Safe Proof Checking in Type Theory with Y

Herman Geuvers[1], Erik Poll[2], and Jan Zwanenburg[2]

[1] Computer Science Department, Eindhoven University of Technology, P.O. Box 513, 5600 MB Eindhoven, The Netherlands, `herman@win.tue.nl`

[2] Computer Science Department, University of Nijmegen, P.O. Box 9010, 6500 GL Nijmegen , The Netherlands, `{erikpoll,janz}@cs.kun.nl`

Abstract. We present an extension of type theory with a fixed point combinator Y. We are particularly interested in using this Y for doing unbounded proof search in the proof system. Therefore we treat in some detail a typed λ-calculus for higher order predicate logic with inductive types (a reasonable subsystem of the theory implemented in [Dowek e.a. 1991]) and show how bounded proof search can be done in this system, and how unbounded proof search can be done if we add Y. Of course, proof search can also be implemented (as a tactic) in the meta language. This may give faster results, but asks from the user to be able to program the implementation. In our approach the user works completely in the proof system itself. We also provide the meta theory of type theory with Y that allows to use the fixed point combinator in a *safe* way. Most importantly, we prove a kind of *conservativity result*, showing that, if we can generate a proof term M of formula φ in the extended system, and M does not contain Y, then M is already a proof of φ in the original system.

1 Introduction

In theorem provers based on type theory, we are always looking for an explicit *proof-object*, i.e. if we want to prove the formula φ, we are in fact looking for a term M such that $M : \varphi$. (M is of type φ.) Such a term M then corresponds to a derivation in standard natural deduction (and can be translated to a proof in natural language text). This has the advantage that, besides the proof engine telling us that the formula is provable, the engine also produces – interactively with the user – a proof term that can be checked independently. As a matter of fact, the program for *checking* a proof object is relatively simple: it is a type checking algorithm for a strongly dependent-typed language. This conforms with the basic idea that *finding* a proof is difficult – hence this is done interactively, whereas *verifying* an alleged proof is simple.

The interaction with the proof engine usually exists in a set of goal-directed tactics. So, we try to construct a proof-term by looking at the structure of the goal to be proved. Of course, one can define more powerful tactics, especially when we are dealing with a decidable fragment of the logic. An example is the 'Tauto' tactic in Coq, that automatically solves (i.e. constructs proof-terms) for first order propositional logic (and a bit beyond).

Tactics like Tauto are built-in in the engine, but the user can also define his/her own tactics, by programming them in the meta-language of the proof system. To do so, the user has to know the meta-language and the way the proof system is implemented in it quite well. This makes it in general quite hard to program one's own tactics. In this paper we present a kind of 'tactic' that can be programmed in the proof system itself, which allows *searching* for proof-terms. So no knowledge of the implementation is required. We also present two examples of its use and the underlying explanation of the method in terms of the proof system (the typed λ-calculus that is implemented in the proof engine).

The method we present can also be implemented as a tactic in the meta-language, and then it can certainly be made much faster. We believe that it is nice that a 'search-tactic' can be safely implemented in the language of the proof system itself, which makes it much easier to apply for a user.

Due to the expressiveness of typed λ-calculus, a lot of 'proof search' can be defined already in the proof system itself. E.g. if we have a decidable predicate Q over `nat` (i.e. a proof term P of type $\forall n{:}\mathtt{nat}.Q(n) \vee \neg Q(n)$), then we can do a *bounded search* for an element $m \leq N$ such that $Q(m)$ holds. The idea is to iterate P up to N times until we find an $m : \mathtt{nat}$ for which $Pm = \mathtt{inl}\ t$; then $t : Q(m)$. Note that this m will also be the smallest n for which $Q(n)$ holds.

An *unbounded search* can also be defined if we add a *fixed point combinator* to the typed λ-calculus. In the example above: using the fixed point combinator, we can iterate P without bound, until we find an $m : \mathtt{nat}$ such that $Pm = \mathtt{inl}\ t$, and then $Q(m)$ holds. Adding a fixed point combinator is of course a real extension of the proof system: as the underlying typed λ-calculus is strongly normalizing, no fixed point combinator can be defined in it. In this paper we show that adding a fixed point combinator Y is *safe*. This is done by showing that the addition of Y yields a *conservative extension*. That is, if $\vdash_S$ denotes derivability in some typed λ-calculus and $\vdash_{S+Y}$ denotes derivability in S extended with Y, then

$$\Gamma \vdash_{S+Y} M : A \Rightarrow \Gamma \vdash_S M : A,$$

for Γ, M and A not containing Y. (Of course we do *not* have conservativity in the *logical* sense: $\exists_M(\Gamma \vdash_{S+Y} M : A) \not\Rightarrow \exists_M(\Gamma \vdash_S M : A)$, for Γ and A not containing Y.) Now, in order to show that adding Y is *safe*, let φ be a formula in a certain context Γ (both φ and Γ in the system S). Suppose we have constructed a proof-term $P : \varphi$ in $S + Y$, so P may possibly contain Y. Now we let P reduce until we find a term P' that does not contain Y. Then, due to the *subject reduction* property for $S + Y$ (which we will prove), we have $\Gamma \vdash_{S+Y} P' : \varphi$ and hence $\Gamma \vdash_S P' : \varphi$ by conservativity.

How exactly the fixed point combinator is used to perform proof search will be detailed in the paper by some examples. The proof search is in fact performed by the reduction of the fixed point combinator, so, in terms of the previous paragraph, the search is in the reduction from P to P'.

The conservativity of $S + Y$ over S will be proved for arbitrary functional Pure Type Systems S. Functional Pure Type Systems cover a large class of typed λ-calculi, among which we find the simple typed λ-calculus, dependent typed λ-calculus, polymorphic λ-calculus (known as system F) and the Calculus of Constructions. The core of the proof system of Coq is also a functional Pure Type System. We believe that the conservativity of $S+Y$ over S will extend to the whole proof system of Coq, which is a functional Pure Type System extended with inductive types.

2 Theorem Proving in Typed λ calculus

In this section we briefly introduce the notion of Pure Type System and give some examples of how theorem proving is done in such a system. Our main focus will be on the system λPREDω. This is a typed λ-calculus that faithfully represents constructive higher order predicate logic. To motivate this we give some examples of derivable judgements in λPREDω. For more information on Pure Type Systems and typed λ-calculus in general, we refer to [Barendregt 1992] and [Geuvers 1993].

Pure Type Systems or PTSs were first introduced by Berardi [Berardi 1990] and Terlouw [Terlouw 1989a], with slightly different definitions. The advantage of the class of PTSs is that many known systems can be seen as PTSs. So, many specific results for specific systems are immediate instances of general properties of PTSs. In the following we will mention a number of these properties.

Definition 1. *For $\mathcal{S}$ a set, the so called* sorts, *$\mathcal{A} \subset \mathcal{S} \times \mathcal{S}$ (the* axioms*) and $\mathcal{R} \subset \mathcal{S} \times \mathcal{S} \times \mathcal{S}$ (the* rules*), the* Pure Type System *$\lambda(\mathcal{S}, \mathcal{A}, \mathcal{R})$ is the typed λ-calculus with the following deduction rules.*

$$(\text{sort}) \quad \vdash s_1 : s_2 \qquad\qquad \textit{if } (s_1, s_2) \in \mathcal{A}$$

$$(\text{var}) \quad \frac{\Gamma \vdash A : s}{\Gamma, x{:}A \vdash x : A}$$

$$(\text{weak}) \quad \frac{\Gamma \vdash A : s \quad \Gamma \vdash M : C}{\Gamma, x{:}A \vdash M : C}$$

$$(\Pi) \quad \frac{\Gamma \vdash A : s_1 \quad \Gamma, x{:}A \vdash B : s_2}{\Gamma \vdash \Pi x{:}A.B : s_3} \qquad \textit{if } (s_1, s_2, s_3) \in \mathcal{R}$$

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

$$(\text{app}) \quad \frac{\Gamma \vdash M : \Pi x{:}A.B \quad \Gamma \vdash N : A}{\Gamma \vdash MN : B[N/x]}$$

$$(\text{conv}_\beta) \quad \frac{\Gamma \vdash M : A \quad \Gamma \vdash B : s}{\Gamma \vdash M : B} \qquad\qquad A =_\beta B$$

If $s_2 \equiv s_3$ in a triple $(s_1, s_2, s_3) \in \mathcal{R}$, we write $(s_1, s_2) \in \mathcal{R}$. In these rules, the expressions are taken from the set of pseudoterms T, *defined by*

$$\mathsf{T} ::= \mathcal{S} \mid \mathsf{V} \mid (\Pi\mathsf{V}{:}\mathsf{T}.\mathsf{T}) \mid (\lambda\mathsf{V}{:}\mathsf{T}.\mathsf{T}) \mid \mathsf{T}\mathsf{T}.$$

The pseudoterm A is typable *if there is a context Γ and a pseudoterm B such that $\Gamma \vdash A : B$ or $\Gamma \vdash B : A$ is derivable.*

In the following, we will mainly be dealing with the PTS λPREDω, which is defined as follows.

λPREDω	
$\mathcal{S}$	$\mathsf{Set}, \mathsf{Type}^s, \mathsf{Prop}, \mathsf{Type}^p$
$\mathcal{A}$	$(\mathsf{Set} : \mathsf{Type}^s), (\mathsf{Prop} : \mathsf{Type}^p)$
$\mathcal{R}$	$(\mathsf{Set}, \mathsf{Set}), (\mathsf{Set}, \mathsf{Type}^p), (\mathsf{Type}^p, \mathsf{Type}^p),$
	$(\mathsf{Prop}, \mathsf{Prop}), (\mathsf{Set}, \mathsf{Prop}), (\mathsf{Type}^p, \mathsf{Prop})$

The idea is that Set is the sort (universe) of 'small' sets, Prop is the sort of propositions Type^p is the sort of 'large' sets (so Prop is a large set) and Type^s is the sort containing just Set.

We briefly explain the rules. The rule $(\mathsf{Prop}, \mathsf{Prop})$ is for forming the implication: $\varphi \rightarrow \psi$ for $\varphi, \psi : \mathsf{Prop}$. With $(\mathsf{Set}, \mathsf{Type}^p)$ one can form $A \rightarrow \mathsf{Prop} : \mathsf{Type}^p$ and $A \rightarrow A \rightarrow \mathsf{Prop} : \mathsf{Type}^p$, the domains of unary predicates and binary relations over A. $(\mathsf{Type}^p, \mathsf{Type}^p)$ allows to extend this to higher order predicates and relations, like $(A \rightarrow \mathsf{Prop}) \rightarrow \mathsf{Prop} : \mathsf{Type}^p$, the domain of predicates over predicates over A, and $(A \rightarrow A \rightarrow \mathsf{Prop}) \rightarrow \mathsf{Prop} : \mathsf{Type}^p$. The rule $(\mathsf{Set}, \mathsf{Prop})$ allows the quantification over small sets (i.e. A with $A : \mathsf{Set}$): one can form $\Pi x{:}A.\varphi$ (for $A : \mathsf{Set}$ and $\varphi : \mathsf{Prop}$), which is to be read as a universal quantification. $(\mathsf{Type}^p, \mathsf{Prop})$ allows also higher order quantification, i.e. over large sets, e.g. $\Pi P{:}A \rightarrow \mathsf{Prop}.\varphi : \mathsf{Prop}$. Using $(\mathsf{Set}, \mathsf{Set})$ one can define function types like the type of binary functions: $A \rightarrow A \rightarrow A$, but also $(A \rightarrow A) \rightarrow A$, which is usually referred to as a 'higher order function type'.

We motivate the definition by giving some examples of mathematical notions that can be formalised in λPREDω.

Example 1.

1. $\mathtt{nat}{:}\mathsf{Set}, 0{:}\mathtt{nat}, >{:}\mathtt{nat} \rightarrow \mathtt{nat} \rightarrow \mathsf{Prop} \vdash \lambda x{:}\mathtt{nat}.x>0 : \mathtt{nat} \rightarrow \mathsf{Prop}$. Here we see the use of λ-abstraction to define a predicate.
2. $\mathtt{nat}{:}\mathsf{Set}, 0{:}\mathtt{nat}, S{:}\mathtt{nat} \rightarrow \mathtt{nat} \vdash$ $\Pi P{:}\mathtt{nat} \rightarrow \mathsf{Prop}.(P0) \rightarrow (\Pi x{:}\mathtt{nat}.(Px \rightarrow P(Sx))) \rightarrow \Pi x{:}\mathtt{nat}.Px : \mathsf{Prop}$. This is the induction formula written down in λPREDω as a term of type Prop.
3. $A{:}\mathsf{Set}, R{:}A \rightarrow A \rightarrow \mathsf{Prop} \vdash \Pi x, y, z{:}A.Rxy \rightarrow Ryz \rightarrow Rxz : \mathsf{Prop}$. (This formula expresses transitivity of R.)
4. $A{:}\mathsf{Set} \vdash \lambda R, Q{:}A \rightarrow A \rightarrow \mathsf{Prop}.\Pi x, y{:}A.Rxy \rightarrow Qxy :$ $(A \rightarrow A \rightarrow \mathsf{Prop}) \rightarrow (A \rightarrow A \rightarrow \mathsf{Prop}) \rightarrow \mathsf{Prop}$. (This relation between binary relations on A expresses inclusion of relations.)

5. $A{:}\mathsf{Set} \vdash \lambda x, y{:}A.\Pi P{:}A \to \mathsf{Prop}.(Px \to Py) : A \to A \to \mathsf{Prop}$.
   This binary relation on A is also called 'Leibniz equality' and is usually denoted by $=_A$, denoting the domain type explicitly.
6. $A{:}\mathsf{Set}, x, y{:}A \vdash \lambda r : x =_A y.\lambda P{:}A \to \mathsf{Prop}.r(\lambda z{:}A.Pz \to Px)(\lambda q{:}Px.q) : x =_A y \to y =_A x$. The proof of symmetry of Leibniz equality.

The rules of Pure Type Systems give the flexibility to define subsystems in a rather easy way by restricting the set $\mathcal{R}$. The Pure Type System λPRED2, representing second order predicate logic, is defined from λPREDω by removing $(\mathsf{Type}^p, \mathsf{Type}^p)$ from the $\mathcal{R}$. (Then we can no longer form higher order predicates and relations.) To obtain first order predicate logic, we remove $(\mathsf{Type}^p, \mathsf{Prop})$ from λPRED2, which forbids quantification over second order domains (predicates, relations). Other well-known typed λ-calculi that can be described as a PTS are simple typed λ-calculus, polymorphic typed λ-calculus (also known as system F) and the Calculus of Constructions.

2.1 Properties of Pure Type Systems

An important motivation for the definition of Pure Type Systems is that many important properties can be proved for all PTSs at once. Here we list the most important properties and discuss them briefly. Proofs can be found in [Geuvers and Nederhof 1991] and [Barendregt 1992]. Here we only mention the ones that are needed for the proof of conservativity of the extension of a PTS with a fixed point combinator.

In the following, unless explicitly stated otherwise, $\vdash$ refers to derivability in an arbitrary PTS. Furthermore, Γ is a *correct* context means that $\Gamma \vdash M : A$ for some M and A.

Proposition 1 (Church-Rosser (CR)).
The β-reduction is Church-Rosser on the set of pseudoterms T.

Proposition 2 (Correctness of Types (CT)).
If $\Gamma \vdash M : A$ then $\Gamma \vdash A : s$ or $A \equiv s$ for some some $s \in \mathcal{S}$.

Proposition 3 (Subject Reduction (SR)).
If $\Gamma \vdash M : A$ and $M \twoheadrightarrow_\beta N$, then $\Gamma \vdash N : A$.

Proposition 4 (Predicate Reduction (PR)).
If $\Gamma \vdash M : A$ and $A \twoheadrightarrow_\beta A'$, then $\Gamma \vdash M : A'$.

There are also many (interesting) properties that hold for specific PTSs or specific classes of PTSs. We mention one of these properties.

Definition 2. *A PTS $\lambda(\mathcal{S}, \mathcal{A}, \mathcal{R})$ is* functional, *also called* singly sorted, *if the relations A and R are functions, i.e. if the following two properties hold*

$$\forall s_1, s_2, s_2' \in \mathcal{S}(s_1, s_2'), (s_1, s_2') \in \mathcal{A} \Rightarrow s_2 = s_2',$$
$$\forall s_1, s_2, s_3, s_3' \in \mathcal{S}(s_1, s_2, s_3), (s_1, s_2, s_3') \in \mathcal{R} \Rightarrow s_3 = s_3'$$

The PTSs that we have encountered so far are functional. So are all PTSs that are used in practice. Functional PTSs share the following nice property.

Proposition 5 (Unicity of Types for functional PTSs (UT)). *For* functional *PTSs, if* $\Gamma \vdash M : A$ *and* $\Gamma \vdash M : B$*, then* $A =_\beta B$*.*

A less interesting, very basic, property, but one needed in a proof later, is:

Proposition 6 (Π-generation). *Let* $\Gamma \vdash \Pi x{:}A.B : s$*. Then there exists a rule* $(s_1, s_2, s) \in \mathcal{R}$ *such that* $\Gamma \vdash A : s_1$ *and* $\Gamma, x{:}A \vdash B : s_2$

An important property of a type system is that types can be *computed*, i.e. there is an algorithm that given Γ and M, computes an A for which $\Gamma \vdash M : A$ holds, and if there is no such A, returns 'false'. This is usually referred to as the *type inference problem.*

There are two important properties that ensure that type inference is decidable: Church-Rosser for β-reduction and Normalization for β-reduction. Of course, when adding a fixed point combinator, normalization is lost. In the next section we will discuss why, for the relevant fragment of the system, type checking is still decidable.

2.2 Inductive Types

We briefly treat the extension of λPREDω with inductive types, by giving some examples and how they are used. λPREDω + inductive types does not fully cover the type system of Coq, but quite a bit of it. At least it covers enough to be able to describe our examples of proof search in the next section. The scheme we give is roughly the one first introduced in [Coquand and Mohring 1990] and implemented in [Dowek e.a. 1991].

We first give the (very basic) example of natural numbers $\mathtt{nat}$. One is allowed to write down the following definition.

$$\begin{array}{ll} \mathsf{Inductive\ \ definition}\ \mathtt{nat} : \mathsf{Set} := & \\ & 0 : \mathtt{nat} \\ & S : \mathtt{nat} \to \mathtt{nat}. \end{array}$$

to obtain the following rules.

$$(\mathrm{elim}_1)\ \frac{\Gamma \vdash A : \mathsf{Set} \quad \Gamma \vdash f_1 : A \quad \Gamma \vdash f_2 : \mathtt{nat} \to A \to A}{\Gamma \vdash \mathsf{Rec}_{\mathtt{nat}} f_1 f_2 : \mathtt{nat} \to A}$$

$$(\mathrm{elim}_2)\ \frac{\Gamma \vdash P : \mathtt{nat} \to \mathsf{Prop} \quad \Gamma \vdash f_1 : P0 \quad \Gamma \vdash f_2 : \Pi x{:}\mathtt{nat}.Px \to P(Sx)}{\Gamma \vdash \mathsf{Rec}_{\mathtt{nat}} f_1 f_2 : \Pi x{:}\mathtt{nat}.Px}$$

$$(\mathrm{elim}_3)\ \frac{\Gamma \vdash A : \mathsf{Type}^p \quad \Gamma \vdash f_1 : A \quad \Gamma \vdash f_2 : \mathtt{nat} \to A \to A}{\Gamma \vdash \mathsf{Rec}_{\mathtt{nat}} f_1 f_2 : \mathtt{nat} \to A}$$

The rule (elim_1) allows the definition of functions by primitive recursion. The rule (elim_2) allows proofs by induction. The rule (elim_3) allows the definition of predicates (on $\mathtt{nat}$) by induction. To make sure that the functions defined by the (elim) rules compute in the correct way, Rec has the following reduction rule.

$$\mathsf{Rec}_{\mathtt{nat}} f_1 f_2 0 \longrightarrow_\iota f_1$$
$$\mathsf{Rec}_{\mathtt{nat}} f_1 f_2 (St) \longrightarrow_\iota f_2 t(\mathsf{Rec}_{\mathtt{nat}} f_1 f_2 t)$$

The additional ι-reduction is also included in the *conversion*-rule (conv), where we now have as a side-condition '$A =_{\beta\iota} B$'. The subscript in $\mathsf{Rec}_{\mathtt{nat}}$ will be omitted, when clear from the context.

An example of the use of (elim_1) is in the definition of the 'double' function d, which is defined by

$$d := \mathsf{Rec}\, 0(\lambda x{:}\mathtt{nat}.\lambda y{:}\mathtt{nat}.S(S(y))).$$

Now, $d0 \twoheadrightarrow_{\beta\iota} 0$ and $d(Sx) \twoheadrightarrow_{\beta\iota} S(S(dx))$. The predicate of 'being even', $\mathsf{even}(-)$, can be defined by using (elim_3):

$$\mathsf{even}(-) := \mathsf{Rec}\,(\top)(\lambda x{:}\mathtt{nat}.\lambda\alpha{:}\mathsf{Prop}.\neg\alpha).$$

Here, $\neg\varphi$ is defined as $\varphi \to \bot$. We obtain indeed that

$$\mathsf{even}(0) \twoheadrightarrow_{\beta\iota} \top,$$
$$\mathsf{even}(Sx) \twoheadrightarrow_{\beta\iota} \neg\mathsf{even}(x)$$

An example of the use of (elim_2) is the proof of $\Pi x{:}\mathtt{nat}.\mathsf{even}(dx)$. Say that true is some canonical inhabitant of type $\top$. Using $\mathsf{even}(d(Sx)) =_{\beta\iota} \neg\neg\mathsf{even}(dx)$ we also find that the term $\lambda x{:}\mathtt{nat}.\lambda h{:}\mathsf{even}(dx).\lambda z{:}\neg\mathsf{even}(dx).zh$ is of type $\Pi x{:}\mathtt{nat}.\mathsf{even}(dx) \to \mathsf{even}(d(Sx))$. So we conclude that

$$\vdash \mathsf{Rec}\,\mathsf{true}(\lambda x{:}\mathtt{nat}.\lambda h{:}\mathsf{even}(dx).\lambda z{:}\neg\mathsf{even}(dx).zh) : \Pi x{:}\mathtt{nat}.\mathsf{even}(dx).$$

Another well-known example is the type of lists over a domain D. This is usually defined as a *parametric inductive type*, taking the domain as a parameter of the inductive definition. The type of *parametric lists* can be defined as follows.

$$\begin{array}{ll} \mathsf{Inductive\ \ definition\ List} : \mathsf{Set} \to \mathsf{Set} := & \\ & \mathsf{Nil} : \Pi D{:}\mathsf{Set}.(\mathsf{List} D) \\ & \mathsf{Cons} : \Pi D{:}\mathsf{Set}.(\mathsf{List} D) \to D \to (\mathsf{List} D). \end{array}$$

Which generates the following elimination rules and reduction rule.

$$(\mathrm{elim}_1)\ \frac{\Gamma \vdash D : \mathsf{Set} \quad \Gamma \vdash A : \mathsf{Set} \quad \Gamma \vdash f_1 : A \quad \Gamma \vdash f_2 : (\mathsf{List} D) \to D \to A \to A}{\Gamma \vdash \mathsf{Rec}_{\mathsf{List}} f_1 f_2 : (\mathsf{List} D) \to A}$$

$$(\mathrm{elim}_2)\ \frac{\begin{array}{ll} \Gamma \vdash D : \mathsf{Set} & \Gamma \vdash f_1 : P(\mathsf{Nil} D) \\ \Gamma \vdash P : (\mathsf{List} D) \to \mathsf{Prop} & \Gamma \vdash f_2 : \Pi x{:}(\mathsf{List} D).\Pi d{:}D.Px \to P(\mathsf{Cons} x d) \end{array}}{\Gamma \vdash \mathsf{Rec}_{\mathsf{List}} f_1 f_2 : \Pi x{:}(\mathsf{List} D).Px}$$

$$(\mathrm{elim}_3)\ \frac{\Gamma \vdash D : \mathsf{Set} \quad \Gamma \vdash A : \mathsf{Type}^p \quad \Gamma \vdash f_1 : A \quad \Gamma \vdash f_2 : (\mathsf{List} D) \to D \to A \to A}{\Gamma \vdash \mathsf{Rec}_{\mathsf{List}} f_1 f_2 : (\mathsf{List} D) \to A}$$

$$\mathsf{Rec}_{\,\mathsf{List}} f_1 f_2 (\mathsf{Nil} D) \longrightarrow_\iota f_1$$
$$\mathsf{Rec}_{\,\mathsf{List}} f_1 f_2 (\mathsf{Cons} D l d) \longrightarrow_\iota f_2 l d (\mathsf{Rec}_{\,\mathsf{List}} f_1 f_2 l)$$

Note that to be able to write down the type of the constructors Nil and Cons, we need to add the rule $(\mathsf{Type}^s, \mathsf{Set})$ to λPREDω. The constructors Nil and Cons have a dependent type. It turns out that this situation occurs more often. We treat another interesting example: the Σ-type. Let $B : \mathsf{Set}$ and $Q : B \to \mathsf{Prop}$ and suppose we have added the rule $(\mathsf{Prop}, \mathsf{Set})$ to our system.

$$\begin{array}{l} \mathsf{Inductive\ \ definition}\ \mu : \mathsf{Set} := \\ \qquad\qquad\qquad \mathsf{In} : \Pi z{:}B.(Qz) \to \mu. \end{array}$$

$$(\mathrm{elim}_1)\ \frac{\Gamma \vdash A : \mathsf{Set} \quad \Gamma \vdash f_1 : \Pi z{:}B.(Qz) \to A}{\Gamma \vdash \mathsf{Rec}_{\,\mu} f_1 : \mu \to A}$$

$$(\mathrm{elim}_2)\ \frac{\Gamma \vdash P : \mu \to \mathsf{Prop} \quad \Gamma \vdash f_1 : \Pi z{:}B.\Pi y{:}(Qz).P(\mathsf{In} z y)}{\Gamma \vdash \mathsf{Rec}_{\,\mu} f_1 : \Pi x{:}\mu.(Px)}$$

$$(\mathrm{elim}_3)\ \frac{\Gamma \vdash A : \mathsf{Type}^p \quad \Gamma \vdash f_1 : \Pi z{:}B.(Qz) \to A}{\Gamma \vdash \mathsf{Rec}_{\,\mu} f_1 : \mu \to A}$$

The ι-reduction rule is

$$\mathsf{Rec}_{\,\mu} f_1 (\mathsf{In} b q) \longrightarrow_\iota f_1 b q$$

Now, taking in (elim_1) B for A and $\lambda z{:}B.\lambda y{:}(Qz).z$ for f_1, we find that $\mathsf{Rec}\,(\lambda z{:}B.\lambda y{:}(Qz).z)(\mathsf{In} b q) \twoheadrightarrow b$. Hence we define $\pi_1 := \mathsf{Rec}\,(\lambda z{:}B.\lambda y{:}(Qz).z)$. Now, taking in (elim_2) $\lambda x{:}\mu.Q(\pi_1 x)$ for P and $\lambda z{:}B.\lambda y{:}(Qz).y$ for f_1, we find that $\mathsf{Rec}\,(\lambda z{:}B.\lambda y{:}(Qz).y) : \Pi z{:}\mu.Q(\pi_1 z)$. Also, $\mathsf{Rec}\,(\lambda z{:}B.\lambda y{:}(Qz).y)(\mathsf{In} b q) \twoheadrightarrow q$. Hence we define $\pi_2 := \mathsf{Rec}\,(\lambda z{:}B.\lambda y{:}(Qz).y)$ and we remark that μ together with In (as pairing constructor) and π_1 and π_2 (as projections) represents the Σ-type. In the rest of this article, we will just use the Σ-type, and will write $\langle n, p \rangle$ for the pair of n and p.

An example of an inductively defined proposition is the disjunction. Given φ and ψ of type Prop, $\varphi \vee \psi$ can be defined as follows.

$$\begin{array}{l} \mathsf{Inductive\ \ definition}\ \varphi \vee \psi : \mathsf{Prop} := \\ \qquad\qquad\qquad \mathtt{inl}\ : \varphi \to (\varphi \vee \psi) \\ \qquad\qquad\qquad \mathtt{inr}\ : \psi \to (\varphi \vee \psi) \end{array}$$

We add the $(\mathsf{Prop}, \mathsf{Set})$ rule to λPREDω, because we want to have the first two elimination rules.

$$(\mathrm{elim}_1)\ \frac{\Gamma \vdash A : \mathsf{Set} \quad \Gamma \vdash f_1 : \varphi \to A \quad \Gamma \vdash f_2 : \psi \to A}{\Gamma \vdash \mathsf{Rec}_{\,\vee} f_1 f_2 : (\varphi \vee \psi) \to A}$$

$$(\mathrm{elim}_2)\ \frac{\Gamma \vdash P : \mathsf{Prop} \quad \Gamma \vdash f_1 : \varphi \to P \quad \Gamma \vdash f_2 : \psi \to P}{\Gamma \vdash \mathsf{Rec}_{\,\vee} f_1 f_2 : (\varphi \vee \psi) \to P}$$

$$\mathsf{Rec}_{\vee} f_1 f_2 (\mathtt{inl}\ q) \longrightarrow_{\iota} f_1 q,$$
$$\mathsf{Rec}_{\vee} f_1 f_2 (\mathtt{inr}\ q) \longrightarrow_{\iota} f_2 q.$$

As usual we write

$$\begin{aligned} \mathtt{case}\ t\ \mathtt{of}\ \ & \mathtt{inl}\ (p) \Rightarrow M_1 \\ & \mathtt{inr}\ (p) \Rightarrow M_2 \end{aligned}$$

for $\mathsf{Rec}_{\vee}(\lambda p{:}\varphi.M_1)(\lambda p{:}\psi.M_2)t$.

Similarly one can define the disjoint union of two small sets, $A + B$ for $A, B : \mathsf{Set}$, inductively. We will ambiguously use the same notations for the constructors of $A + B$.

3 Proof Search in Type Theoretic Theorem Provers

We treat two examples of proof search in Coq. We try to avoid using Coq-syntax and describe the examples in terms of λPREDω with inductive types. The first example is a search for a term $n : \mathtt{nat}$ such that $Q(n)$ holds, where Q is a decidable predicate. So we let $Q : \mathtt{nat} {\rightarrow} \mathsf{Prop}$ and we assume we have a term

$$P : \Pi n{:}\mathtt{nat}.Q(n) \vee \neg Q(n).$$

Now, we want to iterate P to find the n and the proof of $Q(n)$. First suppose that we hope to find the n before N, so we want to iterate P at most N times ($N : \mathtt{nat}$).

Definition 3. *For $A : \mathsf{Set}$, $a : A$ and $n : \mathtt{nat}$ we define $Y_a^n : (A{\rightarrow}A){\rightarrow}A$ as follows.*

$$Y_a^n := \lambda f{:}A{\rightarrow}A.\mathsf{Rec}_{\mathtt{nat}}\, a(\lambda x{:}\mathtt{nat}.f)n.$$

The following is easily verified.

$$\begin{aligned} Y_a^0 f &\twoheadrightarrow_{\beta\iota} a, \\ Y_a^{n+1} f &\twoheadrightarrow_{\beta\iota} f(Y_a^n f), \\ Y_a^n f &\twoheadrightarrow_{\beta\iota} f^n(a), \end{aligned}$$

where $f^n(a)$ denotes, as usual, n times application of f on a.

Now define

$$\begin{aligned} F :\equiv \lambda g{:}\mathtt{nat}{\rightarrow}\mathtt{nat}.\lambda n{:}\mathtt{nat}.\mathtt{case}\ (Pn)\ \mathtt{of}\ \ & \mathtt{inl}\ (p) \Rightarrow n \\ & \mathtt{inr}\ (p) \Rightarrow g(n+1). \end{aligned}$$

So, $F : (\mathtt{nat}{\to}\mathtt{nat}){\to}(\mathtt{nat}{\to}\mathtt{nat})$. Now, let $N : \mathtt{nat}$ and let $I := \lambda x{:}\mathtt{nat}.x$. Then

$$\begin{array}{rll}
Y_I^N F0 \twoheadrightarrow_{\beta\iota} & 0 & \text{if } P0 \twoheadrightarrow_{\beta\iota} \mathtt{inl}\ (p) : Q(0), \\
& Y_I^{N-1} F1 & \text{if } P0 \twoheadrightarrow_{\beta\iota} \mathtt{inr}\ (p) : \neg Q(0), \\
Y_I^{N-1} F1 \twoheadrightarrow_{\beta\iota} & 1 & \text{if } P1 \twoheadrightarrow_{\beta\iota} \mathtt{inl}\ (p) : Q(1), \\
& Y_I^{N-2} F2 & \text{if } P1 \twoheadrightarrow_{\beta\iota} \mathtt{inr}\ (p) : \neg Q(1), \\
Y_I^{N-2} F2 \twoheadrightarrow_{\beta\iota} & 2 & \text{if } P2 \twoheadrightarrow_{\beta\iota} \mathtt{inl}\ (p) : Q(2), \\
& Y_I^{N-3} F3 & \text{if } P2 \twoheadrightarrow_{\beta\iota} \mathtt{inr}\ (p) : \neg Q(2), \\
\dots & & \\
Y_I^1 F(N-1) \twoheadrightarrow_{\beta\iota} & N-1 & \text{if } P(N-1) \twoheadrightarrow_{\beta\iota} \mathtt{inl}\ (p) : Q(N-1), \\
& Y_I^0 FN & \text{if } P(N-1) \twoheadrightarrow_{\beta\iota} \mathtt{inr}\ (p) : \neg Q(N-1), \\
Y_I^0 FN \twoheadrightarrow_{\beta\iota} & N. &
\end{array}$$

So, if $Y_I^N F0 \twoheadrightarrow_{\beta\iota} n$ with $n < N$, then $Q(n)$ holds and $P(n) \twoheadrightarrow_{\beta\iota} \mathtt{inl}\ (p)$ with p a proof of $Q(n)$. If $Y_I^N F0 \twoheadrightarrow_{\beta\iota} N$, then $\neg Q(n)$ for all $n < N$.

The method above works if we know an upperbound to the n that we want to search for. Another option is to start a (possibly non-terminating) search. This can be done by adding the fixed point combinator to λPREDω with inductive types. We define the extension very generally for PTSs.

Definition 4. *Let $S = \lambda(\mathcal{S}, \mathcal{A}, \mathcal{R})$ be a PTS and let s be a sort of the system S. The system $S + Y^s$ is obtained by adding the following rule.*

$$\frac{\Gamma \vdash A : s \quad \Gamma \vdash f : A{\to}A}{\Gamma \vdash Y^s f : A}$$

The β-reduction is extended with

$$Y^s f \longrightarrow_Y f(Y^s f).$$

We sometimes omit the superscript s, if we know which sort we are talking about. If we add Y^s for all sorts, we just talk about $S + Y$.

In λPREDω with inductive types and Y^{Set}, we can now program an arbitrary proof search. (We omit the superscript Set.) With the above definition for F we obtain

$$YFI : \mathtt{nat}$$

and if $YFI \twoheadrightarrow_{\beta\iota Y} n$ with n a normal form, then we know that $Q(n)$ holds and $Pn \twoheadrightarrow_{\beta\iota Y} \mathtt{inl}\ (p)$ with $p : Q(n)$. (Moreover, we know that n is the smallest m for which $Q(m)$ holds, but only on a meta-level: the proof term does not represent this information. If YFI does not terminate, there is no n for which $Q(n)$ holds.)

We may wonder whether the extension of a type system with Y is safe. This will be the subject of the next section. Here we consider one more application of the proof search method, where we want to verify whether Q holds for $n = 0, 1, \dots, N$.

Suppose again we have our decidable predicate Q with P a proof term of type $\Pi n{:}\mathtt{nat}.Q(n) \vee \neg Q(n)$. If we want to prove that Q holds for $n = 0, 1, \ldots, N$, we do not just want to verify that fact, but also store the proof terms of $Q(0), Q(1), \ldots, Q(N)$. Moreover, if $Q(n)$ fails for an $n \leq N$, we want to return this n. Now abbreviate

$$\mathsf{List} := \mathsf{List}(\Sigma x{:}\mathtt{nat}.Q(x)).$$

Define the function F as follows.

$$\begin{array}{rl} F :\equiv \lambda g.\lambda n{:}\mathtt{nat}.\mathtt{case}\ P(n)\ \mathtt{of} & \mathtt{inl}\ (p) \Rightarrow (\mathtt{case}\ g(n+1)\ \mathtt{of} \\ & \qquad \mathtt{inl}\ (l) \Rightarrow \mathtt{inl}\ ((\mathsf{Cons}\langle n, p\rangle l) \\ & \qquad \mathtt{inr}\ (m) \Rightarrow \mathtt{inr}\ (m)) \\ & \mathtt{inr}\ (p) \Rightarrow \mathtt{inr}\ (n). \end{array}$$

Here, the type of g is $\mathtt{nat}{\rightarrow}(\mathsf{List} + \mathtt{nat})$, with List as above, the type of lists over $\Sigma x{:}\mathtt{nat}.Q(x)$. (For readability we have omitted the Set-parameter in Cons.) So,

$$F : (\mathtt{nat}{\rightarrow}(\mathsf{List} + \mathtt{nat})){\rightarrow}(\mathtt{nat}{\rightarrow}(\mathsf{List} + \mathtt{nat})).$$

Now, iterating F $N+1$ times on $\lambda x{:}\mathtt{nat}.\mathtt{inl}\ (\mathsf{Nil})$ will either result in a sequence

$$[\langle 0, p_0\rangle, \langle 1, p_1\rangle, \ldots, \langle N, p_N\rangle]$$

with $p_i : Q(i)$ for each i, or in a term $n : \mathtt{nat}$ with $n \leq N$, $Pn \twoheadrightarrow_{\beta\iota} \mathtt{inr}\ (p)$ and $p : \neg Q(n)$. Obviously, in this example one will never wish to use the fixed point combinator, as we are doing a *bounded* search.

4 Meta-theory of Pure Type Systems with Y

Most of the meta-theoretical properties of PTSs are not affected by the inclusion of a fixpoint combinator Y. (The obvious exception is strong normalization, of course!) In particular:

Proposition 7 (Church-Rosser ($\mathbf{CR}_Y$)).
The βY-reduction is Church-Rosser on the set of pseudoterms T.

Proposition 8 (Correctness of Types ($\mathbf{CT}_Y$)).
If $\Gamma \vdash_{S+Y} M : A$ then $\Gamma \vdash_{S+Y} A : s$ or $A \equiv s$ for some some $s \in \mathcal{S}$.

Proposition 9 (Subject Reduction ($\mathbf{SR}_Y$)).
If $\Gamma \vdash_{S+Y} M : A$ and $M \twoheadrightarrow_{\beta Y} N$, then $\Gamma \vdash_{S+Y} N : A$.

Proposition 10 (Unicity of Types for functional PTSs ($\mathbf{UT}_Y$)).
For functional *PTSs, if $\Gamma \vdash_{S+Y} M : A$ and $\Gamma \vdash_{S+Y} M : B$, then $A =_{\beta Y} B$.*

In addition to the properties above, to prove conservativity of Y we also need the (very basic) ones below.

Proposition 11 **(Π_Y-generation).** *Let $\Gamma \vdash_{S+Y} \Pi x{:}A.B : s$. Then there exists a rule $(s_1, s_2, s) \in \mathcal{R}$ such that $\Gamma \vdash_{S+Y} A : s_1$ and $\Gamma, x{:}A \vdash_{S+Y} B : s_2$*

Proposition 12 **(axiom$_Y$-generation).**
Let $\Gamma \vdash_{S+Y} s : s'$ with $s, s' \in \mathcal{S}$. Then $(s, s') \in \mathcal{A}$.

All the properties of PTSs with Y above can be proved in exactly the same way as for PTSs. The trick to proving Conservativity (1 below) is to prove the following, slightly weaker, property. A direct proof of Conservativity by induction on derivations fails.

Lemma 1. *For* functional *PTSs, if $\Gamma \vdash_{S+Y} M : A$ with Γ and M not containing Y, then $\Gamma \vdash M : A'$ for some A' with $A \twoheadrightarrow_{\beta Y} A'$.*

Proof. Induction on the derivation of $\Gamma \vdash_{S+Y} M : A$. The interesting cases are the abstraction and application rule:

– Suppose the last step in the derivation is

$$\frac{\Gamma \vdash_{S+Y} M : \Pi x{:}A.B \quad \Gamma \vdash_{S+Y} N : A}{\Gamma \vdash_{S+Y} MN : B[N/x]}$$

By the IH $\Gamma \vdash M : C$ for some C with $\Pi x{:}A.B \twoheadrightarrow_{\beta Y} C$. So, C is a Π-abstraction, say $C \equiv \Pi x{:}A'.B'$. Then $A \twoheadrightarrow_{\beta Y} A'$ and $\twoheadrightarrow_{\beta Y} B'$. By Proposition 6, $\Gamma \vdash A' : s_1$ for some s_1. By the IH $\Gamma \vdash N : A''$ for some A'' with $A \twoheadrightarrow_{\beta Y} A''$. As $A' =_{\beta Y} A''$ and A', A'' do not contain Y, we conclude $A' =_\beta A''$ (using $\mathrm{CR}_{\beta Y}$). Hence $\Gamma \vdash N : A'$ by the (conv) rule. Now, $\Gamma \vdash MN : B'[N/x]$ by the (app) rule and indeed $B[N/x] \twoheadrightarrow_{\beta Y} B'[N/x]$.

– Suppose the last step in the derivation is

$$\frac{\Gamma, x{:}A \vdash_{S+Y} M : B \quad \Gamma \vdash_{S+Y} \Pi x{:}A.B : s}{\Gamma \vdash_{S+Y} \lambda x{:}A.M : \Pi x{:}A.B}$$

By the IH on the first premise $\Gamma, x{:}A \vdash M : B'$ for some B' with $B \twoheadrightarrow_{\beta Y} B'$. Unfortunately we cannot use the IH on the second premise – $\Gamma \vdash_{S+Y} \Pi x{:}A.B : s$ – as $\Pi x{:}A.B$ may contain Y. We reason as follows: $\Gamma \vdash A : s_1$ (for some s_1). By CT (Proposition 2), $\Gamma, x{:}A \vdash B' : s_2$ for some s_2 (i) or $B' \equiv s'$ for some s' (ii). Looking at these two cases:

(i) As $\vdash_S \subseteq \vdash_{S+Y}$ we know $\Gamma \vdash_{S+Y} A : s_1$ and $\Gamma, x : A \vdash_{S+Y} B' : s_2$. Using SR_Y we find $\Gamma \vdash_{S+Y} \Pi x{:}A.B' : s$. Combining this with Proposition 11 and UT_Y we conclude $(s_1, s_2, s) \in \mathcal{R}$. So $\Gamma \vdash \Pi x{:}A.B' : s$.

(ii) As $\vdash_S \subseteq \vdash_{S+Y}$ we know $\Gamma \vdash_{S+Y} A : s_1$. Using SR_Y we find $\Gamma \vdash_{S+Y} \Pi x{:}A.B' : s$. Combining this with Proposition 11 and UT_Y we conclude $(s_1, s_2, s) \in \mathcal{R}$ and $\Gamma, x : A \vdash_{S+Y} B' : s_2$ for some s_2. Since $B' \equiv s'$, $s' : s_2$ must be an axiom, so $\Gamma, x : A \vdash B' : s_2$. Hence $\Gamma \vdash \Pi x{:}A.B' : s$.

Now $\Gamma \vdash \lambda x{:}A.M : \Pi x{:}A.B'$ by the (λ) rule and $\Pi x{:}A.B \twoheadrightarrow_{\beta Y} \Pi x{:}A.B'$.

Conservativity is an easy consequence of the lemma above.

Corollary 1 (Conservativity for functional PTSs).
Consider a functional PTS. Let $\Gamma \vdash_{S+Y} M : A$ *with* Γ, M, *and* A *not containing* Y. *Then* $\Gamma \vdash M : A$.

Proof. By the previous lemma $\Gamma \vdash M : A'$ for some A' with $A \twoheadrightarrow_{\beta Y} A'$. By CT_Y we have $\Gamma \vdash_{S+Y} A : s$ or $A \equiv s$ for some $s \in \mathcal{S}$. In the second case, $A' \equiv A \in \mathcal{S}$, so $\Gamma \vdash M : A$. In the first case, $\Gamma \vdash A : s$ by the previous Lemma, so $\Gamma \vdash M : A$ by the conversion rule.

With respect to the issue of *decidability of type inference*: in general, the addition of a fixed point combinator will make type inference undecidable. This is because Y allows us to define all partial recursive functions. So, if $F : \mathtt{nat} \to \mathtt{nat}$ is some closed term, representing a partial recursive function and we take $\Gamma = P : \mathtt{nat} \to \mathsf{Prop}, x : P0$, then

$$x : P(F0) \text{ iff } F0 = 0.$$

So, a type inference algorithm would give us a decision algorithm for the value of partial recursive functions on 0, quod non: type inference is undecidable.

Nevertheless, we still want to be able to edit the term YF that defines our proof search. That is, we would like to be able to interactively construct YF and have it type checked by the proof engine. If we look back at the examples in Section 3, we see that the 'proof search terms' YF that are given here *can* be type checked: If we apply the usual type-checking algorithm to these terms, the Y-reduction, which is the only possible source for infinite reductions (and hence undecidability), is never used.

To be more precise: the proof search terms that we have constructed can all be type-checked in the system $\lambda\mathrm{PRED}\omega$, where Y is treated as a constant that takes a term of type $A \to A$ to a term of type A. (The $\lambda\mathrm{PRED}\omega$-type-checking algorithm applies immediately to this small extension. Alternatively one could extend $\lambda\mathrm{PRED}\omega$ with the rule $(\mathsf{Type}^s, \mathsf{Set})$. Then put $Y : \Pi\alpha{:}\mathsf{Set}.(\alpha \to \alpha) \to \alpha$ in the context and use the type-checking algorithm for this extension of $\lambda\mathrm{PRED}\omega$.)

This is a general situation: in the phase of constructing the proof search term we can treat Y as a constant (without reduction behaviour). So then we are dealing with well-known type systems. When we have constructed the proof search term, we let it reduce and if this results in a normal form, the conservativity property, Corollary 1, guarantees that we have found a proof in the original type system.

5 Conclusions and Related Work

We have presented a method for proof search inside the proof system of higher order predicate logic with inductive types. We have tested our method by some examples using the proof engine Coq. See [Zwanenburg e.a. 1999] for the examples; the methods turn out to be reasonably fast. In our first example we are looking for a 'witness' n of the property Q, using $Y_I^N F0$, which iterates F up to

N times, starting from 0. One could do this similarly in the meta language of the proof system (the implementation language), which may be faster, but also requires a lot of knowledge of the implementation and experience in programming in the meta-language.

We have also presented the underlying theory, why doing an unbounded search (by adding a fixed point combinator) does not spoil the logical proof system. The addition of a fixed point combinator to the Calculus of Constructions (CC) has also previously been studied in [Audebaud 1991]. His goal is to overcome the problem with the second order definable datatypes in CC, so he is using the fixed point mainly to be able to define data types (of type Set in our system) that have the desirable properties. We don't have to do that, because we use the extension with inductive types, which provides us with the necessary data types. Moreover, we are especially interested in using the fixed point combinator to define (potentially) infinite computations to search for witnesses and proof-objects.

References

[Audebaud 1991] P. Audebaud, Partial Objects in the Calculus of Constructions, in *Proceedings of the Sixth Annual Symp. on Logic in Computer Science*, Amsterdam 1991, IEEE, pp. 86 – 95.

[Barendregt 1992] H.P. Barendregt, Lambda calculi with Types. In *Handbook of Logic in Computer Science*, eds. Abramski et al., Oxford Univ. Press, pp. 117 – 309.

[Berardi 1990] S. Berardi, Type dependence and constructive mathematics, Ph.D. thesis, Universita di Torino, Italy.

[Coquand and Mohring 1990] Th. Coquand and Ch. Paulin-Mohring Inductively defined types, In P. Martin-Löf and G. Mints editors. *COLOG-88 : International conference on computer logic, LNCS 417.*

[Dowek e.a. 1991] G. Dowek, A. Felty, H. Herbelin, G. Huet, Ch. Paulin-Mohring, B. Werner, The Coq proof assistant version 5.6, user's guide. INRIA Rocquencourt - CNRS ENS Lyon.

[Geuvers 1993] H. Geuvers, Logics and Type Systems, Ph.D. Thesis, University of Nijmegen, 1993.

[Geuvers and Nederhof 1991] J.H. Geuvers and M.J. Nederhof, A modular proof of strong normalisation for the calculus of constructions. *Journal of Functional Programming*, vol 1 (2), pp 155-189.

[Terlouw 1989a] J. Terlouw, Een nadere bewijstheoretische analyse van GSTT's (incl. appendix), Manuscript, Faculty of Mathematics and Computer Science, University of Nijmegen, Netherlands, March, April 1989. (In Dutch)

[Zwanenburg e.a. 1999] J. Zwanenburg and H. Geuvers, Example of Proof Search by iteration in Coq, url:
`http://www.cs.kun.nl/~janz/proofs/proofSearch/index.html`