Formal Reasoning about Communication Systems I

Embedding ML into Type Theory

Christoph Kreitz

Abstract. We present a semantically correct embedding of a subset of the OCAML programming language into the type theory of NUPRL. The subset is that needed to build the ENSEMBLE group communication system. We describe the essential methodologies for representing language constructs by type-theoretical expressions. Tactics representing derived inference rules and a programming logic for these constructs will be discussed as well as algorithms for translating an OCAML-program into NUPRL-objects and vice versa.

The formal representations and the translation algorithms will serve as the foundation for the development of automated reasoning tools for the verification and optimization of a group communication systems.

Contents

1 Introduction . . . 1

2 Preliminaries . . . 4
- 2.1 The NuPRL Proof Development System . . . 4
- 2.2 Relating OCaml and Type Theory . . . 6

3 Embedding OCaml-Constructs into NuPRL . . . 10
- 3.1 Representing Types and Expressions . . . 10
- 3.2 Dealing with Variable-sized Expressions . . . 14
- 3.3 Patterns . . . 16
- 3.4 Local and Global Definitions . . . 19
- 3.5 Imperative Features . . . 20

4 Derived Inference Rules . . . 21
- 4.1 The Type System . . . 21
- 4.2 The Computation System . . . 26
- 4.3 Strategies . . . 28
- 4.4 Implementation Issues . . . 29

5 Translating between OCaml and NuPRL . . . 31
- 5.1 Parsing and Preprocessing OCaml-code . . . 33
- 5.2 Constructing Terms and Library Objects . . . 34
- 5.3 Translating from NuPRL into OCaml source code . . . 39

6 Conclusion . . . 40

A Example proof . . . 43

B The NuPRL-Theory `Ocaml` . . . 47

List of Figures

1 Ensemble Protocol Layers 2
2 Verifying and Reconfiguring Communication Systems in NuPRL 3
3 Structure of the NuPRL System 6
4 Ensemble Code from the Stability Layer 7
5 Initialization procedure of Ensemble's Stability Layer 29
6 Translating between OCaml and NuPRL: General Methodology 32
7 Code for Constructing the Representation of `init` 35

List of Tables

1 Patterns and their Type-Theoretical Representation 18
2 Computation Rules for OCaml-expressions 27

Formal Reasoning about Communication Systems I
Embedding ML into Type Theory

Christoph Kreitz

Department of Computer Science
Cornell-University, Ithaca, NY 14853-7501
kreitz@cs.cornell.edu

Abstract. We present a semantically correct embedding of a subset of the OCAML programming language into the type theory of NUPRL. The subset is that needed to build the ENSEMBLE group communication system. We describe the essential methodologies for representing language constructs by type-theoretical expressions. Tactics representing derived inference rules and a programming logic for these constructs will be discussed as well as algorithms for translating an OCAML-program into NUPRL-objects and vice versa.

The formal representations and the translation algorithms will serve as the foundation for the development of automated reasoning tools for the verification and optimization of a group communication systems.

1 Introduction

Group communication via computer networks has become increasingly important over the past years. Applications can be identified in a wide range of businesses as well as in military intelligence analysis, command and control (see [3] for a general treatment). Because of the variety of applications group communication systems not only have to be efficient and secure but also flexible enough to be adaptable to any specific task.

ENSEMBLE is a group communication toolkit which has been implemented by Mark Hayden [9] in the Objective Caml programming language [15] in order to provide a concise and clear 'reference' implementation of flexible and secure communication systems. It is based on the system HORUS [18] which is a redesign of the widely used ISIS system [4]. ENSEMBLE, like its predecessor HORUS, has a layered architecture. It is broken up into 40 small functions, called *protocol layers,* which are linked by simple input/output channels and can be combined almost arbitrarily (see figure 1). Although layering of protocols typically leads to serious performance problems, the efficiency of layered group communication systems can be improved even beyond the one of monolithic systems by analyzing common sequences of operation and reconfiguring the system accordingly [10].

Hayden chose OCAML as implementation language of ENSEMBLE not only for the purpose of clarity and flexibility but also to enable formal reasoning about the system's properties and symbolic processing of its components. Automated reasoning tools can check the protocols for errors and thus cause the system

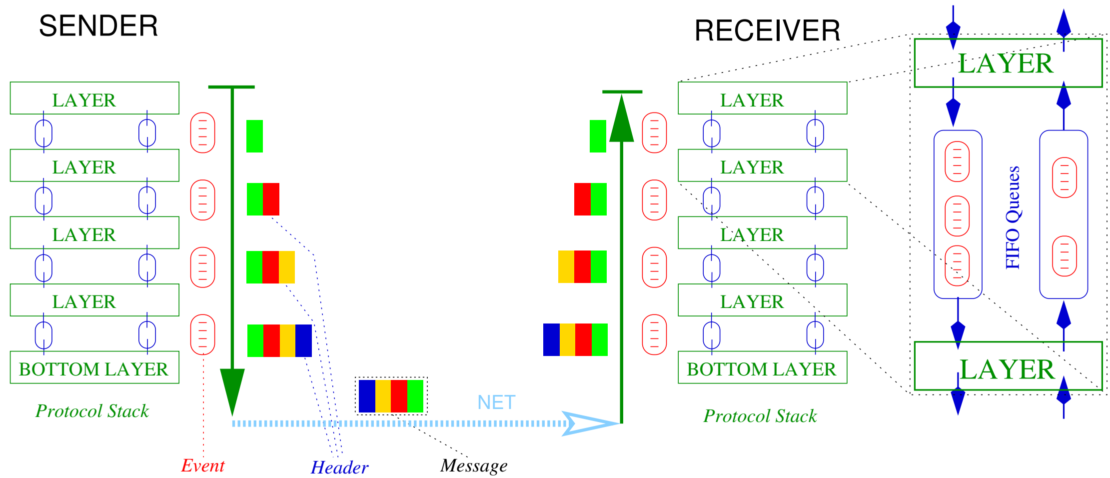

Fig. 1. *Protocol layers ensure or check individual properties of a group communication (such as membership, leader election, stability of messages received, etc.) by adding or analyzing message headers. Individual layers are executed as I/O automata, with pairs of event queues connecting adjacent layers.* (adapted from [9])

to become more reliable. Formal proofs of system properties can be used as documentation thereby making the system's behavior more predictable. Finally, fast-track reconfiguration (i.e. symbolic optimization), which is error-prone if performed by hand, is an area which can be particularly well supported by reasoning techniques in order to be made secure and efficient.

The NuPRL proof development system [6] provides a platform well suited for developing construction methodologies for improving distributed system security. Its underlying formal calculus (Type Theory) already includes formalizations of the fundamental concepts of mathematics, data types, and programming and is similar in most details to the ML programming language: NuPRl's term language is essentially a core ML.. The system itself supports interactive and semi-automatic formal reasoning, evaluation of programs, language extensions by user-defined concepts, and an extendible library of verified knowledge from various domains. It has been used for large verifications of a logic synthesis tool [1] and the SCI cache coherency protocol [12].

In order to tailor NuPRL as a reasoning tool for verifying and reconfiguring communication systems we have to solve several subtasks (see figure 2).

- First we have to embed the programming language OCaml into the type theory of NuPRL and thus provide a formal semantics for OCaml. The embedding has to be targeted towards the intended application and omit features such as a complete treatment of imperative behavior which are not used in Ensemble and cause unnecessary complications in a type-theoretical formalization. By this we avoid having to deal with problems which are mainly of theoretical interest.
- Second we have to provide techniques for reasoning about OCaml-programs within NuPRL. This means implementing *derived inference rules* for each OCaml-construct as programmed applications of type-theoretical inferences. These rules will be the foundation of all other kinds of reasoning.

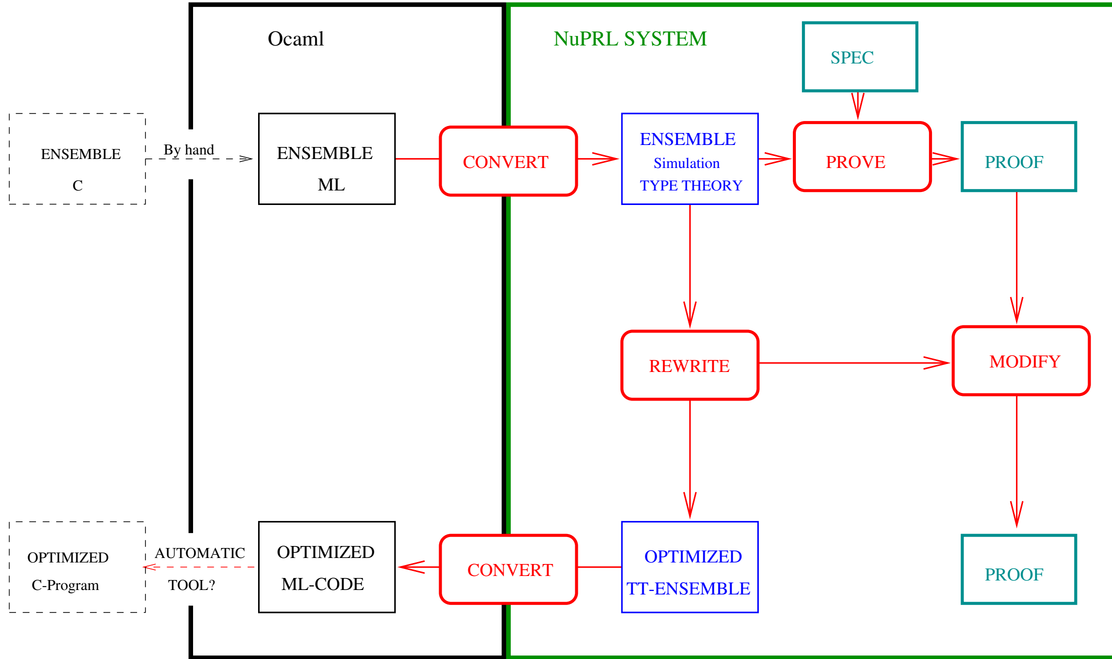

Fig. 2. Verifying and Reconfiguring Communication Systems in NuPRL

- Third we have to provide *tools* for automatically converting OCAML-programs into type-theoretical expressions and vice versa.
- Fourth we have to build automated reasoning tools for program optimization programs such as simplification, conditional rewriting, etc.
- Finally we have to develop methods for formally specifying the properties of ENSEMBLE layers and for automated reasoning about them.

Since the resulting reasoning system shall eventually be handled by system experts who are not necessarily logicians, comprehensibility will be a major design issue. Formal proofs have to be readable if they shall serve as system documentation and serve to support interactively guided verification and reconfiguration. NuPRL expressions simulating OCAML-programs should look like OCAML-code and make the underlying type theory nearly invisible. Applying inference rules should always result in OCAML-expressions and never reveal type-theoretical peculiarities. Therefore general techniques already present in the NuPRL system cannot always be used but have to be replaced by reasoning methods tailored towards the particular application.

In this paper we give a broad overview of our solutions for the first three tasks which are the foundation for all research on automated techniques. In section 2 we shall present a brief summary of NuPRL and discuss differences between OCAML and type theory which have relevance in ENSEMBLE's implementation. Section 3 discusses the embedding of OCAML into the type theory of NuPRL. Section 4 describes the inference system for the embedded language. Section 5 describes the conversion algorithms. We conclude with a few remarks on related work and future research.

2 Preliminaries

2.1 The NuPRL Proof Development System

We shall briefly summarize the essentials of NuPRL's type theory and system features which are important for automating the translation between OCAML and type theory. For a complete account we refer to [6, 13, 14, 5].

NuPRL's type theory is based on Martin-Löf's inductive semantics; it is related to his *intuitionistic type theory* [16, 17] but has been reformulated and extended to support programming concepts as well as proof development on a computer. It provides formalizations of a constructive higher order logic, an elementary programming language, and fundamental data types and has been shown correct with respect to a formalization of Martin-Löf's semantics [2].

The basic objects of reasoning are *types* and *members* of types. Types are built from the atomic types $\mathbb{Z}$, `Atom`, `Void` and *type constructors* such as the function space $S{\rightarrow}T$, a dependent function space $x{:}S{\rightarrow}T[x]$ (the *type* $T[x]$ depends on the input *value* x of a function), (dependent) product types $x{:}S{\times}T[x]$ and $S{\times}T$, disjoint union $S{+}T$, lists T `list`, subtypes $\{x{:}S \mid T[x]\}$, quotient types $\texttt{x,y}{:}S\texttt{//}E[x,y]$, recursive data types $\texttt{rec}(x\,.\,T[x])$, (extensional) equality between members of a type $s{=}t \in T$, and an ordering on integers $s{<}t$. The latter two are propositions which are formally treated as types.[1] A cumulative hierarchy of universes $\mathbb{U}\{i\}$, representing the notion of typehood, enables higher order reasoning in a simple and natural way.

Associated with each type are forms for constructing *canonical members.* A λ-abstraction $\lambda x\,.\,t[x]$ is a canonical member of the function space $S{\rightarrow}T$, provided $t[x]$ is a member of T whenever x is a member of S. A pair $<s,t>$ is a member of $S{\times}T$ whenever s is a member of S and t one of T. Injections $\texttt{inl}(s)$, $\texttt{inr}(t)$ construct the values in $S{+}T$, the empty list `[]` and prepending elements to lists $t\texttt{::}l$ construct members of T `list`, etc. The only canonical member of $s{=}t$ `in` T and $s{<}t$ is the term `Ax`, provided the corresponding propositions are true. Otherwise these types are empty as is the type `Void`.

Non-canonical forms are provided for making use of members. Examples are application $t(s)$ for functions, uniform projection $\texttt{let}\ <x,y>=e\ \texttt{in}\ t[x,y]$ for pairs, an extended conditional $\texttt{case}\ e\ \texttt{of}\ \texttt{inl}(x)\texttt{=>}\,l[x]\ \mid\ \texttt{inr}(y)\texttt{=>}\,r[y]$ for elements of a union type, induction forms for integers and lists, etc.

These forms give rise to a system for *computing* the value of each term as in the λ-calculus. Certain combinations of non-canonical and canonical forms are designated as *reducible* to a *contractum*: $\lambda x\,.\,t[x])(s)$ reduces to $t[s/x]$ (free occurrences of x in $t[x]$ are replaced by s); $\texttt{let}\ <x,y>\ =\ <a,b>\ \texttt{in}\ t[x,y]$ reduces to $t[a,b/x,y]$; $\texttt{case}\ \texttt{inl}(s)\ \texttt{of}\ \texttt{inl}(x)\ \texttt{=>}\ l[x]\ \mid\ \texttt{inr}(y)\ \texttt{=>}\ r[y]$ reduces to $l[s/x]$; induction forms reduce either to the base case or a recursive call; etc.

[1] Viewing propositions as types inhabited by proofs also allows to simulate logical connectives by type constructors. $P \wedge Q$ can be expressed by $P{\times}Q$: a pair of proofs for both P and Q is a proof of $P \wedge Q$. $P \Rightarrow Q$ can be simulated by $P{\rightarrow}Q$, $P \vee Q$ by $P{+}Q$, $\neg P$ by $P{\rightarrow}\texttt{Void}$, $\forall x{:}T.P[x]$ by $x{:}T{\rightarrow}P[x]$, and $\exists x{:}T.P[x]$ by $x{:}T{\times}P[x]$.

Thus besides being a higher order logic type theory is a simple programming language consisting of about 70 predefined fundamental expressions.

Conservative extensions of this language by user-defined concepts are supported by NUPRL's definition mechanism which for presentation purposes has been divided into abstractions and display forms. An *abstraction* has the form

$$operator\text{-}id\{parameters\}(bound\text{-}subterms) \;\equiv\; type\text{-}theoretical\ expression$$

This introduces a new term with name *operator-id* and placeholders for certain parameters and subterms (in which certain variables may be bound) and defines it to be equal to the given type theoretical expression on the right hand side. The abstraction describes the *internal* representation of a newly introduced concept and all formal reasoning will be based on this representation.

Despite the very rigid internal syntax, the appearance of abstract type-theoretical terms on a computer screen can easily be modified via *display forms* into whatever seems appropriate. Since NUPRL uses a structure editor and a hypertext-mechanism for managing terms (instead of a parser) it is even possible to suppress certain information from the display and to handle it as implicit assumptions only (e.g. $i{=}j$ instead of $i{=}j \in \mathbb{Z}$) – as it is often done in mathematical textbooks. An iteration of some term may receive a different appearance (e.g. $\lambda x,y.t[x,y]$ instead of $\lambda x.\lambda y.t[x,y]$), and it is possible to define rules for formatting and parenthesizing. This makes the presentation of formal expressions almost as flexible as the textbook notation.[2]

Formal reasoning in type theory is based on the notion of *sequents*. These are objects of the form $x_1{:}T_1,\ldots,x_n{:}T_n \vdash C$ which should be read as "Under the *hypotheses* that the x_i are variables of type T_i a member of the *conclusion* C can be constructed". In NUPRL a proof for such a sequent is developed in a *top-down* fashion. Proof rules *refine* a goal sequent, obtaining subgoal sequents whose proofs would suffice to validate the original goal. Refinement rules are given by schemata with placeholders for lists of hypotheses and conclusions.

$\Delta \vdash C$ `BY` *rule-name & optional parameters*
  $\Delta_1 \vdash C_1$
  ...
  $\Delta_n \vdash C_n$

expresses the fact that $\Delta \vdash C$ is provable if all the subgoals $\Delta_i \vdash C_i$ are. A set of refinement rules describing how to build members, make use of them, compute them, or deal with well-formedness exists for each type construct.[3]

The development of formal proofs in NUPRL is supported by a highly visual *proof-editor* which operates on proof trees. After entering the initial goal, a

[2] We shall always use the display form instead of the internal representation unless it is necessary to present information which is suppressed from the display. Placeholders in formal expressions will be *emphasized* while `typewriter`-font is used elsewhere.

[3] Note that sequents implicitly describe an algorithm constructing a member C and that the rules are designed to construct this algorithm as well. NUPRL has built-in mechanisms for extracting and evaluating this algorithm.

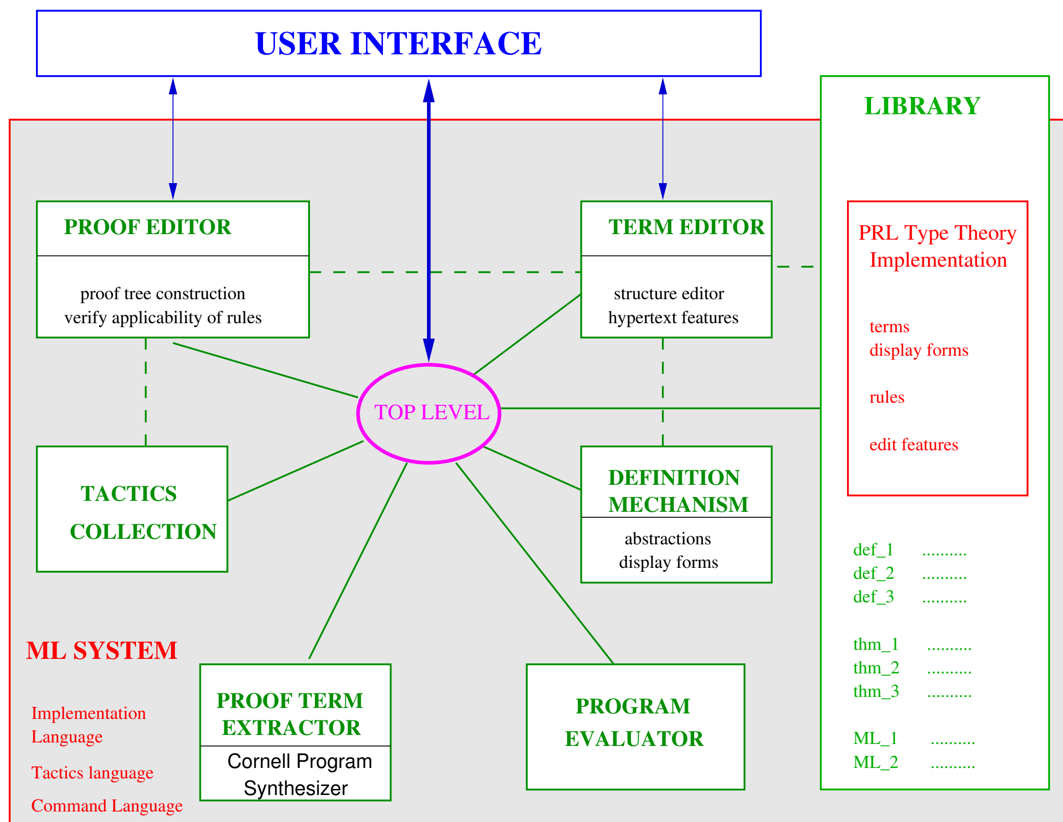

Fig. 3. Structure of the NuPRL System

user subsequently types in rule names (and possibly some arguments) while the system generates the corresponding subgoals. Proofs can also be constructed by *tactics*, i.e. by meta-level programs[4] which guide the application of refinement rules. Tactics are necessary for structuring a formal proof by automatically filling in details and hiding uninteresting proof parts. They can be used as proof search strategies or for simulating the inference rules of user-defined concepts. High-level applications are usually based on tactics instead of primitive rules.

Primitive rules, abstractions, display forms, theorems (including proofs), meta-level programs, and comments are stored as objects of the system's *library* which can be viewed as computerized mathematical textbook. This library already contains many standard extensions of type theory including theories about (natural, rational, and real) numbers, booleans, lists, segments, and functions which are valuable for all further developments.

2.2 Relating OCAML and Type Theory

Objective Caml [15] is a member of the ML language family which descends from the meta-language of the Edinburgh LCF system [8]. It is a strongly typed, (almost) functional language which has recently been extended by a module

[4] The meta-language used for this purpose is the original programming language ML, defined in [8, 13], which is the ancestor of all ML-dialects including OCAML.

system. At its core it is very similar to the language of type theory but has a different syntax and contains many more features which make it a real programming language. It is supported by standard tools like a parser, compiler, debugger etc.

In this section we shall discuss features of OCAML which do not already occur in type theory and investigate to what extent they need to be embedded into NUPRL. This obviously depends on how they are used within the implementation the ENSEMBLE. In order to illustrate the typical differences between OCAML and type theory we use an excerpt from ENSEMBLE's stability protocol layer presented in figure 4.

```
let hdlrs s (ls,vs) {up_out=up; upnm_out=upnm; dn_out=dn;
 .                     dnlm_out=dnlm; dnnm_out=dnnm }        =
 :
 let up_hdlr ev abv hdr =
  match getType ev,hdr with
  | ECast, NoHdr ->
     let origin = (getOrigin ev) in
       array_incr s.ncasts origin ;
       if s.explicitack
        then up (set name ev [Ack (RankSeqno(origin,s.ncasts.(origin)))])
                abv
        else (s.my_row.(origin) <- s.ncasts.(origin) ;
              up ev abv
             )
  | ECast, Unrel -> up ev abv
  | _, NoHdr     -> up ev abv
  | _, _         -> failwith "bad up event"
 :
in {up_in=up_hdlr; uplm_in=uplm_hdlr; upnm_in=upnm_hdlr;
    dn_in=dn_hdlr; dnnm_in=dnnm_hdlr }
```

Fig. 4. ENSEMBLE Code from the Stability Layer

- **Data type constructors** such as arrays, records, variant records, queues, etc. and the corresponding expressions are predefined in OCAML but are not in type theory. `s.ncasts.(origin)`, for instance, selects the element with index `origin` from an array which is the component `ncasts` of the record `s`.

  Most of these data types have several type-theoretical equivalents, and we have to investigate which of them captures their semantics in a natural way and supports a simple representation of the corresponding expressions.
- OCAML has a more **flexible syntax** than type-theory. For instance, the expression `{up_in=up_hdlr; uplm_in=uplm_hdlr; ...}` at the bottom of figure 4 denotes a record where the field `up_in` has the value `up_hdlr`, the component `uplm_in` has the value `uplm_hdlr` etc. A record expression can have an arbitrary number of arguments and implicitly makes a comparison between field names.

In contrast to that, each type-theoretical term must have a fixed number of subterms. Implicit parameters are not allowed. Nevertheless, we can establish an equivalence between variable-sized OCAML-expressions and *iterated applications* of more elementary type-theoretical abstractions and use display forms to suppress information which should be kept implicit.

As a consequence the type-theoretical representation of OCAML is not based on an equivalence between an OCAML-construct and a single abstraction but on a formal equivalence between OCAML-constructs and meta-level methods for applying *several* NUPRL-abstractions whose combination has the same semantics and display as the the original OCAML-construct. We will refer to these meta-level methods as *term-generators*.

– OCAML supports **pattern matching** for assigning values to variables in certain target expressions. In figure 4 this feature is used explicitly in the case expression `match getType ev,hdr with`... and implicitly in the head of the function `hdlrs` where the argument `(ls,vs)` matches against input pairs and `{up_out=up;..;dnnm_out=dnnm}` matches against input records.

Patterns, although similar to conventional expressions, have no meaning by themselves but only provide a method for binding variables in a target expression and for computing values to be assigned to these variables via a structural analysis of an expression against which they will be matched.

In the formal embedding we must therefore distinguish between patterns and expressions and provide a representation of the evaluation mechanism for pattern matching in terms of type-theoretical abstractions. Technically this turns out to be the most demanding aspect of the language embedding. It requires excessive use of display forms for suppressing implicit information (see section 3.3) although semantically the solution can be made very clean.

– OCAML supports **imperative features** such as assigning values to a field of some record as in `s.my_row.(origin) <- s.ncasts.(origin)`. Programs would become extremely inefficient if a copy of a record would have to be produced whenever a field is modified. Consequently compound statements (separated by a semicolon) and loops are also supported.

Type theory does not allow imperative behavior. This is no principal problem since the we only want to *reason* about OCAML-programs. A formal representation based on a copy semantics will be sufficient. Nevertheless, a complete formalization of imperative behavior would require a general model for managing the contents of reference variables.

Fortunately ENSEMBLE-programs do not require such a general model. Only the state of a protocol layer, a record identified by the name `s`, is modified imperatively. There are no other variables to which values will be assigned. This gives room for a much simpler solution (see section 3.5).

– OCAML supports raising and capturing **exceptions**, a concept whose formal representation would be extremely difficult. ENSEMBLE uses these features in a very restricted way. Exceptions are raised only if illegal events occur. In this case the function `failwith` sends a specific failure message to the

outside (i.e. to some outer program not be involved in the reasoning process). Failures caused by matching are usually caught by the final case of the case expression. A failure while matching against the head of a function is not intended and would indicate a serious programming error. Thus it is not necessary to reason about the consequences of failure. Instead both the verification and the fast-track-reconfiguration shall prove that these cases cannot occur.

- Another observation is that ENSEMBLE does not make use of **while-loops** or unrestricted recursion. Although both concepts can easily be represented using the Y-combinator, they usually make formal reasoning about programs rather complicated. For the sake of completeness we have provided representations for these concepts but they will not be used and it will not be necessary to invent automated reasoning strategies for dealing with them.
- The **module system** of OCAML does not introduce a new language construct but a means for structuring the code. This allows, for instance, using the same name for different functions in different modules and supports a clear and uniform presentation of the protocol layers of ENSEMBLE.

  Modules play only an indirect role in the embedding. They have to be mapped onto the flat name space of NUPRL's library and different functions must become different library objects even if their display form (i.e. the name) is the same. Name space management (see section 5.2) is mainly an issue of the translation algorithm and not of the formal representation itself.

All the above considerations have to be taken into account in our formal representation of the subset of OCAML which is relevant for reasoning about the ENSEMBLE system. Technically, establishing the equivalence between OCAML and type theory means extending the library of the NUPRL system by abstractions, display forms, and meta-level programs for term generators such that the NUPRL-expression created by a term generator has the identical semantics and appearance as the OCAML-construct it shall represent. This requires working on both the object-level and the meta-level of NUPRL simultaneously.

- On the object-level we have to provide proper abstractions representing the meaning of (parts of) OCAML-constructs, appropriate display forms, and (proven) theorems describing the type of the newly introduced terms. The latter will be essential when we have to reason about these terms. All these abstractions, display forms, and theorems are collected as a part of the library designated as the theory `Ocaml` (see appendix B).
- On the meta-level we have to provide term-generators which build NUPRL-expressions representing a specific OCAML-construct from the above abstractions as well as operators for analyzing the NUPRL-expressions. These meta-level programs refer to a single abstraction if there is a one-to-one correspondence between an OCAML-construct and an abstraction. Otherwise several abstractions have to be involved to build the NUPRL-expression.

  For technical convenience these meta-level programs are kept in two separate files (`Ocaml-internal.ml, Ocaml.ml`) that will be loaded by NUPRL.

- To prepare the automatic conversion of specific OCAML-code we also have to provide meta-level programs that can generate new abstractions and display forms. These *object-generators* are necessary for building representations of user-defined OCAML-programs automatically, which is the key for reasoning about these programs. This also involves issues like name space management and keeping information about what is known and what should not be known while generating the representation of a module.

  Object-generators are kept in the file `Ocaml-parse.ml`.
- Representing of user-defined OCAML-programs also requires providing verified theorems describing their type which is necessary in future reasoning steps. Thus a system for typing basic and user-defined OCAML-constructs needs to be set up (see section 4.1 and 4.2). This means writing tactics which represent derived inference rules for reasoning about OCAML-expressions.

  Tactics are kept in two files, one for the type system (`Ocaml-rules.ml`) and the other for the computation system (`Ocaml-compute-rules.ml`).

A third level will come in when we will actually translate OCAML-code into NUPRL-objects and vice versa. Since OCAML-programs are just files containing some program text we must first parse this text and then hand the the information gained during the process over to NUPRL. This means combining the OCAML-parser with the meta-level of NUPRL and requires working on the meta-level of OCAML as well.

3 Embedding OCAML-Constructs into NUPRL

In this section we shall discuss the formal representation of the programming language OCAML in type theory. We shall first summarize the type-theoretic equivalences of the most frequently used OCAML-types and expressions[5] , their formal definitions, and the corresponding term-generators. Afterwards we shall describe the key techniques which were necessary to establish this equivalence between OCAML and type theory.

3.1 Representing Types and Expressions

Each formalization of a language construct from OCAML involves providing one or several abstractions, display forms, a term generator, and meta-level programs for analyzing the generated term. Term generators[6] will usually be prefixed with a `mk_` as in `mk_int, mk_number, mk_add`, etc. Functions prefixed with `dst_` usually invert the corresponding term generator and yield subterms and possibly

[5] Unfortunately notions like *type, term, expression*, etc occur both in OCAML and type theory. We will try to minimize confusion by prefixing them from time to time.

[6] It is generally advisable to use term constructors instead of instantiating abstractions within NUPRL's term editor, particularly if there is a nontrivial correspondence between an OCAML-construct and a rather complex type-theoretical expression.

additional information. Functions prefixed with `is_` check whether a given term represents a specific OCAML-construct.

Since OCAML is conceptually close to type theory there are many constructs which can be embedded straightforwardly. In many cases only the syntactical presentation needed to be changed. We shall summarize them by types, beginning with atomic OCAML-types.

- The data type unit contains a single element, denoted by `()`, and is already contained in the standard library of NUPRL. A matcher (see section 3.3) had to provided for the pattern `()` which is used in function declarations and matches arbitrary expressions.
- The data type bool with elements `true` and `false` needed only a few syntactical adjustments. The conditional now reads `if` e `then` a `else` b, conjunction and disjunction are p `&` q and p `or` q. Negation is a function `not: bool -> bool` instead of a boolean operator containing a subterm. Matchers had to be defined for the constant patterns `true` and `false`.
- **Integers** (int) come with a rich variety of predefined expressions. Natural and negative numbers `0, 1, -1, 2, -2, ...` can be converted into number terms using the generator `mk_num`, standard operations are `+, -, *, /, mod` and the tests `=, <, >, <>, <=, >=` which yield *boolean* values as results. Each number also corresponds to a constant pattern.
- **Floating point numbers** (float) are embedded only in a very limited way. They are only necessary for dealing with time. They are represented as pairs consisting of an integer and a list of digits and can be generated from a token like '`-24.55`' using the generator `mk_FloatNum`.
- Characters (char) and **strings** (string) are necessary for composing messages in ENSEMBLE-programs. Characters are represented by token-terms, i.e. members of the predefined type `Atom`. Strings are list of characters. A string term "*string*" can be generated using the function `mk_StringExpr` which is based on several NUPRL-abstractions since the size of strings is not fixed (see section 3.2 for methodology). String concatenation str_1 ^ str_2 and character selection str `.[`i`]` are expressed by predefined list-functions.

  Other functions from the OCAML-module String have been embedded automatically using the translation algorithm described in section 5.

With the exception of product and function types, OCAML's type constructors such as lists, arrays, queues, and user-defined type constructors are functions from (products of) types to types whose application is written in a postfix-notation `(`T_1 `* .. *` T_n`)` *Constr*. We have introduced a separate abstraction for type constructor application in order to distinguish it from the usual function application. We also introduced formal representations of the class of all types (TYPES) and the classes of all n`-ary type constructors`. These two concepts do not belong to the Objective Caml programming language but are essential for reasoning about OCAML types. They can be defined via the universe $\mathbb{U}_1$.

- **Products** T_1 `* .. *` T_n can be variable-sized and contain tuples a_1`, .. ,` a_n as elements. Both can be generated iteratively from NUPRL's product types

and pairs using a display forms without parentheses. For convenience we have added term generators `mk_PROD` and `mk_Tuple` which generate multiple products and tuples in a single step. Tuples also correspond to patterns which are discussed explicitly in section 3.3.

- **Function spaces** $\underline{T_1 \texttt{ -> .. -> } T_n}$ can be variable-sized as well as function application $f\ e_1 \ldots e_n$. Both are simple iterations of predefined NuPRL-terms and can be generated conveniently via `mk_FUN` and `mk_Apply`.

  OCAML has two constructs for *defining* functions, both involving pattern matching. `function` p_1`->`t_1 `|..|` p_n`->`t_n expects a single input expression which is subsequently matched against the p_i. If matching is successful for some p_j then the free variables in t_j will be instantiated accordingly and t_j will be evaluated. To create this term one has to apply `mk_function` to a list of term pairs $\hat{p}_i, t_i$. `mk_function` will automatically convert each $\hat{p}_i$ into the corresponding pattern p_i. This allows generating function definitions without having to understand the peculiarities of pattern matchers. The other construct, `fun` $p_1 \ldots p_n$ `->` t, is a shorthand notation for function definitions with only one case. It accepts an arbitrary number of input values and is created by applying `mk_fundef` to a list of pattern terms $\hat{p}_i$ and a term t.

  Strongly related is the concept of *local definitions*. `let` p `=` e `in` t matches the expression e against the pattern p and binds variables in t accordingly. `let` $f\ p_1 \ldots p_n$ `=` t `in` t' describes a local binding of a function variable. `let rec` $f\ p_1 \ldots p_n$ `=` t `in` t' even allows a recursive binding of f in t. The first two expressions are created using `mk_let`, the last via `mk_letrec`.

  Function definition is also related to the *case expression*, the fundamental matching construct. `match` e `with` p_1`->`t_1 `|..|` p_n`->`t_n matches the expression e against the p_i and instantiates the first term t_j where matching was successful. It is created using `mk_MatchWith` which, in addition to a list of term pairs for the p_i and t_i, expects the term e as input.

  Further details of the the above constructs are discussed in section 3.4.

- The **list** type constructor <u>`list`</u> is a 1-ary type constructor in the above sense which requires a slight modification of NuPRL's list constructor. Constant list expressions `[`e_1`;..`e_n`]` are variable-sized and have to be generated via `mk_ListExpr`. The cons-operation t`::`l and list concatenation l_1 `@` l_2 are already predefined. Functions such as computing the `length` of a list or selecting its `nth` element are already contained in the NuPRL-library and had only to be converted from operations into functions. List expressions and the cons-operation also correspond to patterns.

- The **array** type constructor <u>`array`</u> can be represented in several ways. One option is to use lists which would make the two essential operations, storing values at certain positions (a`.(`i`)` `<-` *expr*) and retrieving values from arrays (a`.(`i`)`) rather unnatural. Representing arrays by finite functions from {`0..lg-1`} into the type T proved to have several disadvantages when reasoning about operations on arrays including an inconsistency with the typings described in [15, section 15.2]. We finally decided to represent them by pairs

of natural numbers (the length) and functions from the space `int` $\rightarrow T$.[7]

Constant array expressions `[|`e_1`;..`e_n`|]` are created via `mk_ArrayExpr` as usual. a`.(`i`)` is represented by function application. a`.(`i`) <-` *expr* is imperative and will only be used in the context of states. Its representation is based on the value substitution a`[`$i \leftarrow$ *expr*`]` (which does not belong to OCAML), the functional counter-piece of an assignment. All other operations from the module Array such as `length`, `create`, `create_matrix`, `to_list`, `of_list`, etc. are represented as functions.

The above type constructors have a comparably simple structure and involve only a fixed number of data types. Records and variant records are much more complex. They combine arbitrarily many types and address them by labels.

– A **record** is a collection of values v_1,..,v_n addressed by field labels l_1,..,l_n. Each value can have its own type and a record type $\{l_1 : T_1 ; \ldots ; l_n : T_n\}$ is the type of all records $\{l_1 = v_1 ; .. ; l_n = v_n\}$ whose values v_i are of type T_i.

  Since records are similar to tuples one might select the product T_1`*..*`T_n as a representation of $\{l_1 : T_1 ; .. ; l_n : T_n\}$ and provide a selector function for each label l_i in order to address a component. But this representation has several drawbacks. When translating OCAML-programs into NUPRL we would have to create a series of selector and assignment functions for each type declaration which would create a significant overhead and be quite unnatural. Furthermore, the nested projections in the selector functions would make formal reasoning about records unnecessary complicated.

  In mathematics, indexed product types (Π-types $\Pi_{x \in I} T_x$) are a natural counter part to records since they select a particular type through an index. Π-types are identical with NUPRL's dependent function types which hence are the best choice for representing records. A record $\{l_1 = v_1 ; .. ; l_n = v_n\}$ can be expressed by a function r on fields which on input l_i yields the value v_i. The type of this function is `l:FIELDS` $\rightarrow$ `T(l)` where `T` is a higher-order function which on input l_i yields the type T_i. We chose `Atom`, the type of token terms, as representation of `FIELDS`.

  To create record expressions `mk_RecordExpr` step-wisely builds a case expression from the function table where the unit value `()` is used as default. The record type $\{l_1 : T_1 ; .. ; l_n : T_n\}$ can be created similarly using `unit` as default type. Addressing a record component r`.`f can be realized by applying the function r to the field f. The imperative assignment r`.`f `<-` *expr* will only be used in the context of states. As in the case of arrays we introduce value substitution r`[`$f \leftarrow$ *expr*`]` for representing it.

  Record expressions are also used as patterns. They match against records which have at least the mentioned fields and match with each sub-pattern.

[7] The concrete representation of a data type and its operations is not really important as long as it can be proven to have certain desired properties. These properties, the interface of the data type, are essential for reasoning purposes while the 'implementation' only guarantees the existence of a faithful representation of the interface.

- **Variant records** are elements from a collection of operators $c_i(e_i)$ called *constructors* which are identified by their name c_i and may have arguments e_i from a type T_i. This collection is described by a variant record type c_1 `of` T_1 `|..|` c_n `of` T_n. We write c_i instead of c_i `of` T_i if there are no arguments. Variant records are often used to mark alternatives and constructors frequently occur as patterns of a case expression.

  Intuitively one might select disjoint unions $T_1|..|T_n$ for a representation but this leads to the same drawbacks as representing record types by products. Instead, indexed union types (Σ-types $\Sigma_{x \in I} T_x$) which are the same as NuPRL's dependent product types are the most reasonable choice. A constructor $c_i(e_i)$ is represented by the pair $<c_i,e_i>$. The variant record type c_1 `of` T_1 `|..|` c_n `of` T_n is constructed similarly to the record type (via `mk_VariantType`) but we use `unit` for constructors without arguments and `Void` as default type for unmentioned labels.

OCaml-**declarations** of user defined types and functions introduce a name for a newly defined object and bind it to a certain expression. In NuPRL this is done by adding the corresponding abstractions and display forms to the library. Since the outer appearance of these formal definitions differs from the style used in OCaml we have introduced a few notations for declarations.

The declarations `type` T `=` e and `type` $(a_1,..,a_n)$ T `=` e are used to define types and type constructors. They are represented by e and $\lambda a_1,..,a_n.e$. The name T has no meaning but will be identical to the name of the NuPRL-object containing the declaration. Similarly `let` x `=` e is represented by e and `let` f $p_1..p_n$ `=` e by `fun` $p_1..p_n$ `->` e, adding x and f as syntactical sugar.

Our embedding currently consists of about 500 library objects for abstractions, display forms, and theorems containing typing information and 1500 lines of meta-level code for programming the term-generators. A detailed description of the library can be found in appendix B.

3.2 Dealing with Variable-sized Expressions

While many OCaml-constructs can be described by a single abstraction and a simple display form which arranges the available information in a convenient form, most variable sized expressions require a combination of several abstractions whose display forms suppress implicit information in order to make a NuPRL expression look like the OCaml-construct it represents. In some cases, variable sized expressions just need a simple iteration of one basic definition. Tuples $a_1,..,a_n$ can be generated by composing pairs a,b without any modifications. List expressions $[e_1;..;e_n]$ can be generated from the cons-operation $t::l$ using the iteration feature in display forms which replaces `::` by `;` and puts square brackets around the whole expression if l is the empty list `[]`.

In many other cases, however, variable-sized expressions require an iteration over several expressions, display forms for iteration, and some additional internal information which is handled only internally. A typical example is the construction of a record $\{l_1=v_1;..;l_n=v_n\}$.

Example 1. As discussed above, a record expression $\{l_1{=}v_1;..;l_n{=}v_n\}$ is represented by a function which takes labels (i.e. token terms) as arguments and returns the value v_i if the input is l_i. The canonical representation of such a function is a type-theoretical expression of the kind

```
λl. if l=l1 then v1 else
      if l=l2 then v2 else
          .
        if l=ln then vn else   ()
```

The unit value () is given as default value since we have to provide some value for labels other than $l_1,..,l_n$. Thus the representation consists of three parts: the λ-abstraction, one call of the conditional for each label, and the default case. Only the labels l_i and the values v_i are to appear when displaying the record expression. This means that the running variable `l` and several other informations must be supressed from the display.

- The default case must not be displayed on the screen at all.
- `if l=`l_i `then` v_i `else ...` must be presented as $l_i{=}v_i$. A semicolon needs to be added if the expression after the `else` is another conditional.
- The outermost λ-abstraction `λl. if ...` must hide the '`λl.`' and display only the remaining term '`{if ...}`' in brackets.

All three forms are special for building records and thus require separate abstractions and display forms. It should be noted that the variable `l` which is bound by the λ-abstraction has to occur as subterm in *all* conditionals but this information has to be kept implicit. One might select a fixed variable name in the abstraction-objects of the conditional and the outermost λ-abstraction in order to achieve that. However, it seems to be better to have the term generator select an appropriate variable name and pass it on to the λ-abstraction and all conditionals which are necessary for building the record. These considerations have led to the following abstractions (`*A`) and display forms (`*D`).

```
*A RecordExpr_Null:  RecordExpr_Null{}()    ≡ ()
*D RecordExpr_Null_df:                      ≡ RecordExpr_Null{}()
*A RecordExpr_Step:  RecordExpr_Step{}(l; lab; v; next)
                                            ≡ if l=lab then v else next
*D RecordExpr_Step_df:
                     lab = v next  ≡ RecordExpr_Step{}(l; lab; v; next)
           #Hd a :: lab = v;#      ≡ RecordExpr_Step{}(l; lab; v; #)
           #Tl a :: lab = v next   ≡ RecordExpr_Step{}(l; lab; v; next)
           #Tl a :: lab = v;#      ≡ RecordExpr_Step{}(l; lab; v; #)
*A RecordExpr:       RecordExpr{}(l.next[l])  ≡ λl.next[l]
*D RecordExpr_df:    {next}                   ≡ RecordExpr{}(l.next[l])
```

In the display form for `RecordExpr_Step` we make use of iteration. The first form shows how a single occurrence of the conditional is to be presented. The second is the head of an iteration where # is a placeholder for another abstraction of the same kind. The third is a terminating tail form and the last is a tail form in which the iteration will be continued. Note that the placeholder l occurs only on the

right hand side but is suppressed from the display. In the abstraction `RecordExpr` the subterm l.*next*[l] on the left hand side means that free occurrences of l in *next* will be bound in this abstraction.

The algorithm for the term generator is now easy to define. Given a list of fields (tokens) and values (terms) we iteratively instantiate `RecordExpr_Step` using the same field variable (as a term) again and again and finally apply `RecordExpr_Null`. The result will be taken as subterm for `RecordExpr` which again uses the same variable. Formally this algorithm is described as follows

```
let mk_RecordExpr  field_expr_list  =
   let mk_RecordExpr_Null =    mk_simple_term 'RecordExpr_Null' []
   and mk_RecordExpr_Step lab v next =
         mk_simple_term 'RecordExpr_Step'
                             [ FieldVarTerm; mk_field lab; v; next]
in
   mk_std_term 'RecordExpr' [ [FieldVar], reduce2  mk_RecordExpr_Step
                                                    mk_RecordExpr_Null
                                                    field_expr_list
                             ]
```

Many variable-sized expressions such as record types, variant records, array expressions, case expressions, function definitions etc. can be generated using the same methodology as described in this example. Only the construction of patterns for local and global definitions requires a much more complex technique a technique which heavily relies on supressing implicit data.

3.3 Patterns

Pattern matching is a very convenient way of assigning values to certain free variables in some target expression. The language OCAML provides several constructs which perform pattern matching among which the *case expression* is the most fundamental one. Evaluating

$$\texttt{match}\quad expr\ \texttt{with}\quad p_1\ \texttt{->}\ t_1 \\ \quad | \quad \vdots \\ \quad | \quad p_n\ \texttt{->}\ t_n$$

subsequently tries to match the expression *expr* against the patterns $p_1 \ldots p_n$. Matching succeeds if the free variables of p_i can be instantiated by values such that the instantiated pattern is identical to *expr*. In this case the (target) expression t_i will be evaluated. The evaluation of t_i takes place in an environment that binds the free variables of p_i according to the result of the matching.

The outer appearance of a pattern is identical to that of a conventional expression except for the fact that only variables, constants, Or-patterns, and certain canonical expressions such as tuples, list and record expressions, etc. are allowed. Semantically, a pattern serves an entirely different purpose: it has to bind free variables in a target expression and to provide structural information which allows decomposing *expr* in order to determine values for the free variables.

While running OCAML the distinction between expressions and patterns can be captured by looking at the language constructs from the outside. This will eventually lead to different evaluation mechanisms. In a type-theoretical simulation of OCAML, however, we have to represent the semantics of patterns explicitly on the *object level*, i.e. by NUPRL-terms which semantically are different from the corresponding expressions but have the same outer appearance.

Computationally, a pattern can be viewed as a matching function (briefly *matcher*) which takes an expression *expr* and a target t, performs a structural analysis of *expr*, and returns an instance of t where the free variables of the pattern are instantiated according to the result of matching. A pattern x, for instance, contains the free variable x and matches agains any expression. Applying the matcher to *expr* and t thus results in $t[expr/x]$. This idea will be fundamental for our treatment of matching. Matchers will be terms which have an input expression and a target expression as subterms. However, we need to address a few problems which cause the technicalities to become somewhat complex.

To make a NUPRL-representation look like the original OCAML program both the input and the target expression must not be displayed. This can be achieved by appropriate display forms but one should keep in mind that the display on the screen differs greatly from the data which are handled internally.

Most patterns are not atomic like variable or constant patterns but are constructed from other patterns. Thus a matcher can have several other matchers as (second-order!) subterms and its meaning must be defined with respect to them. A paired pattern (p_1, p_2), for instance, consists of two sub-matchers and expects the input expression *expr* to be a pair (e_1, e_2). The expression e_1 together with the target expression t will be handed down to the matcher p_1 and e_2 together with the result of matching[8] to p_2.

The number of free variables to be bound by a pattern can vary greatly and does not depend only on the outer structure of a pattern. Therefore the free variables must be bound already in the target expression by some λ-abstraction which also has to be hidden from the display. The order of hidden variable bindings must fit their occurrence in the pattern. Variable instantiation can then be represented by function application.

Since all type-theoretical expressions must have a meaning, a failing match cannot be expressed directly. Instead we must express success and failure by a boolean value which will be passed to the outside and can then be analyzed by language constructs that are based on matching.

Taking all these considerations into account a pattern can formally be defined via a collection of NUPRL-abstractions and display forms. The abstraction will define the internal structure and the semantics of a matcher while the display form describes the pattern it represents.

Example 2. The abstraction for paired patterns introduces a term with the operator identifier `Product__Match_Pair` and four formal subterms, separated by

[8] Since the environment which binds variables of t cannot be handled separately, t has to be threaded through these two applications.

```
x                            ≡ (true, t expr)
_                            ≡ (true, t)
p1[e1;t1] | p2[e2;t2]        ≡ let b',t' = p1[expr;t] in
                                  if b' then (true, t') else p2[expr;t]
()                           ≡ (true, t)
true                         ≡ (expr, t)
false                        ≡ (not expr, t)
n                            ≡ (expr = n, t)
[]                           ≡ (null(expr), t)
p1[e1;t1]::p2[e2;t2]         ≡ rec-case(expr) of
                                  [] → (false, t)
                                  | e1::e2 → R.let b1,t1 = p1[e1;t] in
                                     if b1 then p2[e2;t1] else (false, t1)
p1[e1;t1], p2[e2;t2]         ≡ let e1,e2 = expr in
                                  let b1,t1 = p1[e1;t] in
                                     if b1 then p2[e2;t1] else (false, t1)
                             ≡ (true, t)
{l=p1[e1;t1] p2[e2;t2]}      ≡ let b1,t1 = p1[expr.l; t] in
                                  if b1 then p2[expr;t1] else (false, t1)
constr(p1[e1;t1])            ≡ let nc(arg) = expr in
                                  if nc = constr then p1[arg;t] else (false, t)
constr                       ≡ let nc(nullvar) = expr in (nc = constr, t)
```

Table 1. Patterns and their Type-Theoretical Representation

a semicolon: the input expression *expr*, the target expression t and two sub-matchers p_1 and p_2. The sub-matchers must contain two free variables — for the input and target expressions which will be passed down — and hence are second-order variables. The right hand side of the abstraction formalizes the above intuition of paired matchers while the display form suppresses *expr*, t, and the variable bindings and causes the matcher to appear as simple pair.

```
Product__Match_Pair{}(.expr; e1,t1.p1[e1;t1], e2,t2.p2[e2;t2]; .t)
       ≡ let e1,e2 = expr in
           let b1,t1 = p1[e1;t] in
             if b1 then p2[e2;t1] else (false, t1)

p1, p2  ≡  Product__Match_Pair{}(.expr; e1,t1.p1[e1;t1], e2,t2.p2[e2;t2]; .t)
```

Table 1 shows the formal definitions of all the pattern constructs[9] used in the embedding. The display forms are given on the left hand side of the equivalence symbol (*expr* and t are always suppressed) and the formal semantics of each expression is given on the right. Record patterns $\{l_1{=}p_1;..;l_n{=}p_n\}$, like record expressions, are variable-sized and thus require two definitions (c.f. example 1).

[9] These definitions are not intended for interactive use but for the automatic translation of OCAML-code into NUPRL. The second-order variables for sub-matchers and the requirement to bind the target expression with hidden-lambda bindings make it rather difficult to enter correct simulations of OCAML-pattern. Should there be a need to type in a pattern during a derivation one should create it from the corresponding expression using the term-generator `build_Matcher_term`.

3.4 Local and Global Definitions

Based on the embedding of pattern matching we can now explain the meaning and representation of the language constructs that involve matching. These are the case expression, function definitions, local definition of constants and (recursive) functions, and global declarations of user-defined constructs.

- A case expression with a single case `match` e `with` p `->` t is the easiest version of matching. The pattern p is to be matched against e and the result will be the corresponding instance of t. Since this is exactly the effect of applying the matcher which represents the pattern p to e and t we only have to remove the boolean value which expresses success or failure of the matching and return the computed term.

  ```
  match e with p -> t   ≡ let b,t' = p[e;t] in t'
  ```

  This definition is based on the convention that ENSEMBLE-programs are not allowed to fail (which can be checked while *reasoning*). Matching must be used in a way that it succeeds eventually. This can be ensured by adding further cases, the last one involving a universal match.
- The full **case expression** `match` e `with` p_1 `->` t_1 `| .. |` p_n `->` t_n can be represented as iterated conditional which matches e against a pattern and hands it down to the remaining cases if matching has failed. The final case *must* succeed. The internal representation of the case expression is thus

  ```
  let b1,t1' = p1[e;t1] in if b1 then t1' else
    let b2,t2' = p2[e;t2] in if b2 then t2' else
              ⋮
        let bn,tn' = pn[e;tn] in tn'
  ```

  and can be created in the same way as record expressions (see example 1).
- The **function definition** `function` p_1 `->` t_1 `| .. |` p_n `->` t_n is similar to the case expression. λ`e.match e with` p_1 `->` t_1 `|..|` p_n `->` t_n has the same meaning and is used for representing it.
- `fun` p_1 `..` p_n `->` t can be seen as a shorthand notation for

  ```
  function p1 -> function p2 -> ... function pn -> t
  ```

  Its representation uses an iteration of λ-abstractions combined with the base case of the case expression. Two additional definitions are necessary for creating the `fun` at the beginning and the arrow between the last pattern and the target expression t. A more sophisticated display mechanism would make this superfluous.
- A **local definition** `let` p `=` e `in` t binds the variables of p locally in t. Except for syntactical differences this is the same as `match` e `with` p `->` t.

  `let` f p_1 `..` p_n `=` t `in` t' binds the variable f locally in t to a function which matches its inputs against the patterns p_i and returns an instance of t. This is the same as `let` f `= fun` p_1 `..` p_n `->` t `in` t' and will be created in the same way as the function definition. Only the syntactical appearance has to be be different.

- A local recursive definition, `let rec` f $p_1 \,..\, p_n$ `=` t `in` t', does not only bind the occurences of f in t' but also recursively in t. Semantically, f is the fixpoint of $\lambda f.$ `fun` $p_1 \,..\, p_n$ `->` t. We have to make use of the Y-combinator to express it. Its type-theoretical representation is

  `let` f `=` Y ($\lambda f.$ `fun` $p_1 \,..\, p_n$ `->` t) `in` t'

  Except for a definition involving the Y-combinator we can proceed as before.
- A **global declaration** of an individual constant `let` x `=` e has to be represented as a meta-level construct using *object generators* instead of term generators. The intended meaning of this construct is to declare a constant which can be addressed by the name x and whose value is e. In NuPRL this can only be done by creating an abstraction object with name x which is defined to be equivalent to e. The '`let` x `=`' in the definition is just syntactical sugar which does not have a meaning in the object level of type theory.

  A more complex version `let` p `=` e would bind several constants at once through pattern matching thus generating several abstractions. This feature is not used within ENSEMBLE and we have not implemented it yet.

  A global function declaration `let` f $p_1 \,..\, p_n$ `=` t binds the name f to the function `fun` $p_1 \,..\, p_n$ `->` t while `let rec` f $p_1 \,..\, p_n$ `=` t binds it to Y ($\lambda f.$ `fun` $p_1 \,..\, p_n$ `->` t). Except for syntactical differences this can be handled as above.

3.5 Imperative Features

Due to the limited use of imperative features in ENSEMBLE, their formal representation in type theory can be kept comparably simple. Statements may modify only the state s of a protocol layer which is a record containing various informations about incoming, outgoing, and not yet processed events. Thus statements can be expressed as functions modifying the state, i.e. as elements of the type `STATES -> STATES`. As usual, statements are built from assignments and various programming structures

- The basic assignment consists of a modification of the components of the state s. $s.f$ `<-` *expr* is represented as $\lambda s.s[f \leftarrow expr]$, i.e. as function returning a state whose component f has been modified (which is easy to represent because states are dependent functions).
- Sequencing statements $stmt_1;stmt_2$ means composing the corresponding functions, i.e. λ`s.` $stmt_2$ ($stmt_1$ `s`). The conditional `if` e `then` a `else` b remains unchanged. Loops `for` i=e_1 [down]`to` e_2 `do` *stmt* `done` can be expressed by induction on integers. The while loop `while` c `do` *stmt* `done` is represented by `Y` (λ`while.if` c `then` *stmt*`;` `while else id`), where `id` is the identity function.

Exceptions are an imperative feature which would be rather difficult to represent in the purely functional setting of type theory. ENSEMBLE raises exceptions only through the function `failwith` which sends a failure message to the outside. Since these messages will not be processed at all, it is sufficient to represent `failwith` as a statement with no specific effect.

4 Derived Inference Rules

Tools for automated reasoning about OCAML-programs will essentially rely on three basic mechanisms: *typechecking*, *computation*, and *logical reasoning*. While all three mechanisms are in principle already supported by the NUPRL system the difficulty lies in making the underlying type theory invisible such that a user of the reasoning tools will not be burdened by unfamiliar notions. Therefore we have designed a system of inference rules for typechecking and computating OCAML-expressions and implemented them as tactics which are based on the type-theoretical representations given in the previous section. We have also developed a small set of simple strategies for automatic typechecking and evaluation of OCAML-programs. These tools will be the foundation for implementing more sophisticated techniques for semi-automatic verification and optimization of OCAML-programs within NUPRL.

In the following we shall first summarize the type system and the computation system for the embedded subset of OCAML. We shall then present the techniques which were necessary to realize the inference rules as tactics of the NUPRL proof development system and discuss efficiency problems and solutions.

4.1 The Type System

The type of an expression is one of its fundamental properties. In a suffienctly expressive type system such as type theory every property of a program can be expressed within its type. A type system describes the inference rules which are used for proving that an expression belongs to some given type or that a complex type expression does in fact denote a type or a type constructor. It provides the basis for reasoning about the properties of OCAML-expressions.

For all OCAML-operators which do not require subterms, i.e. functions and fixed constants, the type is explicitly given in a *wellformedness* theorem that has been proven while creating the formal definition of the operator. These operators can be handled by the rules `PreDefinedObjectMem` and `PreDefinedTypeMem` which look up the type in the theorem. No subgoals need to be generated. These rules also provide a convenient way of automatically extending the type system to arbitrary user-defined types and functions since the translation algorithm will automatically generate *wellformedness* theorems for them.

Most other fixed-size operators have explicit wellformedness theorems which involve universally quantified variables. The corresponding inference rules will generate subgoals which are left to be proven and should be presented explicitly. For variable-sized and other complex expressions there is no explicit wellformedness theorem because they are composed out of several abstractions in a way which cannot be described by a fixed expression within a theorem.

All rulenames consist of a name for the operator or type and a postfix `Mem`, expressing that they prove the expression to be a member of some type. To simplify interaction with the system, all rules about type-expressions are collected under the name `CamlTypehood` and all other rules under the name `CamlMembership`.

Types and Type Constructors The following rules prove typehood without creating subgoals. A variable is a type if it is declared to be one; atomic OCAML-types are in fact types; `list` and `array` are type constructors, i.e. functions which map types into types. Other predefined types or type constructors can be validated if there is a typehood theorem which is identical to the conclusion.

```
Δ, T:TYPES ⊢ T ∈ TYPES
  by VarMem

Δ ⊢ unit   ∈ TYPES        Δ ⊢ bool ∈ TYPES        Δ ⊢ int    ∈ TYPES
  by UnitMem                by BoolMem              by IntMem

Δ ⊢ float  ∈ TYPES        Δ ⊢ char ∈ TYPES        Δ ⊢ string ∈ TYPES
  by FloatMem               by CharMem              by StringMem

Δ ⊢ list  ∈ 1-ary TYPE-constructors
  by ListMem

Δ ⊢ array ∈ 1-ary TYPE-constructors
  by ArrayMem

Δ ⊢ T ∈ TYPES                       Δ ⊢ T ∈ n-ary TYPE-constructors
  by PreDefinedTypeMem*               by PreDefinedTypeMem*
```

Rules for complex type constructs decompose the typehood proof into typehood problems for their subterms. The rules decompose variable sized constructs, i.e. multiple products and function spaces, record types, variant record types, and the application of an n-ary type constructor in a single step. The rule for module calls simply reduces the long name of a type to the local name. Type declarations will be decomposed into the defining type expression.

```
Δ ⊢ T1 *..* Tn ∈ TYPES                Δ ⊢ T1 ->..-> Tn ∈ TYPES
  by ProductMem                         by FunctionMem
  Δ  ⊢ T1 ∈ TYPES                       Δ  ⊢ T1 ∈ TYPES
       .                                     .
  Δ  ⊢ Tn ∈ TYPES                       Δ  ⊢ Tn ∈ TYPES

Δ ⊢ {l1:T1;..;ln:Tn} ∈ TYPES          Δ ⊢ l1 of T1 |..| ln of Tn ∈ TYPES
  by RecordMem                          by VariantMem
  Δ  ⊢ T1 ∈ TYPES                       Δ  ⊢ T1 ∈ TYPES
       .                                     .
  Δ  ⊢ Tn ∈ TYPES                       Δ  ⊢ Tn ∈ TYPES

Δ ⊢  Module.T ∈ TYPES                 Δ ⊢ (T1 *..* Tn) Constr ∈ TYPES
  by LongIdMem                          by TypeApplyMem
  Δ  ⊢ T ∈ TYPES                        Δ  ⊢ Constr ∈ n-ary TYPE-constructors
                                        Δ  ⊢ T1 ∈ TYPES
                                             .
                                        Δ  ⊢ Tn ∈ TYPES

Δ ⊢ type T = expr ∈ TYPES             Δ ⊢ type (a1,..,an) T = expr
  by TypeDeclMem                                ∈ n-ary type constructors
  Δ ⊢  expr ∈ TYPES                     by TypeDeclMem
                                        Δ,a1:TYPES,..,an:TYPES ⊢ expr ∈ TYPES
```

Note that the rule `VariantMem` generates typehood subgoals for the argument types of a constructor (label) l_i. If the constructor does not have arguments then no subgoal will be generated for it.

Basic expressions. Most of the rules for Unit expressions, boleans, integers, floats, characters, and strings are obvious. They simply decompose an expression or check whether it is actually a natural or floating point number, a character, or a string. Variables are accepted as of type T if they have been declared so and Predefined objects can be validated if there is a typing theorem which is identical to the conclusion.

$\Delta,\ t{:}T \ \vdash t \in T$
by `VarMem`

$\Delta \vdash t \in T$
by `PreDefinedObjectMem`*

$\Delta \vdash \texttt{()} \in \texttt{unit}$
by `AxMem`

$\Delta \vdash \texttt{true} \in \texttt{bool}$
by `TrueMem`

$\Delta \vdash \texttt{false} \in \texttt{bool}$
by `FalseMem`

$\Delta \vdash \texttt{if}\ e\ \texttt{then}\ a\ \texttt{else}\ b \in C$
by `CondMem`
$\Delta \vdash e \in \texttt{bool}$
$\Delta,\ e\texttt{=true} \ \vdash a \in C$
$\Delta,\ e\texttt{=false} \vdash b \in C$

$\Delta \vdash p\ \texttt{\&}\ q \in \texttt{bool}$
by `AndMem`
$\Delta \vdash p \in \texttt{bool}$
$\Delta \vdash q \in \texttt{bool}$

$\Delta \vdash p\ \texttt{or}\ q \in \texttt{bool}$
by `OrMem`
$\Delta \vdash p \in \texttt{bool}$
$\Delta \vdash q \in \texttt{bool}$

$\Delta \vdash n \in \texttt{int}$
by `NatnumMem`

$\Delta \vdash \texttt{-}i \in \texttt{int}$
by `MinusMem`
$\Delta \vdash i \in \texttt{int}$

$\Delta \vdash i\ op\ j \in \texttt{int}$
by *opname*`Mem`
$\Delta \vdash i \in \texttt{int}$
$\Delta \vdash j \in \texttt{int}$

where op $\in$ {`+`, `-`, `*`, `/`, `mod`, `=`, `<`, `>`, `<>`, `<=`, `>=`}
where opname $\in$ {`Add`, `Sub`, `Mul`, `Div`, `Mod`, `Eq`, `Lt`, `Gt`, `Neq`, `Le`, `Ge`}

$\Delta \vdash x\texttt{.}d \in \texttt{float}$
by `FloatnumMem`

$\Delta \vdash \texttt{'}a\texttt{'} \in \texttt{char}$
by `CharConstMem`

$\Delta \vdash \texttt{"}string\texttt{"} \in \texttt{string}$
by `StrConstMem`

$\Delta \vdash str_1 \hat{}\, str_2 \in \texttt{string}$
by `StrCatMem`
$\Delta \vdash \texttt{str}_1 \in \texttt{string}$
$\Delta \vdash \texttt{str}_2 \in \texttt{string}$

The rule `PreDefinedObjectMem` can also be used for dealing with many standard OCAML functions whose abstract representation in NUPRL does not have subterms. For all these functions there are typing theorems in the library.

Lists and Arrays. Most operations on lists and arrays are functions and are handled by `PreDefinedObjectMem`. Only a few expressions need explicit rules.

$\Delta \vdash \texttt{[}e_1\texttt{;..;}e_n\texttt{]} \in T\ \texttt{list}$
by `ListExprMem`
$\Delta \vdash e_1 \in T$
.
$\Delta \vdash e_n \in T$

$\Delta \vdash a\texttt{::}l \in T\ \texttt{list}$
by `ListConsMem`
$\Delta \vdash a \in T$
$\Delta \vdash l \in T\ \texttt{list}$

$\Delta \vdash l_1\ \texttt{@}\ l_2 \in T\ \texttt{list}$
by `ListConcatMem`
$\Delta \vdash l_1 \in T\ \texttt{list}$
$\Delta \vdash l_2 \in T\ \texttt{list}$

$\Delta \vdash \texttt{[|}e_1\texttt{;..;}e_n\texttt{|]} \in T\ \texttt{array}$
by `ArrayExprMem`
$\Delta \vdash e_1 \in T$
.
$\Delta \vdash e_n \in T$

$\Delta \vdash a\texttt{.(}i\texttt{)} \in T$
by `ArrayGetMem`
$\Delta \vdash a \in T\ \texttt{array}$
$\Delta \vdash i \in \texttt{int}$

Note that `ArrayGetMem` does not check array bounds. This is consistent with the typing of `Array.get` in the OCAML-system. We have decided to deal with array bounds only during a verification process. The rule `ArraySubstMem` below does not immediately characterize an OCAML-construct but will be helpful when reasoning about assignments.

$\Delta \vdash a[i \leftarrow \mathit{expr}] \in T$ `array`
  by `ArraySubstMem`
  $\Delta \vdash a \in T$ `array`
  $\Delta \vdash i \in$ `int`
  $\Delta \vdash \mathit{expr} \in T$

Products, Functions, Records, and Modules. Some rules about functions and records require that the type of the function or record is provided as an additional argument. This is somewhat inconvenient for building reasoning tools. We have provided a second version of these rules which tries to find the required type automatically. In `RecordGetMem` this involves proving exactly what would usually be left as subgoal. Therefore the rule has no subgoals anymore. Since many record types are introduced via type declarations we also allow a record rules to reduce a type to a record type if necessary.

$\Delta \vdash$ `Module.t` $\in T$
  by `LongIdMem`
  $\Delta \vdash t \in T$

$\Delta \vdash a_1,..,a_n \in T_1 * .. * T_n$
  by `TupleMem`
  $\Delta \vdash a_1 \in T_1$
  $\vdots$
  $\Delta \vdash a_n \in T_n$

$\Delta \vdash f\ e \in T$
  by `ApplyM` S
  $\Delta \vdash f \in S$ `->` T
  $\Delta \vdash e \in S$

$\Delta \vdash f\ e_1\ \ldots\ e_n \in T$
  by `ApplyMem`
  $\Delta \vdash f \in S_1$ `->` .. S_n `->` T
  $\Delta \vdash e_1 \in S_1$
  $\vdots$
  $\Delta \vdash e_n \in S_n$

$\Delta \vdash \{l_1 = v_1; ..; l_n = v_n\} \in T$ *where T reduces to* $\{l_{i_1}:T_{i_1};\ ..;\ l_{i_n}:T_{i_n}\}$
  by `RecordExprMem`
  $\Delta \vdash l_1 \in T_1$
  $\vdots$
  $\Delta \vdash l_n \in T_n$

$\Delta \vdash r.f \in T_f$
  by `RecordGetM` $\{l_1:T_1; ..; f:T_f; ..; l_n:T_n\}$
  $\Delta \vdash$ r $\in \{l_1:T_1; ..; f:T_f; ..; l_n:T_n\}$

$\Delta \vdash r.f \in T_f$
  by `RecordGetMem`

$\Delta \vdash r[f \leftarrow \mathit{expr}] \in T$ *where T reduces to* $\{l_1:T_1; ..; f:T_f; ..; l_n:T_n\}$
  by `RecordSubstMem`
  $\Delta \vdash r \in T$
  $\Delta \vdash \mathit{expr} \in T_f$

$\Delta \vdash c(\mathit{arg}) \in T$ *where T reduces to* c_1 `of` T_1 | .. | c `of` T_c | .. | c_n `of` T_n
  by `VariantConstrMem`
  $\Delta \vdash \mathit{arg} \in T_c$

$\Delta \vdash c \in T$ *where T reduces to* c_1 `of` T_1 | .. | c | .. | c_n `of` T_n
  by `VariantConstrMem`

Expressions involving Patterns. For typing purposes patterns can be viewed as a complex version of λ-abstraction. Patterns, however, may bind several variables at once and it requires greater effort to bind these variables correctly. In order to do so, the type of the pattern as a whole must be known. We can then decompose this type in order to determine the typing of the individual variables.

By $\ulcorner p{:}S \urcorner$ we denote a yet unknown typing of the variables of the pattern p whose combination will be of type S. This is a meta-notation which in a concrete rule application has to be replaced by the result of evaluating it (see below). Some of the rules require that the type S is provided as additional argument. Again we provide a second version in which this type will be found automatically.

$\Delta \vdash$ `match` e `with`
$\quad p_1$`->`t_1 `|..|` p_n`->`$t_n \ \in\ T$
`by MatchWithM` S
$\Delta,\ \ulcorner p_1{:}S\urcorner \vdash t_1 \ \in\ T$
.
$\Delta,\ \ulcorner p_n{:}S\urcorner \vdash t_n \ \in\ T$
$\Delta \vdash e \ \in\ S$

$\Delta \vdash$ `match` e `with`
$\quad p_1$`->`t_1 `|..|` p_n`->`$t_n \ \in\ T$
`by MatchWithMem`
$\Delta,\ \ulcorner p_1{:}S\urcorner \vdash t_1 \ \in\ T$
.
$\Delta,\ \ulcorner p_n{:}S\urcorner \vdash t_n \ \in\ T$

$\Delta \vdash$ `function` p_1`->`t_1 `|..|` p_n`->`$t_n \ \in\ S$ `->` T
`by FunctionPatMem`
$\Delta,\ \ulcorner p_1{:}S\urcorner \vdash t_1 \ \in\ T$
.
$\Delta,\ \ulcorner p_n{:}S\urcorner \vdash t_n \ \in\ T$

$\Delta \vdash$ `fun` $p_1 \,..\, p_n$ `->` $e \ \in\ S_1$ `->` $..S_n$ `->` T
`by FunPatternMem`
$\Delta,\ \ulcorner p_1{:}S_1\urcorner, .., \ulcorner p_n{:}S_n\urcorner \vdash e \ \in\ T$

$\Delta \vdash$ `let` p `=` e `in` $t \ \in\ T$
`by LetPatternM` S
$\Delta, \ulcorner p{:}S\urcorner \vdash t \ \in\ T$
$\Delta \vdash e \ \in\ S$

$\Delta \vdash$ `let` p_1`=`e_1 `and` $..p_n$`=`e_n `in` $t \ \in\ T$
`by LetPatternMem`
$\Delta, \ulcorner p_1{:}S_1\urcorner, .., \ulcorner p_n{:}S_n\urcorner \vdash t \ \in\ T$
$\Delta \vdash e_1 \ \in\ S_1$
.
$\Delta \vdash e_n \ \in\ S_n$

$\Delta \vdash$ `let` $f\ p_1..p_n$ `=` e `in` $t \ \in\ T$
`by LetFunPatternM` S_1 `->` $..S_n$ `->` S
$\Delta,\ \ulcorner p_1{:}S_1\urcorner, .., \ulcorner p_n{:}S_n\urcorner \vdash e \ \in\ S$
$\Delta,\ f{:}\ S_1$ `->` $..S_n$ `->` $S \vdash t \ \in\ T$

$\Delta \vdash$ `let` $f\ p_1..p_n$ `=` e `in` $t \ \in\ T$
`by LetFunPatternMem`
$\Delta,\ f{:}\ S_1$ `->` $..S_n$ `->` $S \vdash t \ \in\ T$

$\Delta \vdash$ `let` p `=` $t \ \in\ T$
`by LetDeclMem`
$\Delta \vdash t \ \in\ T$

$\Delta \vdash$ `let` $f\ p_1..p_n$ `=` $t \ \in\ S_1$ `->` $..S_n$ `->` T
`by LetDeclMem`
$\Delta,\ \ulcorner p_1{:}S_1\urcorner, .., \ulcorner p_n{:}S_n\urcorner \vdash t \ \in\ T$

The rules `MatchWithMem` and `LetFunPatternMem` have to determine the type of the expression e in order to provide the required type S. It is not necessary to prove the goal $e \in S$ again. `LetDeclMem` deals with user-defined object and functions and is similar to `LetPatternMem` and `LetFunPatternMem` except that is causes a global binding of names instead of a local one. The left version of it fails if $\ulcorner p{:}S\urcorner$ does not lead to a typing.

Typing Variables in a Pattern. The algorithm for determining a typing for the individual variables of a pattern can be described by the following rule system. It terminates whenever variables, constants, `()`, `_`, or 'Or-patterns' are encountered. Note that Or-patterns, in contrast to case expressions, are not allowed to have free variables.

Δ, $\ulcorner x\texttt{:}S\urcorner \vdash C$
  by `MatchVar`
  Δ, $x\texttt{:}S \vdash C$

Δ, $\ulcorner \texttt{_:}S\urcorner \vdash C$
  by `MatchAll`
  $\Delta \vdash C$

Δ, $\ulcorner c\texttt{:}S\urcorner \vdash C$
  by `MatchConst`
  $\Delta \vdash C$

Δ, $\ulcorner \texttt{():}S\urcorner \vdash C$
  by `MatchUnit`
  $\Delta \vdash C$

Δ, $\ulcorner p_1 \texttt{|} p_2 \texttt{ : } S\urcorner \vdash C$
  by `MatchOr`
  $\Delta \vdash C$

Δ, $\ulcorner p_1\texttt{,}p_2 \texttt{ : } S\texttt{*}T\urcorner \vdash C$
  by `MatchPair`
  Δ, $\ulcorner p_1\texttt{:}S\urcorner$, $\ulcorner p_2\texttt{:}T\urcorner \vdash C$

Δ, $\ulcorner p_1\texttt{::}p_2 \texttt{ : } S \texttt{ list}\urcorner \vdash C$
  by `MatchListCons`
  Δ, $\ulcorner p_1\texttt{:}S\urcorner$, $\ulcorner p_2\texttt{:}S \texttt{ list}\urcorner \vdash C$

Δ, $\ulcorner \texttt{[}p_1\texttt{;..;}p_n\texttt{]} \texttt{ : } S \texttt{ list}\urcorner \vdash C$
  by `MatchListExpr`
  Δ, $\ulcorner p_1\texttt{:}S\urcorner\texttt{,..,}\ulcorner p_n\texttt{:}S\urcorner \vdash C$

Δ, $\ulcorner c(p)\texttt{:}c_1 \texttt{ of } T_1 \texttt{ |..| } c \texttt{ of } T_c \texttt{ |..| } c_n \texttt{ of } T_n\urcorner \vdash C$
  by `MatchConstr`
  Δ, $\ulcorner p\texttt{:}T_c\urcorner \vdash C$

Δ, $\ulcorner l_1\texttt{=}p_1\texttt{;..;}l_k\texttt{=}p_k \texttt{ : } \{l_{i_1}\texttt{:}T_{i_1}\texttt{;..;}l_{i_n}\texttt{:}T_{i_n}\}\urcorner \vdash C$ *where* $k \leq n$
  by `MatchRecordExpr`
  Δ, $\ulcorner p_1\texttt{:}T_1\urcorner\texttt{,..,}\ulcorner p_k\texttt{:}T_k\urcorner \vdash C$

Type Inference. The above rules provide the *framework* for reasoning about Ocaml programs but they do not explain how to *find* a proof. Since typechecking OCAML expressions is decidable we have elaborated an algorithm which automatically determines the type of an OCAML-expression on the meta-level of NUPRL[10] and provides a proof plan for verifying this type. Currently our type inference algorithm has about the same capabilities as the OCAML typechecker. Future versions shall also perform *extended typechecking*, which means addressing problems such as array bounds and other logical constraints which usually have to be checked during an optimization or the verification of a program.

4.2 The Computation System

While the type system is good for analyzing the extensional properties of a program, computation rules are necessary for analyzing its concrete behavior. Since all OCAML-constructs are represented completely in terms of type-theoretical expressions we could simply use NUPRL's evaluator and reduce these expressions as long as possible. This would, however, lead to uncontrollable results. In many cases the evaluator would not stop at an OCAML-value but continue and make the type-theoretical representation visible.

[10] Note that the rules `ApplyMem`, `RecordGetMem`, and `LetPatternMem` require that the type of an expression can be found automatically.

Redex	**Contractum**
`if true then` a `else` b	a
`if false then` a `else` b	b
`true &` q	q
`false &` q	`false`
p `& true`	p
p `& false`	`false`
`true or` q	`true`
`false or` q	q
p `or true`	`true`
p `or false`	p
`not true`	`false`
`not false`	`true`
$-i$, $i+j$, $i-j$, $i*j$, i/j, i `mod` j	*as usual*
$i=j$, $i<>j$, $i<j$, $i<=j$, $i>j$, $i>=j$	*as usual*
"$a_1..a_n$" ^ "$b_1..b_m$"	"$a_1..a_n b_1..b_m$"
"$a_1..a_n$".[i]	a_i
$a::[e_1;..;e_n]$	$[a;\ e_1;..;e_n]$
$[e_1;..;e_n]$ @ $[t_1;..;t_m]$	$[e_1;..;e_n;\ t_1;..;t_m]$
`length` $[e_1;..;e_n]$	n
`nth` $[e_1;..;e_n]\ i$	e_i
$[\vert e_1;..;e_n \vert].(i)$	e_i
`length` $[\vert e_1;..;e_n \vert]$	n
$[\vert e_1;..;e_n \vert]\ [i \leftarrow e]$	$[\vert e_1;\ e_2;.\ e..\ e_n \vert]$
`create` $n\ x$	$[\vert x;..;x \vert]$ *(n-times)*
`create_matrix` $n\ m\ x$	$[\vert\ [\vert x;..;x \vert];\ ..\ ;\ [\vert x;..;x \vert]\ \vert]$
`to_list` $[\vert e_1;..;e_n \vert]$	$[e_1;..;e_n]$
`of_list` $[e_1;..;e_n]$	$[\vert e_1;..;e_n \vert]$
`match` e `with` p_1`->`$t_1\ \vert..\vert p_n$`->`t_n	$t_i[\ \lceil e/p_i \rceil\ ]$ *first matching pattern is* p_i
`(function` p_1`->`$t_1\ \vert..\vert p_n$`->`t_n`)` e	$t_i[\ \lceil e/p_i \rceil\ ]$ *first matching pattern is* p_i
`(fun` $p_1..p_n$ `->` t`)` $e_1..e_n$	$t[\ \lceil e_1/p_1 \rceil,..,\lceil e_n/p_n \rceil\ ]$
`(fun` $p_1..p_n$ `->` t`)` $e_1..e_m$	p_n `-> t[` $\lceil e_1/p_1 \rceil,..,\lceil e_m/p_m \rceil$ `]`
`let` $p_1=e_1$ `and` $..\ p_n=e_n$ `in` t	$t\ [\ \lceil e_1/p_1 \rceil,..,\lceil e_n/p_n \rceil\ ]$
`let` $f\ p_1..p_n$ `=` e `in` t	t `[fun` $p_1..p_n$ `->` $e\ /\ f$`]`
$\{l_1=v_1;..;\ l_n=v_n\}.l_i$	v_i
$\{l_1=v_1;..;\ l_n=v_n\}.[l_i \leftarrow e]$	$\{l_1=v_1;..;l_i=e;..;l_n=v_n\}$
`(let` $f\ p_1..p_n$ `=` t`)` $e_1..e_n$	$t[\ \lceil e_1/p_1 \rceil,..,\lceil e_n/p_n \rceil\ ]$

Table 2. Computation Rules for OCAML-expressions

We therefore have provided a set of computation rules for each reducible OCAML-expression. In contrast to the typing rules these computation rules can operate on an arbitrary subterm of a program and require the address of that subterm to be provided as argument. This is essential for optimizing programs which – after inserting certain assumptions about their input values – contain reducible expressions. All computation rules are collected under the name `RedAt` but could also be called by their individual names.

Table 2 describes the computation system by listing the reducible OCAML-expressions and the results of applying one evaluation step. In the arithmetic reductions i and j are placeholders for integer expressions built from numbers and `-`, `+`, `*`, `/`, and `mod`. In reductions involving pattern matching the notation $[\ \ulcorner e_1/p_1 \urcorner, .., \ulcorner e_n/p_n \urcorner\ ]$ describes a substitution where the free variables of the pattern p_i are instantiated by the subterms of e_i which result from matching p_i against e_i. This substitution is defined as the result of evaluating the following reduction rules.

$[\ \ulcorner e/x \urcorner\ ]$	$\mapsto [\ e/x\ ]$	
$[\ \ulcorner e/_ \urcorner\ ]$	$\mapsto [\ ]$	
$[\ \ulcorner e\,/\,p_1 \vert p_2 \urcorner\ ]$	$\mapsto [\ \ulcorner e/p_1 \urcorner\ ]$	*if this succeeds*
	$[\ \ulcorner e/p_2 \urcorner\ ]$	*otherwise*
$[\ \ulcorner e/() \urcorner\ ]$	$\mapsto [\ ]$	
$[\ \ulcorner e/\mathtt{true} \urcorner\ ]$	$\mapsto [\ ]$	*if* $e=\mathtt{true}$
$[\ \ulcorner e/\mathtt{false} \urcorner\ ]$	$\mapsto [\ ]$	*if* $e=\mathtt{false}$
$[\ \ulcorner e/n \urcorner\ ]$	$\mapsto [\ ]$	*if* $e=n$
$[\ \ulcorner a::l\,/\,p_1::p_2 \urcorner\ ]$	$\mapsto [\ \ulcorner a/p_1 \urcorner\ ,\ \ulcorner l/p_2 \urcorner\ ]$	
$[\ \ulcorner [e_1;..;e_n]\,/\,p_1::p_2 \urcorner\ ]$	$\mapsto [\ \ulcorner e_1/p_1 \urcorner,\ \ulcorner [e_2;..;e_n]\,/\,p_2 \urcorner\ ]$	
$[\ \ulcorner [e_1;..;e_n]\,/\,[p_1;..;p_n] \urcorner\ ]$	$\mapsto [\ \ulcorner e_1/p_1 \urcorner, .., \ulcorner e_n/p_n \urcorner\ ]$	
$[\ \ulcorner e_1,..,e_n\,/\,p_1,..,p_n \urcorner\ ]$	$\mapsto [\ \ulcorner e_1/p_1 \urcorner, .., \ulcorner e_n/p_n \urcorner\ ]$	
$[\ \ulcorner \{l_1{=}v_1;..;l_n{=}v_n\}\,/\,\{l_{i_1}{=}p_1;..;l_{i_k}{=}p_k\} \urcorner\ ]$	$\mapsto [\ \ulcorner v_{i_1}/p_1 \urcorner, .., \ulcorner v_{i_k}/p_k \urcorner\ ]$	
$[\ \ulcorner \mathit{constr}(e)\,/\,\mathit{constr}(p) \urcorner\ ]$	$\mapsto [\ \ulcorner e/p \urcorner\ ]$	
$[\ \ulcorner \mathit{constr}/\mathit{constr} \urcorner\ ]$	$\mapsto [\ ]$	

Matching will fail (i.e. return **false** in the boolean component) if evaluating $[\ \ulcorner e_1/p_1 \urcorner, .., \ulcorner e_n/p_n \urcorner\ ]$ leads to a situation where none of the above rules is applicable and a proper substitution has not been generated yet. This requires some meta-level analysis in the reduction rules.

We intend to extend the rule system by a collection of *symbolic computation rules* on the basis of distributive laws such as $\mathtt{length}(a::l)\ =\ 1 + \mathtt{length}(l)$. These rule will support partial evaluation and add flexibility to the reasoning tools for symbolic optimization.

4.3 Strategies

In addition to the derived inference rules we have developed a few strategies which automate simple reasoning problems such as validating the type of a user-defined OCAML-program. These strategies will become a vital part of more sophisticated reasoning tools which shall be developed in the future.

In a typing problem $t\ \in\ T$ the tactics `CamlTypehood` and `CamlMembership` analyze the outer structure of the term t and determine the specific rule which needs to be applied. They are helpful for developing a proof step by step, particularly when investigating the reason for a failed proof.

The tactic `CamlWF` applies to theorems about the type of a given expression and proves them completely (provided they are correct). Its major applications are the wellformedness theorems which are generated by the translation process (section 5.2) but it can also be used for educational purposes. `CamlWF` generates

```
let init () ls, vs
 = let acks = Array.create_matrix ls.nmembers ls.nmembers 0
   in
     { explicitack = Param.bool vs.params "stable_explicit_ack";
       sweep = Param.time vs.params "stable_sweep";
       failed = Array.create ls.nmembers false;
       ncasts = Array.create ls.nmembers 0;
       my_row = acks.(ls.rank);
       acks = acks;
       next_gossip = Time.invalid;
       blocking = false;
       block_ok = false;
       dbg_mins = Array.create ls.nmembers 0;
       dbg_maxs = Array.create ls.nmembers 0;
       byp_hdr = NoHdr
     }
```

Fig. 5. Initialization procedure of ENSEMBLE's Stability Layer

a proof tree which consists only of applications of derived inference rules. This proof tree will usually be hidden but can be made visible on demand.

An example of such an automatically created proof is given in appendix A. It validates the type of the initialization procedure of ENSEMBLE's stability layer whose OCAML-implementation is presented in figure 5 (this is also the display form of the NUPRL-representation of `init`).

The tactic `RedAt` subsumes all the computation rules for OCAML-expressions. It expects a subterm address as argument and determines the reduction which can be applied to that subterm. The tactic `Red` tries to find the first reducible subterm and applies the appropriate reduction to it. `Red` does not require an argument but cannot be guided towards a particular expression. Finally, the tactic `RedAll` applies reductions to a term as long as it contains reducible subterms.

4.4 Implementation Issues

The NUPRL proof development system requires that all implementations of proof techniques are based on applying type-theoretical inference rules or proven lemmata. This guarantees that the proof techniques are actually correct. The implementation of a derived inference rule for an OCAML-construct therefore has to be based on its type-theoretical representation. For constructs which have a simple representation the implementation is straightforward. We only have to unfold the definition, apply the type theoretical rules corresponding to the representation, and fold back definitions in some of the subgoals. Often it also suffices to instantiate the corresponding typing theorem and leave the typing of the instances for universally quantified variables as subgoals.

Records, variant records, and patterns lead to rather complex implementations of the inference rules. This is due to the complex semantics of these constructs. Analyzing a record expression (cf. example 1), for instance, means decomposing a hidden λ-abstraction, a conditional for each label, and the default

case. In each step we have to deal with wellformedness subgoals[11] which are left by the type-theoretical inference rules. The autotactic of the NuPRL-system can handle some of them automatically but often requires assistance.

The implementation of rules for expressions with patterns can be based on the typing rules for pattern given at the end of section 4.1. The only additional complications come from the various iterations which are involved (many cases or several patterns at once). Thus a rule, which from the outside such appears to be a single inference step actually performs a series of up to *several thousand primitive inferences*.

Computation rules could often be implemented as straightforward applications of NuPRL's evaluation mechanism and unfolding of definitions. We had to make sure that only a certain number of evaluation steps can be applied. Otherwise the evaluation might end in some term which does not represent an OCAML expression. Therefore each rule had to be programmed individually.

Again, the evaluation of expressions with patterns was the most complex one. We had to control the reductions very specifically and to evaluate the boolean value indicating failure or success in each step. The implementation essentially follows the rules given at the end of section 4.2.

The tactics `CamlTypehood`, `CamlMembership`, and `RedAt` are programmed as an exhaustive case analysis which identifies the operator of the term or the kind of the redex that has to be handled. `CamlWF` consists of a few initialization steps and repeated applications of either `CamlTypehood` or `CamlMembership`. The tactic `Red` goes through a term in left-most, depth-first order in order to find a reducible expression. `RedAll` is simply a repeated application of `Red`.

A major problem is speed. Most inference rules will be applied interactively and should compute their subgoals in a short period of time. The implementation of these rules, however, has to perform all the primitive type-theoretical inferences which are necessary to establish the relation between the original goal and its subgoals. Wellformedness subproblems make the situation particularly awkward because they deal with problems that seem unrelated to the initial question but require many basic inferences. This causes a considerable delay in the execution of the rule and much time had to be spent on methods reducing the number of wellformedness problems during the execution of the rule. Decomposing the record of the function `init` (see the proof in appendix A), for instance, originally took 190 seconds of time on a Sparc 20/71 and required 100 megabytes of space[12] for searching the proof tree.

We have discovered that more than 98% of the proof efforts go into proving wellformedness problems. For experimental purposes, NuPRL can skip subgoals by applying a rule which is only valid in "development mode". We have developed

[11] The rich expressivity of type theory makes the question whether a complex expression is syntactically well-formed undecidable. Wellformedness has to be shown during a proof. This leads to additional subgoals in many inference rules.

[12] The response time would have been even worse if the whole proof could not have been constructed within the memory. The need for swapping between memory and disk on smaller machines would cause an additional delay of up to a factor of 5.

a second set of implementations for all inference rules which can be executed only in development mode. They skip all wellformedness subgoals which can *easily* be identified as such. With these rules the tactic `CamlWF` generated the *whole* proof in appendix A within 5.5 seconds, using only 4.4 megabytes of memory. We expect that another improvement of factor 25–50 will be possible by representing derived inferences as primitive rules[13] which are justified externally.

An interesting application of this methodology would be a separation of proof development and proof execution. One would provide two implementations of each inference mechanism, one for creating proofs which are completely based on primitive type-theoretical inferences, and a fast version which only performs the essential step without validating it explicitly. The latter could then be used for interactive proof development and automated proof search with acceptable response times. Once a derivation is finished, a formally complete proof could be constructed from it in the background where speed is a less important requirement. This cooperation between *trusted refiners* and background proof processes will be supported by future releases of the NuPRL proof development system.

5 Translating between OCAML and NuPRL

In the previous sections we have described the embedding of fundamental constructs from the OCAML programming language and derived inference rules for reasoning about OCAML-programs. In order to reason about the ENSEMBLE communication toolkit we have to represent its implementation in terms of the embedded constructs. This representation should be generated fully automatically because the code of the system is likely to change over the years and protocol layers will be composed in various ways depending on the particular application. On the other hand, fast track reconfigurations of application specific communication systems generated in NuPRL have to be converted back into OCAML source code in order to become applicable. Hence we had to develop algorithms which automatically translate user-defined OCAML-programs into abstraction objects of the NuPRL library and vice versa. The general methodology of these translations is illustrated in figure 6.

The translation algorithm from OCAML into NuPRL has to perform a syntax analysis of the program text and to convert it into structured terms which then will be stored as library objects. In order to ensure faithfulness to the programming environment used for executing ENSEMBLE-code, the first step should be performed by the original OCAML-parser. The second obviously has to be realized on the meta-level of NuPRL. Both steps are strongly related and had to be approached simultaneously.

- Parsing OCAML-programs leads to a syntax tree which is already a structured description of the program text. Unfortunately the syntax tree is an object of the OCAML-programming environment which has a different structure than

[13] NuPRL-LIGHT [11], a modular re-implementation of the NuPRL refiner also influencing its next release, supports the creation inference rules as system primitives.

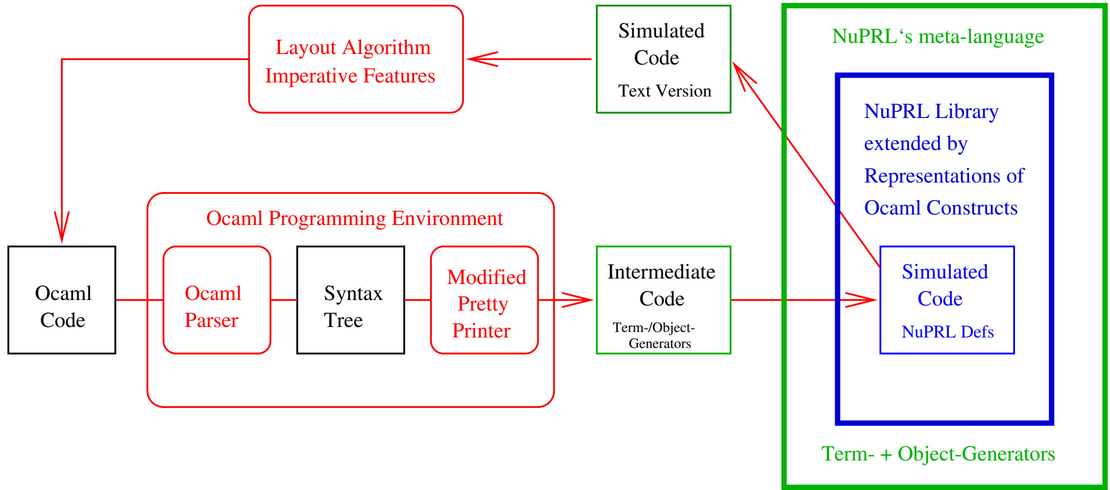

Fig. 6. Translating between OCAML and NUPRL: General Methodology

the meta-level of NUPRL. Therefore the OCAML-parser cannot build syntax trees that can immediately be interpreted by the NUPRL-system[14] but has to send its results as text. This text will describe program code which can be executed by the NUPRL system and eventually leads to the construction of type-theoretical terms and library objects.

– The programs generated by the OCAML-parser should not only create type theoretical terms but also abstraction objects and display forms. These are necessary for storing the generated terms permanently in the library under a name given by the user. Without them no references to existing user-defined functions could be made within the type-theoretical representation of a program. Typing theorems should be generated and proven in order to automatically expand the type inference system to all user-defined functions and constants. We have provided a set of meta-level programs which will generate these objects and cause the OCAML-parser to create the corresponding text for each top-level declaration.

  Besides providing object-generators we also have to deal with a few issues that cannot be handled during the parsing process but require a deeper analysis. In particular we have to manage the name space to avoid conflicts between declarations which occur in different modules under the same name and to analyze the role of identifiers.

The translation from NUPRL into OCAML is comparably simple and independent from the other two steps because it converts structured objects into unstructured ones. Since the embedded programs are displayed as genuine OCAML-code it only has to build a print representation of the terms contained in the library which we wish to convert.

[14] The combination between the OCAML-parser and the meta-level of NUPRL would be less complicated if the meta-language of NUPRL would not be the original ML language but OCAML as in NUPRL-LIGHT [11]. In this case structural information could be passed directly.

5.1 Parsing and Preprocessing OCAML-code

Taking the original parser of OCAML for analyzing the structure of program text is not only more efficient but also the only way of ensuring the faithfulness with respect to the programming environment which is used for running the ENSEMBLE communication system. A modifiable, isolated version of the OCAML-parser is the CAMLP4 parser-preprocessor [7] which parses a source file, generates an abstract syntax tree, and returns the result either as pretty-printed text or as binary dump. To run, it loads separate files holding operations which define parsing and printing. These files can be provided by a user if there is need for a modified input syntax or a different method for printing the result.

For our purposes, CAMLP4 is almost the ideal tool. We 'only' had to write a module for converting the abstract syntax tree into the source code of the corresponding term- and object-generators. This means generating pieces of text for each possible content of a syntax tree node, distinguishing the various kinds of identifiers, language expressions, patterns, types, signature items, module types, and module expressions (see [7, Appendix A]). Although not all of these are used within the implementation of ENSEMBLE, about 80 cases had to be distinguished.

We have decided to modify one of the existing pretty printing modules for this purpose instead of creating the conversion module from scratch. Some amount of pretty printing was necessary anyway to keep the generated 'intermediate code' comprehensible and to allow the detection of typing errors and similar mistakes during the development phase.

Except for these considerations the solution was straightforward. To give an example how a syntax tree node is analyzed and converted into meta-language code for the NuPRL-system we present a piece of the module which deals with types, indicated by the so-called quotations `<:ctyp_<..>>` and `<:ctyp<..>>`.

```
(fun curr next t _ k ->
  match t with
  [(****** TYPE VARIABLES ******)
    <:ctyp_< '$s$ >> ->
      [: `S LO " (Typ_Var `"; `S NO (var_escaped s); `S NO "`) "; k :]
   (****** TYPE IDENTIFIERS / Constants ******)
  | <:ctyp< $lid:s$ >> ->
      [: `HVbox [: `S LO " (Typ_Id `"; `S NO s; `S NO "`) ":]; k :]
  ⋮
  (****** RECORD TYPES ******)
  | <:ctyp< { $list:ftl$ } >> ->
      [: `S LR "(Typ_record [";
         `HVbox [: labels [: :] ftl "" [: :]  :];
         `S RO "] )" ;
         k  :]
  ⋮
  (****** FUNCTION TYPES ******)
  | <:ctyp< $x$ -> $y$ >> ->
      [: `S LR "(Typ_fun ";
         `next x "" [: :];
         curr y "" [: `S LR ")" :];
         k  :]
```

The module matches the actual syntax tree `t` against possible structures and emits a sequence of text blocks, formatted by various commands. Sometimes subterms need to be converted into text and inserted in between and some priorities have to be settled for this purpose (using `next` and `curr`). To avoid a possible confusion between expressions, patterns, types, modules, and top-level declarations when the created meta-language text is evaluated in NuPRL we have created term-generators with different prefixes.

Figure 7 shows an example of meta-level code which has been generated by the parser-preprocessor module. The code is the result of translating the OCAML-implementation of the initialization procedure of ENSEMBLE's stability layer (see figure 5). It will create a type-theoretical representation of the function `init` whose display form will be the same as the original program.

A parser is restricted to a syntactical analysis of a single text file and cannot solve problems which arise when linking the code of several modules.

- *Name Resolution*: parsing can only determine the local name of a user-defined function or type. If the same name occurs in different modules it must be guaranteed that different objects are created and that references to that name will always be bound to the intended object.
- *The Role of Identifiers*: parsing cannot determine whether an identifier refers to a local variable or a user-defined operator. It is necessary to analyze bindings of variable names in order to decide whether an identifier shall be represented as variable or not. In the latter case a reference to the appropriate library object needs to be established.
- *Overloading*: some operations like equality can have different meanings, depending on the type of the arguments. Type inference is required before the correct implementation can be selected.

Solutions for these problems had to be provided at the meta-level of NuPRL during the evaluation of the term- and object-generators

5.2 Constructing Terms and Library Objects

After parsing and preprocessing the meta-level of NuPRL has to assign meaning to the generated intermediate code and evaluate it. While representations of the individual language constructs can be created using term-generators (see section 3.1) top-level declarations must lead to the construction of abstractions, display forms, and typing theorems. We therefore had to provide a set of meta-level programs which will create such library objects and also address the above-mentioned issues which could not be handled during the parsing process.

Name Resolution. The Objective Caml programming language supports a modular approach to programming and allows to use the same name for different operators in different modules. To avoid ambiguities, an operator name must be prefixed by the module name (with a dot as separator) unless the operator belongs to the local module or to modules which were opened at the beginning of the module. In contrast to that, the library of the NuPRL system has essentially a flat name space and does not support duplicate names.

```
DECLARE_BINDING
  [ (Pat_Id `init`); (Pat_Const `()`);
        ((Pat_Tup [(Pat_Id `ls`);(Pat_Id `vs`) ]))
  ], (mk_Stmt [
      (mk_let [
        [ (Pat_Id `acks`) ],
          (mk_Stmt [
            (mk_apply (mk_apply (mk_apply
               (mk_RemoteId  `Array` `create_matrix`)
                (mk_RecordGet (mk_Id `ls`) `nmembers` ) )
                 (mk_RecordGet (mk_Id `ls`) `nmembers` ) )
                  (mk_num 0) )
            ] ) ]
      (mk_Stmt [
       (mk_RecordExpr [
        `explicitack`, (mk_apply (mk_apply
                          (mk_RemoteId  `Param` `bool`)
                           (mk_RecordGet (mk_Id `vs`) `params` ) )
                            (mk_StringExpr "stable_explicit_ack") );
        `sweep`,       (mk_apply (mk_apply
                          (mk_RemoteId  `Param` `time`)
                           (mk_RecordGet (mk_Id `vs`) `params` ) )
                            (mk_StringExpr "stable_sweep") );
        `failed`,      (mk_apply (mk_apply
                           (mk_RemoteId  `Array` `create`)
                            (mk_RecordGet (mk_Id `ls`) `nmembers` ) )
                             (mk_Const `false`) );
        `ncasts`,      (mk_apply (mk_apply
                           (mk_RemoteId  `Array` `create`)
                            (mk_RecordGet (mk_Id `ls`) `nmembers` ) )
                             (mk_num 0) );
        `my_row`,      (mk_ArrayGet (mk_Id `acks`)
                             (mk_RecordGet (mk_Id `ls`) `rank` ));
        `acks`,        (mk_Id `acks`) ;
        `next_gossip`, (mk_RemoteId  `Time` `invalid`);
        `blocking`,    (mk_Const `false`) ;
        `block_ok`,    (mk_Const `false`) ;
        `dbg_mins`,    (mk_apply (mk_apply
                             (mk_RemoteId  `Array` `create`)
                              (mk_RecordGet (mk_Id `ls`) `nmembers` ) )
                               (mk_num 0) );
        `dbg_maxs`,    (mk_apply (mk_apply
                             (mk_RemoteId  `Array` `create`)
                              (mk_RecordGet (mk_Id `ls`) `nmembers` ) )
                               (mk_num 0) );
        `byp_hdr`,     (mk_Const `NoHdr`)  ] ) ] )
      ) ] )
;;
```

Fig. 7. Code for Constructing the Representation of `init`

In order to separate the operators of different modules from each other each top-level declaration of a type, function, or constant will be stored in an object whose name is built from the name of the local module and the name of the declaration.[15] Types, which are allowed to have the same name as functions of the same module, will additionally be prefixed by the keyword `Type_`. The display form of all generated objects will consist only of the name of the declaration. Otherwise local references will not be displayed as such. Thus the initialization procedure of ENSEMBLE's stability layer (see figure 5) will be stored within the abstraction `Stable_DOT_init` and displayed as `init`.

If a reference includes the name of a module the corresponding NUPRL-object can be identified easily. Otherwise we need to know which modules are currently open and have to search for the corresponding objects in the library. Name spaces can therefore not be managed without caches for the name of the local module and for currently open modules. The former will be set when importing the module into NUPRL while the contents of the other will be determined when executing the translation of the OCAML-command `open` *module*.

Identifying the Role of Identifiers. Because of lexical conventions OCAML's parser is able to distinguish constants from other identifiers but it cannot determine whether an identifier refers to a variable or a user-defined operator. This decision can only be made in later steps of the compilation when variable bindings are being analyzed.

In our translation we have to simulate this process on the meta-level of NUPRL. The decision about the nature of an identifier cannot be made by a term-generator for it but only in the context of the whole expression in which it occurs. If there is a binding occurrence of the identifier then it must be a variable since local bindings have priority over global declarations. Otherwise it is either a user definition or some predefined function such as '`not`', '`failwith`' etc. Neither OCAML nor NUPRL allow free variables in top-level expressions.

Technically, terms are constructed bottom-up and the term-generator for identifiers will be executed before its context is known. Therefore it has to generate a special identifier term without an external meaning and postpone the decision about the role of the identifier to the generators for expressions with bindings and top-level declarations. The generators for function declarations, function definitions, local definitions, case expressions, and for-loops will interpret variable bindings by turning the corresponding identifier terms into variables. Identifier terms which have not been converted when an abstraction is being formed must refer to an operator and the reference is built according to the guidelines explained above.

This requires a few meta-level functions which can detect identifiers in arbitrarily nested subterms and convert them into variables or operator references. These functions will be called by modified term generators for language constructs with bindings and whenever an abstraction is being generated.

[15] NUPRL 4.2 does not support dots within object names. Instead we use the naming convention *module*`_DOT_`*name*.

Overloading. In many programming languages the equality function is universally being denoted by the symbol '=' although it has different implementations for integers, floating point numbers, strings, etc. Parsing will identify '=' as equality function but it cannot yet determine the concrete algorithm which has to be executed at runtime since this requires finding the type of the function's arguments. In statically typed languages like OCAML this can usually be done at compile time while object-oriented languages require a type analysis at runtime.

Overloading is one of the key features of modular programming languages[16] and related to the problem of name spacing. As in the latter case we have to find a reference to the operator (i.e. implementation) that can be applied to the given expressions. But overloading requires to choose between several alternatives. In our translation we have to use type inference for this purpose and to assign a reference to the correct abstraction accordingly.

This method is, however, limited to monomorphic occurrences of overloaded functions. Otherwise we would have to find a reference to a polymorphic implementation. ENSEMBLE uses overloading only in this restricted sense.

Extended Term Generators. Term generators as described in section 3.1 were originally designed independently from the translation process in order to obtain a unique representation of each OCAML-construct. It has turned out that in some cases the OCAML-parser leads to a finer analysis of terms while in others it does not distinguish well enough. Thus besides extending some of the term generators by name space management procedures we had to implement additional constructors which exist only for translation purposes.

Constructors defined by variant records and constant operators like `()`, `[]`, `true`, `false`, `None` are only identified as constants with a certain name. The generator `mk_Const` must make a finer distinction and create the appropriate NUPRL-terms. Similarly assignments are only identified as such. We need to analyze the term which shall be modified by the assignment, check whether values are allowed to be assigned to it at all (see section 3.5), and then create the corresponding assignment abstraction.

On the other hand types, patterns, and expressions can be separated from each other during the parsing process. This is particularly helpful when dealing with identifiers and constants while in most other cases generators for types and patterns turn out to be the same as the original term generators. For the sake of clarity in the intermediate code we have kept the distinction in our conversion module and provided additional term generators prefixed with `Typ_` or `Pat_`.

Object Generators. Each top-level declaration in a module, including comments and opening of other module must lead to the construction of library ob-

[16] Overloading is not to be confused with *polymorphism* which can be handled easily in type theory. A polymorphic function like the identity is a reference to a *single* algorithm which can be applied to objects of various types. An overloaded function like equality refers to *different* implementations for objects of different types. Overloading supports an elegant and semantically well structured programming style.

jects. Comments and commands for opening other modules are converted into comment objects. The latter also affect the cache of currently open modules. For declarations of types, functions, and constants the translation must define an abstraction whose right hand side looks like the original declaration, turn the declaration's name into a display form, and create a verified theorem which states the type of the newly defined object.

Since the basic mechanism is the same in all these cases we have written a universal object generator `DECLARE` which expects as inputs a name, its defining expression, and a term describing its (possibly polymorphic) type. `DECLARE` creates references for unidentified identifiers in the defining expression and creates all the required objects while making sure that naming conventions are respected. The tactic `CamlWF` (see section 4.3) is used to prove the typing theorem. A fourth parameter is necessary to distinguish types from operators while keeping the name unchanged.

Each kind of top-level declarations has its own kind of object generator which is built on top of `DECLARE`. The generator `DECLARE_BINDING`, which has been used in figure 7, expects as input a list consisting of (a variable containing) the name of an operator to be declared, possibly some input patterns, and the target term which defines the new operator. Using the term generator `mk_LetDecl` for operator declarations and an appropriate binding of the variables of the patterns it constructs the defining term of the new object. The term describing its type will be created via type inference and all the information is passed to the function `DECLARE` as shown below.

```
let DECLARE_BINDING (declared.input_patterns), target  =
  let name = dvar (dst_var declared)
  and bound_target = mk_LambdaBindings input_patterns target
  in
    let ab_rhs = mk_LetDecl name bound_target
    and type   = wf_goal_of name bound_target
    in
       DECLARE nulltok name ab_rhs type
```

The object generators `DECLARE_REC_BINDING` (recursive function declaration) and `DECLARE_TYPE` (type declaration) are implemented in a similar fashion.

Determining the Type of Generated Expressions. The type of a type declaration can easily be identified by the number n of type parameters involved in the declaration and then be checked with `CamlWF`. The type is either `Types` if there are no parameters or n`-ary type declarations`.

The type of a newly defined object could be determined by a universal type inference algorithm (see section 4.1) before it is being checked with `CamlWF`. In the context of top-level declarations, however, there is a much more efficient method. We can make use of the fact that in a well-structured modular system like Ensemble each module implementation is accompanied by a module interface which contains a typing of each operator. By translating this interface and loading it into another cache before the module itself is being translated we can use the typing information effectively.[17]

Another interesting application of module interfaces is an *abstract translation* of OCAML-modules into NUPRL which currently is only possible in development mode. It provides a possibility for reasoning about programs containing certain operators without having to translate their implementation. An abstract translation just introduces new terms and typing theorems for the operators of a module without providing a type-theoretical definition of the term or a proof of the theorem. Technically, the abstraction object will have some dummy term as right hand while the typing theorem is proven by the special rule of the development mode. The object generator `DECLARE_ABSTRACTLY` will create such an abstract declaration from a signature interface contained in the cache.

Importing Modules. Modules are imported as a whole using the meta-level function `IMPORT`. Applying this function initializes the caches and causes the parser to convert a module file and the corresponding interface file, if present, into intermediate code. This code will then be executed which may result in loading certain meta-level data of other modules in order to determine which 'remote' constructors and record fields should be known within the current module. After all types and operators have been converted into objects of the library, the meta-level data of the current module will be stored as a library object as well.

Importing a module usually results in the creation of several hundred library objects and is rather time consuming. The conversion of ENSEMBLE's stability layer and the 19 modules on which its implementation depends, for instance, generated more than 1800 library objects and took about 15 minutes on a Sparc 20/71. Most of this time is needed for creating and checking library objects while the time for generating the intermediate code was comparably short.

5.3 Translating from NUPRL into OCAML source code

The translation from NUPRL into OCAML requires only a mechanism for generating a print representation of structured terms which is identical to its display on a computer screen. Fortunately the NUPRL system already provides such a mechanism for printing libraries and proofs (see the appendices) and we only had to write a procedure which automatically selects the objects which shall be printed while suppressing unwanted information.

Only the right hand sides of abstractions should be selected for printing. Theorems, display forms, meta-level objects, etc. contain information which was valuable for representing OCAML-programs and reasoning about them but does not contribute to the program itself. If a program has been reconfigured then only the reconfigured version should be printed. Comments may be printed if desired. All these requirements can be implemented straightforwardly.

[17] One might also go one step further and use the translation mechanism to connect the OCAML-typechecker as external refiner to the NUPRL-system which will be used whenever type information is needed. Since type inference is a major aspect of formal reasoning this method may improve the efficiency of reasoning about larger pieces of code and lead to an even stronger consistency with the OCAML-runtime system.

6 Conclusion

We have presented an embedding of the Objective Caml programming language into the type theory of the NuPRL proof development system which is based on a direct formalization of the (intuitive) semantics of OCAML language constructs in terms of type-theoretical expressions. We also have implemented (derived) inference rules for reasoning about the embedded OCAML-constructs as programmed applications of type-theoretical inferences. We also developed a type inference algorithm as well as a few simple strategies for typechecking and evaluating OCAML-programs within NuPRL. Finally we have provided algorithms for automatically translating between original OCAML-code and its type-theoretical simulation. Together, these tools and formalizations form the foundation for the development of automated tools for reasoning about group communication systems.

In our work we have focused on a comprehensible representation of a reasonably large subset of OCAML. Meta-level term constructors and appropriate display forms for type-theoretical abstractions make sure that embedded OCAML-programs appear to be genuine OCAML-code. The type theoretical peculiarities of the formalization usually remain hidden from a user but may be revealed on demand. By formalizing only a subset of OCAML which is targeted towards the particular application we could avoid the overhead of a formal model of the complete OCAML-language. Full imperative features, exceptions, unrestricted recursion, and other language constructs with an extremely complex functional representation are not used in the implementation of the ENSEMBLE group communication toolkit and were not represented.

Despite this restriction, our techniques are general enough to support the development of automated reasoning tools not only for communication systems but for all systems which can be implemented in a clean mathematical fashion (especially algorithms based on finite state systems). It will be possible for system experts to use such tools interactively even without knowledge about type theory while on the other hand a partial automation of the reasoning process becomes practically feasible.

In the future we will develop automated reasoning tools as support for a fast-track reconfiguration and verification of group communication systems which are constructed with the ENSEMBLE communication toolkit. These tasks will require the realization of several inference techniques on top of the basic inference system. Fast-Track reconfiguration is essentially symbolic optimization under a given set of assumptions and can be realized by *(conditional) rewriting*. For this purpose we need to develop strategies for context analysis and *automatic simplification* and other program transformation techniques.

Program verification, on the other hand, requires techniques for reasoning about program properties. Among those, typechecking is the most fundamental one since many basic properties involve reasoning about the type of certain objects. We intend to develop an algorithm for *extended typechecking* which will be able to deal with all elementary program properties which are usually checked at runtime. In addition to that we need to investigate methods for the *formal*

specification of distributed systems in a sequential theory and apply them for specifying the properties of ENSEMBLE's protocol layers. Proof presentation and *automated proof structuring* will also be an important research issue. A formal proof verifying a specified system property can serve as documentation of the system which provides precise insights into the system's behavior. Tools for browsing formal proofs shall allow to present this information at various levels of detail thus supporting both the beginner and the expert. Finally a full *programming logic*[18] for reasoning about OCAML-constructs needs to be explicitly formulated and automated techniques for constructing proofs in that logic have to be elaborated. The field of automated deduction provides a variety of decision procedures and proof search techniques which could be adapted and specialized for this purpose.

Acknowledgments

I would like to thank Mark Hayden and Jason Hickey for sharing insights about OCAML and the CAMLP4 parser-preprocessor as well as for insightful discussions about various alternatives in the formalization of OCAML. I would also like to thank Ozan Hafizogullari for elaborating the type inference algorithm mentioned at the end of section 4.1.

Last but not least I would like to thank Bob Constable for organizing the project, for all the efforts he took to get me involved in it, and for the many helpful and inspiring comments on draft versions of this report.

[18] It should be noted that by embedding OCAML into type theory we get full access to the semantics of an OCAML-program. The programming logic implicitly contained in the embedding is therefore much more general than Hoare Logic or similar logics for reasoning about programs: it enables reasoning about a program's structure (e.g. for reconfiguration) and embedding program pieces into arbitrary mathematical formulae. This logic, however, needs to be made explicit in the form of verified lemmata about the logical laws of OCAML-constructs and proof tactics for automated reasoning. Elementary Hoare Logic, including the laws for composition and loops, has already been formulated and proven in NUPRL.

References

1. M. Aagaard and M. Leeser. Verifying a logic synthesis tool in Nuprl. In *Proceedings of the Workshop on Computer-Aided Verification*, Lecture Notes in Computer Science. Springer Verlag, 1993.
2. Stuart Allen. A non-type-theoretic definition of Martin-Löf's types. In David Gries, editor, *LICS-87 — Proceedings of the Second Annual Symposium on Logic in Computer Science*, pages 215–224. IEEE Computer Society Press, 1987.
3. K. P. Birman. *Building Secure and Reliable Network Applications.* Manning Publishing Co. & Prentice Hall, 1996.
4. K.P. Birman and R. van Renesse. *Reliable Distributed Computing with the Isis Toolkit.* IEEE Computer Society Press, 1994.
5. Robert L. Constable. The structure of NuPRL's type theory. In M. Broy and H. Schwichtenberg, editors, *Logic of Computation.* Springer Verlag, 1997.
6. Robert L. Constable, Stuart F. Allen, H. Mark Bromley, W. Rance Cleaveland, J. F. Cremer, Robert W. Harper, Douglas J. Howe, Todd B. Knoblock, Nax Paul Mendler, Prakash Panangaden, Jim T. Sasaki, and Scott F. Smith. *Implementing Mathematics with the NuPRL proof development system.* Prentice Hall, 1986.
7. Daniel de Rauglaudre. *Camlp4 version 0.5.* Institut National de Recherche en Informatique et en Automatique, 1997.
8. Michael J. Gordon, Robin Milner, and Christopher P. Wadsworth. *Edinburgh LCF: A mechanized Logic of Computation.* Number 78 in Lecture Notes in Computer Science. Springer Verlag, 1979.
9. Mark Hayden. *Ensemble Reference Manual.* Cornell University, 1996.
10. Mark Hayden and Robbert vanRenesse. Optimizing layered communication protocols. Technical Report TR 96-1613, Cornell University. Department of Computer Science, 1996.
11. Jason Hickey. NuPRL–Light: An implementation framework for higher-order logics. In W. McCune, editor, *Proceedings of the 14th Conference on Automated Deduction*, Lecture Notes in Artificial Intelligence. Springer Verlag, 1997.
12. D.J. Howe. Importing mathematics from HOL into NuPRL. In J. von Wright, J. Grundy, and J. Harrison, editors, *Theorem Proving in Higher Order Logics*, number 1125 in Lecture Notes in Computer Science, pages 267–282. Springer Verlag, 1996.
13. Paul Jackson. *NuPRL ML Manual.* Cornell University. Department of Computer Science, 1993.
14. Paul Jackson. *The Nuprl Proof Development System, Version 4.2: Reference Manual and User's Guide.* Cornell University. Department of Computer Science, 1994.
15. Xavier Leroy. *The Objective Caml system release 1.05.* Institut National de Recherche en Informatique et en Automatique, 1997.
16. Per Martin-Löf. Constructive mathematics and computer programming. In *6-th International Congress for Logic, Methodology and Philosophy of Science, 1979*, pages 153–175. North–Holland, 1982.
17. Per Martin-Löf. *Intuitionistic Type Theory*, volume 1 of *Studies in Proof Theory Lecture Notes.* Bibliopolis, 1984.
18. R. van Renesse, K. Birman, and S. Maffeis. Horus: A flexible group communication system. *Communications of the ACM*, 39(4):76–83, 1996.

A Example proof

```
⊢ init ∈ unit -> View.local * View.state -> state
BY unfold `Stable_DOT_init`
|
⊢ let init () ls, vs
|  = let acks = Array.create_matrix ls.nmembers ls.nmembers 0
|      in
|        { explicitack = Param.bool vs.params "stable_explicit_ack";
|          sweep = Param.time vs.params "stable_sweep";
|          failed = Array.create ls.nmembers false;
|          ncasts = Array.create ls.nmembers 0;
|          my_row = acks.(ls.rank);
|          acks = acks;
|          next_gossip = Time.invalid;
|          blocking = false;
|          block_ok = false;
|          dbg_mins = Array.create ls.nmembers 0;
|          dbg_maxs = Array.create ls.nmembers 0;
|          byp_hdr = NoHdr
|        }
|   ∈ unit -> View.local * View.state -> state
BY LetDeclMem
|
1. ls: View.local
2. vs: View.state
⊢ let acks = Array.create_matrix ls.nmembers ls.nmembers 0
|  in
|    { explicitack = Param.bool vs.params "stable_explicit_ack";
|      sweep = Param.time vs.params "stable_sweep";
|      failed = Array.create ls.nmembers false;
|      ncasts = Array.create ls.nmembers 0;
|      my_row = acks.(ls.rank);
|      acks = acks;
|      next_gossip = Time.invalid;
|      blocking = false;
|      block_ok = false;
|      dbg_mins = Array.create ls.nmembers 0;
|      dbg_maxs = Array.create ls.nmembers 0;
|      byp_hdr = NoHdr
|    }
|   ∈ state
BY LetPatternMem
|\
| 3. acks: int array array
| ⊢ { explicitack = Param.bool vs.params "stable_explicit_ack";
| |     sweep = Param.time vs.params "stable_sweep";
| |     failed = Array.create ls.nmembers false;
| |     ncasts = Array.create ls.nmembers 0;
| |     my_row = acks.(ls.rank);
| |     acks = acks;
| |     next_gossip = Time.invalid;
| |     blocking = false;
| |     block_ok = false;
| |     dbg_mins = Array.create ls.nmembers 0;
| |     dbg_maxs = Array.create ls.nmembers 0;
| |     byp_hdr = NoHdr
| |   }
| |    ∈ state
1 BY RecordExprMem
```

```
| |\
| | ⊢ Param.bool vs.params "stable_explicit_ack" ∈ bool
1 2 BY ApplyMem
| | |\
| | | ⊢ Param.bool ∈ Param.tl -> string -> bool
1 2 3 BY LongIdMem
| | | |
| | | ⊢ bool ∈ Param.tl -> string -> bool
1 2 3 BY PreDefinedObjectMem
| | |\
| | | ⊢ vs.params ∈ Param.tl
1 2 3 BY RecordGetMem
| |  \
| |   ⊢ "stable_explicit_ack" ∈ string
1 2   BY StringExprMem
| |\
| | ⊢ Param.time vs.params "stable_sweep" ∈ Time.t
| | |
1 2 BY ApplyMem
| | |\
| | | ⊢ Param.time ∈ Param.tl -> string -> Time.t
1 2 3 BY LongIdMem
| | | |
| | | ⊢ time ∈ Param.tl -> string -> Time.t
1 2 3 BY PreDefinedObjectMem
| | |\
| | | ⊢ vs.params ∈ Param.tl
1 2 3 BY RecordGetMem
| |  \
| |   ⊢ "stable_sweep" ∈ string
1 2   BY StringExprMem
| |\
| | ⊢ Array.create ls.nmembers false ∈ bool array
1 2 BY ApplyMem
| | |\
| | | ⊢ Array.create ∈ nmembers -> bool -> bool array
1 2 3 BY LongIdMem
| | | |
| | | ⊢ create ∈ nmembers -> bool -> bool array
1 2 3 BY PreDefinedObjectMem
| | |\
| | | ⊢ ls.nmembers ∈ nmembers
1 2 3 BY RecordGetMem
| |  \
| |   ⊢ false ∈ bool
1 2   BY FalseMem
| |\
| | ⊢ Array.create ls.nmembers 0 ∈ seqno array
1 2 BY ApplyMem
| | |\
| | | ⊢ Array.create ∈ nmembers -> int -> seqno array
1 2 3 BY LongIdMem
| | | |
| | | ⊢ create ∈ nmembers -> int -> seqno array
1 2 3 BY PreDefinedObjectMem
| | |\
| | | ⊢ ls.nmembers ∈ nmembers
1 2 3 BY RecordGetMem
| |  \
| |   ⊢ 0 ∈ int
1 2   BY NatnumMem
```

```
| |\
| | ⊢ acks.(ls.rank) ∈ seqno array
1 2 BY ArrayGetMem
| | |\
| | | ⊢ acks ∈ int array array
1 2 3 BY VarMem
| |  \
| |   ⊢ ls.rank ∈ int
1 2   BY RecordGetMem
| |\
| | ⊢ acks ∈ seqno array array
1 2 BY VarMem
| |\
| | ⊢ Time.invalid ∈ Time.t
1 2 BY LongIdMem
| | |
| | ⊢ invalid ∈ Time.t
1 2 BY PreDefinedObjectMem
| |\
| | ⊢ false ∈ bool
1 2 BY FalseMem
| |\
| | ⊢ false ∈ bool
1 2 BY FalseMem
| |\
| | ⊢ Array.create ls.nmembers 0 ∈ seqno array
1 2 BY ApplyMem
| | |\
| | | ⊢ Array.create ∈ nmembers -> int -> seqno array
1 2 3 BY LongIdMem
| | | |
| | | ⊢ create ∈ nmembers -> int -> seqno array
1 2 3 BY PreDefinedObjectMem
| | |\
| | | ⊢ ls.nmembers ∈ nmembers
1 2 3 BY RecordGetMem
| |  \
| |   ⊢ 0 ∈ int
1 2   BY NatnumMem
| |\
| | ⊢ Array.create ls.nmembers 0 ∈ seqno array
1 2 BY ApplyMem
| | |\
| | | ⊢ Array.create ∈ nmembers -> int -> seqno array
1 2 3 BY LongIdMem
| | | |
| | | ⊢ create ∈ nmembers -> int -> seqno array
1 2 3 BY PreDefinedObjectMem
| | |\
| | | ⊢ ls.nmembers ∈ nmembers
1 2 3 BY RecordGetMem
| |  \
| |   ⊢ 0 ∈ int
1 2   BY NatnumMem
|  \
|   ⊢ NoHdr ∈ header
1   BY VariantConstrMem
```

```
\
 ⊢ Array.create_matrix ls.nmembers ls.nmembers 0 ∈ int array array
 BY ApplyMem
 |\
 | ⊢ Array.create_matrix ∈ nmembers -> nmembers -> int -> int array array
 1 BY LongIdMem
 |  |
 | ⊢ create_matrix ∈ nmembers -> nmembers -> int -> int array array
 1 BY PreDefinedObjectMem
 |\
 | ⊢ ls.nmembers ∈ nmembers
 1 BY RecordGetMem
 |\
 | ⊢ ls.nmembers ∈ nmembers
 1 BY RecordGetMem
  \
   ⊢ 0 ∈ int
   BY NatnumMem
```

B The NuPRL-Theory Ocaml

Comment objects add structure to the library and make it more comprehensible.

```
*C Ocaml_begin        ******************************************************
*C Ocaml_begi_        **                      Theory OCAML                 **
*C Ocaml_beg__        ******************************************************

*D O_fail_df          fail== O_fail
*A O_fail             fail ==  any 0
*T O_fail_wf          ⊢ ∀T:U1. fail ∈ T
*D LongId_df          <module:tok>.<t:term:*>== LongId{<module>:t}(<t>)
*A LongId             $module.t ==  t
*T LongId_wf          ⊢ ∀T:U1. ∀t:T. ∀modules:Atom.  modules.t ∈ T

*C Ocaml_classes1     ======================================================
*C Ocaml_classes2     Classes for Ocaml Types / Type constructors
*C Ocaml_classes3     ======================================================

*D TYPES              TYPES== U1

*D TypeConstr1_df     1-ary Type Constructors== TypeConstr1
*A TypeConstr1        1-ary Type Constructors == TYPES -> TYPES
*T TypeConstr1_wf     ⊢ 1-ary Type Constructors ∈ U2
*D TypeConstr2_df     2-ary Type Constructors== TypeConstr2
*A TypeConstr2        2-ary Type Constructors == TYPES*TYPES -> TYPES
*T TypeConstr2_wf     ⊢ 2-ary Type Constructors ∈ U2
*D TypeConstr3_df     3-ary Type Constructors== TypeConstr3
*A TypeConstr3        3-ary Type Constructors == TYPES*TYPES*TYPES -> TYPES
*T TypeConstr3_wf     ⊢ 3-ary Type Constructors ∈ U2
*D TypeConstr4_df     4-ary Type Constructors== TypeConstr4
*A TypeConstr4        4-ary Type Constructors == TYPES*TYPES*TYPES*TYPES -> TYPES
*T TypeConstr4_wf     ⊢ 4-ary Type Constructors ∈ U2
*D TypeConstr5_df     5-ary Type Constructors== TypeConstr5
*A TypeConstr5        5-ary Type Constructors == TYPES*TYPES*TYPES*TYPES*TYPES -> TYPES
*T TypeConstr5_wf     ⊢ 5-ary Type Constructors ∈ U2
*D TypeConstr6_df     6-ary Type Constructors== TypeConstr6
*A TypeConstr6        6-ary Type Constructors == TYPES*TYPES*TYPES*TYPES*TYPES*TYPES -> TYPES
*T TypeConstr6_wf     ⊢ 6-ary Type Constructors ∈ U2

*D TypeApply_df       <T:Type> <TC:TypeConstr>== TypeApply(<TC>; <T>)
*A TypeApply          T TC ==  TC (T)
*T TypeApply_wf1      ⊢ ∀TC:1-ary Type Constructors. ∀T:TYPES.  T TC ∈ TYPES
*T TypeApply_wf2      ⊢ ∀TC:2-ary Type Constructors. ∀T1,T2:TYPES.  (T1,T2) TC ∈ TYPES
*T TypeApply_wf3      ⊢ ∀TC:3-ary Type Constructors.
                         ∀T1,T2,T3:TYPES.  (T1,T2,T3) TC ∈ TYPES
*T TypeApply_wf4      ⊢ ∀TC:4-ary Type Constructors.
                         ∀T1,T2,T3,T4:TYPES.  (T1,T2,T3,T4) TC ∈ TYPES
*T TypeApply_wf5      ⊢ ∀TC:5-ary Type Constructors.
                         ∀T1,T2,T3,T4,T5:TYPES.  (T1,T2,T3,T4,T5) TC ∈ TYPES
*T TypeApply_wf6      ⊢ ∀TC:6-ary Type Constructors.
                         ∀T1,T2,T3,T4,T5,T6:TYPES. (T1,T2,T3,T4,T5,T6) TC ∈ TYPES

*T TypeSubst          ⊢ ∀S,T:TYPES.  S = T ∈ TYPES ⇒ (∀t:S. t ∈ T)

*C DECLARATIONS0      +----------------------------------------------------+
*C DECLARATIONS1      | Constructs for introducing new types / functions    |
*C DECLARATIONS2      +----------------------------------------------------+

*D LetDecl_df         let <name:tok> <expr:expr> == LetDecl{<name>:t} (<expr>)
*A LetDecl            let $name  expr ==  expr
*D LetRecDecl_df      let rec <name:tok> <e:expr> == LetRecDecl{<name>:t}(<e>)
*A LetRecDecl         let rec $name e  ==  r

*D TypeDecl0_df       type <name:tok> = <T:type>== TypeDecl0{<name>:t}(<T>)
*A TypeDecl0          type $name = type ==  type
*D TypeDecl1_df       type '<tp:var> <name:tok> = <T:type>== TypeDecl1{<name>:t}(<tp>.<T>)
*A TypeDecl1          type 'tp $name = T[tp] ==  λtp.T[tp]
```

```
*D TypeDecl2_df       type ('<tp1:var>, '<tp2:var>) <name:tok> = <T:type>
                       == TypeDecl2{<name>:t} (<tp1>,<tp2>.<T>)
*A TypeDecl2          type ('tp1, 'tp2) $name = T[tp1; tp2]
                       == λT2.let <tp1,tp2> = T2 in T[tp1; tp2]
*D TypeDecl3_df       type ('<tp1:var>, '<tp2:var>, '<tp3:var>) <name:tok> = <T:type>)
                       == TypeDecl3{<name>:t} (<tp1>,<tp2>,<tp3>.<T>)
*A TypeDecl3          type ('tp1, 'tp2, 'tp3) $name = T[tp1; tp2; tp3])
                       == λT3.let tp1,tp2,tp3 = T3 in T[tp1; tp2; tp3]
*D TypeDecl4_df       type ('<tp1>, '<tp2>, '<tp3>, '<tp4>) <name:tok> = <T:type>)
                       == TypeDecl4{<name>:t} (<tp1>,<tp2>,<tp3>,<tp4>.<T>)
*A TypeDecl4          type ('tp1, 'tp2, 'tp3, 'tp4) $name = T[tp1; tp2; tp3; tp4])
                       == λT4.let tp1,tp2,tp3,tp4 = T4 in T[tp1; tp2; tp3; tp4]
*D TypeDecl5_df       type ('<tp1>, '<tp2>, '<tp3>, '<tp4>, '<tp5>) <name:tok> = <T:type>
                       == TypeDecl5{<name>:t} (<tp1>,<tp2>,<tp3>,<tp4>,<tp5>.<T>)
*A TypeDecl5          type ('tp1, 'tp2, 'tp3, 'tp4, 'tp5) $name = T[tp1; tp2; tp3; tp4; tp5]
                       == λT5.let tp1,tp2,tp3,tp4,tp5 = T5 in T[tp1; tp2; tp3; tp4; tp5]
*D TypeDecl6_df       type ('<tp1>,'<tp2>,'<tp3>,'<tp4>,'<tp5>,'<tp6>) <name:tok> = <T:type>
                       == TypeDecl6{<name>:t} (<tp1>,<tp2>,<tp3>,<tp4>,<tp5>,<tp6>.<T>)
*A TypeDecl6          type ('tp1,'tp2,'tp3,'tp4,'tp5,'tp6) $name = T[tp1;tp2;tp3;tp4;tp5;tp6]
                       == λT6.let tp1,tp2,tp3,tp4,tp5,tp6 = T6 in T[tp1;tp2;tp3;tp4;tp5;tp6]
*C Matching1          ---------------------------------------------------------
*C Matching2          PATTERN MATCHING
*C Matching3          ---------------------------------------------------------

*D Matching_Union_df  <t1:pattern-term> | <t2:pattern-term>== Matching_Union(<t1>; <t2>)
*A Matching_Union     t1 | t2 ==  t1 + t2
*D Match__Var_df      <x:variable-name>== Match__Var{<x>:t}(<expr>; <t>)
*A Match__Var         $x ==  (true, t expr)
*D Match__All_df      _== Match__All(<expr>; <t>)
*A Match__All         _ ==  (true, t)
*D Match__Alternatives_df     <p1:matcher> | <p2:matcher>
                       == Match__Alternatives(<expr>; <e1>,<t1>.<p1>; <e2>,<t2>.<p2>; <t>)
*A Match__Alternatives        p1[e1; t1] | p2[e2; t2]
                       == let <b',t'> = p1[expr; t] in if b' then (true, t') else p2[expr; t]
*C Matching4          ..........................................................
*C Matching5          Hidden lambda abstractions are necessary to keep the ML style
*C Matching6          ..........................................................
*D Match__HideLambda_df       <t:t>== Match__HideLambda(<x>.<t>)
*A Match__HideLambda          t[x] ==  λx.t[x]
*C Ocaml_data1        =========================================================
*C Ocaml_data2        Basic Ocaml Data Structures
*C Ocaml_data3        =========================================================

*C Ocaml_unit1        ---------------------------------------------------------
*C Ocaml_unit2        UNIT
*C Ocaml_unit4        ---------------------------------------------------------
*D Unit_df                    unit== Unit
*D Null_df                    ()== Ax
*D Unit__Match_All_df         ()== Unit__Match_All(<expr>; <t>)
*A Unit__Match_All            () ==  (true, t)
*C Bool1              ---------------------------------------------------------
*C Bool2              BOOLEANS
*C Bool4              ---------------------------------------------------------
*D O_bool                     bool== 𝔹
*D O_true                     true== tt
*D O_false                    false== ff
*D O_cond                     if <p:bool> then <e1:expr> == if <p> then <e1> else () fi
*T O_cond_wf1                 ⊢ ∀p:bool. ∀A:TYPES. ∀a,b:A. if p then a else b  ∈ A
*D O_and                      p & q==p ∧_b q
*D O_or                       p or q==p ∨_b q
*D O_not_df                   not== O_not
*A O_not                      not ==  λp.¬_b p
*T O_not_wf                   ⊢ not ∈ bool -> bool

*D Bool__Match_True_df        true== Bool__Match_True(<expr>; <t>)
*A Bool__Match_True           true ==  (expr, t)
*D Bool__Match_False_df       false== Bool__Match_False(<expr>; <t>)
*A Bool__Match_False          false ==  (¬_b expr, t)
```

```
*C Int1               ------------------------------------------------------
*C Int2               INTEGERS
*C Int4               ------------------------------------------------------
*D O_int              int== ℤ
*D O_add              <a:int>+<b:int>== <a> + <b>
*D O_pred_df          pred== O_pred
*A O_pred             pred ==  λx.x - 1
*T O_pred_wf          ⊢ pred ∈ int -> int
*D O_sub              <a:int> - <b:int>== <a> - <b>
*D O_minus            -<a:int>== -<a>
*D O_mul              <a:int> * <b:int>== <a> * <b>
*D O_div              <a:int> / <b:int>== <a> ÷ <b>
*D O_mod              <a:int> mod <b:int>== <a> rem <b>

*D O_eq_df            <i:i> = <j:j>== O_eq(<i>; <j>)
*A O_eq               i = j ==  if i=j then true else false
*T O_eq_wf            ⊢ ∀i,j:int.  i = j ∈ bool
*D O_neq_df           <i:i> <> <j:j>== O_neq(<i>; <j>)
*A O_neq              i <> j ==  if i=j then false else true
*T O_neq_wf           ⊢ ∀i,j:int.  i <> j ∈ bool
*D O_lt_df            <i:i> < <j:j>== O_lt(<i>; <j>)
*A O_lt               i < j ==  if i<j then true else false
*T O_lt_wf            ⊢ ∀i,j:int.  i < j ∈ bool
*D O_gt_df            <i:i> > <j:j>== O_gt(<i>; <j>)
*A O_gt               i > j ==  if j<i then true else false
*T O_gt_wf            ⊢ ∀i,j:int.  i > j ∈ bool
*D O_le_df            <i:i> <= <j:j>== O_le(<i>; <j>)
*A O_le               i <= j ==  if j<i then false else true
*T O_le_wf            ⊢ ∀i,j:int.  i <= j ∈ bool
*D O_ge_df            <i:i> >= <j:j>== O_ge(<i>; <j>)
*A O_ge               i >= j ==  if i<j then false else true
*T O_ge_wf            ⊢ ∀i,j:int.  i >= j ∈ bool

*D Int__Match_Num_df <n:number>== Int__Match_Num(<expr>; <n>; <t>)
*A Int__Match_Num    n ==  (expr = n, t)
*C Float1             ------------------------------------------------------
*C Float2             FLOATING POINT NUMBERS
*C Float4             ------------------------------------------------------
*D Float_df           float== Float
*A Float              float ==  int * {0...9} List
*T Float_wf           ⊢ float ∈ TYPES

*D FloatUp_df         <d:digit><dl:dig_list>== FloatUp(<d>; <dl>)
                      <d:digit>== FloatUp(<d>; [])
*A FloatUp            d dl ==  d::dl
*D FloatNum_df        <n:number>.<dl:dig_list>== FloatNum(<n>; <dl>)
                      <n:number>== FloatNum(<n>; [])
*A FloatNum           n.dl ==  (n, dl)
*C String1            ------------------------------------------------------
*C String2            STRINGS
*C String3            ------------------------------------------------------
*D Char_df            char== Char
*A Char               char ==  Atom
*T Char_wf            ⊢ char ∈ TYPES
*D String_df          string== String
*A String             string ==  char List
*T String_wf          string ∈ TYPES

*D StringGet_df       <s:s>.[<idx:idx>]== StringGet(<s>; <idx>)
*A StringGet          s.[idx] ==  if idx<||s|| then if idx<0 then fail else s[idx] else fail
*T StringGet_wf       ⊢ ∀s:string. ∀i:int.  s.[i] ∈ char
*D StringCat_df       <str1:string>^<str2:string>== StringCat(<str1>; <str2>)
*A StringCat          str1^str2 ==  str1 @ str2
*D StringUp_df        <char:char><chars:chars>== StringUp(<char>; <chars>)
                      <char:char>== StringUp(<char>; [])
*A StringUp           charchars ==  char::chars
*D StringExpr_df      "<strg:strg>"== StringExpr(<strg>)
*A StringExpr         "strg" ==  strg
```

```
*C List1             --------------------------------------------------------------
*C List2             MODULE List: list operations (manual section 15.11)
*C List3             --------------------------------------------------------------
*D List_df           list== List
*A List              list ==  λT.T List
*T List_wf           ⊢ list ∈ 1-ary Type Constructors

*D ListNil           []== []
*D ListExpr_step_df  [<hd:head>;<tl:tail>]== ListExpr_step(<hd>; <tl>)
                     [<hd:head>]== ListExpr_step(<hd>; [])
           #Hd a ::  [<hd:head>;<#:tail>]== ListExpr_step(<hd>; <#>)
           #Tl a ::  <hd:head>;<#:tail>== ListExpr_step(<hd>; <#>)
           #Tl a ::  <hd:head>== ListExpr_step(<hd>; [])
*A ListExpr_step     [hd;tl] ==  hd::tl
*T ListExpr_step_wf  ⊢ ∀T:TYPES. ∀a:T. ∀L:T list.  [a;L] ∈ T list
*D ListCons_df       <t:term>::<L:list>== <t>::<L>
*D ListConcat_df     as @ bs==as @ bs

*D List_DOT_length_df length== List_DOT_length
*A List_DOT_length   length ==  λL.||L||
*T List_DOT_length_wf ⊢ ∀T:TYPES. length ∈ T list -> int
*T List_DOT_length_nonneg      ⊢ ∀T:TYPES. ∀L:T list.  0 ≤ length L
*D List_DOT_nth_df   nth== List_DOT_nth
*A List_DOT_nth      nth ==  λL,i.if i<||L|| then if i<0 then fail else L[i] else fail
*T List_DOT_nth_wf   ⊢ ∀T:TYPES. nth ∈ T list -> int -> T
*D List_DOT_iter_df  iter== List_DOT_iter
*A List_DOT_iter     iter == λf,L.rec-case(L) of [] => λs.s  | a::L' => stmnt.f a; stmnt
*T List_DOT_iter_wf  ⊢ ∀S,T:TYPES. ∀f:S -> T. ∀L:S list.  iter f L ∈ T

*D List___Match_Nil_df       []== List___Match_Nil(<expr>; <t>)
*A List___Match_Nil          [] ==  (null(expr), t)
*D List___Match_Cons_df      <p1:matcher-hd>::<p2:matcher-tl>
                              == List___Match_Cons(<expr>;<e1>,<t1>.<p1>;<e2>,<t2>.<p2>;<t>)
*A List___Match_Cons         p1[e1; t1]::p2[e2; t2]
                              == rec-case(expr) of
                                   [] => (false, t)
                                 | e1::e2 => R.let <b1,t1> = p1[e1; t] in
                                                 if b1 then p2[e2; t1] else (false, t1)
*C Array1            --------------------------------------------------------------
*C Array2            MODULE Array: array operations (manual section 15.2)
*C Array3            --------------------------------------------------------------
*D Array_df          array== Array
*A Array             array ==  λT.lg:ℕ × int -> T
*T Array_wf          ⊢ array ∈ 1-ary Type Constructors

*D ArrayExpr_null_df  == ArrayExpr_null(<idx>)
*A ArrayExpr_null     ==  any idx
*T ArrayExpr_null_wf  ⊢ ∀T:TYPES. (0, λi.) ∈ T array
*D ArrayExpr_step_df  <cont:cont>;<val:val>== ArrayExpr_step(<idx>; <pos>; <val>; <cont>)
            #Hd a :: <#:cont>;<val:val>  == ArrayExpr_step(<idx>; <pos>; <val>; <#>)
            #Tl a :: <cont:cont><val:val>== ArrayExpr_step(<idx>; <pos>; <val>; <cont>)
            #Tl a :: <#:cont>;<val:val>  == ArrayExpr_step(<idx>; <pos>; <val>; <#>)
*A ArrayExpr_step     cont;val ==  if idx=pos then val else cont
*D ArrayExpr_df       [|<cont:cont>|]== ArrayExpr(<size>; <idx>.<cont>)
*A ArrayExpr          [|cont[idx]|] ==  (size, λidx.cont[idx])

*D Array_DOT_length_df        length== Array_DOT_length
*A Array_DOT_length           length ==  λa.let <lg,values> = a in lg
*T Array_DOT_length_wf        ⊢ ∀T:TYPES. length ∈ T array -> int
*D Array_DOT_get_df   <a:array>.(<i:index>)== Array_DOT_get(<a>; <i>)
*A Array_DOT_get      a.(i) ==  let <lg,values> = a in values i
*T Array_DOT_get_wf   ⊢ ∀T:TYPES. ∀a:T array. ∀i:int.  a.(i) ∈ T
*T Array_DOT_get_wf1  ⊢ ∀T:TYPES. ∀a:T array. ∀i:{0..(length a)^-}.  a.(i) ∈ T
*D ArraySubst_df      <a:array>[.(<i:index>)←<expr:expr>]== ArraySubst(<a>; <i>; <expr>)
*A ArraySubst         a[.(i)←expr] == let <lg,values> = a  in
                                          (lg, λidx.if idx=i then expr else a.(idx))
*T ArraySubst_wf      ⊢ ∀T:TYPES. ∀a:T array. ∀i:{0..(length a)^-}. ∀expr:T.
                           a[.(i)←expr] ∈ T array
```

```
*D Array_DOT_set_df   <a:var>.(<i:index>) <- <expr:expr>== Array_DOT_set(<i>; <a>.<expr>)
*A Array_DOT_set      a.(i) <- expr[a] ==  λa.a[.(i)←expr[a]]
*D Array_set_matrixfield_df   <a:var>.(<i:index>).(<j:index>) <- <expr:expr>
                               == Array_set_matrixfield(<i>; <j>; <a>.<expr>)
*A Array_set_matrixfield       a.(i).(j) <- expr[a] ==  λa.a[.(i)←a.(i)[.(j)←expr[a]]]
*D Array_DOT_create_df        create== Array_DOT_create
*A Array_DOT_create           create == λlg,x. if 0<lg then (lg, λi.if i>0 & (lg-1)>i then x)
                                                   else (0, λi.x)
*T Array_DOT_create_wf        ⊢ ∀T:TYPES. create ∈ int -> T -> T array
*D Array_DOT_create_matrix_df create_matrix== Array_DOT_create_matrix
*A Array_DOT_create_matrix    create_matrix ==  λdimx,dimy,e.create dimx (create dimy e)
*T Array_DOT_create_matrix_wf ⊢ ∀T:TYPES. create_matrix ∈ int -> int -> T -> T array array
*D Array_DOT_copy_df          copy== Array_DOT_copy
*A Array_DOT_copy             copy ==  λa.a
*T Array_DOT_copy_wf          ⊢ ∀T:TYPES. copy ∈ T array -> T array
*D Array_DOT_to_list_df       to_list== Array_DOT_to_list
*A Array_DOT_to_list          to_list == λa.I(length a - 1) where I(α) =
                                          when i = α < 0,  L_to_i = I(α+1). []
                                          when α = 0.  a.(0)::[]
                                          when i = α > 0,  L_to_i = I(α-1).
                                            if i<(length a) then L_to_i @ [a.(i)] else L_to_i
                                         end where
*T Array_DOT_to_list_wf       ⊢ ∀T:TYPES. to_list ∈ T array -> T list
*D Array_DOT_of_list_df       of_list== Array_DOT_of_list
*A Array_DOT_of_list          of_list ==  λL.(length L, λidx.nth L idx)
*T Array_DOT_of_list_wf       ⊢ ∀T:TYPES. of_list ∈ T list -> T array

*D ValueGet_df                !<VAR:imperative_VAR>== ValueGet(<VAR>)
*A ValueGet                   !VAR ==  VAR
*D ValueSet_df                <cell:var> := <expr:expr>== ValueSet(<cell>.<expr>)
*A ValueSet                   cell := expr[cell] ==  λcell.expr[cell]
                      -------------------------------------------------------------
*C Product2           PRODUCTS
*C Product3           -------------------------------------------------------------
*D Product_df         <A:type> * <B:type>== <A> × <B>
*D Pair_df            (<A:term>, <B:term>) == <<A>, <B>>
            #Hd R :: (<A:term>, <#:term>)== <<A>, <#>>
            #Tl R :: <A:term>, <B:term>== <<A>, <B>>
            #Tl R :: <A:term>, <#:term>== <<A>, <#>>
*D Product__Match_Pair_df   <p1:matcher1>, <p2:matcher2>
                       == Product__Match_Pair(<expr>; <e1>,<t1>.<p1>; <e2>,<t2>.<p2>; <t>)
*A Product__Match_Pair      p1[e1; t1], p2[e2; t2]
                              == let <e1,e2> = expr in
                                   let <b1,t1> = p1[e1; t]  in
                                    if b1  then p2[e2; t1] else (false, t1)

*C Function1          -------------------------------------------------------------
*C Function2          FUNCTIONS AND BINDINGS
*C Function3          -------------------------------------------------------------
*D Fun                <S:type> -> <T:type>== <S> → <T>
*D FunArgument_df     <p:matcher> <target:term>
                       == FunArgument(<e>,<tVar>.<p>; <target>; <arg>.<expr>)
*A FunArgument        p[e; tVar] target ==  λarg.let <b,t> = p[expr[arg]; target] in t
*D FunBody_df          -> <expr:expr>== FunBody(<expr>)
*A FunBody              -> expr ==  expr
*D FunBinding_df      fun <funexpr:funexpr>== FunBinding(<funexpr>)
*A FunBinding         fun funexpr ==  funexpr
*D FunctionBinding_df function <matchcase:matchcase>== FunctionBinding(<e>.<matchcase>)
*A FunctionBinding    function matchcase[e] ==  λe.matchcase[e]
*D FunApply_df        <F:term:E> <A:term:L> == <F> <A>

*D MatchCase_Last_df  <p:matchcase>  ->  <target:target>
                       == MatchCase_Last(<expr>; <target>; <e>,<tVar>.<p>)
*A MatchCase_Last     p[e; tVar]  ->  target ==  let <b,t> = p[expr; target] in t
*D MatchCase_Step_df  <p:matchcase>  -> <target:target>  |  <cont:continuation>
                       == MatchCase_Step(<expr>; <target>; <e>,<t>.<p>; <e'>.<cont>)
*A MatchCase_Step     p[e; tVar]  ->  target  |  cont[e']
                       == let <b,t> = p[expr; target] in if b  then t else cont[expr]
```

```
*D MatchWith_df       match <expr:expr> with <matchcase:matchcase>
                       == MatchWith(<expr>; <e>.<matchcase>)
*A MatchWith          match expr with  matchcase[e] ==   matchcase[expr]
*D LetBody_df         = <expr:expr>== LetBody(<expr>)
*A LetBody            = expr ==  expr
*D LetPattern_df      let <p:matcher> = <expr:expr>in  <target:target>
                       == LetPattern(<expr>; <target>; <e>,<tVar>.<p>)
*A LetPattern         let p[e; tVar]  = expr in  target == let <b,t> = p[expr; target] in t
*D LetBindFunction_df let <f:var> <f_val:term>in  <target:target>
                       == LetBindFunction(<f>.<target>; <f_val>)
*A LetBindFunction    let f f_val in  target[f] == let f = f_val in target[f])
*D LetBindRecursive_df        let rec <f:var> <f_val:term> in <target:target>
                               == LetBindRecursive(<f>.<target>; <f>.<f_val>)
*A LetBindRecursive           let rec f f_val[f] in target[f] ==
                               = let f = Y (λf.f_val[f]) in target[f])

*C Ocaml_Types1       ============================================================
*C Ocaml_Types2       COMPLEX TYPE CONSTRUCTORS
*C Ocaml_Types8       ============================================================

*D FIELDS_df          FIELDS== FIELDS
*A FIELDS             FIELDS ==  Atom
*T FIELDS_wf          ⊢ FIELDS ∈ TYPES
*D Token_df           <a:token>== "<a>"
*D FieldEq_df         <f1:field> = <f2:field>== FieldEq(<f1>; <f2>)
*A FieldEq            f1 = f2 ==  if f1=f2 then  true else false
*T FieldEq_wf         ⊢ ∀f1,f2:FIELDS.  f1 = f2 ∈ bool
*C Record1            -----------------------------------------------------------
*C Record2            RECORDS
*C Record3            -----------------------------------------------------------
*D RecordDecl_df      RECORDdecl== RecordDecl
*A RecordDecl         RECORDdecl ==  FIELDS -> TYPES
*T RecordDecl_wf      ⊢ RECORDdecl ∈ U2
*D RecordType_Null_df   == RecordType_Null
*A RecordType_Null      == unit
*T RecordType_Null_wf ⊢   ∈ TYPES
*D RecordType_Step_df {<lab:fieldname> :<T:Type> <cont:continuation>}
                       == RecordType_Step(<l>; <lab>; <T>; <cont>)
          #Hd a :: { <lab:fieldname> : <T:Type>; <#:continuation>}
                       == RecordType_Step(<l>; <lab>; <T>; <#>)
          #Tl a :: <lab:fieldname> : <T:Type> <cont:continuation>
                       == RecordType_Step(<l>; <lab>; <T>; <cont>)
          #Tl a :: <lab:fieldname> : <T:Type>; <#:continuation>
                        == RecordType_Step(<l>; <lab>; <T>; <#>)
*A RecordType_Step    {lab :T cont}  ==  if l=lab then  T else cont
*T RecordType_Step_wf ⊢ ∀l,lab:FIELDS. ∀T,cont:TYPES.  l {lab :T cont}  ∈ TYPES
*D RECORDS_df         RECORDS== RECORDS(<TYPE-OF>)
*A RECORDS            RECORDS ==  field:FIELDS → TYPE-OF field
*T RECORDS_wf         ⊢ ∀TYPE-OF:RECORDdecl. RECORDS ∈ TYPES
*D RecordType_df      <TypeCases:TypeCases>== RecordType(<l>.<TypeCases>)
*A RecordType         TypeCases[l] ==  RECORDS
*T RecordType_wf      ⊢ ∀TYPE-OF:RECORDdecl. ∀name:Atom.  TYPE-OF l ∈ TYPES
*T RecordType_wf1     ⊢ ∀TYPE-OF:RECORDdecl. ∀name:Atom. ∀s:TYPE-OF l. ∀field:FIELDS.
                         s field ∈ TYPE-OF field

*D RecordExpr_Null_df  == RecordExpr_Null
*A RecordExpr_Null     ==  ()
*A RecordExpr_Step_df  <lab:lab> = <val:val><cont:cont>
                        == RecordExpr_Step(<l>; <lab>; <val>; <cont>)
           #Hd a :: <lab:lab> = <val:val>;<#:cont>
                        == RecordExpr_Step(<l>; <lab>; <val>; <#>)
           #Tl a :: <lab:lab> = <val:val><cont:cont>
                        == RecordExpr_Step(<l>; <lab>; <val>; <cont>)
           #Tl a :: <lab:lab> = <val:val>;<#:cont>
                        == RecordExpr_Step(<l>; <lab>; <val>; <#>)
*A RecordExpr_Step     lab = val cont ==  if l=lab then  val else cont
*D RecordExpr_df       {<cont:cont>}== RecordExpr(<l>.<cont>)
*A RecordExpr          {cont[l]} ==  λl.cont[l]
```

```
*D RecordGet_df        <r:record>.<f:field>== RecordGet(<r>; <f>)
*A RecordGet           s.f ==  s f
*T RecordGet_wf        ⊢ ∀TYPE-OF:RECORDdecl. ∀r:RECORDS. ∀f:FIELDS.  r.f ∈ TYPE-OF f
*D RecordSubst_df      <r:r>[.<f:field>←<expr:expr>]== RecordSubst(<r>; <f>; <expr>)
*A RecordSubst         r[.f←expr] ==  λfVar.if fVar=f then  expr else r.fVar
*T RecordSubst_wf      ⊢ ∀TYPE-OF:RECORDdecl. ∀r:RECORDS. ∀f:FIELDS. ∀expr:TYPE-OF f.
                         r[.f←expr] ∈ RECORDS
*D RecordSet_df        <s:statevar>.<f:field> <- <expr:expr>== RecordSet(<f>; <s>.<expr>)
*A RecordSet           s.f <- expr[s] ==  λs.s[.f←expr[s]]
*T RecordSet_wf        ⊢ ∀TYPE-OF:STATEdecl. ∀f:FIELDS. ∀expr:TYPE-OF f.
                         s.f <- expr ∈ STATEMENTS
*D RecordSet_arrayfield_df     <s:var>.<f:arrayfield>.(<idx:idx>) <- <expr:expr>
                                == RecordSet_arrayfield(<f>; <idx>; <s>.<expr>)
*A RecordSet_arrayfield        s.f.(idx) <- expr[s] ==  λs.s[.f←s.f[.(idx)←expr[s]]]
*D RecordSet_matrixfield_df    <s:var>.<f:matrixfield>.(<i:idx>).(<j:idx>) <- <expr:expr>
                                == RecordSet_matrixfield(<f>; <i>; <j>; <s>.<expr>)
*A RecordSet_matrixfield       s.f.(i).(j) <- expr[s]
                                == λs.s[.f←s.f[.(i)←s.f.(i)[.(j)←expr[s]]]]

*D Record__Match_Null_df      == Record__Match_Null(<expr>; <t>)
*A Record__Match_Null          ==  (true, t)
*D Record__Match_Step_df      {<f:field> = <p1:matcher-field><p2:matcher-cont>}
                                == Record__Match_Step(<expr>; <f>; <e1>,<t1>.<p1>;
                                                                   <e2>,<t2>.<p2>; <t>)
                    #Hd a :: { <f:field> = <p1:matcher-field>; <#:matcher-cont> }
                                == Record__Match_Step(<expr>; <f>; <e1>,<t1>.<p1>;
                                                                   <e2>,<t2>.<#>; <t>)
                    #Tl a :: <f:field> = <p1:matcher-field> <p2:matcher-cont>
                                == Record__Match_Step(<expr>; <f>; <e1>,<t1>.<p1>;
                                                                   <e2>,<t2>.<p2>; <t>)
                    #Tl a :: <f:field> = <p1:matcher-field>; <#:matcher-cont>
                                == Record__Match_Step(<expr>; <f>; <e1>,<t1>.<p1>;
                                                                   <e2>,<t2>.<#>; <t>)
*A Record__Match_Step         {f = p1[e1; t1]p2[e2; t2] }
                                == let <b1,t1> = p1[expr.f; t] in
                                  if b1  then p2[expr; t1] else (false, t1)
*C Variant1            -----------------------------------------------------------
*C Variant2            VARIANT RECORDS
*C Variant3            -----------------------------------------------------------
*D VariantDecl_df      VARIANTdecl== VariantDecl
*A VariantDecl         VARIANTdecl ==  FIELDS -> TYPES
*T VariantDecl_wf      ⊢ VARIANTdecl ∈ U2
*D VariantType_Null_df == VariantType_Null
*A VariantType_Null     ==  Void
*D VariantType_Step_df <c:constructor> of <T:Type> <cont:continuation>
                        == VariantType_Step(<l>; <c>; <T>; <cont>)
            #Hd a :: <c:constructor> of <T:Type>| <#:continuation>
                        == VariantType_Step(<l>; <c>; <T>; <#>)
            #Tl a :: <c:constructor> of <T:Type> <cont:continuation>
                        == VariantType_Step(<l>; <c>; <T>; <cont>)
            #Tl a :: <c:constructor> of <T:Type>| <#:continuation>
                        == VariantType_Step(<l>; <c>; <T>; <#>)
*A VariantType_Step    c of T cont  ==  if l=c then  T else cont
*D VariantType_StepPure_df     <c:constructor> <cont:continuation> == <l>  <c> of unit <cont>
                    #Hd a :: <c:constructor> | <#:continuation> == <l>  <c> of unit <#>
                    #Tl a :: <c:constructor> <cont:continuation> == <l>  <c> of unit <cont>
                    #Tl a :: <c:constructor> | <#:continuation> == <l>  <c> of unit <#>
*D VARIANTS_df          VARIANTS== VARIANTS(<TYPE_OF>)
*A VARIANTS             VARIANTS == f:FIELDS × TYPE-OF f
*T VARIANTS_wf          ⊢ ∀TYPE-OF:VARIANTdecl. VARIANTS ∈ TYPES
*T VARIANTS_wf1         ⊢ ∀TYPE-OF:VARIANTdecl. ∀s:VARIANTS. ∀f:FIELDS. s f ∈ TYPE-OF f
*D VariantType_df      <TypeCases:TypeCases>== VariantType(<l>.<TypeCases>)
*A VariantType         TypeCases[l] ==  VARIANTS

*D VariantConstr_df    <c:constructor>(<expr:term>) == VariantConstr(<c>; <expr>)
*A VariantConstr       c(expr) ==  (c, expr)
*D Variant_Cconstr     <c:constructor>== <c>(())
*T VariantConstr_wf    ⊢ ∀TYPE-OF:VARIANTdecl. ∀c:FIELDS. ∀obj:TYPE-OF c. c(obj) ∈ VARIANTS
```

```
*D Variant__let_constr_df     let <c:constructor>(<arg:var>) = <expr:expr> in <t:expr>
                               == Variant__let_constr(<expr>; <c>,<arg>.<t>)
*D Variant__let_Cconstr       let <c:constructor> = <expr:expr> in <t:expr>
                               == let <c>(nullvar) = <expr> in <t>
*A Variant__let_constr        let c(arg) = expr in t[c; arg]
                               == let <c,arg> = expr in t[c; arg]
*T Variant__let_constr_wf     ⊢ ∀TYPE-OF:RECORDdecl. ∀vrec:VARIANTS.
                                 let c(arg) = vrec in arg ∈ TYPE-OF c
*D Variant__Match_Nconstr_df  <c:constructor>(<p:matcher>)
                               == Variant__Match_Nconstr(<expr>; <c>; <e1>,<t1>.<p>; <t>)
*A Variant__Match_Nconstr     constr(p[e1; t1]) ==
                               let c(arg) = expr in
                                if c = constr then p[arg; t]  else (false, t)
*D Variant__Match_Cconstr_df  <c:constructor> == Variant__Match_Cconstr(<expr>; <c>; <t>)
*A Variant__Match_Cconstr     constr == let c = expr in (c = constr, t)

*C Basics_begin       **-------------------------------------------------------
*C Basics_begi_       **  Module Basics: Miscellaneous predefined expressions
*C Basics_beg__       **-------------------------------------------------------
*D O_ref_df           ref== O_ref
*A O_ref              ref ==  Abstractly defined
*T O_ref_wf           ⊢ ref ∈ 1-ary Type Constructors
*D O_failwith_df      failwith== O_failwith
*A O_failwith         failwith ==  Abstractly defined
*T O_failwith_wf      ⊢ ∀T:TYPES. failwith ∈ string -> T
*D O_option_df        option== O_option
*A O_option           option ==  type 'a option = None of unit | Some of a
*T O_option_wf        ⊢ option ∈ 1-ary Type Constructors
*D Obj_DOT_Type_t_df  t== Obj_DOT_t
*A Obj_DOT_Type_t     t ==  Abstractly defined
*T Obj_DOT_Type_t_wf  ⊢ t ∈ TYPES
*D Obj_DOT_is_block_df is_block== Obj_DOT_is_block
*A Obj_DOT_is_block   is_block ==  ()
*D Obj_DOT_repr_df    repr== Obj_DOT_repr
*A Obj_DOT_repr       repr ==  ()
*D Obj_DOT_magic_df   magic== Obj_DOT_magic
*A Obj_DOT_magic      magic ==  ()
*C Basics____         **-------------------------------------------------------
*C Basics___d         **   END of Module Basics
*C Basics__nd         **-------------------------------------------------------


*C Ocaml_state1       ---------------------------------------------------------
*C Ocaml_state2       STATES and STATE PROPERTIES
*C Ocaml_state3       ---------------------------------------------------------
*D O_sDecl_df         STATEdecl== O_sDecl
*A O_sDecl            STATEdecl ==  RECORDdecl
*T O_sDecl_wf         ⊢ STATEdecl ∈ 𝕌_2
*D O_sDecl_all        <prop:prop>== ∀TYPE-OF:STATEdecl. <prop>
*D O_states_df        STATES== O_STATES(<TYPE-OF>)
*A O_states           STATES ==  f:FIELDS → TYPE-OF f
*T O_states_wf        ⊢ ∀TYPE-OF:STATEdecl. STATES ∈ TYPES
*T O_states_wf1       ⊢ ∀TYPE-OF:STATEdecl. ∀s:STATES. ∀f:FIELDS.  s f ∈ TYPE-OF f
*D O_sProps_df        ℙ[s]== O_sProps(<TYPE-OF>)
*A O_sProps           ℙ[s] == STATES -> ℙ_1
*C Ocaml_stmtA        ---------------------------------------------------------
*C Ocaml_stmtB        STATEMENTS operate on states only
*C Ocaml_stmtC        ---------------------------------------------------------

*D O_stmt_df          STATEMENTS== O_STMT(<TYPE-OF>)
*A O_stmt             STATEMENTS ==  STATES -> STATES
*T O_stmt_wf          ⊢ ∀TYPE-OF:STATEdecl. STATEMENTS ∈ TYPES
*T O_stmt_wf1         ⊢ ∀TYPE-OF:STATEdecl. ∀s:STATES. ∀cmd:STATEMENTS.  cmd s ∈ STATES
*D O_Assert_df        {{<pre:pre>}} <stmt:stmt> {{<post:post>}}
                       == O_Assert(<pre>; <stmt>; <post>; <TYPE-OF>)
*A O_Assert           {{pre}} stmt {{post}} ==  ∀s:STATES. pre s ⇒ post (stmt s)
*T O_Assert_wf        ⊢ ∀TYPE-OF:STATEdecl. ∀stmt:STATEMENTS. ∀pre,post:ℙ[s].
                         {{pre}} stmt {{post}} ∈ ℙ_1
```

```
*C Ocaml_stmta          ==========================================================
*C Ocaml_stmtb          Basic Ocaml statements
*C Ocaml_stmtc          ==========================================================

*D O_begin_df           begin <stmts:stmts>end== O_begin(<stmts>)
*A O_begin              begin stmts end ==  stmts

*C O_assign             ------- Assignments only to states (see module Record) ----

*D O_compound_df        <stmt1:statement>;<stmt2:statement>== O_compound(<stmt1>; <stmt2>)
*A O_compound           stmt1; stmt2 ==  λs.stmt2 (stmt1 s)
*T O_compound_wf        ⊢ ∀TYPE-OF:STATEdecl. ∀stmt1,stmt2:STATEMENTS.
                           stmt1;stmt2 ∈ STATEMENTS
#T O_compound_vf        ⊢ ∀TYPE-OF:STATEdecl. ∀stmt1,stmt2:STATEMENTS. ∀pre,post,post1:ℙ[s].
                             {{pre}} stmt1 {{post1}}  ∧  {{post1}} stmt2 {{post}}
                               ⇒  {{pre}} stmt1; stmt2 {{post}}

*C O_cond1              ---------- Conditional (see Bool) --------------------------
*T O_cond_wf            ⊢ ∀TYPE-OF:STATEdecl. ∀stmt1,stmt2:STATEMENTS. ∀cond:bool.
                           if cond  then stmt1 else stmt2  ∈ STATEMENTS

*C O_match1             ---------- Case Expressions (see Functions and Bindings) ---

*D O_for_df             FOR <i:var>=<e1:int> to <e2:int> do  <stmt:stmt> done
                         == O_for(<e1>; <e2>; <i>.<stmt>)
*A O_for                FOR i=e1 to e2 do   stmt[i] done
                         == I(e2 - e1) where I(α) =
                              when n = α < 0,  before = I(α+1).  Id
                              when α = 0.  stmt[e1]
                              when  n = α > 0,  before = I(α-1).  before; stmt[e1+n]
                            end where
*T O_for_wf             ⊢ ∀TYPE-OF:STATEdecl. ∀stmt:STATEMENTS. ∀e1,e2:int.
                            FOR i=e1 to e2 do   stmt done ∈ STATEMENTS
*D O_fordown_df         FOR <i:var>=<e1:e1> DOWNTO <e2:e2> DO <stmt:stmt> DONE
                         == O_fordown(<e1>; <e2>; <i>.<stmt>)
*A O_fordown            FOR i=e1 DOWNTO e2 DO stmt[i] DONE
                         == I(e2 - e1) where I(α) =
                              when n = α < 0,  before = I(α+1).  before; stmt[e1+n]
                              when α = 0.  stmt[e1]
                              when  n = α > 0,  before = I(α-1).  Id
                            end where
*T O_fordown_wf         ⊢ ∀TYPE-OF:STATEdecl. ∀stmt:STATEMENTS. ∀e1,e2:int.
                           FOR i=e1 DOWNTO e2 DO stmt DONE ∈ STATEMENTS

*D O_while_df           WHILE <cond:bool> DO <stmt:stmt> DONE== O_while(<cond>; <stmt>)
*A O_while              WHILE cond DO stmt DONE
                         ==  Y (λwhile.if cond  then stmt; while else Id )
*C Ocaml____            **********************************************************
*C Ocaml___d            **                    END of theory OCAML               **
*C Ocaml__nd            **********************************************************
*C Ocaml_end            **
```

This article was processed using the LaTeX macro package with LLNCS style