Theorem Proving and Program Synthesis with Oyster *

Christian Horn † Alan Smaill ‡

Abstract

Martin-Löf type theory provides a formal framework for the construction of verified programs, both specified and written in the type theory. We describe an implementation of the type theory that aims to provide an environment for software engineering using this approach. We illustrate this by describing the synthesis of a simple evaluator for arithmetic expressions in the system.

1 Introduction

There is currently much interest in providing tools for computer-aided reasoning about programs. This can be done using a general purpose theorem prover. For example, the Boyer and Moore theorem prover [1] provides reasoning within the quantifier-free predicate calculus, and programs can be shown correct by formulating properties of the programming language in question within this logic. More recently various logical framework systems have been proposed, where a variety of logics can be treated uniformly. Hence reasoning using program logics can be automated, as in [2].

Another approach is to reason in the first instance not about programs, but about specifications and their satisfiability. Where the specification is given in Martin-Löf Type Theory [3], a program corresponding to the specification can be computed by a simple extraction process from a proof of the satisfiability of the specification. This is the approach taken here. Various realisablity notions may be used to perform this program extraction task for other proof systems, for example Girard's F_ω [4]. Viewed in this way program synthesis becomes a task of theorem proving, and the design decisions for the algorithm correspond to choices of proof rules in a natural way.

There are several implementations of theorem provers based on Martin-Löf Type Theory, for example [5, 6]. In this paper we describe a new implementation which aims to provide support for software engineering in this context.

*This research was supported by SERC Visiting Fellowship GR/F/69703 to the first author and SERC grant GR/E/44598 for the second author.

†Dept. of Mathematics, Humboldt-University, Berlin. Email: C.Horn@uk.ac.edinburgh

‡Dept. of Artificial Intelligence, University of Edinburgh. Email: A.Smaill@uk.ac.edinburgh

universe	type	constructor	destructor
u0	**void**	—	**any**(x)
	unit	**e**	—
		void, **unit**	**if**(x, s, t)
u(U)	$x{:}\,A\#B_x$	(x, y)	**spread**$(x, a{:}\,b{:}\,t_{a,b})$
	$x{:}\,A \rightarrow B_x$	$\lambda(a{:}\,t_a)$	f **of** x
	$x{:}\,A\backslash B_x$	**inl**(a), **inr**(b)	**rec_ind**$(x, a{:}\,s_a, f{:}\,b{:}\,t_{f,b})$
	a **in** T	**axiom**	—
	$a = b$ **in** T		
		$U, x{:}\,A\#B_x$, $x{:}\,A \rightarrow B_x, x{:}\,A\backslash B_x$ a **in** $T, a = b$ **in** T	—

2 The Oyster System

Oyster is a fast, interactive, programmable, theorem proving system. It was originally designed in an attempt to reconstruct the NuPRL system [5]. Around the Oyster kernel [7] we have built over the last two years an environment for theorem proving and program synthesis.

The strength of the system lies in the ability to deal with full higher order logic for the specification and implementation of mathematical theories provided by the type theory. The Oyster approach is especially useful for the development of verified software, but recent work has suggested its suitability also for design and configuration tasks.

2.1 The Structure of the Logic

The Oyster Logic may be considered as yet another formalisation of Martin-Löf type theory. As basic types we assume **void**, representing the empty set, and **unit**, representing the singleton set with the element **e**. **unit** and **void** may be considered as *truth* and *contradiction*. The basic types form the universe **u0** of *decidable truthvalues*, i.e. the only elements of **u0** are **unit** and **void**, and both are well distinguished, so that we have a selector operation on **u0**, which allows case analysis on truthvalues.

Once a universe U is given, the next higher universe **u**(U) is defined to consists of the type U, together with all types that belong to U, and to be closed under propositional, dependent *product*, and *function* and recursive *union* type constructs. Dependent product and function types may be interpreted as existential and universal quantifiers, whereas recursive union types enable us to define interesting domains. We put the usual restrictions on recursive types. The equality in the new universe is fully intensional. Propositional types represent the membership and equality judgements. Typehood is considered as membership to an appropriate universe, which has to be supplied explicitly.

The logic itself is assumed to be closed, in that an extension of the type system requires the generation of a new Oyster system, although this process may be supported. We prefer conservative extensions to the logic, in that the new types together with its operations are explicitly defined and compiled into the Oyster system during the warm-up phase. This has the advantage of keeping the logic and therefore the kernel system small, and makes metalogical investigations easier, whereas the overhead involved during the system start remains altogether below half a second. Notational definitions supplied with the standard systems are:

- the non-dependent product, function and union type constructs, which might be considered as intuitionistic propositional connectives (we use ~ for anonymous variables):

  $A\#B \iff \sim\!: A\#B$, $A \to B \iff \sim\!: A \to B$, and $A\backslash B \iff \sim\!: A\backslash B$;

- some shorthand notations, like $f(x) \iff f$ **of** x for identifiers f explicitly declared in the hypothesis list of a theorem, $\mathbf{fst}(x) \iff \mathbf{spread}(x, u\!:\sim\!: u)$, and $\mathbf{snd}(x) \iff \mathbf{spread}(x, \sim\!: v\!: v)$, for accessing the components of a pair and $\mathbf{decide}(x, a\!: s_a, b\!: t_b) \iff \mathbf{rec\_ind}(x, a\!: s_a, \sim\!: b\!: t_b)$ for distinguishing elements of a plain, i.e. nonrecursive, union type;

- the type of natural numbers is defined as recursive type $\mathbf{nat} \iff n\!: \mathbf{unit}\backslash n$ with constructors $0 \iff \mathbf{inl(e)}$ and $\mathbf{s}(x) \iff \mathbf{inr}(x)$, induction schemas $\mathbf{pind}(x, s, a\!: b\!: t_{a,b})$ and $\mathbf{cvind}(x, a\!: b\!: t_{a,b})$, ordering $a < b$, test on equality $a == b$ and the usual arithmetic operations $a + b$, $a - b$, $a * b$ and a/b;

- the type constructor t **list** $\iff l : \mathbf{unit}\backslash(t\#l)$ with constructors [] and $a.b$ (or using the Prolog-like notation $[a|b]$), a primitive induction schema $\mathbf{list\_ind}(x, s, a\!: b\!: c\!: t_{a,b,c})$, the operations $\mathbf{car}(x) \iff \mathbf{list\_ind}(x, \mathbf{e}, a\!: \sim\!: \sim\!: a)$ and $\mathbf{cdr}(x) \iff \mathbf{list\_ind}(x, \mathbf{e}, \sim\!: b\!: \sim\!: b)$ as well as the list concatenation $x :: y \iff \mathbf{list\_ind}(x, y, a\!: \sim\!: c\!: a.c)$; and

- the type **string**, which is defined as **nat list** and allows to adopt the Prolog convention of manipulating strings "ABC..." as lists of the corresponding ASCII-codes.

2.2 The Structure of Theorems

As is standard, we follow the "Propositions as Types" principle so that propositions in the specification language are treated as types in the type theory. Truth of propositions is identified with inhabitation of the corresponding type. The proof of a theorem consists in the construction of an element of the type denoted by the main goal of the theorem.

We use a sequent calculus formulation of the logic, where the inference rules are attributed with termforms, thereby incrementally constructing an inhabiting element of the type of the main goal during the inference process. The *hypothesis list* in a sequent consists from the logical point of view of a list of types labelled with identifiers symbolically denoting elements inhabiting the types. From the technical point of view the hypothesis list may contain furthermore notational *definitions*, which are treated solely as abbreviations. The unfolding/folding of the definitions is done automatically during the system's input/output, therefore delegating the responsibility for ambiguities and possible confiusion to the user.

The global objects declared in the hypothesis list of a theorem may be considered as parameters to that theorem, thereby implementing polymorphism, or as denotations for properties assumed as general precondition. This allows us to introduce some modularity into the system, since the header allows us to assume intermediate results (though of course these have to be proved subsequently), and to localise the dependency information.

For example the theorem **double_neg** uses a definition for intuitionistic negation, and assumes that the universe $\mathbf{u}(1)$ is inhabited by an (arbitrary) element p, which is a trivial assumption, but might be read otherwise as a *declaration* of p to be an intuitionistic propositional truthvalue.

$$\begin{array}{lll} \textbf{double\_neg}: & [& not(x) \Longleftrightarrow x \rightarrow \textbf{void}, \\ & & p{:}\, \mathbf{u}(1) \\ &] & \vdash p \rightarrow not(not(p)) \end{array}$$

Working inside a theory, many theorems will share common declarations and definitions. We have adopted the concept of *include files* (***.inc**) which allows informal substructures to be created inside a theory by simply sharing assumptions.

2.3 The Structure of Proofs

Proof are constructed in a top-down approach by the application of inference rules starting with the goal to be proved. The inference rules are selected using a rule specification (like **intro**, **elim**, ...), which together with the current goal and the available hypotheses identifies the inference rules. Building a complex proof using only single inference rules makes the proof structure as a whole totally incomprehensible, and therefore unacceptable for our purposes, because we assume mechanized proofs will be accepted by human beings only if they are still (at least in their main contours) comprehensible; this is also important since design of algorithms depends on the place of certain key inferences in the structure of the corresponding proof.

The appropriate size for the proof steps, being neither too fine-grained nor too large for easy comprehension, depends naturally on the constructs involved. Usually such a proof step consists of between several dozen and several hundred rule applications. These proof steps are guided by tactics, *i.e.* programs which control the sequence of applications of inference rules. The *tactic language* consists of the rule specifications and control primitives (**then**, **try**, **or**, **rep**, ...), but may be extended by arbitrary user supplied Prolog code. The standard autotactic, which is mainly equivalent to **rep intro** but involves certain elimination steps, solves many of the tedious details automatically, as for example the theorem **double_neg** thereby implicitly constructing the extract term $\lambda(v_0{:}\, \lambda(v_1{:}\, v_1 \textbf{ of } v_0))$.

During the proof process the user discovers repeated command sequence, for which he can easily define abbreviating macros using the Prolog interface. The next level is to develop sophisticated tactics which automate most of the proof activities in the application domain. Writing such tactics is still an "ordinary" programming task, *i.e.* one can apply principles of structured programming and write the tactics required in a top down approach from scratch, guided simply by the general idea of the proof under consideration. But this works only if the user has already enough experience in writing this kind of meta program. The creation of suitable programming methodologies for writing tactics is still an open task.

2.4 The Structure of Theories

Oyster assumes that theories consist mainly of definitions, theorems with their proofs, and generalised proof methods, so-called *tactics*. As the result of a proof one gets a file containing the extract term of that proof, *i.e.* an executable term of enlarged λ-calculus which represents the computationally interesting aspects of the proof. All the information is contained in several files organized in a hierarchical way, representing inheritance properties. We distinguish files containing theorems (***.thm**), possibly shared hypotheses (***.inc**), the corresponding extract terms (***.ext**) and tactic files that control the proof process (***.tac**).

Often work can be based on prefabricated theories, just like standard modules in software design. If these theories are really well designed, then the collection of tactics supplied with a theory comes together with a *specifications* for the proof methods supplied. Now the idea is that if the user starts from these prefabricated theories then it would be useful to have some automated "watch dog", which instructs him in the application of these tactics, or perhaps applies them automatically [8]. Therefore on top of Oyster a proof planning system, called Clam, has been implemented which searches for combinations of the available tactics [9] appropriate for the goal at hand. During the planning process the tactics are not directly executed, but instead the specification of the tactics and some simple metamathematical reasoning is used to construct a rough proof plan. This proof plan can than be used to guide the inference process immediately or to inspire the manual theorem proving process. Currently we try to develop a sample library of relevant program constructs and their mathematical counterpart. We do not consider proof-planning further in this paper.

2.5 The Structure of the System

The Oyster System is relatively small and fast, but nevertheless highly portable. The System is written in C, Prolog and the Oyster Language itself. It consists mainly of a special purpose Prolog implementation, and a theorem proving shell running on top of it. The source code consists of about 14,000 lines C-Code and 2,500 lines Prolog/Oyster-Code. The executable code of the complete system comprises less than 250kB and requires for optimal performance 8MB of swap space and about 1.5MB resident memory.

In discovering new application domains Oyster is used mainly as a interactive Theorem Prover. That means the user has to develop the ideas and to guide the proof directly. The user interacts with Oyster through a Prolog-like interface, with additional built-in-predicates controling the inference process and the organization of the proof environment. The Oyster interface looks still very primitive. But the invention of a really "brainstorming" theorem proving environment, which supports higher order logic, is still an open task.

Once a proof is finished using some (intelligent or frustrating) user interaction it is very useful to save just these critical parts. The Oyster systems supports the automatic generation of tactic files (***.tac**) out of a proof or subproof. These tactic files describe the inference process in a compact way, and may be considered therefore as something like the "first derivation" of the proof. A tactic file turns out to be usually as comprehensible as the main theorem, and often a look backwards at the end of the proof shows you the unnecessary side steps. It is often very useful to edit the tactic file at the end, because it can serve as a design documentation of the proof. In many cases it is possible to modify a tactic file to adapt an old proof to a slightly modified theorem. The use of tactic files enables a certain form of batch processing, which is very useful in the design of larger theories, where you often have to change your mind about certain details, but is quite practical in the event of overloaded machines too.

As theories are organized using the standard UNIX file system, Oyster makes use of the "make"-facility to keep track of all the dependencies between the definitions and the various theorems of a theory. So after some changes it is usually enough to run the **make**-command in the background, to check for all dependent theorems whether they are affected or can be automatically reproved.

When everything comes to an end, the extract term of a theorem is available in the corresponding ***.ext** file. This extract term can directly be executed from the Oyster system. If it is for example an λ-term describing a function, the execution of that λ-term in the UNIX-environment simulates a program which reads a term from the standard input stream and writes the result of evaluating the λ-term on that argument on the standard output stream. The composition of more complicated functions can than be performed easily using the process and file concept of the operating system enviroment, as we will see in the following small example.

3 Case Study: Development of an Evaluator

In this section we will try do develop a small application program: an evaluator for arithmetic expressions, i.e. a program which reads arbitrary character sequences and answers with the arithmetic value of the expression represented by that sequences, if it has any, or some error string otherwise. We will follow the top-down, refinement style of program development, and will prove the correctness of the implementation steps imvolved. In fact we will try do give a more realistic description of the program development process, which also involves some backtracking. We start with a theorem specifying the type of the function we are interested in:

$$eval : \quad [] \quad \vdash \mathbf{string} \rightarrow \mathbf{string}$$

For reasons of simplicity we assume as input strings natural numbers and fully bracketed expressions of the form $(t_1 + t_2)$, $(t_1 - t_2)$, $(t_1 * t_2)$, and (t_1/t_2). The set of arithmetic expressions can be described as a *recursive type* of the form $term \Longleftrightarrow t{:}\,\mathbf{nat}\backslash(t\#op\#t)$, whereas *op* denotes a four element set explicitly defined as the disjoint union of four one element sets. The task of evaluating a string can then be described as a three stage process: *analysing* the string and representing the result as term, if the string is syntactically wellformed; *evaluating* the term to a number, if the term is arithmetically wellformed; and *printing* that number or an error message in the other cases.

The syntactical analysis again can be regarded as consisting of a *scanning* process, separating the input string in to a list of tokens and thereby abstracting from details of the typographical representation, and a *parsing* process, which folds the list of tokens into a tree structure representing the abstract syntax of the input string. This separation is purely pragmatic, but has proven useful for the kind of formal language we are interested in, although in general there might be other ways of structuring. In our case a standard way of separating lexical and syntactical matters would be the definition of tokens as representing *numbers*, *operator* symbols, parentheses, and other symbols, collapsing all non expected characters into a special error token. The consideration of these subtasks leads to a refined specification:

$$\begin{array}{lll}\mathbf{eval}: & [& op \Longleftrightarrow \mathbf{unit}\backslash\mathbf{unit}\backslash\mathbf{unit}\backslash\mathbf{unit}, \\ & & token \Longleftrightarrow \mathbf{nat}\backslash op\backslash\mathbf{unit}\backslash\mathbf{unit}\backslash\mathbf{unit}, \\ & & term \Longleftrightarrow t{:}\,\mathbf{nat}\backslash(t\#op\#t), \\ & & scanner : \mathbf{string} \rightarrow token\ \mathbf{list}, \\ & & parser : token\ \mathbf{list} \rightarrow (term\backslash\mathbf{unit}), \\ & & evaluator : term \rightarrow (\mathbf{nat}\backslash\mathbf{unit}), \\ & & printer : (\mathbf{nat}\backslash\mathbf{unit}) \rightarrow \mathbf{string} \\ &] & \vdash \mathbf{string} \rightarrow \mathbf{string}\end{array}$$

The implementation step could be carried out immediately, and the correctness proof degrades to a simple type check, which is automatically solved by the standard autotactic:

$$?\!- \quad \mathbf{intro}(\lambda(s{:}\,printer(\mathbf{decide}(parser(scanner(s)), t{:}\,evaluator(t), \tilde{}{:}\,\mathbf{inr}(\mathbf{e})))))$$

Once the abstract syntax of a formal language is given (in our case represented by the type *term*), the concrete syntax (*spelling*) is specified in the easiest way by a mapping $\{.\}$: $term \rightarrow token$ **list** which describes the external representation of a given syntax tree. To elaborate this definition we do need some auxiliary notions for generating tokens, thereby making more explicit our intensions about the languages lexicographical structure. The simplest way, is to collect these additional definitions in a *include* file, which than might be shared between several subtasks.

$$\begin{aligned}
&token \iff \mathbf{nat}\backslash op\backslash\mathbf{unit}\backslash\mathbf{unit}\backslash\mathbf{unit},\\
&int(n) \iff \mathbf{inl}(n),\\
&op(x) \iff \mathbf{inr}(\mathbf{inl}(x)),\\
&lpar \iff \mathbf{inr}(\mathbf{inr}(\mathbf{inl}(\mathbf{e}))),\\
&rpar \iff \mathbf{inr}(\mathbf{inr}(\mathbf{inr}(\mathbf{inl}(\mathbf{e})))),\\
&error \iff \mathbf{inr}(\mathbf{inr}(\mathbf{inr}(\mathbf{inr}(\mathbf{e}))))
\end{aligned}$$

Supposed we have collected these definitions in a file *token.inc*, the task of parsing can be formalized by its inverse *spelling* function:

$$\begin{aligned}
\mathbf{parser}: \; [\;\; & op \iff \mathbf{unit}\backslash\mathbf{unit}\backslash\mathbf{unit}\backslash\mathbf{unit},\\
& \mathbf{include}(token),\\
& term \iff t{:}\,\mathbf{nat}\backslash(t\#op\#t),\\
& spreading(x, p{:}\,q{:}\,r{:}\,t) \iff \mathbf{spread}(x, p{:}\,s{:}\,\mathbf{spread}(s, q{:}\,r{:}\,t))\\
& \{t\} \iff \mathbf{rec_ind}(t, n{:}\,[int(n)], f{:}\,g{:}\\
& \qquad\qquad spreading(g, l{:}\,o{:}\,r{:}\,[lpar] :: f(l) :: [op(o)] :: f(r) :: [rpar]))\\
]\;\; & \vdash parser : (token\;\mathbf{list} \to (term\backslash\mathbf{unit}))\#\\
& \qquad t{:}\,term \to parser(\{t\}) = \mathbf{inl}(t)\;\mathbf{in}\;(term\backslash\mathbf{unit})
\end{aligned}$$

For the syntactical analysis we will use a LR(1)-Algorithm, which can be easily coded as primitive induction over lists. The specification of this algorithm is the essential step in the further development, as it anticipates the recursion schema for implementation and the choice of an appropriate induction schema for verification. For our synthesis proof we introduce the existence of such a utility function $pars : token\;\mathbf{list} \to (term\backslash token)\;\mathbf{list}$ via a lemma. Again it turns out to be useful, first to introduce an additional definition. We end up with two new subgoals: to proof the existence of such a utility function and still to prove the existence of a parser, but now under the (additional) assumption of having a utility function *pars* at hand:

$$\begin{aligned}
&?\!- \quad \mathbf{define}(pars_res \iff (term\backslash token)\;\mathbf{list}).\\
&?\!- \quad \mathbf{lemma}(pars{:}\,(token\;\mathbf{list} \to pars_res)\#\\
&\qquad\qquad t{:}\,term \to s{:}\,token\;\mathbf{list} \to pars(\{t\} :: s) = [\mathbf{inl}(t)] :: pars(s)\;\mathbf{in}\;pars_res).
\end{aligned}$$

$$\begin{aligned}
[1] \quad [] \quad & \vdash pars : (token\;\mathbf{list} \to pars_res)\#\\
& \qquad t{:}\,term \to s{:}\,token\;\mathbf{list} \to pars(\{t\} :: s) = [\mathbf{inl}(t)] :: pars(s)\;\mathbf{in}\;pars_res
\end{aligned}$$

$$\begin{aligned}
[2] \quad [\;\; & pars{:}\,token\;\mathbf{list} \to pars_res,\\
& v_0{:}\,t{:}\,term \to s{:}\,token\;\mathbf{list} \to pars(\{t\} :: s) = [\mathbf{inl}(t)] :: pars(s)\;\mathbf{in}\;pars_res\\
]\;\; & \vdash parser : (token\;\mathbf{list} \to (term\backslash\mathbf{unit}))\#\\
& \qquad t{:}\,term \to parser(\{t\}) = \mathbf{inl}(t)\;\mathbf{in}\;(term\backslash\mathbf{unit})
\end{aligned}$$

The second subgoal can be proved automatically, if we decide to consider only the first item parsed:

$$?\!- \quad \mathbf{take}(\lambda(s{:}\,\mathbf{list\_ind}(pars(s), \mathbf{inr}(\mathbf{e}), p{:}\,\tilde{}\,{:}\,\tilde{}\,{:}\,\mathbf{decide}(p, t{:}\,\mathbf{inl}(t), \tilde{}\,{:}\,\mathbf{inr}(\mathbf{e})))))$$

The main idea of the algorithm is, that we look at the first element of the token list. If this token represents a number, then the result is a term consisting of just that number. If the first token represents a left parenthesis then we will look at the result of parsing the rest of the list. In case there are at least four further items parsed, we check whether these represent a term, an operator symbol, another term and a right parenthesis in just this order, whereas the operator and the right parenthesis are coded as error tokens. If this is so, then we will replace these four items in front of the parsing result with a term constructed just out of these items. If the first token is neither a number nor a left parenthesis it is passend as an error token and the analysis is continued with the next token.

To continue we need first some additional operations, which might be inserted in the appropriate include file or ad hoc introduced using $?-\,\mathbf{define}(\cdots)$ commands. In practice the choice depends on how deeply structured our theorem currently is. In most cases it is easier to edit the main theorem and rerun the complete proof so far, which in the current situation takes 1..2 seconds. If we are run into such tricky situations deeply inside another theorem, it is certainly worth considering the introduction of an auxiliary theorem, which reflects the current situation. Then we can play around in this simplified environment, till we have found a complete proof. At the end we still have the choice either of keeping the auxiliary theorems or copying the proofs into the corresponding positions of the main theorem – just like in ordinary programming, where we have to decide whether we want keep the many procedures resulting from structured design, or to join them at least partially by copying the code in line.

The operations we obviously need are a destructor for tokens, which allows a case analysis corresponding to the token given; a destructor for lists, which allows a case analysis, whether the given list has at least four elements and separates this first four elements from the rest; and a destructor for *pars_res* which test, whether the first four elements of the given parse result represents a term, an operator, a term and a right parenthesis, in just that sequence:

$$
\begin{aligned}
&\mathit{switch\_token}(t, n{:}\,a, o{:}\,b, c, d, e) \Longleftrightarrow\\
&\quad \mathbf{decide}(t, \underbrace{n{:}\,a}_{\mathbf{case}\,:\,\mathbf{nat},}, p{:}\,\mathbf{decide}(p, \underbrace{o{:}\,b}_{op,}, q{:}\,\mathbf{decide}(q, \underbrace{\sim{:}\,c}_{lpar,}, r{:}\,\mathbf{decide}(r, \underbrace{\sim{:}\,d}_{rpar,}, \underbrace{\sim{:}\,e}_{error}))))\\
&\mathit{first4}(f, s, a_1{:}\,a_2{:}\,a_3{:}\,a_4{:}\,g{:}\,t) \Longleftrightarrow\\
&\quad \mathbf{list\_ind}(f, s, a_1{:}\,f_1{:}\,\sim{:}\,\mathbf{list\_ind}(f_1, s, a_2{:}\,f_2{:}\,\sim{:}\\
&\qquad \mathbf{list\_ind}(f_2, s, a_3{:}\,f_3{:}\,\sim{:}\,\mathbf{list\_ind}(f_3, s, a_4{:}\,g{:}\,\sim{:}\,t))))\\
&\mathit{rec\_pars}(f, s, l{:}\,o{:}\,r{:}\,g{:}\,t) \Longleftrightarrow\\
&\quad \mathit{first4}(f, s, a{:}\,b{:}\,c{:}\,d{:}\,g{:}\,\mathbf{decide}(a, \underbrace{l{:}}_{\in term}\,\mathbf{decide}(b, \sim{:}\,s, \underbrace{b_1{:}}_{\in \boldsymbol{token}}\\
&\qquad\quad \mathit{switch\_token}(b_1, \sim{:}\,s, \underbrace{o{:}}_{\in \boldsymbol{op}}\,\mathbf{decide}(c, \underbrace{r{:}}_{\in \boldsymbol{term}}\,\mathbf{decide}(d, \sim{:}\,s, \underbrace{d_1{:}}_{\in \boldsymbol{token}}\\
&\qquad\qquad \mathit{switch\_token}(d_1, \sim{:}\,s, \sim{:}\,s, s, t, s)), \sim{:}\,s), s, s, s)), \sim{:}\,s))
\end{aligned}
$$

With these definitions at hand, we can try the main step:

$$?\!-\quad \mathbf{take}(\lambda(\underbrace{s}_{\in token\ \mathbf{list}}{:}\ \mathbf{list\_ind}(s, [], \underbrace{t}_{\in token}{:}\ \tilde{}:\ \underbrace{f}_{\in \boldsymbol{pars\_res}}{:}\ \cdots$$

$$\cdots switch\_token(t, \underbrace{n}_{\in \mathbf{nat}}{:}\ \underbrace{[\mathbf{inl}(\underbrace{\mathbf{inl}(n)}_{\in term})] :: f}_{\in pars\_res}, \underbrace{o}_{\in op}{:}\ \underbrace{\mathbf{inr}(\ \underbrace{op(o)}_{\in token}\ ) :: f}_{\in pars\_res}, \cdots$$

$$\cdots rec\_pars(f, \underbrace{[\mathbf{inr}(\ \underbrace{lpar}_{\in token}\ )] :: f}_{\in pars\_res}, l{:}\, o{:}\, r{:}\, g{:}\, \underbrace{\mathbf{inl}(\underbrace{\mathbf{inr}((l, o, r))}_{\in term}) :: g}_{\in pars\_res}), \cdots$$

$$\cdots \underbrace{\mathbf{inr}(\ \underbrace{rpar}_{\in token}\ ) :: f}_{\in pars\_res}, \underbrace{\mathbf{inr}(\ \underbrace{error}_{\in token}\ ) :: f}_{\in pars\_res}))))$$

Again it turns out to be more suitable to introduce an abbreviating definition and to try the proof step again, After a few seconds the system stops with four subgoals of the following form (one for each operator):

$$?\!-\quad \mathbf{define}(parse(s) \Longleftrightarrow \mathbf{list\_ind}(s, [], t{:}\,\tilde{}{:}\, f{:}\, \boldsymbol{switch\_token}(t, \cdots))).$$

$$?\!-\quad \mathbf{take}(\lambda(s{:}\, parse(s))).$$

$$\begin{array}{ll}
[& parser \Longleftrightarrow \lambda(x{:}\, parse(x)) \\
& s{:}\ token \\
& v0_{:}\ u(1) \\
& \{p : v0_ \rightarrow p \ \mathbf{in}\ term\} \\
& v1{:}\, p{:}\, v0_ \rightarrow parse(\{p\} :: s) = \mathbf{inl}(p).parse(s)\ \mathbf{in}\ pars_res \\
& v0 : v0_ \\
& v4 : v0_ \\
] & \vdash rec_pars(parse(\{v0\} :: \boxed{\text{operator}}.(\{v4\} :: rpar.s)) = \\
& \qquad \mathbf{inl}((v0, \boxed{\text{operator}}, v4)).parse(s)\ \mathbf{in}\ pars_res
\end{array}$$

which encourage us to support the system with a hint for the appropriate substitution step:

$$\begin{array}{ll}
?\!- & \mathbf{subst}(\mathbf{over}(q : rec_pars(q) = \ldots\ \mathbf{in}\ \ldots), \\
& \qquad \ldots = \mathbf{inl}(v0).\mathbf{inr}(\boxed{\text{operator}}).\mathbf{inl}(v4).\mathbf{inr}(rpar).parse(s)\ \mathbf{in}\ pars_res) \\
[] & \vdash parse(\{v0\} :: \boxed{\text{operator}}.(\{v4\} :: rpar.s)) = \\
& \quad \mathbf{inl}(v0).\mathbf{inr}(\boxed{\text{operator}}).\mathbf{inl}(v4).\mathbf{inr}(rpar).parse(s)\ \mathbf{in}\ pars_res
\end{array}$$

But now we are stuck, because the *induction hypothesis* $v1$ is not applicable in this situation: the system had tried to prove

$$[\ t{:}\, term,\ \ s{:}\, token\ \mathbf{list}\ ] \vdash parse(\{t\} :: s) = \mathbf{inl}(t).parse(s)\ \mathbf{in}\ pars\_res$$

by induction over t. Instead it should have tried:

$$[\, t{:}\, term \,] \vdash s{:}\, token\ \mathbf{list} \rightarrow parse(\{t\} :: s) = \mathbf{inl}(t).parse(s)\ \mathbf{in}\ pars\_res$$

This is a typical flaw of the current implementation, although it should be possible to recognize situations which could be solved by generalization. Currently we have to enforce earliest possible induction over t, by

$$?\!-\quad \mathbf{take}(\lambda(s{:}\,parse(s)))\ \mathbf{then}\ \mathbf{rep}(\mathbf{elim}(p)\ \mathbf{orelse}\ \mathbf{intro})$$

Using this command, the systems replies after about 17 seconds with **complete** reporting 5038 inference steps.

Once all the other (sub)theorems have been proved, we are ready to save the extract term of our main theorem in a file **eval.ext**. The extract terms can be directly interpreted by the oyster system, or compiled into programs which read a string from standard input and produce the required string on standard output. And really we get the expected response, if we feed our program with the arithmetic expressions.

4 Conclusion

We have tried to demonstrate that our approach to the implementation of type theory is on the way to the level needed for realistic applications. Nevertheless, there is still a lot to do. The most interesting aspects seems to be related to the full utilisation of higher order logic, not only for specification, but also for meta programming in the sense of semantically driven program generation, and the introduction of AI technology for guiding and supporting more complex proof and synthesis processes.

References

[1] R.S. Boyer and J.S. Moore. *A Computational Logic Handbook.* Academic Press, 1988. Perspectives in Computing, Vol 23.

[2] I.A. Mason. Hoare's logic in the LF. In *ACM Proc. POPL*, San Diego, 1987.

[3] Per Martin-Löf. Constructive mathematics and computer programming. In *6th International Congress for Logic, Methodology and Philosophy of Science*, pages 153–175, Hanover, August 1979. Published by North Holland, Amsterdam. 1982.

[4] C. Paulin-Mohring. Extracting F_ω's programs from proofs in the calculus of constructions. *ACM Proc. POPL*, 1989. Also in [?].

[5] R.L. Constable, S.F. Allen, H.M. Bromley, et al. *Implementing Mathematics with the Nuprl Proof Development System.* Prentice Hall, 1986.

[6] K. Nordström, B. Petersson and J. Smith. *Programming in Martin-Löf Type Theory.* Oxford University Press, 1990.

[7] C. Horn. The Nurprl proof development system. Working paper 214, Dept. of Artificial Intelligence, Edinburgh, 1988. The Edinburgh version of Nurprl has been renamed Oyster.

[8] A. Bundy, F. van Harmelen, J. Hesketh, and A. Smaill. Experiments with proof plans for induction. *Journal of Automated Reasoning*, 7:303–324, 1991. Earlier version available from Edinburgh as DAI Research Paper No 413.

[9] F. van Harmelen. The CLAM proof planner, user manual and programmer manual. Technical Paper TP-4, Dept. of Artificial Intelligence, Edinburgh, 1989.