conceals most of the complexities of type theory. Interactive construction of proof objects is possible whenever desired, metavariables serve as placeholders which can incrementally be refined until the desired object is complete. For procedures which can reasonably be automated, high-level tactics are available. In this respect, TYPELAB can be understood as a proof assistant which, in addition to the manipulations of formulae traditionally performed by theorem provers, permits to carry out operations on entities such as functions and types.

From a different perspective, TYPELAB can be viewed as a programming environment in which, apart from the execution of programs in the style of functional language interpreters, properties of programs can be specified and verified and in which complex developments can be carried out. Even though these features are currently not advanced very much beyond the stage of a research prototype, it is possible to enter expressions at the top level and evaluate them by reduction to normal form. Function definition is so far limited to (higher-order) primitive recursive functions and no efficient compilation is currently available, but a wide range of practically relevant functions can be coded in a natural style.

TYPELAB supports program development by stepwise refinement: Declarations of types, functions and axioms can be bundled up to specifications respectively mathematical theories. Specifications, as internal objects of the logic, can be handled in complete analogy to other entities of the logic. In particular, they can be parameterized, possibly by other specifications, and can be the domain or range of functions, which in this case can be interpreted as refinement mappings.

The remainder of this report is organized as follows: In Section 2, some concepts of TYPELAB are illustrated by an example, the derivation of executable functions from an abstract specification of a symbol table. Since metavariables play an important role in the interactive construction of proof objects, but lead to some intricacies in dependently-typed logics, the theory behind TYPELAB's metavariables has to be described in greater detail (Section 3). In Section 4, it will be shown how this machinery, developed primarily for interactive proof construction, provides a foundation for an automation of proof search. A comparison with related systems (Section 5) and final remarks (Section 6) conclude this report.

2 An extended example

In this section, it will be necessary to briefly sketch some distinctive aspects of TYPELAB's specification language, because specification language and logic are closely intertwined. The language is based on the Calculus of Constructions (Coquand and Huet, 1988), which comprises a higher-order logic and a functional language with an expressive type system. In particular, types are first-class objects, which yields a form of parametric polymorphism. Types and relations can be defined inductively in a user-friendly syntax, induction principles are generated automatically. Specifications respectively mathematical theories (not to be confounded with the notion of "theory" in the LCF tradition) can be represented by dependent record types (Luo, 1994), elements of these types can be interpreted as realizations of specifications.

The language features sketched so far will in the following be illustrated by a standard example, the development of the specification of a symbol table into an executable version. Our formalization is inspired by Sannella's specification (Sannella, 1989) in Extended ML (see also (Hofmann, 1992)). A symbol table is a data structure in which items, called *attributes*, are stored together with an *identifier* by which they can be retrieved later on.

When working with TYPELAB, a *global context* of declarations and definitions is gradually built up. We assume that this context already contains some elementary vocabulary about natural numbers, Booleans etc. The specification of the symbol table will be parameterized by theories `IDENT` and `ATTRIB`. Their definition can now be added to the global context:

```
IDENT :=
  SPEC
    Ident: Type,
    =_Ident: Ident → Ident → Bool,
    AXIOM equiv_ax: equiv_p =_Ident
  END-SPEC
```

```
ATTRIB :=
  SPEC
    Attrib,Attrib+: Type,
    bottom: Attrib+,
    lift: Attrib → Attrib+,
    AXIOM lift_ax:
          ∀a:Attrib. bottom ≠ (lift a)
  END-SPEC
```

As opposed to the global context, the declarations (and possibly also definitions) in a specification generate a *local context* whose members are not visible from outside, except when explicitly selected. Thus, we declare a type `Ident` and a relation $=_{Ident}$ inside of `IDENT`. In the constructive type theory on which TYPELAB is based, propositions and types are identified, therefore the statement of the axiom is in fact a syntactically sugared declaration of `equiv_ax` of type $=_{Ident}$.

We can now turn to the specification of the symbol table itself. $\mathtt{SYMTAB}_0$ is a specification which is parameterized by the specifications `IDENT` and `ATTRIB`. As already mentioned above, specifications are types, parameterization can therefore simply be expressed as λ-abstraction. $\mathtt{SYMTAB}_0$ specifies a type `Symtab`, a constant `empty`, a function `add` that adds an identifier and an attribute to a symbol table, a function `present_p` that tests whether there is an entry for an identifier in a symbol table, a function `lookup` that retrieves the attribute associated with an identifier from a symbol table, a function `change_attrib` that, when given an identifier and an attribute, modifies the value associated to the identifier to the attribute, and the related axioms. The construct **open** `(ID,ATT)` **in** `B` makes components of `ID` and `ATT` visible in the body `B` so that they can be referenced directly without an explicit selection such as `ID.Ident`.

```
SYMTAB0 := λID:IDENT,ATT:ATTRIB.
 open (ID,ATT) in
  SPEC
   Symtab: Type,
   empty: Symtab,
   add: Ident → Attrib → Symtab → Symtab,
   present_p: Ident → Symtab → Bool,
   lookup: Ident → Symtab → Attrib+,
   change_attrib: Ident → Attrib → Symtab → Symtab,
   AXIOM ax1: ∀i:Ident. (present_p i empty) = false,
   AXIOM ax2: ∀i,is:Ident,as:Attrib,s:Symtab.
                (present_p i (add is as s)) = ((i =Ident is) ∨ (present_p i s)),
   AXIOM ax3: ∀i,is:Ident,as:Attrib,s:Symtab.
                (present_p i (change_attrib is as s)) = (present_p i s),
   AXIOM ax4: ∀i:Ident. (lookup i empty) = bottom,
   AXIOM ax5: ∀i:Ident,a:Attrib,s:Symtab.
                (lookup i (add i a s)) = (lift a),
   AXIOM ax6: ∀i,is:Ident,as:Attrib,s:Symtab. (i =Ident is) = false ⟹
                ((lookup i (add is as s)) = (lookup i s)),
   AXIOM ax7: ∀i:Ident,a:Attrib,s:Symtab. (present_p i s) = true ⟹
                ((lookup i (change_attrib i a s)) = (lift a)),
   AXIOM ax8:∀i,is:Ident,as:Attrib,s:Symtab. (i=Ident is) = false ⟹
                ((lookup i (change_attrib is as s)) = (lookup i s)),
   THEOREM th1: ∀s:Symtab,i:Ident,a:Attrib.
                  (lookup i (add i a s)) ≠ bottom
  END-SPEC
```

Stating the theorem in the last line of the specification generates a *proof obligation* which can be discharged any time later during the development. A theorem can be used even if it has not been proved yet. The mechanisms described in Section 3 prevent circular arguments.

The proof management of TYPELAB keeps track of unsolved obligations. Open obligations can be selected in the graphical user interface and fully or partially solved in any order. If we invoke the TYPELAB prover with the obligation generated for the theorem $\mathtt{th}_1$, we enter the following proof state:

```
 ...
 ID:IDENT, ATT:ATTRIB, Symtab:Type
 ..
 ax5:∀i:Ident,a:Attrib,s:Symtab.(lookup i (add i a s)) = lift a
 ..
 ax8:∀i,is:Ident,as:Attrib,s:Symtab. ..
|--------------------------------------
?SYMTAB0_th1:∀s:Symtab,i:Ident,a:Attrib.
              (lookup i (add i a s)) ≠ bottom
```

Here, the global context is abbreviated by `'...'`, the local assumptions are displayed above the stylized turnstile (some are omitted). The proof obligation is represented by the *metavariable* $\mathtt{?SYMTAB}_0\mathtt{\_th}_1$ which is a placeholder for the proof object that will be constructed. In this case, the proof object does not have computational contents. Proof objects are useful if functions are synthesized together with their correctness proof (see below) or if programs are extracted from proofs.

First we introduce the local assumptions:

```
tlab? intros;
Command succeeded. Goals: 1 new, 1 open.
 ..
 ax5:∀i:Ident,a:Attrib,s:Symtab.(lookup i (add i a s)) = lift a
 ..
 s:Symtab, i:Ident, a:Attrib
|--------------------------------------
 ?1:(lookup i (add i a s)) ≠ bottom
```

We can now use $\mathtt{ax}_5$ as rewrite rule to rewrite `(lookup i (add i a s))` in goal `?1` to `(lift a)`:

```
tlab? rewrite ax5;
Command succeeded. Goals: 1 new, 1 open.
?2:(lift a) ≠ bottom
```

This yields the new goal `?2` with the same local context as of goal `?1`. It can completely be solved using the axiom `lift_ax` out of `ID:IDENT`:

```
tlab? refine lift_ax;
Command succeeded. Goals: 0 new, 0 open. Q.E.D.
```

Now we turn back to the development of $\texttt{SYMTAB}_0$. The first step will be to implement `change_attrib` by a function that adds the identifier-attribute pair to the symbol table if the identifier already occurs in the symbol table, and leaves it unchanged otherwise. This implementation step is achieved by a high-level operator `implement` that takes the name of the component to be implemented (`change_attrib`), the source specification ($\texttt{SYMTAB}_0$) and the expression which implements the function and produces a new specification (called $\texttt{SYMTAB}_1$) and a function (called `sm1to0`) which shows that $\texttt{SYMTAB}_1$ is indeed a realization of $\texttt{SYMTAB}_0$.

```
implement change_attrib in SYMTAB0 by
     λi:Ident,a:Attrib,s:Symtab. if (present_p i s) (add i a s) s
yields SYMTAB1 and sm1to0
```

Note that the **`implement`** operator is not a function defined in the language of TYPELAB, but just a syntactic device that removes from $\texttt{SYMTAB}_0$ the declaration of `change_attrib`, as well as axioms referencing `change_attrib`, textually produces the specification $\texttt{SYMTAB}_1$ and submits it to the type checker (Henke et al., 1995). Thus, except for the components `change_attrib`, $\texttt{ax}_3$, $\texttt{ax}_7$ and $\texttt{ax}_8$, specification $\texttt{SYMTAB}_1$ agrees with $\texttt{SYMTAB}_0$.

The refinement mapping `sm1to0` is generated in a similar spirit: The components that remain unaffected by the implementation are copied, and for `change_attrib`, its proposed implementation is inserted. The refinement mapping then reads as follows:

```
sm1to0 := λID:IDENT,ATT:ATTRIB,sym:(SYMTAB1 ID ATT).
  open (ID,ATT) in
   STRUCT
     Symtab := sym.Symtab, empty := sym.empty, add := sym.add,
     present_p := sym.present_p, lookup := sym.lookup,
     change_attrib := λi:Ident,a:Attrib,s:Symtab.
                          if (present_p i s) (add i a s) s,
     ax1 := sym.ax1, ax2 := sym.ax2, ax4 := sym.ax4,
     ax5 := sym.ax5, ax6 := sym.ax6
   END-STRUCT :: (SYMTAB0 ID ATT)
type = ΠID:IDENT,ATT:ATTRIB. (SYMTAB1 ID ATT) → (SYMTAB0 ID ATT)
```

For given `ID:IDENT` and `ATT:ATTRIB`, the refinement mapping `sm1to0` converts each realization ("model") `sym` of $(\texttt{SYMTAB}_1\ \texttt{ID ATT})$ to a realization of $(\texttt{SYMTAB}_0\ \texttt{ID ATT})$. A specification can be understood as a dependent record type (a Σ type), which generalize Cartesian products. Accordingly, elements of specifications, *structures* in TYPELAB terminology, are generalizations of tuples. As mentioned above, most of the components of $(\texttt{SYMTAB}_1\ \texttt{ID ATT})$, such as `sym.empty`, can simply be copied to form the corresponding component of $(\texttt{SYMTAB}_0\ \texttt{ID ATT})$. For the components $\texttt{ax}_3$, $\texttt{ax}_7$ and $\texttt{ax}_8$, which are required in $(\texttt{SYMTAB}_0\ \texttt{ID ATT})$ but have no counterpart in $(\texttt{SYMTAB}_1\ \texttt{ID ATT})$, the typechecker generates three proof obligations which are handled in analogy to the proof obligations created by theorems. The purpose of the coercion `::(SYMTAB₀ ID ATT)` is just to enable the typechecker to recognize which type the given realization is supposed to have, and to make it generate proof obligations for missing components.

Let's prove the obligation generated for ax_7 now. The proof easily succeeds by introducing assumptions, expanding definitions and invoking equational simplification:

```
 ...
 ax6 := sym.ax6:∀i,is:Ident,as:Attrib,s:sym.Symtab. ..
|--------------------------------------
 ?sm1to0_ax7:∀i:Ident,a:Attrib,s:Symtab. (present_p i s) = true ⟹
                    (lookup i (change_attrib i a s)) = (lift a)
tlab? intros; expandR;
Command succeeded. Goals: 1 new, 1 open.
 ...
 ax6 := sym.ax6:∀i,is:Ident,as:Attrib,s:sym.Symtab. ..
 i:Ident, a:Attrib, s:Symtab
 h1:(present_p i s) = true
|--------------------------------------
 ?5:(lookup i (if (present_p i s) (add i a s) s)) = (lift a)
tlab? eqSimplify;
Command succeeded. Goals: 0 new, 0 open. Q.E.D.
```

The last step of the development consists in producing an executable realization. We choose to implement the symbol table by a list of pairs of identifiers and attributes.

The inductive type of polymorphic lists is predefined in the system by the following statement:

```
ind [T|Type] List : Type := nil:List | cons:T→List→List;
```

As abbreviation we can write `T*` instead of `(List T)` for the type of lists with elements of type `T`. It is now quite straightforward to implement `add` as the function that attaches an identifier-attribute pair to the front of a list.

```
REALIZATION := λID:IDENT,ATTR:ATTRIB.
   open (ID,ATTR) in
     let (Elem := Ident×Attrib) in
       STRUCT
          Symtab := Elem*,
          empty  := (nil Elem),
          add    := λi:Ident,a:Attrib,l:Elem*. (cons ⟨i,a⟩ l)
       END-STRUCT :: (SYMTAB1 ID ATTR)
```

However, it is not so evident how to realize the functions present_p and lookup. We therefore submit the incomplete realization to TYPELAB. Apart from proof obligations for axioms concerning the given function add, the typechecker generates proof obligations that contain metavariables ?present_p and ?lookup which stand for the as yet undefined functions present_p and lookup, respectively. We will now show how TYPELAB can be used to construct the function present_p and to prove its correctness at the same time.

We start with the goal requiring to construct the function present_p and expand the definition of Symtab. It is reasonable to assume that present_p recurses over its second argument, so we start an induction over lists.

```
 ...
 ID:IDENT, ATTR:ATTRIB,
 Elem   := Ident×Attrib:Type
 Symtab := Elem*:Type
 empty  := (nil Elem):Elem*
 add    := λi:Ident,a:Attrib,l:Elem*. (cons ⟨i,a⟩ l)
|---------------------------------------
 ?present_p:Ident→Symtab→Bool
tlab? expandR Symtab; induct 2;
Command succeeded. Goals: 2 new, 2 open.
 ...
 i:Ident, l:Elem*
|---------------------------------------
 ?4:Bool
new subgoal 2 is: ?9:Bool
```

We obtain two new goals, the first one corresponding to the base case. By applying the above commands, TYPELAB has constructed a partial solution for ?present_p:

```
tlab? show proof-term;
  λi:Ident,l1:Elem*. elimList (λl:Elem*. Bool) ?4
     (λe:Elem,l2:Elem*,b:Bool. ?9) l1
```

This partial solution still contains two metavariables (?4 and ?9), corresponding to what the function returns for l_1=nil and for l_1=(cons e l_2), respectively. We leave our goal for a moment, to look at the obligation ?REALIZATION_ax_1 which has been generated for the axiom ax_1 and specifies the behaviour of present_p for an empty symbol table:

```
 ...
 present_p := λi:Ident,l1:Elem*. elimList (λli:Elem*. Bool) ?4
              (λe:Elem,l2:Elem*,b:Bool. ?9) ..
 lookup := ?lookup:Ident→Symtab→Attrib+
|---------------------------------------
 ?REALIZATION_ax1:∀i:Ident. (present_p i empty) = false
```

The local context of this proof obligation contains the partial solution provided for `?present_p` so far. Introducing assumptions and expanding definitions on the right leads to the following goal, from which we can read off the required solution for metavariable `?4` at once:

```
tlab? intros; expandR;
Command succeeded. Goals: 1 new, 1 open.
 ...
 i:Ident
|--------------------------------------
 ?2:(?4 = false)
```

We can use this hint to solve the base case of the original problem. Finishing this part of the proof, we turn to the step case, which can be handled in a similar fashion, however involving some reasoning modulo the reduction relation generated by the list recursor. This reasoning is currently not automated; a more powerful unification procedure would be desirable here.

Altogether, we have synthesized the following function for `present_p`:

$$\lambda \mathtt{i{:}Ident,l_1{:}Elem^*.}\ \ \mathtt{elim}_{List}\ (\lambda \mathtt{l{:}Elem^*.\ Bool})\ \mathtt{false}$$
$$(\lambda \mathtt{e{:}Elem,l_2{:}Elem^*,b{:}Bool.}\ (\mathtt{i} =_{Ident} \mathtt{e.1}) \vee \mathtt{b})\ \mathtt{l_1}$$

As demonstrated by the synthesis of this function, the “executable” and “logical” parts of the calculus interact smoothly, proof construction in both fragments obeys the same principles and can therefore be handled by the same machinery.

After having completed the realization of the symbol table, we can instantiate its parameter theories appropriately, for example by choosing strings as identifiers and natural numbers as attributes.

```
ident :=
  STRUCT
    Ident  := String,
    =_Ident := =_String
  END-STRUCT :: IDENT
```

```
attrib :=
  STRUCT
    Attrib  := Nat,
    Attrib+ := Error Nat,
    bottom  := failure Nat "No entry!",
    lift    := success Nat
  END-STRUCT :: ATTRIB
```

For the realization of attributes we use the predefined inductive type `Error` with the two constructor `success` and `failure` on natural numbers to model the extended domain `Attrib+`. Of course we have to prove the axiom `lift_ax` for this realization.

```
symtab := REALIZATION ident attrib
sym1 := add "Martin" 31 (add "Marko" 28 empty)
```

The following computations confirm that the functions work as desired:

```
tlab> compute (present_p "Marko" sym1);
  true
tlab> compute (lookup "Martin" sym1);
  success Nat 31
tlab> compute (lookup "Frieder" sym1);
  failure Nat "No entry!"
```

The above development of executable functions from an abstract specification has been carried out in order to present the most important constructs of the TYPELAB specification language, to exemplify a certain software development methodology and to demonstrate how interactive and automated proof techniques are applied.

Interactive proof construction is further facilitated by a pleasant graphical user interface. There are separate windows for displaying the global context, the current proof state, the proof tree and much more. The objects presented in the windows are mouse sensitive. For example, context entries can be shrunk or hidden to save space on the screen. Similarly, subtrees of the proof tree can be expanded so that individual steps taken by a complex tactic can be viewed in detail. TYPELAB suggests appropriate proof rules or equations for rewriting if subterms of the current goal are activated.

Equational simplification is the only kind of automation applied in the above example. Apart from that, TYPELAB offers proof search in intuitionistic logic, which will be described at length in Section 4. Based on proof search and equational reasoning, some still quite rudimentary support for inductive proofs has been developed.

3 A calculus with metavariables

The example derivation given in Section 2 has demonstrated the usefulness of metavariables as placeholders for proof objects. The concept of proof object has to be understood in a large sense: A proof object can be a function that is part of a realization of a specification, it can be a witness of an existentially quantified variable or a term which encodes logical reasoning, according to the propositions-as-types principle[1].

For these seemingly disparate notions, type theory provides a unifying framework. For this convenience, a price has to be paid: in a dependently typed logic such as the Calculus of Constructions that TYPELAB is based on, metavariables can depend on one another. The solution for one metavariable, say $?n_1$, can determine the type of another metavariable $?n_2$ and thus influence the set of valid solutions for $?n_2$. A priori, it is not clear which dependencies among metavariables should be admitted, and whether a partial solution provided for

[1]Some of the following motivation makes use of terminology introduced in the later sections.

a metavariable can safely be accepted. The sections below try to clarify these questions by developing a calculus with metavariables.

An additional difficulty arises from the fact that metavariables not only depend on a type, but also on a context that determines which variables can legally occur in the solution of the metavariable. For example, the theorem in specification $\texttt{SYMTAB}_0$ of Section 2 depends on the context that is shown in the initial proof state.

There are mainly two problems when dealing with context-dependent metavariables, illustrated by the following examples:

Example 3.1
Commutativity of instantiation and reduction: Assume that metavariable $?n_1$ is defined to be of type T in a context containing $x : T$, that is, $x : T \vdash ?n_1 : T$. In a naive approach, first reducing the term $trm_1 := (\lambda x : T.\ ?n_1)\ t$ to $?n_1$ and then instantiating the result with term x yields the result x. First instantiating $?n_1$ to x and then reducing yields t (the variable x bound by λ-abstraction is the same object as the variable x bound in the context of the metavariable – de Bruijn indices would give a clearer picture).

$$\begin{tikzcd}
(\lambda x : T.\ ?n_1)\ t \arrow[rr, "\{?n_1 := x\}"] \arrow[d, "\beta"'] & & (\lambda x : T.\ x)\ t \arrow[d, "\beta"'] \\
?n_1 \arrow[r, "\{?n_1 := x\}"] & x & t
\end{tikzcd}$$

Note that this problem is not caused by a particular type system, but arises in any calculus in which there is a notion of β-reduction and in which metavariables depend on a context.

$\diamondsuit$

Example 3.2
Keeping track of type information: Consider a metavariable $?n_2$ defined with the following context and type:

$$A : Type, T : Type, x : T \vdash ?n_2 : T$$

Consider the term $trm_2 := (\lambda T : Type.\ \lambda x : T.\ ?n_2)\ A$ in context A:Type. When first instantiating $?n_2$ with x and then reducing, the resulting $\lambda x : A.\ x$ is easily seen to have type $A \to A$. When first reducing trm_2, however, the question arises what the type of the resulting term $\lambda x : A.\ ?n_2$ should be. $A \to T$ is certainly not correct, as T does not even occur in context A:Type. Claiming that $?n_2$ has type A is also problematic, since then, the term $?n_2$ would have different types (A resp. T) in different contexts. As opposed to the first problem, this difficulty is directly related to the type system and arises in a similar form in any calculus with dependent types.

◇

To overcome these difficulties, we will keep track of substitutions that have been carried out in metavariables. This leads to a notion of *explicit substitution*. After reduction, trm_1 becomes $?n_1^\frown[x := t]$, whereas trm_2 becomes $\lambda x : A.\ ?n_2^\frown[T := A]$. The calculus developed in the following solves the above problems, and it will be shown that it has desirable meta-theoretical properties such as confluence and strong normalization.

A calculus with metavariables and explicit substitutions for Martin-Löf's monomorphic type theory is presented by Magnusson (1995), and some algorithms such as unification are defined and shown to be correct. However, the properties of the calculus are not examined. Ensuring confluence and termination is not just a routine matter; some straightforward definitions of simply typed λ-calculus with explicit substitutions lack these properties (see for example (Lescanne, 1994; Melliès, 1995)).

Section 3.1 is concerned with the behaviour of – not necessarily well-typed – terms, Section 3.2 defines typing rules for metavariables. Alternative approaches are conceivable, for example using a functional encoding of scopes which avoids dependence of metavariables on contexts. Consequently, no explicit substitutions have to be taken into account. In Section 3.4, it will be shown that both representations have the same strength in that there is a one-to-one correspondence between them. From a practical perspective, however, a functional encoding of scopes would usually become very clumsy for all those metavariables nested deeply inside terms, for example proof obligations generated for theorems within specifications or metavariables occurring in incomplete refinement mappings, as in the example of Section 2.

3.1 Term calculus

The term language of the logic is built up from a set $\mathcal{V}$ of variables, from type constants $Prop$ and $Type_i$ ($i \geq 0$), denoting universes of propositions and types, of dependent product types (Π types) which generalize universal quantification and dependent record types (Σ types) which are used to model specifications. On the level of elements of types, we have λ-abstraction, function application, pairing and projection. In addition to these term constructors, already present in the core logic as defined in (Luo, 1994), a set $\mathcal{M}$ of *metavariables* is introduced. Metavariables $?n, ?m, \ldots \in \mathcal{M}$ are placeholders for terms to be constructed.

The syntax of the calculus with metavariables is then defined by the grammar of Figure 1. One of the productions for terms $\mathcal{T}$ permits to build terms of the form $?n^\frown\sigma$, which expresses that the application of substitution σ to metavariables $?n$ has been delayed. Substitutions are generated by production $\mathcal{S}$. By this construction, a substitution effectively becomes a part

$$
\begin{array}{rcl}
\mathcal{T} & ::= & \mathcal{V} \\
 & | & Prop \mid Type_i \\
 & | & \Pi\mathcal{V}:\mathcal{T}.\mathcal{T} \mid \lambda\mathcal{V}:\mathcal{T}.\mathcal{T} \mid (\mathcal{T}\ \mathcal{T}) \\
 & | & \Sigma\mathcal{V}:\mathcal{T}.\mathcal{T} \mid pair_{\mathcal{T}}(\mathcal{T},\ \mathcal{T}) \mid \pi_1(\mathcal{T}) \mid \pi_2(\mathcal{T}) \\
 & | & \mathcal{M}^\frown\mathcal{S} \\
 & & \\
\mathcal{S} & ::= & [\,] \mid [\mathcal{V} := \mathcal{T}] :: \mathcal{S}
\end{array}
$$

Figure 1: Grammar of language with metavariables and explicit substitutions

of a term and can be reasoned about in the calculus. This notion of *explicit substitutions* has to be distinguished from the traditional notion of substitutions, which are defined as meta-operations on terms. It should be remarked that substitutions can only be attached to metavariables and not to arbitrary terms. This distinguishes our calculus from (to our best knowledge) all calculi of explicit substitutions presented in the literature.

To improve readability, we sometimes write $\forall x : A.B$ instead of $\Pi x : A.\ B$, or $A \to B$ if x does not occur in B. Metavariables with empty substitution, $?n^\frown[\,]$, will often be abbreviated as $?n$.

For proving properties of the term calculus, such as confluence, it will be convenient to assume that there is a function *svars* which associates to each $?n \in \mathcal{M}$ a list of variables that may be substituted into $?n$, the substitutions for other variables being discarded (cf. Definition 3.4). When considering well-typed terms in Section 3.2, it will turn out that this list of variables is naturally given by the context on which $?n$ depends.

We can now define explicit (or *internal*) substitutions more formally as follows:

Definition 3.3 (Internal Substitution)
An internal substitution is a list of the form $[x_1 := t_1, \ldots x_n := t_n]$ associating terms to variables. A substitution $\sigma \equiv [x_1 := t_1, \ldots x_n := t_n]$ is *valid* for a metavariable $?n$ if:

1. All the variables x_i are distinct.
2. All the variables x_i are contained in $svars(?n)$.
3. If x_i occurs before x_j in σ, then x_i occurs before x_j in $svars(?n)$.

The *domain* of a substitution is defined to be the set $dom(\sigma) := \{x_1, \ldots x_n\}$.

$\diamondsuit$

Since substitutions in our calculus are only internalized as attachment to metavariables to express that a substitution is delayed, we cannot completely dispense with an external notion of substitution.

Definition 3.4 (External Substitutions)
A substitution $\sigma := \{x := s\}$ homomorphically maps over terms as usual. On metavariables, it is defined as follows (we assume that $x \notin \{y_1 \ldots y_n\}$ and will from now on enforce this requirement when applying substitutions):

- If $x \in svars(?n)$ and x is declared between y_i and y_{i+1} in $svars(?n)$:

$$(?n^\frown[y_1 := t_1, \ldots y_n := t_n])\sigma :=$$

$$?n^\frown[y_1 := t_1\sigma, \ldots y_i := t_i\sigma, x := s, y_{i+1} := t_{i+1}\sigma, \ldots y_n := t_n\sigma]$$

- If $x \notin svars(?n)$:

$$(?n^\frown[y_1 := t_1, \ldots y_n := t_n])\sigma :=$$

$$?n^\frown[y_1 := t_1\sigma, \ldots y_n := t_n\sigma]$$

$\diamondsuit$

In order to distinguish external substitutions from the internalized explicit substitutions, we write the former in braces like $\{\ldots\}$ and the latter in brackets like $[\ldots]$. However, the letters σ and τ will be used indiscriminately for both kinds of substitutions.

We assume that the reader is familiar with standard notions of λ-calculi such as one-step reduction $\rightarrow_1$, parallel one-step reduction $\Rightarrow_1$, reduction $\twoheadrightarrow$ (the reflexive-transitive closure of $\rightarrow_1$) and convertibility $\simeq$ (cf. Barendregt 1984). The Church-Rosser property ensures that two diverging computation paths can always be joined again. If reductions of the strict part of $\twoheadrightarrow$ always terminate (which indeed they do, see Proposition 3.18), then normal forms are unique. This way, convertibility of two terms can be decided, by reducing them to normal form and comparing the normal forms syntactically. The following result can be established by an adaptation of a method originally developed by Martin-Löf and Tait:

Proposition 3.5 (Reduction is Church-Rosser)

The reduction relation $\twoheadrightarrow$ satisfies the Church-Rosser property.

$\diamondsuit$

3.2 Typing

Typing judgements of the form $\Gamma \vdash t : T$ are used to express that term t is of type T in context Γ, where Γ is a list of declarations of the form $x_1 : T_1, \ldots, x_n : T_n$. Typing judgements are inductively generated by typing rules such as the following (λ) rule, which expresses how elements of dependent product types are introduced.

$$\frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x : A.M : \Pi x : A.B} \; (\lambda)$$

By the propositions-as-types principle, which interprets propositions as types and elements of propositions as proofs, the rule can also be understood as saying that a λ-abstraction provides a proof trace for the introduction of a universal quantifier.

As opposed to the situation encountered in less complex calculi (such as the simply-typed λ-calculus), there can be intricate dependencies among metavariables in calculi with dependent types. In particular, the type of one metavariable can depend on the value assigned to another one, and the well-typedness of a context can depend on the value assigned to a metavariable.

Before stating typing rules and examining their properties, some restrictions on dependencies among metavariables have to be imposed which are strong enough to make verification of the correctness of solutions for metavariables possible. The restrictions should be sufficiently liberal so that dependencies among metavariables can be exploited for proof search.

Example 3.6
Consider the $(\exists R)$ rule (cf. Section 4.3):

$$\frac{\Gamma \vdash ?n_1 : A \qquad \Gamma \vdash ?n_2 : P(?n_1)}{\Gamma \vdash ?n_0 : \exists x : A.P(x)} \; (\exists R)$$

Application of this rule introduces two metavariables $?n_1$ and $?n_2$, where $?n_2$ depends on $?n_1$ since $?n_1 \in MV(P(?n_1))$ (here, $MV(.)$ is the set of metavariables occurring in a term or context). If there is a declaration of the form $h : P(a)$ in Γ, then $?n_2$ can be solved with h, leading to an assignment $?n_1 := a$ as a side-effect.

$\diamondsuit$

Example 3.7
Starting from the (academic) formula $\exists P : Prop, Q : P \to Prop, x : P.(P \to Q(x))$, eliminating existential quantifiers (as in the example above) and introducing assumptions, one obtains the following set of metavariables:
$\vdash ?P : Prop$
$\vdash ?Q : ?P \to Prop$
$\vdash ?x : ?P$
$h : ?P \vdash ?n : ?Q(?x)$.

One step that suggests itself now is to equate $?n$ with h and consequently $?P$ with $?Q(?x)$, leading to a new proof problem with $\vdash ?Q : ?Q(?x) \to Prop$ and $\vdash ?x : ?Q(?x)$, in which the dependency of metavariables is cyclic. There is no intuitive interpretation of such a proof problem, nor does it seem clear that tentative solutions of such a proof problem can effectively be verified.

◊

In the sequel, we will permit proof problems with metavariable dependencies of the first kind, but will exclude circularities of the second kind.

Let us emphasize again that a metavariable depends on a context Γ and has a type T, as expressed by the more suggestive notation $\Gamma \vdash ?n : T$. In the course of a proof, metavariables occurring in Γ or T can be instantiated. Therefore, a context and a type are not invariantly assigned to a metavariable $?n$ by functions depending only on $?n$, but they are determined by the proof problem under consideration, as reflected by the following definition.

Definition 3.8 ((Valid) Proof Problem)
A proof problem $\mathcal{P}$ is a triple $(M_{\mathcal{P}}, ctxt_{\mathcal{P}}, type_{\mathcal{P}})$ consisting of:

- A finite set of metavariables $M_{\mathcal{P}}$
- A function $ctxt_{\mathcal{P}}$ assigning a context to each $?n \in M_{\mathcal{P}}$
- A function $type_{\mathcal{P}}$ assigning a term to each $?n \in M_{\mathcal{P}}$

The subscripts in $M_{\mathcal{P}}, ctxt_{\mathcal{P}}, type_{\mathcal{P}}$ will be omitted whenever $\mathcal{P}$ is clear from the context.

For a proof problem $\mathcal{P}$ and $?n_1, ?n_2 \in M_{\mathcal{P}}$, the relation $<_{\mathcal{P}}$ is defined as:
$?n_1 <_{\mathcal{P}} ?n_2 \quad \text{iff} \quad ?n_1 \in MV(ctxt(?n_2))$ or $?n_1 \in MV(type(?n_2))$.
Let $\ll_{\mathcal{P}}$ be defined as the transitive closure of $<_{\mathcal{P}}$.

A proof problem $\mathcal{P}$ is called *valid* if $\ll_{\mathcal{P}}$ is an irreflexive partial order.

◊

The proof problem $\mathcal{P}_1$ with the set of metavariables $\{?n_1, ?n_2\}$ as given in Example 3.6 is valid, with $?n_1 \ll_{\mathcal{P}_1} ?n_2$ since $?n_1 \in MV(P(?n_1))$. Regarding Example 3.7, the proof problem $\mathcal{P}_2 = (M_2, ctxt_2, type_2)$ has $M_2 = \{?x, ?Q\}$, $ctxt_2(?x)$,$ctxt_2(?Q)$ the empty context and $type_2(?x) = (?Q\ ?x)$ and $type_2(?Q) = ?Q(?x) \rightarrow Prop$. $\mathcal{P}_2$ is not valid, since $?x \ll_{\mathcal{P}_2} ?x$.

Unless stated otherwise, we will only consider valid proof problems in the following. Since, for every valid proof problem $\mathcal{P}$, the set $M_{\mathcal{P}}$ is finite and the order $\ll_{\mathcal{P}}$ is transitive and irreflexive, it is also well-founded on $M_{\mathcal{P}}$. This fact can be used to show termination of certain functions involving metavariables. In particular, the definitions made in the following can be shown to be well-defined.

The typing rules for metavariables with explicit substitutions are shown in Figure 2. Typing rules for metavariables are added to the typing rules of the base logic (Luo, 1994) in a "modular" fashion, that is, without making a modification of the base rules necessary. The typing rules for metavariables can be motivated as follows:

$$\frac{ctxt(?n) \vdash type(?n) : Type_j}{ctxt(?n) \vdash ?n^\frown[\,] : type(?n)} \text{ (MV-base)}$$

$$\frac{\begin{array}{c} z \notin dom(\Gamma) \cup dom(\Delta) \cup dom(\sigma) \\ \Gamma \vdash T : Type_j \qquad \Gamma, \Delta \vdash ?n^\frown\sigma : N \end{array}}{\Gamma, z : T, \Delta \vdash ?n^\frown\sigma : N} \text{ (MV-weak)}$$

$$\frac{\Gamma \vdash t : T \qquad \Gamma, x : T, \Delta \vdash ?n^\frown\sigma : N}{\Gamma, \Delta\{x := t\} \vdash (?n^\frown\sigma)\{x := t\} \ : \ N\{x := t\}} \text{ (MV-}\beta\text{-Red)}$$

Figure 2: Typing rules for Metavariables

MV-base A metavariable $?n$ with empty substitution is typecorrect in case its defining type $type(?n)$ and, consequently, its defining context $ctxt(?n)$ are well-typed.

MV-weak The weakening rule, which is admissible for the base logic, is explicitly added for metavariables. Its effect could as well have been encoded into the rules (MV-base) and (MV-β-Red).

MV-β-Red This rule simulates the behaviour of β-reduction. To illustrate its effect, we resume Example 3.2 which leads to a type-incorrect term when treated naively.

Assume, then, that the term $((\lambda T : Type.\ \lambda x : T.\ ?n)\ A)$ has to be reduced to normal form, where $A : Type, T : Type, x : T \vdash ?n : T$. Note that the type of this term is $A \to A$. β-reduction yields the term $\lambda x : A.\ ?n^\frown[T := A]$. The derivation of its type reflects the procedure of β-reduction – with the sole difference that $T : Type$ and $x : T$ are not bound locally by λ-abstraction, but globally in the context:

$$\cfrac{\cfrac{A : Type \vdash A : Type \quad A : Type, T : Type, x : T \vdash ?n^\frown[\,] : T}{A : Type, x : A \vdash ?n^\frown[T := A] : A} \text{ (MV-}\beta\text{-Red)}}{A : Type \vdash \lambda x : A.\ ?n^\frown[T := A] : A \to A} \ \lambda$$

The observation suggested by this example – the type of terms is invariant under reduction – is confirmed by Proposition 3.10 below.

At first glance, the rules do not appear to have an appropriate form for typechecking. In particular, the rule (MV-β-Red) seems to require guessing a substitution $\{x := t\}$. It can however be shown that efficient typechecking of judgements $\Gamma \vdash ?n^\frown\sigma : T$ is indeed possible by starting with the judgement $ctxt(?n) \vdash ?n^\frown[\,] : type(?n)$ and incrementally building up the expected substitution σ by applying rules in the forward direction.

Altogether, the extended calculus preserves the pleasant properties of the base calculus, such as the following, all of which can be proved by rather straightforward inductive arguments:

Proposition 3.9 (Decidability of Type Inference and Type Checking)

- Given a term t and a context Γ, there is an algorithm that determines whether t is well-typed in Γ or not and, in the first case, computes the principal type of t in Γ.
- Given a term t, a type T and a context Γ, there is an algorithm that determines whether $\Gamma \vdash t : T$ holds or not.

$\diamondsuit$

Proposition 3.10 (Subject Reduction)

If $\Gamma \vdash M : A$ and $M \twoheadrightarrow N$, then $\Gamma \vdash N : A$.

$\diamondsuit$

3.3 Solutions of metavariables

Definition 3.11 (Instantiation)

An instantiation[2] $\iota_{\mathcal{P}}$ for a valid proof problem $\mathcal{P} = (M_{\mathcal{P}}, ctxt_{\mathcal{P}}, type_{\mathcal{P}})$ is a function mapping the metavariables in $M_{\mathcal{P}}$ to terms, subject to the requirement that for every metavariable $?n \in M_{\mathcal{P}}$:

- if $\iota_{\mathcal{P}}(?n) = t$ and $?m \in MV(t)$, then $\iota_{\mathcal{P}}(?m) = ?m$

The notation for instantiations is comparable to the notation for substitutions: If $M_{\mathcal{P}} = \{?n_1, \ldots ?n_k\}$, then $\iota_{\mathcal{P}}$ is written as $\{?n_1 := t_1, \ldots ?n_k := t_k\}$. Whenever $\mathcal{P}$ is understood from the context, the subscript is omitted from $\iota_{\mathcal{P}}$.

Instantiations are inductively extended to terms as follows:

- $\iota(x) = x$ for variables x.
- $\iota(Prop) = Prop$, $\iota(Type_i) = Type_i$
- $\iota(Qx : T.M) = Qx : \iota(T).\iota(M)$ for $Q \in \{\lambda, \Pi, \Sigma\}$
- $\iota(f\ a) = (\iota(f)\ \iota(a))$
- $\iota(pair_T(t_1,\ t_2)) = pair_{\iota(T)}(\iota(t_1),\ \iota(t_2))$
- $\iota(\pi_i(t)) = \pi_i(\iota(t))$ for $i = 1, 2$

[2] The term *instantiation* has been chosen to distinguish instantiation of metavariables from substitution of variables

- $\iota(?n^\frown[x_1 := t_1 \ldots x_k := t_k]) = \iota(?n)\{x_1 := \iota(t_1) \ldots x_k := \iota(t_k)\}$

This definition can be extended to contexts in an obvious manner.

$\diamondsuit$

Note in particular that no renaming of bound variables is carried out: As opposed to substitutions $\{x := s\}$, which must not be applied to terms in which x is bound (see Definition 3.4), variables occurring in the term t of an instantiation $\{?n := t\}$ have to be interpreted relative to the defining context of $?n$.

Definition 3.12 (Instantiation of proof problems)
Assume that $\mathcal{P} = (M_\mathcal{P}, ctxt_\mathcal{P}, type_\mathcal{P})$ is a valid proof problem and ι an instantiation for $\mathcal{P}$. The instantiation $\iota(\mathcal{P}) = (M'_\mathcal{P}, ctxt'_\mathcal{P}, type'_\mathcal{P})$ is defined to be the proof problem consisting of:

- $M'_\mathcal{P} := \{?n \in M_\mathcal{P} | \iota(?n) = ?n\}$
- $ctxt'_\mathcal{P}$ is defined as the function which, for $?n \in M'_\mathcal{P}$, yields $\iota(ctxt_\mathcal{P}(?n))$
- $type'_\mathcal{P}$ is defined as the function which, for $?n \in M'_\mathcal{P}$, yields $\iota(type_\mathcal{P}(?n))$

$\diamondsuit$

Intuitively, if $\mathcal{P}$ is a proof problem, then $\iota(\mathcal{P})$ is the proof problem that remains after providing a partial solution ι.

Example 3.13
Assume the proof problem $\mathcal{P}$ is given by:

$$A : Type, P : A \to Prop, a : A, h : (P\ a) \vdash ?n_1 : A$$

$$A : Type, P : A \to Prop, a : A, h : (P\ a) \vdash ?n_2 : (P\ ?n_1)$$

and the instantiation $\iota_\mathcal{P}$ by $\{?n_1 := a\}$. Then $\mathcal{P}' := \iota_\mathcal{P}(\mathcal{P})$ is the proof problem consisting of:

$$A : Type, P : A \to Prop, a : A, h : (P\ a) \vdash ?n_2 : (P\ a)$$

It can be solved by another instantiation $\iota_{\mathcal{P}'} := \{?n_2 := h\}$.

$\diamondsuit$

The notion of instantiation is not related to type correctness. This is remedied by the following definition:

Definition 3.14 (Valid / Welltyped Instantiation)
Let $\mathcal{P} = (M_\mathcal{P}, ctxt_\mathcal{P}, type_\mathcal{P})$ be a valid proof problem.

- An instantiation ι is called *valid* if $\iota(\mathcal{P})$ is a valid proof problem.

- An instantiation ι is called *well-typed* if, for every $?n \in M_{\mathcal{P}}$,

$$\iota(ctxt_{\mathcal{P}}(?n)) \vdash \iota(?n) : \iota(type_{\mathcal{P}}(?n))$$

$\diamondsuit$

As a consequence of the following proposition, it is sufficient to verify instantiations "locally", i.e. for metavariables only, to ensure typecorrectness "globally" for all terms to which instantiations are applied.

Proposition 3.15 (Instantiation preserves typing)
Let $\mathcal{P}$ be a valid proof problem, Γ, t and T be a context resp. terms in which at most metavariables from $\mathcal{P}$ occur, and ι a well-typed instantiation for $\mathcal{P}$.

- If $\Gamma \vdash t : T$ holds, then also $\iota(\Gamma) \vdash \iota(t) : \iota(T)$.
- $\iota(\mathcal{P})$ is a well-typed proof problem.

$\diamondsuit$

These definitions and propositions provide the foundations of incremental proof construction. They ensure that, whenever metavariables are solved with valid, well-typed instantiations, no type-incorrect terms can result.

3.4 Functional encoding of scopes

In the following, a functional representation of scopes will be examined. The general idea is to replace a metavariable $?n$ of type T which depends on assumptions $x_1 : T_1, \ldots, x_k : T_k$ by a metavariable $?F$ which is of functional type $\Pi x_1 : T_1 \ldots \Pi x_k : T_k . T$ and which does not depend on assumptions.

Example 3.16
This procedure can best be illustrated by an example. Consider the following proof problem:

$$A : Type, P : A \rightarrow Prop \vdash ?n_0 : \forall a : A. \exists x : A. (P\, a) \rightarrow (P\, x)$$

This problem can be decomposed into the subproblems:

$$A : Type, P : A \rightarrow Prop, a : A \vdash ?n_1 : A$$

$$A : Type, P : A \rightarrow Prop, a : A \vdash ?n_2 : (P\, a) \rightarrow (P\, ?n_1)$$

It can easily be verified that $?n_1 := a$ and $?n_2 := \lambda y : (P\, a).\, y$ are welltyped instantiations. The problem can also be stated with metavariables $?F_1$ and $?F_2$ which are the functional analogues of $?n_1$ and $?n_2$ and which are defined by:

$$\vdash ?F_1 : \Pi A : Type, P : A \rightarrow Prop, a : A . A$$

$$\vdash ?F_2 : \Pi A : Type, P : A \to Prop, a : A.(P\, a) \to (P\, (?F_1\, A\, P\, a))$$

The solutions of this proof problem are $?F_1 := \lambda A : Type, P : A \to Prop, a : A.a$ and $?F_2 := \lambda A : Type, P : A \to Prop, a : A.\lambda y : (P\, a).\, y$.

◇

This functional translation can be defined more formally as a mapping taking terms t to terms $\bar{t}$ and proof problems $\mathcal{P}$ to proof problems $\overline{\mathcal{P}}$. Furthermore, it can be shown that this mapping is type-preserving and that there is a one-to-one correspondence between solutions of $\mathcal{P}$ and solutions of $\overline{\mathcal{P}}$.

A benefit of a functional representation of scopes is that substitutions cannot take effect in metavariables: $?F\sigma$ will always be the same as $?F$, since all the variables in the domain of σ cannot occur free in a solution of $?F$. Thus, under a functional representation, there is no need to explicitly record substitutions applied to metavariables – altogether, the calculus essentially behaves like a calculus without metavariables. These advantages have to be traded against the difficulties incurred when dealing with metavariables that depend on many local assumptions, such as the metavariables created for theorems in specifications or metavariables created as typecorrectness conditions in refinement mappings (see the examples in Section 2).

A functional encoding of metavariables is used in many algorithms and systems dealing with proof search in higher-order logic. Only recently have there been attempts (Dowek et al., 1995) to restate unification algorithms for the simply typed lambda calculus in terms of calculi of explicit substitutions. The transformation of metavariables into a functional encoding is closely related to "lifting" as presented by Paulson (1989). Miller (1992) examines methods of exchanging existential and universal quantifiers and develops a similar technique called "raising".

By showing that the reduction relations are preserved under the functional translation, it can be shown that, since the calculus without metavariables is strongly normalizing, so is the calculus with metavariables and explicit substitutions. The following lemma shows how to simulate reductions in the calculus with metavariables by reductions in the calculus with metavariables in functional representation, which we will call "essentially metavariable-free".

Lemma 3.17
If $M \Rightarrow_1 M'$ and $M \neq M'$, then $\overline{M} \Rightarrow_1 \overline{M'}$ and $\overline{M} \neq \overline{M'}$.

◇

Proof:
By induction on the generation of $\Rightarrow_1$. □

Proposition 3.18 (Strong Normalization)
The calculus with metavariables and explicit substitutions is strongly normalizing.

◇

Proof:
Assume, to the contrary, that there is a well-typed term M which permits an infinite sequence

of reduction steps: $M \equiv M_0 \Rightarrow_1 M_1 \Rightarrow_1 \ldots$, with $M_i \neq M_{i+1}$. It can be shown that the term $\overline{M}$ is well-typed in the essentially metavariable-free calculus, and by Lemma 3.17, there is an infinite sequence of well-typed terms $\overline{M} \equiv \overline{M_0} \Rightarrow_1 \overline{M_1} \Rightarrow_1 \ldots$, with $\overline{M_i} \neq \overline{M_{i+1}}$. This contradicts the strong normalization property of the essentially metavariable-free calculus.

□

4 Automated Proof Search

In this section, automation of proof search will be examined. The kind of automation we are aiming at should integrate well with the interactive proof construction presented in Section 3, and it should lead to practically useful procedures, even if completeness has to be sacrificed.

Previous work includes investigations carried out by Pym and Wallen (1991) on proof search in the $\lambda\Pi$ calculus. The proof procedure does not make appeal to typing rules as presented in Section 3.2 to ensure that solutions of metavariables are welltyped in the metavariable-free fragment, but instead uses a complex consistency criterion which is based on permutability of rules. Dowek (1993) develops a complete search procedure for all systems of the λ-cube. For the Calculus of Constructions, this proof procedure has a possibly infinite branching factor which can be avoided when giving up completeness. Even though both approaches (Pym and Wallen, 1991) and (Dowek, 1993) use context-dependent metavariables as we do, the problems related to substituting into metavariables are not addressed by a calculus of explicit substitutions.

Contrary to the procedures defined by Pym, Wallen and Dowek, we have deliberately chosen a formulation which includes the usual logical connectives and is not restricted to the elementary term constructors of the logic, viz. essentially Π-abstraction and its non-dependent version, implication. This makes this calculus amenable to traditional proof search techniques and optimizations.

In Section 4.1, we will shortly review unification. In Section 4.2, a sequent (or Tableau) calculus will be developed. Usually, in sequent calculi an Eigenvariable condition has to be satisfied. The discussion in Section 4.3 will show that this proviso need not be enforced explicitly, since all derivations with typecorrect instantiations in the sense of Section 3.3 respect it. The main emphasis in the following is on conveying an idea of how the apparatus developed so far can be used for automated proof search, and not on providing details of efficient search procedures.

4.1 Unification

Let us briefly point out how unification can be defined in our framework so as to produce welltyped instantiations in the sense of Definition 3.14.

Unification of terms t_1, t_2 consists in finding an instantiation for the metavariables occurring in t_1 or t_2 such that t_1 and t_2 are equal modulo convertibility. In order to be able to develop a correctness criterion, we suppose that the metavariables of t_1 and t_2 stem from a welltyped proof problem $\mathcal{P}_0$. It is furthermore assumed that t_1 and t_2 are welltyped in a context Γ, even though their types need not agree at the outset.

Definition 4.1 (Unification Problem)
A *unification problem* $\langle \mathcal{P}_0, \Gamma \vdash t_1 \stackrel{?}{=} t_2 \rangle$ is a pair consisting of a welltyped proof problem $\mathcal{P}_0$ and a unification equation $\Gamma \vdash t_1 \stackrel{?}{=} t_2$, where Γ is a valid context, t_1 and t_2 are terms welltyped in Γ and all the metavariables of Γ, t_1, t_2 are among the metavariables of $\mathcal{P}_0$.

$\diamondsuit$

The rules defining the unification algorithm will use a judgement of the form $\langle \mathcal{P}_0, \Gamma \vdash t_1 \stackrel{?}{=} t_2 \rangle \Rightarrow \mathcal{P}_1; \iota_1$, which expresses that the unification problem $\langle \mathcal{P}_0, \Gamma \vdash t_1 \stackrel{?}{=} t_2 \rangle$ can be (partially) solved by instantiation ι_1, leaving open the metavariables of the proof problem $\mathcal{P}_1$. As an example, consider the unification rule for λ-abstraction in Figure 3.

$$\frac{\begin{array}{c}\langle \mathcal{P}_0, \Gamma \vdash A_1 \stackrel{?}{=} A_2 \rangle \Rightarrow \mathcal{P}_1; \iota_1 \\ \langle \mathcal{P}_1, \iota_1(\Gamma, x_1 : A_1 \vdash B_1 \stackrel{?}{=} B_2\{x_2 := x_1\}) \rangle \Rightarrow \mathcal{P}_2; \iota_2\end{array}}{\langle \mathcal{P}_0, \Gamma \vdash \lambda x_1 : A_1 .\, B_1 \stackrel{?}{=} \lambda x_2 : A_2 .\, B_2 \rangle \Rightarrow \mathcal{P}_2; \iota_2}\ (\lambda - \lambda)$$

$$\frac{\begin{array}{c} ctxt(?n) \vdash t : T \\ \langle \mathcal{P}_0, \Gamma \vdash type(?n) \stackrel{?}{=} T \rangle \Rightarrow \mathcal{P}_1; \iota_1 \\ valid(\iota_1 \cup \{?n := t\}, \mathcal{P}_1) \\ \iota_2 := \iota_1 \cup \{?n := t\} \quad \mathcal{P}_2 := \iota_2(\mathcal{P}_1)\end{array}}{\langle \mathcal{P}_0, \Gamma \vdash ?n \stackrel{?}{=} t \rangle \Rightarrow \mathcal{P}_2; \iota_2}\ (MV - term)$$

Figure 3: First-order unification

In order to unify a metavariable $?n$ and an arbitrary term t (rule (MV-term)), the type of $?n$ and the type T of t are first unified, yielding an instantiation ι_1. It still has to be verified that the instantiation $\iota_1 \cup \{?n := t\}$ is valid, in particular that it passes the occurs check implicit in Definition 3.11 and that the resulting proof problem $\mathcal{P}_2$ satisfies the acyclicity condition of Definition 3.8. It can then be concluded that the instantiation $\iota_2 := \iota_1 \cup \{?n := t\}$ is valid and well-typed in the sense of Definition 3.14.

Altogether, the rules satisfy the following invariants, which can be understood as stating the correctness of the unification algorithm.

Proposition 4.2 (Invariants of Unification)
Assume $\langle \mathcal{P}_0, \Gamma \vdash t_1 \stackrel{?}{=} t_2 \rangle$ is a unification problem. If $\langle \mathcal{P}_0, \Gamma \vdash t_1 \stackrel{?}{=} t_2 \rangle \Rightarrow \mathcal{P}_1; \iota_1$, then:

- ι_1 is a valid, well-typed instantiation for $\mathcal{P}_0$.
- $\iota_1(t_1) \simeq \iota_1(t_2)$
- $\mathcal{P}_1$ is a valid, well-typed proof problem.

$\diamondsuit$

To some extent, the rules can be adapted to incorporate higher-order aspects, for example unification in the simply-typed λ-calculus (see (Huet, 1975; Nipkow, 1993) and also (Pfenning, 1991)). This also entails some modifications of the invariants of Proposition 4.2. Thus, completeness of unification can be achieved for a larger part of the language than the essentially first-order fragment considered above.

An invariant of our unification algorithm is that two terms are only unified after a unifier for their types has been established. In some cases, this prevents simplifications such as leaving equations as constraints that can be solved later on in the course of the proof. This requirement is relaxed in the procedures for the calculus LF of Elliott (1989) and Pym (1990) which, however, leads to considerably more complex algorithms.

4.2 Tableau-style proof search

This section develops a Tableau calculus appropriate for proof search. For a better understanding, the Tableau calculus is first given in a familiar presentation as a set of sequent rules. A formulation as proof transformation system establishes the connection with the terminology of the preceding sections. This section only considers the quantifier-free fragment, whereas Section 4.3 discusses problems related to quantifiers. For lack of space, only some of the rules are presented in an exemplary fashion.

Although the core language of TYPELAB does not include the usual logical connectives and the existential quantifier as term constructors, these can easily be defined by an encoding which dates back to (Prawitz, 1965). For example, $A \wedge B$ can be defined as $\Pi R : Prop.(A \rightarrow B \rightarrow R) \rightarrow R$. It can be shown that this encoding corresponds to the usual definition of the connectives by means of natural deduction rules. Thus, for the above definition of conjunction, the following rules are derivable (with appropriately defined *andI*, *andEl* and *andEr*):

$$\frac{\Gamma \vdash a : A \quad \Gamma \vdash b : B}{\Gamma \vdash (andI\ a\ b) : A \wedge B}\ (\wedge I)$$

$$\frac{\Gamma \vdash p : A \wedge B}{\Gamma \vdash (andEl\ p) : A}\ (\wedge El) \qquad \frac{\Gamma \vdash p : A \wedge B}{\Gamma \vdash (andEr\ p) : B}\ (\wedge Er)$$

For the other connectives, similar rules are derivable.

$$\frac{\Gamma \vdash ?n_1 : A \quad \Gamma \vdash ?n_2 : B}{\Gamma \vdash ?n_0 : A \wedge B} \; (\wedge R)$$

$$\frac{\Gamma, p : A \wedge B, \Gamma', p_A : A, p_B : B \vdash ?n_1 : G}{\Gamma, p : A \wedge B, \Gamma' \vdash ?n_0 : G} \; (\wedge L)$$

Figure 4: Tableau rules for conjunction

Natural deduction rules can be divided into introduction rules (e.g. $(\wedge I)$) which say how a connective can be constructed, and elimination rules (e.g. $(\wedge El)$ and $(\wedge Er)$) which say how a connective can be decomposed. A well-known disadvantage of natural deduction is the lack of a subformula property in its elimination rules. Thus, for performing "backwards" proof search, applying rules from the conclusion towards the premisses, formulae have to be guessed, such as formula B in $(\wedge El)$ and formula A in $(\wedge Er)$.

Sequent calculi as defined by (Gentzen, 1934) replace the schema of introduction-elimination rules by a schema of Left-Right rules in which connectives are decomposed on the left resp. right side of a sequent. In our case, the interpretation of $\Gamma \vdash ?n : T$ as a sequent is obvious. The Right rules are identical to the introduction rules of the natural deduction calculus. An application of a Left rule to a formula F, or more precisely, to a context entry $x : F$ in Γ does not lead to a removal of this context entry from Γ, since x might appear elsewhere in Γ. Rather, new context entries are added at the end of Γ. This twist of representation makes it more appropriate to call our calculus a Tableau calculus.

The rules for conjunction are displayed in Figure 4. Associated with each rule is a term (cf. Figure 5) which spells out how a solution of the metavariable $?n_0$ in the conclusion of the rule can be obtained from solutions of the metavariables ($?n_1$ and $?n_2$ resp. $?n_1$) of the premisses. The correctness of $(\wedge R)$ and $(\wedge L)$, when interpreted as derived rules, can immediately be verified by typechecking their solution terms.

$$\begin{array}{ll} (\wedge R) & ?n_0 := (andI \; ?n_1 \; ?n_2) \\ (\wedge L) & ?n_0 := (\lambda p_A : A, p_B : B.?n_1) \; (andEl \; p) \; (andEr \; p) \end{array}$$

Figure 5: Solutions associated with the proof rules

More formally, Tableau-style proof search can be related to the notions developed in Section 3 by presenting the rules in the form of a proof transformation system. A transformation rule $\mathcal{P}_0; \iota_0 \Longrightarrow \mathcal{P}_1; \iota_1$ (possibly containing side conditions) expresses that proof problem $\mathcal{P}_0$ and instantiation ι_0 can be transformed to proof problem $\mathcal{P}_1$ and instantiation ι_1 when applying a proof rule backwards. The transformation rules corresponding to the proof rules of Figure 4 are displayed in Figure 6[3].

$$
\begin{array}{ll}
 & \mathcal{P} \cup \{\Gamma \vdash ?n_0 : A \wedge B\}; \iota_0 \\
 & \iota_1 := \iota_0 \uplus \{?n_0 := (andI\ ?n_1\ ?n_2)\} \\
\Longrightarrow & \iota_1(\mathcal{P} \cup \{\Gamma \vdash ?n_1 : A,\ \Gamma \vdash ?n_2 : B\}); \iota_1 \\
 & \\
 & \mathcal{P} \cup \{\Gamma, p : A \wedge B, \Gamma' \vdash ?n_0 : G\}; \iota_0 \\
 & \iota_1 := \iota_0 \uplus \{?n_0 := (\lambda p_A : A, p_B : B.?n_1)\ (andEl\ p)\ (andEr\ p)\} \\
\Longrightarrow & \iota_1(\mathcal{P} \cup \{\Gamma, p : A \wedge B, \Gamma', p_A : A, p_B : B \vdash ?n_1 : G\}); \iota_1
\end{array}
$$

Figure 6: Rules of Transformation System

We can state invariants of the proof transformation system, similar to the invariants for unification (Proposition 4.2) but technically more involved. As a special case, the following proposition expresses that an automated proof search procedure which is started on a goal $\Gamma \vdash ?n_0 : G$ and succeeds in solving it completely, constructs a well-typed proof term t for $?n_0$.

Proposition 4.3 (Correctness of Proof Search)
If $\{\Gamma \vdash ?n_0 : G\}; \{\} \Longrightarrow \{\}; \{?n_0 := t\}$, then $\Gamma \vdash t : G$.

$\diamondsuit$

So far, we have neglected questions regarding completeness. The introductory remarks of Section 4 have given some plausibility to the claim that procedures which are complete for the whole logic become too unwieldy to be practically useful. The identification of suitable fragments which lend themselves to complete proof methods is a topic of current research.

4.3 Quantifiers in sequent rules

In traditional presentations of Sequent Calculi, the rules $(\forall R)$ and $(\exists L)$ are usually stated with an "Eigenvariable condition". Typically, the rule $(\forall R)$ is then given by:

$$\frac{\Gamma \vdash B(z)}{\Gamma \vdash \forall x.\ B(x)}\ (\forall R)$$

under the proviso that the fresh variable z does not occur in Γ. In this section, we will examine how the Eigenvariable condition is enforced in our calculus. The quantifier rules are displayed in Figure 7.

[3] The combination $\iota \uplus \kappa$ of two instantiations ι, κ performs the necessary instantiations of κ within ι and vice versa which ensure the idempotency of the resulting instantiation.

$$\frac{\Gamma \vdash ?n_1 : A \qquad \Gamma \vdash ?n_2 : P(?n1)}{\Gamma \vdash ?n_0 : \exists x : A.P(x)}\ (\exists R) \qquad \frac{\Gamma, x : A \vdash ?n_1 : B(x)}{\Gamma \vdash ?n_0 : \Pi x : A.\ B(x)}\ (\Pi R)$$

$$\frac{\Gamma, p : \exists x : T.P(x), \Gamma', y : T, p' : P(y) \vdash ?n_1 : G}{\Gamma, p : \exists x : T.P(x), \Gamma' \vdash ?n_0 : G}\ (\exists L)$$

$$\frac{\begin{array}{c}\Gamma, p : \Pi x : A.\ B(x), \Gamma' \vdash ?n_1 : A \\ \Gamma, p : \Pi x : A.\ B(x), \Gamma', p' : B(?n_1) \vdash ?n_2 : G\end{array}}{\Gamma, p : \Pi x : A.\ B(x), \Gamma' \vdash ?n_0 : G}\ (\Pi L)$$

Figure 7: Tableau rules for quantifiers

Since the argument leading to Proposition 4.3 can be used to show the correctness of the associated proof transformation system, this section does not provide an additional proof, but rather an illustration for one aspect of correctness. Since most of the considerations are independent of typing, we will omit type information whenever convenient.

Example 4.4
As an illustration, we will use the following formulas throughout this section, the first of which: $\forall x.\exists y.x = y$ is valid, the second of which: $\exists x.\forall y.x = y$ is not.

$\diamondsuit$

In traditional Tableau calculi, Skolemization is used to eliminate a universal quantifier by keeping track of existential quantifiers on which it depends. Skolemization[4] expresses that the formula $\exists x.\forall y.P(x,y)$ is valid iff $\exists x.P(x, f(x))$ is valid, where f is a fresh function constant. There are two impediments to using Skolemization in our framework. The first is that it is hard to justify the introduction of a new function constant f proof-theoretically. When considering, in a typed calculus, the transition from $\exists x : A.\forall y : B.\ P(x,y)$ to $\exists x : A.P(x, f(x))$, we make a claim as to the existence of a function $f : A \to B$, for which we have no direct evidence. The second reason for not using Skolemization, of a more practical nature, is that it can blow up formulae and make them difficult to understand.

The method that is implicit in our approach is to describe the dependence of existential variables on universal variables. A proof obligation $x_1 : T_1, \ldots, x_k : T_k \vdash ?n : T$ expresses that the existential variable $?n$ occurs in the scope of the universal variables $x_1, \ldots, x_k$ and can only be solved by terms in which at most $x_1, \ldots, x_k$ occur free. The examples demonstrate the procedure: In the first example, the proof succeeds because $?y$ can be unified with x.

$$\cfrac{\cfrac{x\ :\ T \vdash ?y\ :\ T \qquad x\ :\ T \vdash ?n_2\ :\ x = ?y}{x\ :\ T \vdash ?n_1\ :\ \exists y : T.x = y}\ (\exists R)}{\vdash ?n_0\ :\ \forall x : T.\exists y : T.x = y}\ (\forall R)$$

[4]The dual variant of Skolemization in which existential quantifiers are eliminated and which preserves satisfiability is commonly found in refutational theorem proving.

In the second example, however, $?x$ does not unify with y because y does not occur in the context of $?x$.

$$\dfrac{\vdash ?x \;:\; T \qquad \dfrac{y:T \vdash ?n_2 \;:\; ?x = y}{\vdash ?n_1 \;:\; \forall y:T.?x = y}\,(\forall R)}{\vdash ?n_0 \;:\; \exists x:T.\forall y:T.x = y}\,(\exists R)$$

Again, this dependence could be made explicit by a functional encoding of scopes, as for example in the Isabelle system (Paulson, 1994). The observations of the above example can be generalized to the following proposition, which expresses that no well-typed instantiation of metavariables can violate the Eigenvariable condition:

Proposition 4.5 (Eigenvariable Condition)
Assume $\Gamma, x : A$ is a valid context with occurrences of metavariables $?m_1, \ldots ?m_k$ in Γ. Then there is no well-typed instantiation ι of $?m_1, \ldots ?m_k$ such that x occurs free in $\iota(\Gamma)$

$\diamondsuit$

Proof:
Since $\Gamma, x : A$ is a valid context and ι a valid instantiation, by Proposition 3.15, $\iota(\Gamma)$ is a valid context and thus cannot contain a free occurrence of x. $\square$

It is worth noting that our approach gives a rather direct criterion to verify that Eigenvariable conditions are respected, as opposed to indirect criteria implicit in the methods of Pym and Wallen (1991) and Shankar (1992) which encode permutabilities of rule applications in their proof search procedures. Even though rule permutabilities are not used to ensure correctness, they can be exploited to optimize proof search. In particular, they can help to recognize when goals cannot be satisfied even by application of alternative proof rules, thus avoiding useless backtracking. Details are a topic of current research.

5 Comparison

There are several criteria by which TYPELAB can be compared to other systems, one of them being the underlying logic. Systems which are based on a type theory are NUPRL (Constable et al., 1986), COQ (Barras et al., 1997), LEGO (Pollack, 1994) and ALF (Magnusson and Nordström, 1994). NUPRL and COQ provide powerful automation for fragments of the logic and permit to extract programs from proofs, but do not allow for a direct construction of objects with the aid of metavariables. In LEGO and ALF, proof construction essentially consists in finding appropriate instantiations for metavariables, further automation is not available. TYPELAB aims at a synthesis of these approaches.

Several systems pursue similar objectives as TYPELAB, but use different logics to attain them. HOL (Gordon and Melham, 1993) and ISABELLE (Paulson, 1994) are based on the simply-typed λ-calculus. Using simpler logics eliminates some of the technical difficulties of typechecking and proof search, at the expense of a reduced expressiveness of the language. By using a system based on a sufficiently rich meta logic as logical framework and encoding an object logic in it, much of the expressiveness can be regained. Even though systems like ISABELLE offer considerable support, the effort to provide a high-level proof and development environment tailored to a complex object logic is then comparable to writing a system from scratch.

Another approach to circumvent limitations of the base logic is to provide some extra-logical features. This is exemplified by Isabelle's type classes (Nipkow and Prehofer, 1995) and still more by the PVS system (Owre et al., 1992, 1995): Semantic subtypes and theory expressions which can be parameterized by types and values are grafted on the core logic of PVS, a simply-typed λ-calculus. This gives PVS much of the expressiveness of a dependently-typed calculus. However, having constructs available in the logic and not simulating them on the meta-level often gives a clearer picture of the semantics. Also, limitations of the language are sometimes dictated more by the implementation than by logical necessity. The possibility to parameterize theories by other theories and to define morphisms between theories in TYPELAB are a case in point.

A notion of specification or theory, even though external to the underlying logic, can be found in several systems, for example OBJ (Goguen and Winkler, 1988) (based on an equational logic), IMPS (Farmer et al., 1993) (simply typed λ-calcuus) and PVS. The KIV system ((W.Reif et al., 1998)), based on Dynamic Logic, provides an environment for modular software development. Central concepts are parameterized specifications, implementations and modules, the latter corresponding closely to the refinement mappings of TYPELAB.

The example of Section 2 has shown how a function can be developed together with a proof of its correctness. However, TYPELAB currently does not support extraction of programs from proofs. In the context of the Calculus of Constructions, this topic has been studied extensively by Paulin-Mohring and Werner (1993). The system COQ translates proofs in the Calculus of Constructions to an intermediate language (actually, the System F (Girard et al., 1989)), removing non-constructive proof information and flattening some dependent typing. From this intermediate language, ML code can be generated. As demonstrated by the MINLOG system ((Benl et al., 1998)), program extraction succeeds with quite sparse logical means, and even by using classical reasoning. The resulting programs can be surprisingly efficient, but some non-trivial transformations of the proof are required to obtain them.

6 Conclusions

This report has provided a survey of the TYPELAB system. The example in Section 2 has shown several styles of program development supported by TYPELAB, such as direct coding of a function, *a posteriori* verification and execution of the function, or stepwise refinement of specifications by interleaving the development of a function and solving associated proof obligations.

This example has also served as a motivation for the use of metavariables in the TYPELAB system. Metavariables can occur nested deeply inside terms, depending on a great number of local assumptions. Therefore, common techniques such as a functional encoding of scopes are not appropriate here. Some problems arising from a naive use of metavariables have been identified, and a calculus with explicit substitutions has been presented to solve them (Section 3). This calculus has desirable properties such as confluence and strong normalization, and it provides a foundation for an automation of proof search (Section 4) in that it directly ensures some conditions such as the Eigenvariable proviso for quantifier rules.

References

H. Barendregt. *The Lambda Calculus*. Elsevier Science Publishers, 1984.

B. Barras et al. *The Coq Proof Assistant Reference Manual, Version 6.1*. INRIA Rocquencourt – CNRS - ENS Lyon, 1997.

H. Benl, U. Berger, H. Schwichtenberg, M. Seisenberger, and W. Zuber. Proof theory at work: program development in the Minlog system. In Bibel and Schmitt (1998), chapter 3: Interactive Theorem Proving. To appear.

W. Bibel and P. Schmitt. *Automated Deduction — A Basis for Applications*. Kluwer Academic Publishers, 1998. To appear.

R. Constable et al. *Implementing Mathematics with the Nuprl Proof Development System*. Prentice-Hall, 1986.

T. Coquand and G. Huet. The Calculus of Constructions. *Information and Computation*, 76(2/3):pages 95–120, 1988.

G. Dowek. A complete proof synthesis method for the cube of type systems. *Journal of Logic and Computation*, 3(3):pages 287–315, 1993.

G. Dowek, T. Hardin, and C. Kirchner. Higher-order unification via explicit substitutions. In D. Kozen, editor, *Proceedings LICS'95*, pages 366–374. 1995. Extended abstract.

C. M. Elliott. Higher-order unification with dependent function types. In N. Dershowitz, editor, *Proc. 3rd Intl. Conf. on Rewriting Techniques and Applications*, pages 121–136. Springer LNCS 355, 1989.

W. M. Farmer, J. D. Guttman, and F. J. Thayer. IMPS: An interactive mathematical proof system. *J. of Automated Reasoning*, 11:pages 213–248, 1993.

G. Gentzen. Untersuchungen über das logische Schließen. *Mathematische Zeitschrift*, 39:pages 176–210 and 405–431, 1934.

J.-Y. Girard, Y. Lafont, and P. Taylor. *Proofs and Types*. Cambridge Press, 1989.

J. Goguen and T. Winkler. Introducing OBJ3. Technical Report SRI-CSL-88-9, SRI International, 1988.

M. Gordon and T. Melham. *Introduction to HOL – A theorem proving environment for higher order logic*. Cambridge University Press, 1993.

F. v. Henke, A. Dold, H. Rueß, D. Schwier, and M. Strecker. Construction and deduction methods for the formal development of software. In *KORSO: Methods, Languages, and Tools for the Construction of Correct Software*. Springer LNCS 1009, 1995.

M. Hofmann. Formal development of functional programs in type theory - a case study. Technical Report ECS-LFCS-92-228, University of Edinburgh, 1992.

G. Huet. A unification algorithm for typed lambda-calculus. *Theoretical Computer Science*, pages 27–57, 1975.

P. Lescanne. From $\lambda\sigma$ to λv a journey through calculi of explicit substitutions. In *Proc. POPL'94*, pages 60–69. 1994.

Z. Luo. *Computation and Reasoning*. Oxford University Press, 1994.

L. Magnusson. *The Implementation of ALF - a Proof Editor based on Martin-Löf's Monomorphic Type Theory with Explicit Substitution*. Ph.D. thesis, Chalmers University of Technology, 1995.

L. Magnusson and B. Nordström. The ALF proof editor and its proof engine. In H. Barendregt and T. Nipkow, editors, *Types for Proofs and Programs*, Springer LNCS 806, pages 213–237. 1994.

P.-A. Melliès. Typed λ-calculi with explicit substitutions may not terminate. In *Typed Lambda Calculi and Applications*. Springer LNCS 902, 1995.

D. Miller. Unification under a mixed prefix. *Journal of Symbolic Computation*, 14:pages 321–358, 1992.

T. Nipkow. Functional unification of higher-order patterns. In *Proc. 8th IEEE Symp. Logic in Computer Science*, pages 64–74. 1993.

T. Nipkow and C. Prehofer. Type reconstruction for type classes. *Journal of Functional Programming*, 5(2):pages 201–224, 1995.

S. Owre, J. Rushby, and N. Shankar. PVS: A Prototype Verification System. In D. Kapur, editor, *Proceedings 11th International Conference on Automated Deduction CADE*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752. Springer-Verlag, Saratoga, NY, 1992.

S. Owre, J. Rushby, N. Shankar, and F. von Henke. Formal Verification for Fault-Tolerant Architectures: Prolegomena to the Design of PVS. *IEEE Transactions on Software Engineering*, 21(2):pages 107–125, 1995.

C. Paulin-Mohring and B. Werner. Synthesis of ML programs in the system Coq. *J. Symbolic Computation*, 11:pages 1–34, 1993.

L. Paulson. The foundation of a generic theorem prover. *Journal of Automated Reasoning*, 5:pages 363–397, 1989.

L. Paulson. *Isabelle - a generic theorem prover*. Springer LNCS 828, 1994.

F. Pfenning. Unification and anti-unification in the calculus of constructions. In *Sixth Annual Symposium on Logic in Computer Science*, pages 74–85. IEEE Computer Society Press, 1991.

R. Pollack. *The Theory of LEGO – A proof checker for the Extended Calculus of Constructions*. Ph.D. thesis, University of Edinburgh, 1994.

D. Prawitz. *Natural Deduction – A proof-theoretic study*. Almqvist & Wiksells, 1965.

D. Pym. *Proofs, Search and Computation in General Logic*. Ph.D. thesis, University of Edinburgh, 1990.

D. Pym and L. Wallen. Proof-search in the $\lambda\pi$-calculus. In G. Huet and G. Plotkin, editors, *Logical Frameworks*, pages 311–340. Cambridge University Press, 1991.

D. Sannella. Formal program development in Extended ML for the working programmer. LCFS Report ECS-LCFS-89-102, University of Edinburgh, 1989.

N. Shankar. Proof search in the intuitionistic sequent calculus. In D. Kapur, editor, *Proc. CADE-11*. Springer LNCS 607, 1992.

F. von Henke, M. Luther, and M. Strecker. Interactive and automated proof construction in type theory. In Bibel and Schmitt (1998), chapter 3: Interactive Theorem Proving. To appear.

W.Reif, G. Schellhorn, K. Stenzel, and M. Balser. Structured specifications and interactive proofs with KIV. In Bibel and Schmitt (1998), chapter 3: Interactive Theorem Proving. To appear.