Science of Computer Programming 17 (1991) 119-138
Elsevier

Resolution and type theory

Leen Helmink

Philips Research Laboratories, P.O. Box 80.000, 5600 JA Eindhoven, Netherlands

Communicated by N. D. Jones
Revised March 1991

Abstract

Helmink, L., Resolution and type theory, Science of Computer Programming 17 (1991) 119-138.

In this paper, an inference mechanism is proposed for proof construction in Constructive Type Theory. An interactive system that implements this method has been developed. Some interesting proofs can be derived automatically.

Keywords. Type theory, calculus of constructions, typed lambda calculus, natural deduction.

1. Introduction

A method is presented to perform unification based top down proof construction for Constructive Type Theory, thus offering a well-founded, elegant and powerful underlying formalism for a proof development system. It combines the advantages of Horn clause resolution and higher order natural deduction style theorem proving. No theoretical contribution to Constructive Type Theory is claimed. [18] shows that the method applies to generalised type systems (GTS) [3], and thus applies to many differents variants of type theory, e.g. systems in the families of AUTOMATH [6], Martin-Löf [25], the Calculus of Constructions [9], LF [16], Elf [29]. Here, the method is presented for the Calculus of Constructions. To illustrate the method, a full derivation example is included.

The problem addressed in this paper is to construct an object in a given context, given its type. This amounts to higher order theorem proving. This paper demonstrates that this construction problem can be handled by Horn clause resolution, provided that the set of available Horn clauses is continuously adapted to the context in which the proof is conducted. This rests on a mechanism that provides a simple clausal interpretation for the assumptions in a context. The method is not complete, due to the expressive power of type theory. Although the provided inference steps suggest certain search strategies (*tactics*), these issues are outside the scope of this paper. A proof environment based on the method, named *Constructor*, has been

developed within Esprit project 1222: "Genesis" [18, 19]. *Constructor* implements the method for generalised type systems, a generalisation of the Calculus of Constructions used in this paper. Experiments with this system demonstrate the power and efficiency of the method.

Here is an overview of the paper:

- Section 2 summarizes the Calculus of Constructions, the specific variant of Constructive Type Theory that will be used to explain the inference method.
- Section 3 presents the inference method.
- Section 4 discusses completeness of the proposed method.
- Section 5 illustrates the method by means of a full derivation example.
- Section 6 describes the *Constructor* proof development environment.
- Section 7 discusses related work.
- Section 8 concludes with a discussion.

2. Constructive Type Theory (CTT)

We assume familiarity with Constructive Type Theory [6, 8, 9, 14, 22], a variant of the *propositions as types* paradigm. This section describes the particular system of interest. We shall use a system developed by Coquand, a version of the Calculus of Constructions. It must be emphasized that this is just one of the possible variants for which the presented proof construction method is applicable.

2.1. Terms

The syntactic formation rules for the terms in the system are defined as:

- constant, viz. one of $\{prop, type, kind\}$.
- variable, denoted by an identifier.
- $\lambda x{:}A.B$, typed abstraction, where A and B are terms, and x a variable.
- $\Pi x{:}A.B$, generalized cartesian product of types B indexed over x of type A, where A and B are terms, and x a variable.
- $(A\ B)$, application, where A and B are terms (function and argument). Application associates to the left, so we will write $(a\ b\ c)$ for $((a\ b)\ c)$.

A *typing* is a construction of the form $[t{:}T]$, where t and T are terms. The intuition behind this is that T is the type of t. Types may have types themselves, and we will refer to such expressions as domains. Four levels of expressions are distinguished in the hierarchy of types: 0-, 1-, 2-, and 3-expressions, where n-expressions serve as types of $(n+1)$-expressions $(n=0, 1, 2)$. There is only one 0-expression: the constant "supertype" *kind*. We introduce two predefined constants as primitive 1-expressions (kinds) of the system: the kind *type*, the set containing all "plain" types (2-expressions), and the kind *prop*, that plays an identical role and is treated similarly, but which is inhabited by propositions (2-expressions). The level of

3-expressions contains objects and proofs. In many versions of the Calculus of Constructions, *type* and *prop* are identified (usually denoted "★" or "*Type*"). Here we will explicitly distinguish between them to avoid possible confusion. This is however not essential to the formalism.

2.2. Correctness

In this section, we will give type rules describing what correct terms are. To this end, a notion of contexts is needed first.

A *context* is a list of assumptions and definitions:

$$\text{assdef}_1, \ldots, \text{assdef}_n \quad (n \geq 0)$$

An *assumption* is of the form $[x_i:A_i]$ and introduces x_i as a variable of type A_i. A *definition* is of the form $[x_i \equiv a_i:A_i]$ and introduces x_i as an abbreviation of the correct term a_i of type A_i. Variables may be declared at most once. The free variables occurring in a_i and A_i must have been declared earlier in the context, i.e., $\mathcal{FV}(A_i) \subseteq \{x_1, \ldots, x_{i-1}\}$ and $\mathcal{FV}(a_i) \subseteq \{x_1, \ldots, x_{i-1}\}$.

We will use Γ as a metavariable over contexts. Well-formedness of a context Γ will be denoted *well-formed*(Γ). We will write $\Gamma, [x:A]$ to denote the context Γ extended with the assumption $[x:A]$, and Γ_1, Γ_2 for the concatenation of contexts Γ_1 and Γ_2. The symbol "$\emptyset$" represents the empty context. A *sequent* is an expression of the form $\Gamma \vdash [a:A]$, denoting that $[a:A]$ is a correct typing in the well-formed context Γ. The predicate *constant*(x) denotes that x is one of the constants *type*, *prop* or *kind*. We will write $B[a/x]$ to denote substitution of the term a for the free occurrences of the variable x in the expression B. We will write "$=_{\beta\delta}$" to denote the transitive reflexive closure of β- and δ-reduction. β-reduction corresponds to the usual notion in typed lambda-calculus, and δ-reduction denotes local expansion of definitions that have been introduced in the context. Note that both β- and δ-reduction are context-dependent. We assume equality modulo α-conversion whenever necessary. This can be achieved by using De Bruijn indices [5] or Barendregt's variable convention [2].

Well-formedness of contexts and correct typing of terms is inductively defined as:

[G]	*well-formed*($\emptyset$)	(empty)
[1a]	If *well-formed*(Γ) then $\Gamma \vdash [\textit{type}:\textit{kind}]$.	(kinds)
[1b]	If *well-formed*(Γ) then $\Gamma \vdash [\textit{prop}:\textit{kind}]$.	
[2a]	If $\Gamma \vdash [A:K]$ and *constant*(K) then *well-formed*($\Gamma, [x:A]$).	(introduction)
[2b]	If $\Gamma \vdash [a:A]$ then *well-formed*($\Gamma, [c \equiv a:A]$).	
[3a]	If *well-formed*(Γ) and $[x:A] \in \Gamma$, then $\Gamma \vdash [x:A]$.	(selection)
[3b]	If *well-formed*(Γ) and $[c \equiv a:A] \in \Gamma$, then $\Gamma \vdash [c:A]$.	
[4]	If $\Gamma, [x:A] \vdash [b:B]$ then $\Gamma \vdash [\lambda x:A.b:\Pi x:A.B]$.	(λ-abstraction)
[5]	If $\Gamma, [x:A] \vdash [B:K]$ and *constant*(K) then $\Gamma \vdash [\Pi x:A.B:K]$.	(Π-abstraction)

[6] If $\Gamma \vdash [a:A]$ and $\Gamma \vdash [b:\Pi x{:}A.B]$ then $\Gamma \vdash [(b\,a):B[a/x]]$. (application)
[7] If $\Gamma \vdash [a:A]$ and $\Gamma \vdash [B:K]$ and *constant*(K) and $A =_{\beta\delta} B$ then $\Gamma \vdash [a:B]$. (type conversion)

Details and properties of the described system can be found in [8, 9, 17, 22, 24, 3].

2.3. *Interpretation and use*

If, for a dependent product type $\Pi x{:}A.B$, x does not occur free in B ($x \notin \mathcal{FV}(B)$), the type simplifies, as usual, to the ordinary *function type* $A \to B$. In case A:*prop*, A is considered a proposition and a typing $[a:A]$ is interpreted as: a is a proof for A, i.e., a proposition plays the role of the type of its proofs. This means that a proposition is considered valid if and only if it is inhabited. The system described contains intuitionistic higher order predicate logic. For example, if B:*prop*, then $\Pi x{:}A.B$ can be interpreted as the universally quantified proposition $\forall x{:}A.B$. If x does not occur free in B and A:*prop* and B:*prop*, then $\Pi x{:}A.B$ can be interpreted as the intuitionistic implication $A \Rightarrow B$. The Calculus of Constructions formalism thus provides a definition language that can be and has been used to formalize and mechanically verify many parts of mathematics. Texts in this language are written in the form of *theories* (*books* in AUTOMATH terminology). Theories are CTT contexts. For a field of interest, assumptions allow the axiomization of the primitive notions, whereas the definitions allow abbreviation of derived notions like lemmas.

3. CTT proof construction method

It is a well-known fact, that correctness of sequents $\Gamma \vdash [t:T]$ (t has type T in context Γ) in Constructive Type Theory and related systems is decidable, even feasibly decidable, and several proof checkers exist that mechanically determine correctness for given CTT theories [23, 10, 8]. For a proof construction system, the objective is not to verify whether a given object has a certain given type, but, for a given type, to attempt construction of an inhabitant of this type. More precisely: given a context (theory) Γ and a type A, the objective is to construct an object p such that $\Gamma \vdash [p:A]$. For propositions, this corresponds to finding a proof object.

The central problem with goal directed proof construction in Constructive Type Theory is that direct backward chaining with the correctness rules of Section 2.2 is hardly possible. Therefore, the approach is to extract from the given formalization a sound set of derived rules, that do allow easy backward inference. These derived rules then serve as the primitive proof steps of the system.

3.1. *Horn clause derivations*

In the method, CTT sequents will be derived using Horn clause resolution. In goal directed proving, the idea is to start off with a goal to be proven, and to replace goals by appropriate subgoals by resolution with inference rules [31]. Horn clause

inference rules consist of a (possibly empty) set of antecedents $S_1 \ldots S_k$ and one conclusion S. Horn clauses will be denoted: $S \Leftarrow S_1 \ldots S_k$. In the method, antecedents and conclusions will all be CTT sequents. Sequents may contain logical variables over CTT terms. To avoid confusion with CTT variables, we will denote logical variables by identifiers prefixed with a "#" symbol. Logical variables are considered to be universally quantified over clauses. A term is grounded if it does not contain logical variables. A context will be called grounded if it contains grounded terms only. The meaning of a Horn clause is that *if* instantiations for the logical variables can be found, such that the antecedent sequents are correct, *then* the associated consequent is a correct sequent. A Horn clause will be considered *valid* if its meaning is correct with respect to the correctness rules of CTT. Unification determines how two rules can be combined into derivations. A derivation for a sequent is either a Horn clause concluding that sequent, or it is a derivation where an antecedent sequent S_i is replaced by the antecedents of a Horn clause concluding S_i', after unifying S_i and S_i'. If no antecedents are left in a derivation, the derivation is complete and serves as a justification for the correctness of its conclusion sequent. Derivations correspond to derived Horn clauses and have the same interpretation.

The queries considered consist of one goal of the form $\Gamma \vdash [\#P{:}A]$, where Γ must be a grounded, well-formed context, A must be a grounded, correct domain, and $\#P$ is a logical variable. For a given query S, the derivation process starts with the trivial derivation "$S \Leftarrow S$", which is obviously valid. The objective is to transform this derivation by resolution with valid Horn clauses, until the derivation "$S' \Leftarrow$" is reached. S' is then a correct instance of S.

Derivations and Horn clauses will be of the form: $\Gamma \vdash [p{:}A] \Leftarrow \Gamma_1 \vdash [p_1{:}A_1] \ldots \Gamma_n \vdash [p_n{:}A_n]$. The following invariant properties will hold for all derivations (but not necessarily for Horn clauses):

1. All contexts Γ and $\Gamma_1 \ldots \Gamma_n$ will be grounded and well-formed.
2. $\Gamma \subseteq \Gamma_1, \ldots, \Gamma \subseteq \Gamma_n$.
3. For any logical variable $\#P$ occurring in the type field A_i of a subgoal $\Gamma_i \vdash [p_i{:}A_i]$, $\#P \in \{p_1 \ldots p_{i-1}\}$. If an object field p_i of a subgoal $\Gamma_i \vdash [p_i{:}A_i]$ is not a logical variable, then for any logical variable $\#P$ occurring in p_i, $\#P \in \{p_1 \ldots p_{i-1}\}$.
4. For any logical variable $\#P$ occurring in the object field p of the conclusion $\Gamma \vdash [p{:}A]$, $\#P \in \{p_1 \ldots p_n\}$.

The first property reflects the fact that construction always takes place within a known context. The second property states that contexts can be extended during backward proving. The third property ensures that logical variables are "introduced before use". The fourth property guarantees that the conclusion of a derivation will be grounded when all subgoals have been solved (the type field A of a derivation conclusion is grounded from the start). For our queries $\Gamma \vdash [\#P{:}A]$ this implies that an object $\#P$ of the requested type A has been constructed. Note that the trivial derivations that correspond to our queries of interest have all the required properties.

It turns out that we can avoid a problem that usually arises with inference rules over sequents, viz. unification over contexts. The reason for this is that in contrast to general purpose higher order theorem provers as presented in e.g. [27, 28 and 13], the method inferences at the object level, not at the meta-level. This is possible because the method is specialized for type theory only; it is not a generic inference method over arbitrary sequents. Contexts will be treated in a special way, and it is sufficient to unify over typings. Our goals, that denote CTT sequents, will be regarded as tuples with a typing and a context.

For the method to work, it is necessary that contexts occurring in derivations are grounded, so that (1) we do not unify over contexts at all, and (2) we can extract the necessary object level horn clauses directly from contexts. The exact consequences of this restriction will be alluded to later.

In the subsequent sections, valid Horn clauses will be derived. These Horn clauses will not violate the given invariant properties for derivations. Some of the Horn clauses correspond directly to correctness rules for type theory and are of interest to all subgoals in a derivation. The problem is that no suitable Horn clause can be given for the application rule (rule 6), on account of the substitution. A solution for this problem is presented, that rests on a mechanism to provide a direct clausal interpretation for the assumptions in a context. The Horn clauses thus obtained cover derivation steps that can construct the necessary application terms. This is the crux of the method. For any given subgoal $\Gamma_i \vdash [p_i{:}A_i]$, this mechanism, when applied to the context Γ_i, allows derivation of a set of valid Horn clauses that are candidates for resolution on this particular subgoal. A proposal for this mechanism was first suggested by Ahn [1].

3.2. Unification and type conversion

Unification determines whether a clause is applicable in a given situation. Unification will also handle the type conversion rule (rule 7), dealing with equality of types. It is important to observe that it is sufficient to provide unification over typings, not over contexts (although context information is of course relevant for β- and δ-reductions during unification). Unifying typings $[P{:}A]$ and $[P'{:}A']$ will be achieved by unifying the objects P and P' and subsequently unifying the types A and A'. For types, the unification is with respect to β-and δ-equality. Although β-equality for objects is not explicit in the correctness rules, it is also desirable to identify β- (and δ-) equivalent terms. This is justified by the *closure under reduction* property, and corresponds to proof normalization [8, 10, 16]. Because we unfold derived clauses completely, it is desirable to augment object unification with outermost η-equality, to ensure reachability of objects in η-normal form.

Unification for expressions in typed λ-calculus with respect to α, β and possibly η-conversion requires complete higher order unification. For simply typed λ-calculus this problem has a possibly infinite set of solutions and is known to be semidecidable, in the sense that if two terms do not unify, search algorithms for unifiers may diverge [21]. [21] also gives a complete algorithm for this problem in simply typed λ-calculus.

In the more complicated case of the Calculus of Constructions, where types are also terms in typed λ-calculus, it is not yet completely clear. Elliott [12] and Pym [30] have independently extended Huets algorithm to dependent types for the logical framework LF (see also [29]). Dowek [11] is working towards extending the algorithm up the Barendregt "cube" [3], a theory that gives a classification of type systems and that includes the Calculus of Constructions. It is too early to report results. For implementations of the method, sound approximations for higher order unification can always be used. Such approximations can be very usable in practice (Section 6 gives an example) although they affect completeness, of course.

3.3. Kinds rule

The *kinds* rule (rule 1) is directly equivalent to the valid Horn clauses:

$$\Gamma \vdash [type{:}kind] \Leftarrow.$$

$$\Gamma \vdash [prop{:}kind] \Leftarrow.$$

For any subgoal $\Gamma_i \vdash [p_i{:}A_i]$, such a rule applies if $[p_i{:}A_i]$ unifies with $[type{:}kind]$ or $[prop{:}kind]$. Γ is not treated as a logical variable. Because contexts in our derivations are always grounded and well-formed, the well-formedness check on Γ_i (required in rule 1) is needless.

3.4. Lambda abstraction rule

The lambda abstraction rule (rule 4) corresponds to the valid Horn clause:

$$\Gamma \vdash [\lambda x{:}\#A.\#B : \Pi x{:}\#A.\#T] \Leftarrow \Gamma, [x{:}\#A] \vdash [\#B{:}\#T].$$

The typing of a goal of the form $\Gamma_i \vdash [P{:}\Pi x{:}A.T]$ may be unified with the typing in the conclusion of this rule, unifying P with $\lambda x{:}\#A.\#B$ and resulting in the new stripped subgoal $[\#B{:}\#T]$, to be solved in the context Γ_i extended with the typing $[x{:}\#A]$. Γ does not play the role of a logical variable. To ensure that this new context is grounded and well-formed, the restriction is imposed that A must be grounded and correct domain, thus preventing logical variables over domains to be introduced in the context. For example, this rule cannot be used to find a proof for an implication $\#A \Rightarrow B$, because this would introduce an unknown assumption $[p{:}\#A]$ in the context.

3.5. Pi abstraction rule

The pi abstraction rule (rule 5) corresponds to the valid Horn clause:

$$\Gamma \vdash [\Pi x{:}\#A.\#B : \#K] \Leftarrow \Gamma, [x\#A] \vdash [\#B{:}\#K].$$

For goals of the form $\Gamma_i \vdash [\Pi x{:}A.B{:}K]$, application of this rule results, after unification of typings, in a stripped subgoal $[B{:}K]$, to be solved in the context Γ_i extended with the typing $[x{:}A]$. K must be a constant. Though this check has to

be postponed if K is not grounded, it can be demonstrated that this need never occur, and K must be either *type* or *prop*. To ensure that the new context is grounded and well-formed, again the restriction is imposed that A must be a grounded and correct domain. For example, this prevents using this rule on a goal $\Gamma_i \vdash [\#P{:}prop]$.

3.6. Derived clauses

The application rule (rule 6) cannot be translated directly to a valid Horn clause, on account of the substitution. A solution is offered, for which the following theorem is essential:

Correctness of a sequent of the form

$$\Gamma \vdash [C : \Pi x_1 : A_1 \ldots \Pi x_n : A_n . B]$$

is equivalent to the validity of the Horn clause:

$$\begin{aligned}\Gamma, \Gamma' \vdash [(C\ \#x_1 \ldots \#x_n) : B[\#x_1/x_1, \ldots, \#x_n/x_n]] \Leftarrow\\ \Gamma, \Gamma' \vdash [\#x_1 : A_1]\\ \Gamma, \Gamma' \vdash [\#x_2 : A_2[\#x_1/x_1]]\\ \ldots\\ \Gamma, \Gamma' \vdash [\#x_n : A_n[\#x_1/x_1, \ldots, \#x_{n-1}/x_{n-1}]].\end{aligned}$$

where $\#x_1, \ldots, \#x_n$ are the logical variables of the Horn clause. The context Γ, Γ' denotes any well-formed context extension of Γ. Note that all possible occurrences of the CTT variables x_i have been replaced by corresponding logical variables $\#x_i$. For a complete proof of this theorem the reader is referred to [18]. For $n = 0$, the set of premises is empty and the application in the consequent simplifies to the object C. The selection rule (rule 3a and 3b) justifies that the theorem is in particular applicable to all introductions and definitions occurring in any well-formed context Γ, i.e.

For all CTT variables c, if

$$[c : \Pi x_1 : A_1 \ldots \Pi x_n : A_n . B] \in \Gamma$$

or

$$[c \equiv C : \Pi x_1 : A_1 \ldots \Pi x_n : A_n . B] \in \Gamma$$

this theorem guarantees that:

$$\begin{aligned}\Gamma \vdash [(c\ \#x_1 \ldots \#x_n) : B[\#x_1/x_1, \ldots, \#x_n/x_n]] \Leftarrow\\ \Gamma \vdash [\#x_1 : A_1]\\ \Gamma \vdash [\#x_2 : A_2[\#x_1/x_1]]\\ \ldots\\ \Gamma \vdash [\#x_n : A_n[\#x_1/x_1, \ldots, \#x_{n-1}/x_{n-1}]].\end{aligned}$$

This result is now used to interpret the introductions and definitions in a context in this clausal form, thus providing the possible application candidates. The idea

is to "unfold" the top level Π-abstractions for context elements to clauses. For any goal $\Gamma \vdash E$, all clauses thus obtained from the grounded context Γ are available as valid Horn clauses for resolution on this goal. It is important to note that the context Γ is the same for the consequent and the antecedents of the Horn clauses thus obtained. Because the context is not affected by application of these clauses, it is sufficient to resolve the typing E of a goal with the typing of the consequent of the clause, and pass the context Γ directly to the antecedents. Henceforth, we will omit the context parameter Γ for these clauses.

Although for a given context element of type $\Pi x_1 : A_1 \ldots \Pi x_n : A_n . B$ (where B is not itself a Π-abstraction) the theorem gives $n+1$ different valid Horn clauses, it is sufficient to provide the completely unfolded clause with n antecedents. If the clause has been unfolded too far to unify directly with a sequent, this can be compensated by first resolving the sequent with the lambda abstraction rule. The small price to pay is that the resulting proofs may contain η-redexes. If desired, these can be reduced immediately. See also Section 5.

Note that this mechanism now covers both the application rule (rule 6) and the selection rule (rule 3). Rule 2 (context introduction) is handled by the λ-abstraction rule and the Π-abstraction rule. Note that these rules are the only ones that can extend contexts.

4. Completeness

An interesting question is whether the method is complete in the sense that a top down derivation can be constructed for all correct inhabitants (modulo object conversion) of a given type (checking is complete). Note that completeness is of course determined by the completeness of the higher-order unification procedure. But what exactly are the consequences of the restriction that we impose on the context, viz. that it is always grounded?

We already saw that our queries of interest are not affected by the restriction. Now consider the effect of the restriction during the inferencing process. It should be clear that only the lambda rule and the pi rule are affected by the restriction, as they may extend a context during resolution. For issues related to completeness (at least in a non-deterministic sense), the following observation is important: due to the third invariant property on derivations, we only need to consider goals where the type field of the conclusion is grounded, because any logical variable $\#P$ occurring there can be instantiated by first solving the associated goal where $\#P$ is introduced. This implies that the partiality of the lambda-abstraction rule poses no fundamental restrictions, because it can be circumvented by postponing resolution on the goal in question. It is clear that the restriction on the applicability of the pi-abstraction rule poses real limitations: it explicitly restricts querying for arbitrary 2-expressions, i.e., it refuses to enumerate all possible Π-abstracted propositions or types. For instance, not all solutions $\#A$ to a subgoal of the form $\Gamma \vdash [\#A{:}prop]$ will be

constructed. This is related to a fundamental problem. The expressive power of CTT is such, that it does allow the construction of proofs that involve e.g. induction loading, where some stronger proposition is needed to construct a proof for a weaker one. The resulting proofs will not have the subformula property. Automatic top down construction of such proofs is unattainable in general, because an 'oracle' is needed. Since it is possible to enumerate all propositions (or types), the method can be made complete, in a non-deterministic sense, by replacing the pi rule by an enumerator algorithm. This approach is however pointless for practical purposes. The interactive assistance of users is essential for such proofs.

In summary, the method as presented is not complete. However, the restriction only affects construction of propositions and types. This is directly related to a fundamental problem in theorem proving.

5. Derivation example

As an example, consider the following correct theory Γ_0:

$[nat : type]$,

$[0 : nat]$,

$[s : \Pi x{:}nat.\ nat]$,

$[< : \Pi x{:}nat.\ \Pi y{:}nat.\ prop]$,

$[axiom\ 1 : \Pi x{:}nat.\ (< x\ (s\ x))]$,

$[trans : \Pi x{:}nat.\ \Pi y{:}nat.\ \Pi z{:}nat.$

$\quad \Pi p{:}(< x\ y).\ \Pi q{:}(< y\ z).\ (< x\ z)]$,

$[ind : \Pi p : \Pi x{:}nat.\ prop.$

$\quad \Pi g : (p\ 0).$

$\quad \Pi h : \Pi n{:}nat.\ \Pi hyp{:}(p\ n).\ (p\ (s\ n)).$

$\quad \Pi z : nat.\ (p\ z)]$,

$[pred1 \equiv \lambda x{:}nat.\ (< 0\ (s\ x)) : \Pi x{:}nat.\ prop]$

To elucidate the method, a top-down derivation is now presented for the theorem $\forall y{:}nat.\ (pred1\ y)$, i.e., a proof object $\#P$ is sought, such that $\Gamma_0 \vdash [\#P : \Pi y{:}nat.\ (pred1\ y)]$. The associated trivial derivation for the query is:

$$\Gamma_0 \vdash [\#P : \Pi y{:}nat.\ (pred1\ y)] \Leftarrow \Gamma_0 \vdash [\#P : \Pi y{:}nat.\ (pred1\ y)].$$

The Horn clauses available for resolution are:

$$\Gamma \vdash [\lambda x{:}\#A.\#B : \Pi x{:}\#A.\#T] \Leftarrow \Gamma, [x{:}\#A] \vdash [\#B{:}\#T].$$

$$\Gamma \vdash [\Pi x{:}\#A.\#B : \#K] \Leftarrow \Gamma, [x{:}\#A] \vdash [\#B{:}\#K].$$

viz. the lambda abstraction rule and the pi abstraction rule (the kinds rule is of no relevance to this example). These Horn clauses are always available for goals. For

a given goal, they are extended with Horn clauses that can be obtained from the unfolded context elements of the goal. For the antecedent in the given trivial derivation this means:

$$[nat : type] \Leftarrow .$$

$$[0 : nat] \Leftarrow .$$

$$[(s\ \#X) : nat] \Leftarrow [\#X : nat].$$

$$[(<\#X\ \#Y) : prop] \Leftarrow [\#X : nat][\#Y : nat].$$

$$[(axiom1\ \#X) : (<\#X\ (s\ \#X))] \Leftarrow [\#X : nat].$$

$$[(trans\ \#X\ \#Y\ \#Z\ \#P\ \#Q) : (<\#X\ \#Z)] \Leftarrow$$

$$[\#X : nat][\#Y : nat][\#Z : nat]$$

$$[\#P : (<\#X\ \#Y)][\#Q : (<\#Y\ \#Z)].$$

$$[(ind\ \#P\ \#G\ \#H\ \#Z) : (\#P\ \#Z)] \Leftarrow$$

$$[\#P : \Pi x : nat.\ prop][\#G : (\#P\ 0)]$$

$$[\#H : \Pi n : nat.\ \Pi hyp : (\#P\ n).\ (\#P\ (s\ n))][\#Z : nat].$$

$$[(pred1\ \#X) : prop] \Leftarrow [\#X : nat].$$

where Γ_0 has been omitted from the sequents. Similar clauses can be constructed for extensions of Γ_0. The only rule applicable to our derivation is the lambda abstraction rule. Resolution gives:

$$\Gamma_0 \vdash [\#P : \Pi y : nat.\ (pred1\ y)] \Leftarrow \Gamma_0, [y : nat] \vdash [\#P' : (pred1\ y)].$$

instantiating $\#P$ to $\lambda y : nat.\#P'$. In the extended context of the subgoal, a derived clause for y ($[y : nat] \Leftarrow .$) is now available. Resolution with the induction clause (*ind*) from the context gives:

$$\Gamma_0 \vdash [\#P : \Pi y : nat.\ (pred1\ y)] \Leftarrow$$

$$\Gamma_0, [y : nat] \vdash [pred1 : \Pi x : nat.\ prop]$$

$$\Gamma_0, [y : nat] \vdash [\#G : (pred1\ 0)]$$

$$\Gamma_0, [y : nat] \vdash [\#H : \Pi n : nat.\ \Pi hyp : (pred1\ n).\ (pred1\ (s\ n))]$$

$$\Gamma_0, [y : nat] \vdash [y : nat].$$

instantiating $\#P'$ to (*ind* $\#P''\ \#G\ \#H\ \#Z$), $\#P''$ to *pred1*, and $\#Z$ to y. Note that this

requires higher order unification. The first subgoal is grounded and can be checked, but this goal can also be solved after resolution with the lambda abstraction rule, provided that the unification knows that *pred1* is equivalent to $[x{:}nat](pred1\ x)$ (outermost η-equivalence). The last subgoal is solved directly with the Horn clause for y from the context. The derivation thus becomes:

$$\Gamma_0 \vdash [\#P{:}\Pi y{:}nat.\,(pred1\ y)] \Leftarrow$$

$$\Gamma_0, [y{:}nat] \vdash [\#G{:}(pred1\ 0)]$$

$$\Gamma_0, [y{:}nat] \vdash [\#H{:}\Pi n{:}nat.\,\Pi hyp{:}(pred1\ n).(pred1\ (s\ n))].$$

The first subgoal resolves with the Horn clause for *axiom1* from the context, instantiating $\#G$ to $(axiom1\ 0)$ and leaving $[0{:}nat]$ as trivial subgoal that can be resolved immediately. The remaining subgoal is stripped twice with the lambda abstraction rule. The derivation is now:

$$\Gamma_0 \vdash [\#P{:}\Pi y{:}nat.\,(pred1\ y)] \Leftarrow \Gamma_1 \vdash [\#H'{:}(pred1\ (s\ n))].$$

instantiating $\#H$ to $\lambda n{:}nat.\lambda hyp{:}(pred1\ n).\,\#H'$. Γ_1 stands for Γ_0, $[y{:}nat]$, $[n{:}nat]$, $[hyp{:}(pred1\ n)]$. The remaining proof obligation is now resolved with the context clause for transitivity (*trans*):

$$\Gamma_0 \vdash [\#P{:}\Pi y{:}nat.\,(pred1\ y)] \Leftarrow \Gamma_1 \vdash [0{:}nat]$$

$$\Gamma_1 \vdash [\#Y{:}nat]$$

$$\Gamma_1 \vdash [(s\ (s\ n)){:}nat]$$

$$\Gamma_1 \vdash [\#P1{:}(<0\ \#Y)]$$

$$\Gamma_1 \vdash [\#Q{:}(<\#Y\ (s\ (s\ n))].$$

instantiating $\#H'$ to $(trans\ 0\ \#Y\ (s\ (s\ n))\ \#P1\ \#Q)$. The first and third subgoal are eliminated with context clauses from Γ_1 for *0*, *s* and *n*. Because these subgoals are grounded, this amounts to checking. Resolving the last subgoal with *axiom1* instantiates $\#Q$ to $(axiom1\ (s\ n))$ and $\#Y$ to $(s\ n)$. The derivation has become:

$$\Gamma_0 \vdash [\#P{:}\Pi y{:}nat.\,(pred1\ y)] \Leftarrow \Gamma_1 \vdash [(s\ n){:}nat]$$

$$\Gamma_1 \vdash [\#P1{:}(<0\ (s\ n))]$$

$$\Gamma_1 \vdash [(s\ n){:}nat].$$

The proof is completed with the context clauses for *s*, *n* and *hyp*. The complete

proof $\#P$ is now:

$\lambda y{:}nat.(ind\ pred1$

$(axiom1\ 0)$

$\lambda n{:}nat.\lambda hyp{:}(pred1\ n).$
$(trans\ 0\ (s\ n)\ (s\ (s\ n))\ hyp\ (axiom1\ (s\ n)))$

$y).$

Note that this proof is an η-redex. This is due to the fact that derived clauses are unfolded as far as possible here, thus constructing applications that are provided with the full number of arguments. If outermost η-conversion of objects is provided, the η-normal proof can also be derived.

6. The *Constructor* proof environment

This section gives a short description of an interactive proof environment, named *Constructor*, that implements an inference machine based on the described method. The machine enforces correctness of proof construction in generalised type systems, including the Calculus of Constructions. The mouse-based interface has been built using the *Genesis* system, the tool generator that resulted from Esprit project 1222. Details of the *Constructor* system can be found in [18–20]. Here, we will explain its most important features.

When using *Constructor*, there is always a global context present, which is the theory that formalizes a domain of interest. A proof editor is provided in which conjunctions of queries can be posed, and that admits application of correct proof steps. Queries are interpreted in the global context. Queries are typings of the form $[A{:}B]$. Figures 1 and 2 present some screen images of the system (for the syntax used in the figures see Section 6.1).

Both interactive user-guided inference and automatic search are possible and may intermingle. In interactive mode the user may, for a selected goal, choose a clause from a menu with resolution candidates. Optionally, the system checks instantiated subgoals that may arise after resolution steps. Currently only one default search strategy or *tactic* (*tactical* in LCF terminology) is present for automatic search. It uses a consecutively bounded depth-first search strategy. The maximum search depth must be specified interactively. Alternative solutions are generated upon request. Facilities to "undo" user or tactic choices are provided. The resolution method itself is used (by way of bootstrapping) for correctness checking of contexts and local context introductions. Completed derivations may be added to the global context as lemmas. To this end, they must be given a name and will be available for use in subsequent queries. It is possible to "freeze" definitions, i.e., to hide their contents and treat them as axioms. In case of clash of variable names (α-clash), unique

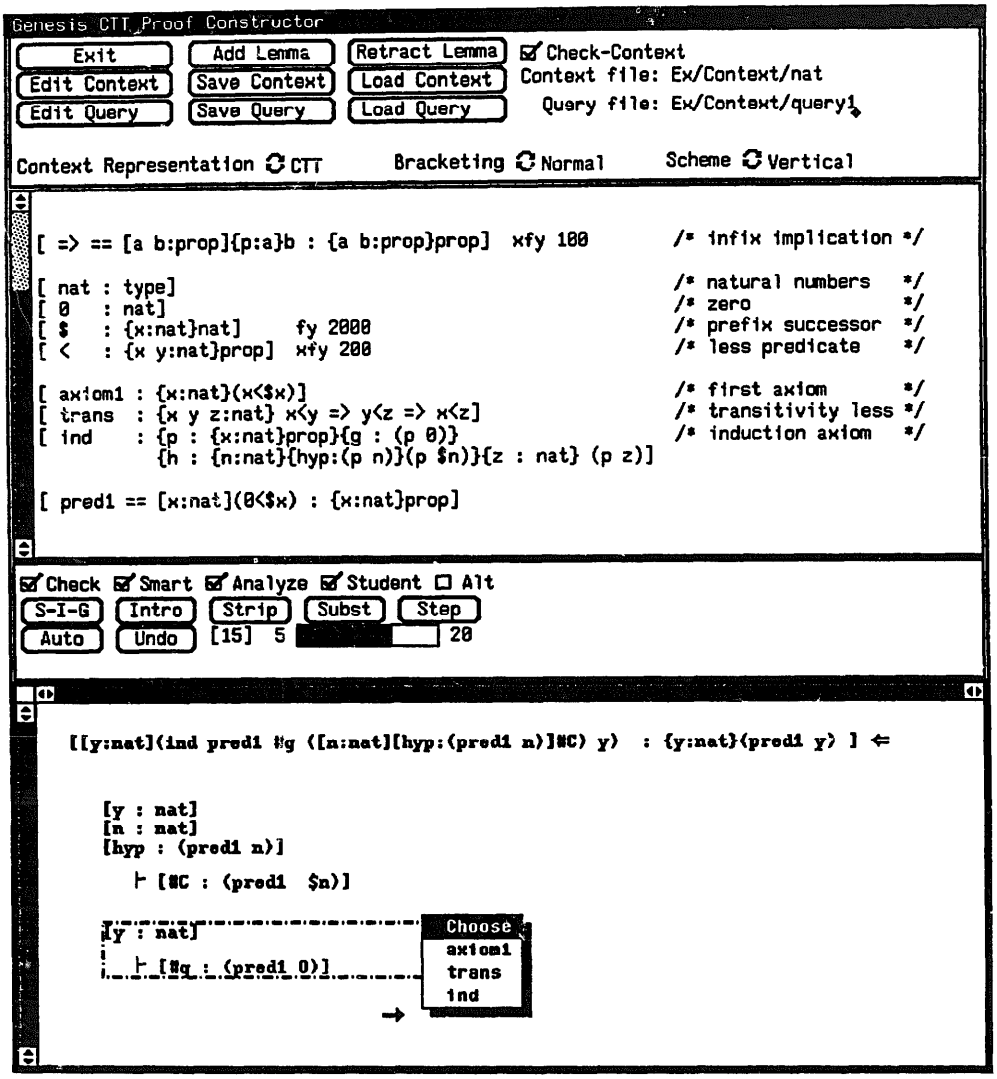

Fig. 1. Here, we are in the middle of an interactive session, solving the query from the example in the previous section. The second window contains the global context. For each goal, candidate clauses for resolution can be selected from a menu. The proof under construction is collected in the head of the derivation. Note the different local context extensions for the different subgoals.

variable names are generated by numbering. Textual editing of theories and queries is provided in the environment itself.

The special handling of contexts can be implemented efficiently. For example, translation of context elements to clauses only needs to be done once, because the main theorem guarantees that clauses remain valid in extended contexts (this is due to the fact that CTT is monotonic). Verifying well-formedness of contexts can be done incrementally. Contexts can be shared amongst goals.

Apart from the application rule (rule 6) and the selection rule (rule 3), which are handled by the method proposed, the *Constructor* system is parameterized with respect to the correctness rules for type theory, i.e., the system can handle different

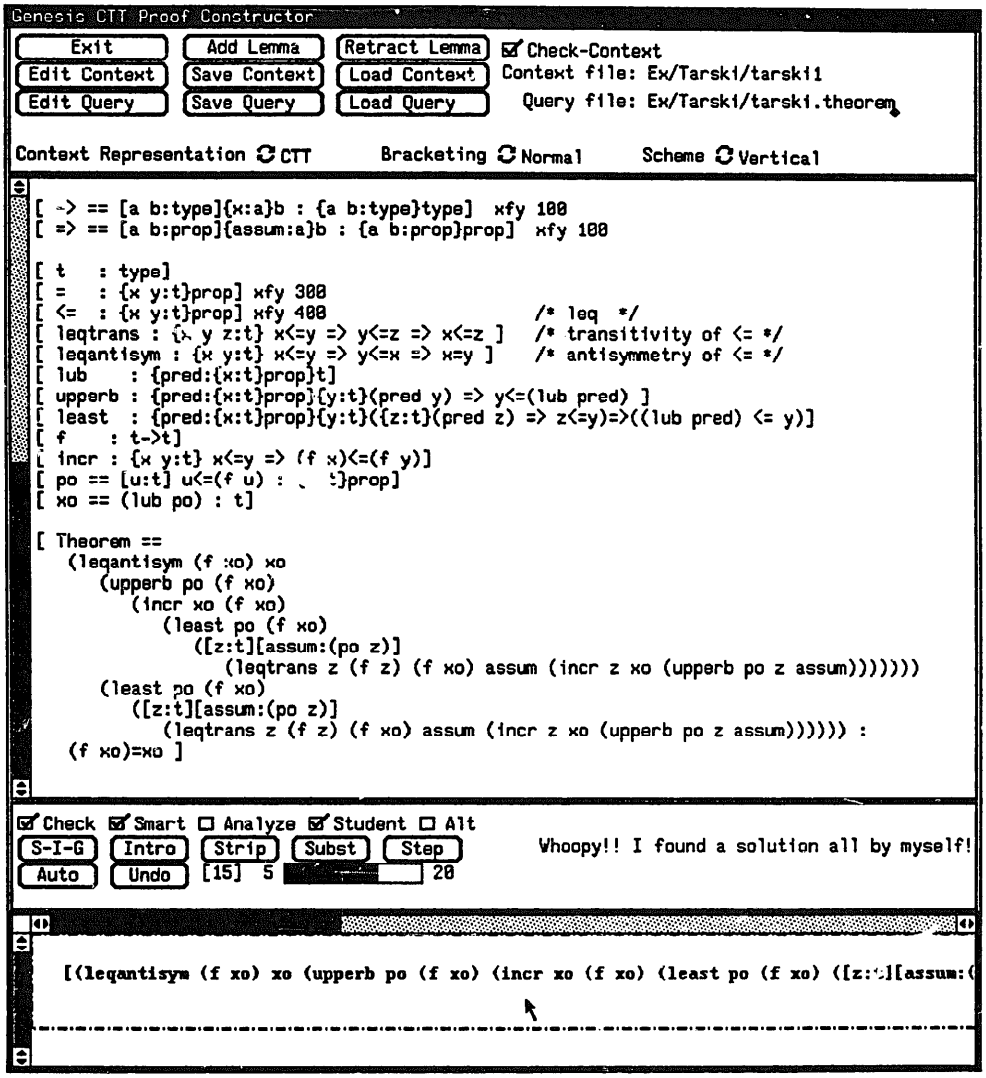

Fig. 2. Proving Tarski's Lemma [22]. After automatically constructing a proof of the fixed point property for the given witness, the proof has been added to the context as a definition (named "Theorem"). The proof is conducted in a version of the Calculus of Constructions. The associated completed derivation in the bottom window is only partly visible here. The search strategy confirms that the proofs given by Huet [22] are indeed the shortest proofs.

versions and extensions of type theory. For example, it poses no problems to interpret various dialects of AUTOMATH [6, 10].

6.1. Technical details

In *Constructor*, typed abstraction is denoted $[x{:}A]B$, whereas typed product is denoted as $\{x{:}A\}B$. Multiple variable introductions are permitted, e.g. $[x\,y{:}A]B$ denotes $[x{:}A][y{:}A]B$. Variables and definitions that are introduced can be declared

as fix operators, much like in a Prolog fashion. For example,

"[⇒ = = [*a b:prop*]{*p:a*}*b*:{*a b:prop*}*prop*] *xfy 100*"

declares "⇒" (implication) as a right associative infix operator with priority level *100*. Application is treated as a left-associative infix operator. Bracketing is used in the usual way to over-rule priorities. Logical variables are prefixed with a "#" symbol.

The built-in unification procedure implements a simple approximation of higher order unification with the following characteristics:

- *Higher order structural matching*

First order unification where logical variables for functors match structurally, e.g. "(#F 0)" unifies with "(suc 0)" yielding unifier #F = suc.

- *Alpha conversion*

The unification is modulo the name of bound variables, e.g. "[x:nat]x" equals "[y:nat]y".

- *Beta conversion*

The built in unification procedure will reduce β-redexes if necessary. To ensure that the reduction is always sound, a goal is added to demonstrate that the argument will have the required domain type, in the context in question.

- *Delta conversion*

The built in unification procedure will do δ-reduction on definitions if necessary, i.e., it may expand abbreviating names.

The unification also recognizes outermost η-equality for objects, so that it can use the lambda abstraction rule to verify given application objects where the functor has not been provided with the full number of arguments. This can be regarded as the inverse operation of "unfolding" Π-abstractions to clauses. The implemented unification procedure will always yield at most one unifier. If complicated higher order unification is required, two options are available: (1) Provide appropriate auxiliary definitions to obtain the desired result (cf. *pred1* in the derivation example), (2) Interactively substitute a template of the desired proof object by hand. Checking objects (also those that cannot be constructed by the unification) is always possible, because all relevant terms are known then.

The *Constructor* system automatically proves the example from the previous section without delay. As another example, the system constructs the proof for Tarski's Lemma as formalised by Huet [22], a famous example from constructive mathematics, either by first proving the lemmas as proposed in [22], or by direct automatic construction of the complete proof. The example runs within seconds. The search strategy confirms that the proofs given by Huet [22] are indeed the shortest proofs. See also Fig. 2. As an example of a large proof (1500 lines), the system has been used to interactively construct a proof of Girard's paradox, formalising a proof in [4]. This proof was conducted in a GTS known as λU. The system's contribution to this effort was considerable. All lemmas were constructed top-down,

in close cooperation between user and machine. In a scenario like this, the system does the clerical work of automatically solving the "easy" lemmas and proof obligations, while the user selects the crucial proof steps. The final proof consists of 110 definitions, 63 of which are lemmas. If the proof is converted into a single λ-term, its size measured 36 thousand characters. When the lemmas are δ-reduced, the term size increases by 5. Finally, when all definitions are expanded, the size of the resulting term is 72 times the size of the original term. For the complete proof term derived by *Constructor*, see [4].

The performance of the system is good. On SUN 4 workstations, the system performs on an average 4000 unifications every second (including possible β and δ-reductions). Currently, no clause compilation takes place. Automatic construction of the proofs shown in the figures runs in seconds.

7. Related systems

The Nuprl system by Constable et al. [7] offers an interesting and impressive interactive proof development environment that is also based on type theory (Martin-Löf's Type Theory [25]). It is a significant improvement over the LCF proof system [15] that strongly influenced it. There is an inconvenience in the Nuprl system that has been overcome by the method proposed in this document. Proof construction in Nuprl is not based on unification based clausal resolution. The inference rules are the correctness rules for the underlying type theory, and implications in a context cannot be used directly as derived rules to resolve goals. Defining a new rule requires a detailed knowledge of the system and the programming language ML. Goals are posed in the empty context, and there is no notion of a theory describing the domain of interest and defining the proper axioms and inference rules. Thus, the desired context hypotheses are given as implications with the goal, and introduced later with introduction rules. As unification is not directly available, derivation of new hypotheses by instantiating others is often demanded, i.e., variables need to be given that could have been calculated. Automated theorem proving has not yet been accomplished with the system [7, p. 13].

One of the most powerful existing proof systems is *Isabelle* [27, 28]. Comparison to this system is difficult, because Isabelle is a generic theorem prover, whereas the proof method proposed here is dedicated to only one single proof formalism, viz. type theory. The same remark can be made for a comparison to the work of Fetty and Miller on theorem provers [13].

To the knowledge of the author, three systems are currently under development that have similar objectives as the method proposed here, viz. assistance for proof construction in type theory. One of them is the LEGO system by Pollack in Edinburgh. First prototype implementations of this system are operational. Another effort is the ALF system, that is implemented by Nordstrom and Coquand. Finally, Dowek [11] is developing an implementation of a mathematical vernacular for the

Calculus of Constructions. As these systems are still in development, it is too early for a comparison.

8. Discussion

The method presented combines the power of type theory with the advantages of resolution inference. The advantages of resolution inference are that it is simple and that it can be implemented efficiently.

Because proof construction is not decidable, strategic information has to be provided by users, either in the form of interactive choices, or in the form of algorithms (tactics). Resolution inference allows easy writing of tactics. The method presented may have potential to be used as a logic programming language, that includes all the essential features of e.g. Prolog, but that also provides typing, higher order facilities (this implies correct handling of expressions containing binders) and the use of local assumptions, thereby creating the possibility to handle queries containing universal quantification or implication (much like λProlog [26]). Note that the resulting derivations are in a natural deduction style.

As a meta-language, the CTT formalism is suitable to specify logical systems. The method presented makes the object level inference rules of a logic directly available for resolution instead of just the underlying correctness rules of CTT, thus offering the appropriate inference level. The abbreviation mechanism provides the possibility for hiding and the use of derived lemmas.

The requirement that contexts are always grounded in derivations is essential to the method, because it avoids the problem of unification over contexts and prevents the undesired generation of new axioms, while permitting extraction of necessary derived Horn clauses. The consequences of this restriction are directly related to a fundamental problem in higher order theorem proving.

It should be noted that proofs constructed by the method are in β-normal form, aside from definitions. In other words, the proofs constructed are cut-free. The proofs are not guaranteed in η-normal form, unless outermost η-reduction on objects is provided.

Actual implementations of proof system can efficiently handle many issues. A version of such a proof system, *Constructor*, automatically constructed some non-trivial proofs.

For an elaborate theoretical treatment of the proof construction method proposed in this paper, the reader is referred to [18]. That paper gives correctness proofs for the inference method, and the method is extended towards Barendregt's generalised type systems (GTS) [3].

Of course fully automated theorem proving cannot be the objective of proof systems. Proof construction is an *interactive* game between user and proof system, where the system plays the role of an interactive assistant. In this approach, the user does the essential proof steps. This includes proposing useful lemmas and

selecting crucial inference steps during the proof process. The machine task consists of the verification of details and solving the "easy" lemmas and (sub-)goals during the assembly of the main proof. Machines cannot do everything.

Acknowledgements

The author owes much gratitude to the referee and to Jan Bergstra, Loe Feijs, Bert Jutting, Ton Kalker, Frank van der Linden and Rob Wieringa for numerous suggestions and corrections. Gratitude is also indebted to Gilles Dowek, Gerard Huet, Dale Miller, Frank Pfenning and Randy Pollack for useful discussions and comments. Special thanks are due to René Ahn, who contributed greatly to this work, to Marcel van Tien, who implemented most of *Constructor*, and to Paul Gorissen, who provided the dynamic parsing facilities for the system. Finally, the author wants to thank Henk Barendregt, for many stimulating and clarifying discussions.

A previous version of this article appeared as [20].

References

[1] R.M.C. Ahn, Some extensions to Prolog based on AUTOMATH. Internal Philips technical note nr. 173/85, 1985.

[2] H. Barendregt, *The Lambda-Calculus: Its Syntax and Semantics* (North-Holland, Amsterdam, 1981).

[3] H. Barendregt, Introduction to generalised type systems, *Proceedings of the 3rd Italian Conference on Theoretical Computer Science*, Mantova, 1989 (World Scientific Publ.). Also to appear in: *J. Functional Programming.*

[4] H. Barendregt, Lambda calculi with types, to appear in: S. Abramsky, D. Gabbay and T. Maibaum, eds., *Handbook of Logic in Computer Science* (Oxford Univ. Press, Oxford, 1991).

[5] N.G. De Bruijn, Lambda-calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem, *Indag. Math.* **34** (5) (1972) 381-293.

[6] N.G. De Bruijn, A survey of the project Automath, in: J.P. Seldin and J.R. Hindley, eds., *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalisms* (Academic Press, New York, 1980).

[7] R.L. Constable, et al., *Implementing Mathematics with the Nuprl Proof Development System* (Prentice Hall, Englewood Cliffs, NJ, 1986).

[8] T. Coquand, Une Theory des Constructions, Thèse de troisième cycle, Université de Paris VII, 1985.

[9] T. Coquand and G.P. Huet, Constructions. a higher order proof system for mechanizing mathematics, *EUROCAL85*, Linz, Lecture Notes in Computer Science **203** (Springer, Berlin, 1985).

[10] D.T. van Daalen, The Language Theory of Automath. Ph. D. Dissertation, Eindhoven University of Technology, Dept. of Mathematics, 1980.

[11] G. Dowek, A proof synthesis algorithm for a mathematical vernacular, *Preliminary Proceedings of the First Annual Workshop on Logical Frameworks*, Antibes, May 1990.

[12] C.M. Elliott, Higher-order unification with dependent function types, *RTA-89*, Chapel Hill, Lecture Notes in Computer Science **355** (Springer, Berlin, 1989).

[13] A. Felty and D.A. Miller, Specifying theorem provers in a higher-order logic programming language, *CADE-9*, Argonne, Lecture Notes in Computer Science **310** (Springer, Berlin, 1988) 61-80.

[14] S. Fortune, et al., The expressiveness of simple and second-order type structures, *J. ACM* **30** (1) (1983) 151-185.

[15] M.J., Gordon, R. Milner and C.P. Wadsworth, *Edinburgh LCF*, Lecture Notes in Computer Science **78** (Springer, Berlin, 1979).

[16] R. Harper., F. Honsell and G. Plotkin, A framework for defining logics, *Second Annual Symposium on Logic in Computer Science*, Ithaca (IEEE, New York, 1987) 194-204.

[17] R. Harper and R. Pollack, Type checking, universe polymorphism, and typical ambiguity in the calculus of constructions, *Tapsoft '89*, Barcelona, Lecture Notes in Computer Science **352**, Vol. 2 (Springer, Berlin, 1989).

[18] L. Helmink and R. Ahn, Goal directed proof construction in type theory, Internal Philips technical note nr. 229/88, 1988. Also available in: G. Huet and G.D. Plotkin, eds., *Preliminary Proceedings of the First Annual Workshop of Logical Frameworks*, Antibes, May 1990 (Cambridge Univ. Press, 1991).

[19] L. Helmink and M. van Tien, Genesis constructive logic machine: user's guide. Available as document 28.5 of Esprit project 1222: 'Genesis'.

[20] L. Helmink, Resolution and type theory, *Proceedings of European Symposium on Programming*, Copenhagen, Lecture Notes in Computer Science **432** (Springer, Berlin, 1990).

[21] G.P. Huet, A unification algorithm for typed λ-calculus, *Theoret. Comput. Sci.* **1** (1975) 27-57.

[22] G.P. Huet, Induction principles formalized in the calculus of constructions, *Tapsoft '87*, Pisa, Lecture Notes in Computer Science **250**, Vol. 1 (Springer, Berlin, 1987).

[23] L.S. Jutting, A Translation of Landau's "Grundlagen" in AUTOMATH, Ph.D. Dissertation, Eindhoven University of Technology, Dept of Mathematics, 1976.

[24] L.S. Jutting, Normalization in Coquand's system, Internal Philips technical note nr. 156/88, 1988.

[25] P. Martin-Löf, *Intuitionistic Type Theory* (Bibliopolis, Napoli, 1984).

[26] G. Nadathur and D.A. Miller, An overview of λProlog. In: R. Kowalski and K. Bowen, eds., *Logic Programming: Proceedings of the Fifth International Conference and Symposium* (MIT Press, Cambridge, MA, 1988), Vol. 1, 820-827.

[27] L.C. Paulson, Natural deduction as higher-order resolution, *J. Logic Programming* **3** (1986) 237-258.

[28] L.C. Paulson, The foundation of a generic theorem prover, *J. Automated Reasoning* **5** (1989) 363-397.

[29] F. Pfenning, Elf: a language for logic definition and verified meta-programming, *Fourth Annual Symposium on Logic in Computer Science* (IEEE, New York, 1989) 313-322.

[30] D. Pym, A unification algorithm for the logical framework, LFCS report, Laboratory for Foundations of Computer Science, University of Edinburgh, Nov. 1988.

[31] J.A. Robinson, A machine oriented logic based on the resolution principle, *J. ACM* **12** (1) (1965) 23-49.