The proof-assistant Yarrow

Citation for published version (APA):

Document status and date:
Published: 01/01/1998

Document Version:
Publisher’s PDF, also known as Version of Record (includes final page, issue and volume numbers)

Please check the document version of this publication:
• A submitted manuscript is the version of the article upon submission and before peer-review. There can be important differences between the submitted version and the official published version of record. People interested in the research are advised to contact the author for the final version of the publication, or visit the DOI to the publisher’s website.
• The final author version and the galley proof are versions of the publication after peer review.
• The final published version features the final layout of the paper including the volume, issue and page numbers.

Link to publication

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

• Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
• You may not further distribute the material or use it for any profit-making activity or commercial gain
• You may freely distribute the URL identifying the publication in the public portal.

If the publication is distributed under the terms of Article 25fa of the Dutch Copyright Act, indicated by the “Taverne” license above, please follow below link for the End User Agreement:
www.tue.nl/taverne

Take down policy
If you believe that this document breaches copyright please contact us at: openaccess@tue.nl
providing details and we will investigate your claim.
The Proof-assistent
Yarrow

by

J. Zwanenburg

ISSN 0926-4515

All rights reserved
editors: prof.dr. R.C. Backhouse
prof.dr. J.C.M. Baeten

Reports are available at:
http://www.win.tue.nl/win/cs

Computing Science Reports 98/11
Eindhoven, July 1998
The Proof-assistant
Yarrow

Jan Zwanenburg
Department of Mathematics and Computing Science
Eindhoven University of Technology
P.O.Box 513, 5600 MB Eindhoven, The Netherlands
e-mail janz@win.tue.nl
July 10, 1998

Abstract

Yarrow is an interactive proof assistant based on the theory of Pure Type Systems, a family of typed lambda calculi. Yarrow has been designed as a flexible environment for experimentation with various typed lambda calculi. It offers both graphical and textual interfaces. It has been coded entirely in Haskell, making use of the Fudget library for the graphical interface. In this paper we concentrate on the software architecture of Yarrow, in particular the use of monads, the coupling of user interface and proof engine, polymorphic output routines, and flexible representations of lambda terms. We also treat the presentation of proofs in the flag-style format.
1 Introduction

In this paper we describe the system Yarrow, an interactive proof assistant based on the theory of Pure Type Systems, a family of typed lambda calculi. In typed lambda calculi, theorems and proofs can be represented as well-typed terms, proof checking amounts to type checking, and proof construction to the construction of a term of a given type. These properties make typed lambda calculi well suited as formal basis of systems that support interactive construction of proofs, so-called proof assistants. Some well-known proof assistants of this kind are Coq [Coq97], LEGO [LP92], and Alfa [Hal97].

Pure Type Systems (PTSs) are a family of typed lambda calculi with common structural properties [Bar92]. Different members of the family can be selected by means of suitable parameter settings, and this feature makes PTSs particularly suited for experimentation with calculi of different strengths. For practical purposes a facility for adding global or local definitions is indispensable. Extension of PTSs with a definition facility leads to so-called DPTSs as presented in [SP94].

The system Yarrow has been designed as a flexible environment for experimentation with DPTSs. It can handle a large class of DPTSs, the so-called bijective DPTSs [Pol93] with a finite specification, which includes all systems of the lambda cube [Bar92]. A typical Yarrow session consists of: selecting a DPTS and loading several modules of definitions and theorems based on this DPTS, after which a number of proof tasks are carried out. Each proof task consists of the interactive construction of a term which is well typed with respect to the loaded context. Besides a conventional command line interface, Yarrow also offers a graphical interface with windows for global context and proof tasks, menu and mouse selection of tactics and subterms.

Yarrow has been coded entirely in the purely functional language Haskell [Tho96, Pet97], making use of the Fudget library [HC95] for the graphical interface. In this report we discuss various interesting aspects of the Yarrow software architecture:

1. Monads to mimic impure features in a purely functional language. The Yarrow system consists of several layers, such as the type checker, which needs error handling, the proof engine which needs a notion of state, and the user interface on top which requires IO. Error handling, state and IO can all be handled by means of monads [WBS9]. So we defined three layers of monads: error monads, state monads and a combination of error and state monads. By using in each layer of the program exactly the monad that is necessary, every layer gets exactly the resources it is entitled to.

2. Support for multiple interfaces: The Yarrow proof engine has been coded entirely in Haskell and can run on many different platforms. It has been designed in such a way that it can cooperate with various user interfaces. Two such interfaces have been developed. First, a simple command line interface, which can be used on any platform supporting Haskell, because this interface is based on the standard IO monads [Tho96]. Second, a graphical interface based on the Fudget library, which is a Haskell library available for a limited number of platforms. The coupling between user interface and engine is very thin, consisting of just a single function.

3. Generic output routines: In the command line interface, strings representing composite terms are composed from strings representing the parts. In the graphical user interface, views representing composite terms are composed from views representing the parts. These two representations are generated by means of a single set of polymorphic output routines.

4. Flexible term representations: An important aspect of proof assistants is the implementation of terms with bound variables and related notions such as substitution. The particular implementation used in Yarrow is such, that binding and substitution are handled in a basic layer of the system, independent of the kind of binder. This makes it easy to add new binding constructs, such as Sigma types or bounded quantifications.
In section 2 we give the design issues in more detail. Section 3 is devoted to the definition of the monads, and the advantages of this approach. In section 4 we describe the global architecture of Yarrow, which consists of three main components. Section 5 treats the top level user interface, and the communication with the engine, which is itself described in section 6. The third component consists of the service routines, including parsing and printing, and is described in section 7. Other features of Yarrow, not important for the architecture, but of interest to the user, are described in section 8. We give the conclusion in section 9.

2 Design Issues

Yarrow is designed to be a flexible proof-assistant. In this section we make this notion more precise, and identify the problems we have to solve to achieve this.

A proof-assistant is an interactive system. For every interactive system implemented in a purely functional language, a major design issue is how to handle state and errors. We elaborate a bit on this subject. Errors — also known as exceptions — may occur in many places in the implementation. The whole program will benefit from a uniform approach. The middle and upper levels of a proof-assistant have to be able to work with a state, that represents amongst other the global context (theorems that have already been proved), and the proof that is currently worked on. The top level of a proof-assistant must be able to perform input and output (IO), to handle the keyboard, the screen, the file system, and, in case of a GUI, the mouse.

Now we will specify what we mean with flexible. We consider the following aspects concerning the interface:

- The system should be set up in such a way, that it is possible to have several user interfaces available, that all work together with the same kernel.
- It should be easy to change notation, independently of the actual user interface.

We also wish flexibility in the type system that is implemented.

- The user can select the PTS he wants to use at run-time. It is not necessary to allow every PTS, but the common ones, e.g. the simply typed lambda-calculus, the second-order lambda calculus (system F), and other PTSs in the lambda-cube [Bar92] should be available.
- It should be easy to add other term constructions.

3 Monads

The problems concerning errors, state and IO can be solved by the use of monads, see [Wad92] for an extensive introduction. A monad is a type-constructor M with a pair of polymorphic functions,

\[ \text{unit}_M :: a \rightarrow M a \]
\[ \text{bind}_M :: M a \rightarrow (a \rightarrow M b) \rightarrow M b \]

Intuitively, a value of type \( M T \) is a calculation which yields a value of type \( T \) as result. It may have some additional effects, depending on the actual monad \( M \). The expression \( \text{unit}_M x \) denotes a calculation which just yields \( x \) as result, without any additional effects. The expression \( \text{bind}_M x f \) denotes the combination of \( x \) and \( f \): First \( x \) is calculated, which delivers a result \( r \). Then the calculation of \( f r \) takes place. Typically, a monad has some additional functions that introduce the additional effects we spoke about.

Now we introduce three particular monads that play an important role in the implementation of Yarrow.

The simplest monad is the error (exception) monad \( \text{Err} \). A value in such a monad is either a value of some type, if no error has occurred, or an error message. Apart from the usual functions on monads, the error monad has two other functions. First, a function \( \text{errE} \) that, given an
error message, produces a value in this error monad. Second, a function handleE that allows the programmer to handle (capture, catch) errors and leave the monad.

```
data Err a = Success a | Error ErrorMessage
    -- Err is a monad
unitE x = Success x
bindE (Error m) f = Error m
bindE (Success r) f = f r
errE :: ErrorMessage -> Err a
handleE :: Err a -> (ErrorMessage->b) -> (a->b) -> b
```

The next monad is the state and error monad State. From now on we will abbreviate this to state monad. A value v in such a monad is a state transformer: Given an initial value, v produces a pair of a new state and a value in the error monad. In the definition of State, we have parameterized over the type of the state. So State s is a monad with states of type s. Apart from the usual functions on monads, the state monad has functions for reading and updating the state. It also has a function that generates errors and one that handles errors, but the latter has a slightly different type than its Err counterpart, since we don’t want to lose the state if an error occurs.

```
type State s a = s -> (s,Err a)
    -- For every s, (State s) is a monad
unitS :: a -> State s a
bindS :: State s a -> (a -> State s b) -> State s b
fetchS :: State s s
updateS :: (s->s) -> State s s
errS :: ErrorMessage -> State s a
handleS :: State s a -> (ErrorMessage -> State s b) ->
    (a -> State s b) -> State s b
```

The last monad contains IO, state and errors. It is a straightforward combination of Haskell’s built-in IO monads and the state monads given above.

```
type Imp s a = s -> IO (s,Err a)
    -- For every s, (Imp s) is a monad
    -- It has functions
```
From now on, when we use the notion IO monad, we mean this Imp.

We have used Haskell's type classes [WB89, Jon95] to define overloaded functions unit and bind so that all monads can be used in exactly the same way (this makes a slight modification of State and Imp necessary). In fact, we could follow this approach even further, by making overloaded functions fetch and update for monads that have a state (State and Imp).

Why didn't we only define the Imp monad, since everything we can do with Err and State can also be done with Imp? One big advantage of this layered approach is security. We give each routine just the monad it needs, and not something more powerful. In this way a routine cannot access resources (i.e. IO or state) that it is not entitled to. We give a few examples of this. The typing routine uses the Err monad, so it cannot change the state, nor can it do any IO. The implementations of tactics use the State monad, because they should change the state concerning a proof, but may not do IO; IO is implemented only in the user interface. So by considering the resources a routine is entitled to, we decide which monad a routine will use. During the implementation, we shifted only a few, auxiliary, routines from one monad to another, so the distinction between different kinds of monads gave rise to only little additional effort.

Another advantage of the layered approach is the possibility to use monads without IO within a non-monadic function. This is quite important, because the graphical user interface is based on the Fudgets system, which doesn't work with IO monads at all. Therefore, it is instructive that the proof-engine does not use IO monads, but at most state monads, which can be converted to ordinary functions (with the state as argument and as result). Thus the graphical interface can invoke the proof-engine. The different layers of monads are sketched in figure 1 on page 4. An arrow from block A to block B means that a function living in A can invoke functions living in B. In other words, a function in B can be converted to one in A.

In the following sections, we show that monads are used throughout the implementation of Yarrow. We will see that higher layers in the implementation use higher level monads.

4 System architecture

Figure 2 shows the architecture of Yarrow. Each block is a module with a certain functionality, and each arrow indicates a dependency. The block labeled "Engine" is the part that defines the objects of our system (terms, contexts, specification of PTSs) and the functions that manipulate these objects, like the typing routine, the tactics, and the routines that extend the context.

The block labeled "Service routines" consists of all sorts of routines needed for the user interface without actually performing any IO. This includes printing and parsing routines, and the displaying of help texts. The service routines depend on the engine in a rather trivial way. They only use the representation of the objects and some elementary functions on these representations.

The block labeled "Top level user interface" (TLI) contains the main loop of the program. This handles input by the user, sends the appropriate messages to the engine, and presents the results of these messages on the screen. The TLI uses several service routines, but only one function of
the engine. The combination of the TLI with the service routines forms the user interface. We have split the user interface in these two parts because of modularity; every pair of TLI and service routines can be combined into a user interface.

The following three sections each describe one of the blocks and the connection with other blocks.

5 Top level user interface

In section 5.1 we describe the communication between the TLI and the engine. Currently, there are two top level interfaces available. A command line interface (section 5.2) and a graphical user interface (section 5.3).

5.1 Communication

In this section we describe how the communication between the top level user interface and the Yarrow engine is implemented. From the viewpoint of the TLI, the engine is just a database to which queries can be sent, which will return a certain result. All possible queries are packed into a datatype called Query, and all possible results into Result. The only function from the engine available to the TLI is doQuery, which handles all queries. So the communication between the TLI and the engine can be visualized as in figure 3.

Since we work in a purely functional language, the engine does not own a state. How can the engine then change the context, for example? The answer is that doQuery is a state transformer. Concretely:

\[
\text{doQuery} :: \text{Query} \rightarrow \text{State} \text{ EngineState Result}
\]

So another way of viewing the information flow is that a pair \((\text{query}, \text{engineState})\) is sent to the engine, which responds with a pair \((\text{result}, \text{engineState}')\). It is the responsibility of the TLI that the next query is paired with this new \text{engineState}'

Now we will consider the datatypes Query and Result in more detail. They are set up in such a way, that the TLI never has to perform manipulations on terms, so that all computational functionality resides in the engine. A stylized subset of the queries and results is depicted in figure 4, in total there are about 30 queries and 20 results. They are grouped around five subjects. The following list gives these subjects and one or two representative queries with their associated results.

1. Proof-tasks. The query \(Q\text{ProveVar} (v, t)\) starts a new proof-task, where \(t\) is the goal, and \(v\) is the name of the goal. The engine gives as result \(R\text{ProofTask taskId toProve}\). The variable taskId contains the identification of the proof-task (there may be several proof-tasks), and toProve contains all information about the proof-task, i.e. the proofTerm, and all the subgoals. The query \(Q\text{Tactics taskId tacticTerm}\) performs the tactic associated with tacticTerm on proof-task taskId, with result \(R\text{Tactic toProve}\).
data Query =
  QProveVar (Vari,Term) |
  QTactic TaskId TacticTerm |
  QLoadModule ModuleName String |
  QDeclareVars ([Vari],Term) |
  QGiveGlobContext |
  QSetTypingSystem System |
  QGiveType Term |
...

data Result =
  RProofTaskId TaskId ToProve |
  RTactic ToProve |
  RModulesAre [ModuleInfo] |
  RLoadList [ModuleName] |
  RGlobContextIs GlobContext |
  RTypingSystemOk |
  RTypes (Term,Term,Sort) |
  RError ErrorMessage |
...

Figure 4: A subset of the queries and results

2. Loading and saving of modules (i.e. a group of related definitions). The query QLoadModule name contents request the loading of module name. Since the engine cannot do any IO, the TLI also gives as parameter the contents of the file associated with name. Usually, this leads to the result RModulesAre, which indicates that loading is done, and gives the new list of currently loaded modules.

But if the module imports other modules mods, a result RLoadList mods is returned, to which the TLI reacts by issuing QLoadModule queries. Here, the narrow communication channel and the inability of the engine to perform IO make the implementation of loading of modules awkward. This is severed by implementing the printing of status messages, that indicate which module is currently being loaded. All of this is forced through the narrow channel, but the resulting code is not elegant.

3. The global context. The query QDeclareVars adds one or more declarations to the context, and QGiveGlobContext requests the current context. Both result in RGlobContextIs.

4. The parameters of the system. The query QSetTypingSystem asks for a change in PTS, and this is answered with RTypingSystemOk.

5. Calculation of normal forms or types. For example, QGiveType t requests calculation of the type of t, and RTypes returns t with its type and sort.

So each query is associated with a small number of results (usually one). Apart from that, the result RError is always allowed.

The big advantage of having only this narrow communication channel between the TLI and the kernel is modularity. This reveals itself in three ways:

- The connection between the engine and the TLI is very sharply defined.
- The TLI uses IO monads or fudgets (both with state) in order to perform IO. Since there is only one communication channel, the conversion between the ordinary functional types of the engine on one hand, and the IO monads or fudgets on the other hand, is limited to one place in the program.
- Since different queries can have the same sort of result, the output for these queries will be uniform.

5.2 Command line interface

The actual implementation of the textual interface is not very interesting. Using the IO monads defined in section 3, it is quite straightforward to implement the command line interface.
Therefore we concentrate on how this top level user interface is used, which is quite similar to Coq and LEGO. We do this by a simple example.

Suppose the user is working on a proof of $\forall P, Q, R. (P \implies Q \implies R) \implies (Q \implies P \implies R)$. After a few steps, the goal is to prove the following.

\begin{verbatim}
P : *
Q : *
R : *
H : P->Q->R
\end{verbatim}

Above the dashed line is the local context of the goal; these declarations and assumptions may be used to prove the proposition under the line. The user now types intros to perform the $\rightarrow$ introduction tactic as often as possible.

\begin{verbatim}
P : *
Q : *
R : *
H : P->Q->R
H1 : Q
H2 : P
\end{verbatim}

The $\rightarrow$ elimination tactic on $H$ is now invoked as follows.

\begin{verbatim}
  $ apply H
  2 goals

  P : *
  Q : *
  R : *
  H : P->Q->R
  H1 : Q
  H2 : P

  R

  2) Q
\end{verbatim}

At this point there are two goals. The first goal is to prove $P$ in the local context shown; the second goal is to prove $Q$, but its local context is not shown. (In this case it is the same context as for $P$). Here we abandon the proof.

5.3 Graphical user interface

The Fudgets library [HC95] is used to implement the graphical user interface. Fudgets offers mechanisms to construct and combine windows, buttons, menus and other window gadgets. The implementation of the graphical user interface is described in detail in [Rai97].

We treat again the example given in section 5.2, but now in the graphical user interface. The starting point is shown in figure 5. There are two windows, one for the global context (on the left), and one for the proof we are working on (on the right). For this example, all actions take place in the latter. The main area of this window is divided into three parts. The top part shows the local context of the current goal, the middle part shows the current goal itself, and the bottom part shows the other goals. On the right-hand side of the window are several buttons
for invoking commonly used tactics. A complete list of tactics can be found under the menu bar entries Tactics and Special. The far bottom of the window consists of a command line, where the user can type in commands, and below that, a status line which displays possible errors.

The user invokes the introduction tactic by clicking on the button labeled Intro. This results in an immediate change in the main area of the window. The user then selects the variable H in the local context by clicking on it, and clicks on the button labeled Apply. These actions form the elimination tactic. The same effect can be achieved in several other ways. First, by clicking Apply without having selected a term. This causes a pop-up window to appear, in which H can be typed in. Second, by selecting the apply tactic from the menu-bar entry Tactics. Third, by selecting H, summoning the pop-up menu, and selecting the Apply H option from this menu. Last, by typing Apply H in the command line on the bottom of the screen.

The graphical user interface has the usual advantages of GUls over CLls. One particular example for Yarrow is that unfolding one occurrence of a defined variable (e.g. in the goal) is achieved by clicking on this occurrence and clicking the Unfold-button. The user of Yarrow with the CLI has to count which occurrence he wants to unfold, and type in this number as parameter to the unfold tactic.

6 Engine

The engine is subdivided into four blocks, as illustrated in figure 6, where the arrows symbolize dependency. The blocks on the right define the datatypes representing terms, contexts and PTSs (on the lower level) and representing tactics (on the upper level). The blocks on the left define the algorithms working on these datatypes; the typing and reduction routines etc. on the lower level, and the implementation of tactics and other commands on the upper level. So the lower level concerns terms, and the upper level concerns commands. We now treat each of the block in more detail, and state which monads are used.

The representation of PTSs, terms and contexts is given a separate treatment in section 6.1.

All routines working on terms are non-monadic or use the error monad. Often some part of the
Implementatio
Typing
Reduction

Implementation
of tactics

Representation
of tactics

figure 6: Architecture of the engine

state (e.g. the current PTS) is given as argument to these routines. The most important routine is the typing algorithm. Essentially, we have used Poll's algorithm for bijective PTSs [Pol93], since this class of bijective PTSs includes all systems in the lambda-cube [Bar92].

The block defining the representation of tactics is necessary because of two reasons. First, it allows all tactics to be covered by one query (see section 5.1). Second, tactics are printed in a summary, when a proof is finished.

Finally, the implementations of tactics and other commands use mostly state monads. The function doQuery (see section 5.1) is also of this form; it acts as a sort of a traffic warden. It inspects the query and calls the appropriate command.

6.1 Representation of terms

A major design issue when implementing a proof assistant is how to represent terms. The most important decision to be made, is how bound variables must be handled. The most common possibilities are:

**Named variables** Annotate each binder with the bound variable, and use identical representations for bound and free variables. Care must be taken for name clashes and substitutions, but these subtleties are hidden in a few routines.

**de Bruijn indices** Represent a bound variable by an index, which is the number of binders between the occurrence of the variable and its binder.

First we observe that names have to be retained anyway, since the clearest way to print bound variables is using names. That means α-conversion is still needed, in order to print terms in an unambiguous way. This is of crucial importance, since the user assigns meaning to the term as it is printed, and not as it is represented internally.

So the simplest way of representing bound variables is using names (it isn't necessarily the most efficient). Therefore we have chosen this approach in Yarrow.

Another issue is how to represent terms, so that they may easily be extended with other term constructions than those of PTSs, e.g. records and existential types. We chose to set up our datatype representing terms in a way reminiscent of Combinatory Reduction System [KvOvR93]. We consider three categories of terms.
• The **basic category** contains terms that do not have any subterms. For any Combinatory Reduction System, variables belong to this category. For PTSs, sorts also belong to this category.

• The **non-binder category** contains terms that have a number of subterms, and do not bind any variables in those subterms. For PTSs, only applications belong to this category. Applications have two subterms.

• The **binder category** contains terms that bind one variable of some type in one subterm called the body. Furthermore, they may have a number of subterms in which the variable is not bound. For DPTSS, abstractions, II-types and local definitions belong to this category. The definiens in a local definition is an example of a subterm in which the variable is not bound.

This definition of binder category could be generalized so that multiple variables may be bound, but this is not necessary for our purposes.

The small disadvantage that this representation is slightly less efficient in execution space and time, is hugely offset by the advantage that critical routines like substitution, determination of free variables, and changing a bound variable (α-conversion) are written once and for all, and never need to be adapted. Furthermore, routines like unification and checking for convertibility of terms become more uniform.

The actual Haskell code that defines terms:

```haskell
type Item = (Vari, Term, Sort)           -- Var with type and sort
data Term = Basic BasicCat |               -- No subterms
       Nonb NonbCat [Term] |              -- Non-binders with
       Bind BindCat [Term] Item Term       -- Binders with subterms,
                                           -- var and type, and body

data BasicCat = Vr Vari |                  -- Two subterms
               Srt Sort |                    -- No additional subterms
               Hole Hnum

data NonbCat = App                        -- No additional subterms
               All |                      -- No additional subterms
               Delta                    -- One additional subterm

data BindCat = Abs |                      -- No additional subterms
               All |                      -- No additional subterms
               Delta                    -- One additional subterm

We always annotate a variable with its sort (the type of its type), hence Item is a triple. We have a basic construct called Hole which is used for the construction of proof terms.

**Example 1** The term let x:=three:Nat. even x is encoded as:

```
Bind Delta [Basic (Vr three)] (x, Basic (Vr Nat), *)
    (Nonb App [Basic (Vr even), Basic (Vr x)])
```

where we assume three, Nat, even and x are of type Vari, and * is of type Sort.

For routines like type-checking, that are specific for DPTS-terms (instead of being uniform for general terms with bounded variables), we supply constructor and destructor functions for each form of DPTS terms. For example, the constructor for application takes two terms t and u and returns the application Nonb App [t, u]. The corresponding destructor takes a term t and delivers a boolean stating whether t is an application, and the subterms of t if applicable.

Contexts are represented in a similar vein as terms, so that it is easy to move a binder into the context.
7 Service routines

The service routines are divided in six modules, as illustrated in figure 7. We give a short explanation of the modules.

The module “Representation of commands” defines the representation of the textual commands, so that these commands are available to all top level interfaces. In fact, the GUI uses this module, so that the user may also type in commands, instead of using the mouse.

The two modules that define the parser routines are quite straightforwardly implemented. The parser routines are written in an imperative style, using state monads, although parser combinators may be more elegant (e.g. [Hut92]).

The remaining modules “Printing terms”, “Printing tactics” and “Help texts” are all concerned with output, and are given a separate treatment in section 7.1, in which we explain how these routine are made independent of the actual TLI.

7.1 Polymorphic printing routines

Although the various interfaces differ greatly in outward appearance, they actually have many structural similarities which can be exploited by making the output routines polymorphic. For instance, when outputting a term as a value of some type a (e.g. String or Graphics), we need two functions. One function turns an atomic piece of text into an a, so this function has type String -> a. It is used for outputting variables or symbols (like \ and ->). The other function concatenates a number of a's together, so it has type [a] -> a. It is used to print composite terms. So, a representation using type a is characterized by a value of tuple type (String->a, [a]->a), hence the print routine for terms is declared as

printTerm :: Display a -> Term -> a
type Display a = (String->a, [a]->a)

A similar approach has been followed for generation of output in the help system. In this case the print routine is declared as:

printHelp :: Display a -> HelpText -> a

Actually, both printing routines are a bit more complicated, to cater for selection of subterms and clicking on links in the help texts.
8 Other features

In this section we give a short description of features of Yarrow that are not very relevant for the architecture of the system, but are quite relevant to users. We first treat briefly some miscellaneous points, and then elaborate on the presentation of proofs using flags.

- Yarrow does not allow dependent goals. This means that a goal may not depend on the proofterm to be found for another goal. In Coq and LEGO, dependent goals are allowed in a restricted way; in Alfa dependent goals are always allowed, this makes a calculus of explicit substitution necessary.

- There are special tactics for predicate logic, and for rewriting using the Leibniz equality. Before these tactics can be used, the user has to indicate the lemmas that express the introduction and elimination rules for these connectives. In this way, these special tactics are independent of the actual PTS.

- There are two simple mechanisms to increase readability of A-terms. First, we have infix notation, quite similar to Haskell. Second, we have a mechanism for implicit arguments. These mechanisms work very well together. E.g. we can now write $x + 0 = x$ instead of $\text{eq } \text{Nat } (\text{plus } x \text{ } 0) \ x$.

- There are no inductive types (they do not belong to the framework of PTSs).

- There is not an auto-tactic, which performs automatic proof-search.

- There is extensive on-line help available in both interfaces.

8.1 Flags

Yarrow has the ability to print proofs in the flag-style format [Ned90]. This is a formal notation for proofs which makes it clear which hypotheses are valid at each point in the proof. Every hypothesis is written in a box. Connected to this box is a "flagpole", which indicates the scope of this hypothesis. The justification for every proposition is written behind it; a justification typically consists of a logical construct (e.g. $\land, \lor$), a letter that indicates whether this construct is Introduced or Eliminated, and references to the lines or theorems which the current line depends on. For an example of this style, see figure 8, which proves in a certain context that the insert function keeps ordered lists ordered. We prefer this more formal notation to a textual presentation, because the flag-style format is clearer, more concise, and the propositions are not embedded within English "prose". A very similar notation is used in Jape [SB96], where this layout is used to build proofs interactively.

The algorithm that produces this presentation of a proof from a proof-object is quite similar to the ones described in [CKT95] and [Cos96], although they produce proofs in pseudo natural language. The basic algorithm is natural and simple: the presentation of a proof-term $p$ is a composition of the presentations of the subterms of $p$. However, this produces quite lengthy proofs. A big improvement is the combination of similar steps into one step, e.g. in line 20 of figure 8, two steps are contracted into one (VE of line 9 with term $b$, and $\Rightarrow$E of the result with line 19). Up to this improvement, the algorithm in [Cos96] and ours are similar. An important difference is that their algorithm works for the Calculus of Inductive Constructions, whereas ours works for PTSs.

9 Conclusion

The complete proof-assistant has been written in Haskell. The suitability of this language varies over different parts of the program. The engine and service routines benefit from Haskell features like polymorphism, higher order functions, and especially user-defined datatypes. For the top level user interfaces, these features do not play such an important role. The command line interface with
its IO monads is imperative in style and could, in itself, equally well be written in an imperative language. The graphical user interface uses the Fudgets library, which allows easy construction of windows from components like buttons and text-fields. However, Fudgets offers only a very basic functionality; some essential features, like saving files, and many fancy features, like dragging, are absent. Furthermore, it is slow and memory demanding. In our experience, the concepts of the Fudgets library made programming a GUI relatively easy and enjoyable, but the library is not ripe enough to be used in a "real-world" application.

Monads are introduced in [Wad92] to mimic impure features, like errors, state and IO. These features are necessary in a proof-assistant, and therefore we used monads. This turned out to be a good decision: monads allow a flexible, elegant and uniform treatment of errors, state and IO. However, in a big program, such as Yarrow, we cannot just use one monad throughout the program, since it would give the whole program access to all resources (state, IO). Therefore we defined different layers of monads, and each layer of the program gets exactly the monad it needs. For example, the typing routine uses error monads, and cannot make any changes in the state, nor perform any IO. Haskell's type classes [Jon95] make a consistent use of monads possible. We expect that this approach with different layers of monads can be applied to many big programs written in Haskell.

Another important issue is how to design the proof-assistant in such a way that several user interfaces can be used with one engine. We decided to create a very narrow communication channel of queries and results, that are handled by one function of the engine. This approach is successful. It helps to separate the tasks of the total program, and allowed us to implement the user interfaces quite independently from the engine. Furthermore, it promotes uniformity, e.g. similar commands always give the same format of output. The strict adherence to this discipline made the implementation of a few commands with a lot of IO (viz. loading and saving of modules) complicated, but the other commands are well-suited to this approach.

We have claimed Yarrow is a big program. Let us make this more precise. The engine consists of about 7600 lines of code. About half of this is for the kernel, which includes representation of terms, reduction, typing and unification. The service routines are implemented in 2500 lines of Haskell. The command line interface is about 1000 lines of code, which is rather small compared with the graphical user interface, consisting of about 5000 lines. This brings the total program size to 11000 to 15000 lines. The most complicated routines lie in the kernel of the program, particularly an efficient reduction routine and an unification routine. Also much effort has gone into the apply and rewrite tactics.

All in all, the coding of a complete proof-assistant in Haskell has been a successful experiment. We intend to extend Yarrow with records and subtyping, and the graphical user interface with proof by pointing. Yarrow is electronically available from the world wide web [Zwa97].

Acknowledgements

Many thanks to Kees Hemerik for his stimulation and help in writing this article, and to Eric Ruijmakers for implementing the graphical user interface. We also wish to thank a few anonymous referees.

References


Computing Science Reports

In this series appeared:

<table>
<thead>
<tr>
<th>Report No.</th>
<th>Authors</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>96/01</td>
<td>M. Voorhoeve and T. Basten</td>
<td>Process Algebra with Autonomous Actions, p. 12.</td>
</tr>
<tr>
<td>96/02</td>
<td>P. de Bra and A. Aerts</td>
<td>Multi-User Publishing in the Web: DreSS, A Document Repository Service Station, p. 12</td>
</tr>
<tr>
<td>96/04</td>
<td>S. Mauw</td>
<td>Example specifications in phi-SDL.</td>
</tr>
<tr>
<td>96/06</td>
<td>W.M.P. van der Aalst and T. Basten</td>
<td>Life-Cycle Inheritance A Petri-Net-Based Approach, p. 18.</td>
</tr>
<tr>
<td>96/07</td>
<td>M. Voorhoeve</td>
<td>Structural Petri Net Equivalence, p. 16.</td>
</tr>
<tr>
<td>96/11</td>
<td>T. Laan</td>
<td>AUTOMATH and Pure Type Systems, p. 30.</td>
</tr>
<tr>
<td>96/12</td>
<td>F. Kamareddine and T. Laan</td>
<td>A Correspondence between Nuprl and the Ramified Theory of Types, p. 12.</td>
</tr>
<tr>
<td>96/13</td>
<td>T. Borghuis</td>
<td>Prioreterangan in Type Logic in Modal Pure Type Systems, p. 61</td>
</tr>
<tr>
<td>96/15</td>
<td>M.A. Reniers and J.J. Vereijken</td>
<td>Completeness in Discrete-Time Process Algebra, p. 139.</td>
</tr>
<tr>
<td>96/16</td>
<td>E. Boiten and P. Hoogendijk</td>
<td>Nested collections and polytypism, p. 11.</td>
</tr>
<tr>
<td>96/17</td>
<td>P.D.V. van der Stok</td>
<td>Real-Time Distributed Concurrency Control Algorithms with mixed time constraints, p. 71.</td>
</tr>
<tr>
<td>96/18</td>
<td>M.A. Reniers</td>
<td>Static Semantics of Message Sequence Charts, p. 71</td>
</tr>
<tr>
<td>96/19</td>
<td>L. Feijs</td>
<td>Algebraic Specification and Simulation of Lazy Functional Programs in a concurrent Environment, p. 27.</td>
</tr>
<tr>
<td>96/22</td>
<td>W.M.P. van der Aalst</td>
<td>Structural Characterizations of sound workflow nets, p. 22.</td>
</tr>
<tr>
<td>96/23</td>
<td>M. Voorhoeve and W. van der Aalst</td>
<td>Conservative Adaption of Workflow, p. 22</td>
</tr>
<tr>
<td>96/24</td>
<td>M. Vaccari and R.C. Backhouse</td>
<td>Deriving a systolic regular language recognizer, p. 28</td>
</tr>
<tr>
<td>97/06</td>
<td>M. Frantzen</td>
<td>Tools for the Construction of Correct Programs: an Overview, p. 33.</td>
</tr>
<tr>
<td>97/07</td>
<td>J.C.M. Baeten and J.A. Bergstra</td>
<td>Bounded Stacks, Bags and Queues, p. 15.</td>
</tr>
</tbody>
</table>