Building Reliable, High-Performance Networks with the Nuprl Proof Development System

Christoph Kreitz
Department of Computer Science, Cornell-University
Ithaca, NY 14853-7501, USA
(*e-mail:* `kreitz@cs.cornell.edu`)

Abstract

Proof systems for expressive type theories provide a foundation for the verification and synthesis of programs. But despite their successful application to numerous programming problems there remains an issue with scalability. Are proof environments capable of reasoning about large software systems? Can the support they offer be useful in practice?

In this article we answer this question by showing how the NUPRL proof development system and its rich type theory have contributed to the design of reliable, high-performance networks by synthesizing optimized code for application configurations of the ENSEMBLE group communication toolkit. We present a *type-theoretical semantics* of OCAML, the implementation language of ENSEMBLE, and tools for automatically importing system code into the NUPRL system. We describe reasoning strategies for generating verifiably correct *fast-path optimizations* of application configurations that substantially reduce end-to-end latency in ENSEMBLE. We also discuss briefly how to use NUPRL for checking configurations against specifications and for the design of reliable adaptive network protocols.

1 Introduction

Advanced type systems have greatly increased our ability to produce reliable software. In programming languages, type checking, certifying compilers, and extended static checking (Schneider *et al.*, 2000; Leino, 2000) help detecting subtle errors at compile time. In theorem proving, type theories provide the logical foundation for proving programs correct and synthesizing algorithms from formal specifications. Numerous proof assistants have been built for these expressive formalisms and been used successfully in a variety of applications in mathematics and programming. Some of the most prominent of these systems are ALF (Altenkirch *et al.*, 1994; ALFA), COQ (Dowek & et. al, 1991; Coq), HOL (Gordon & Melham, 1993; HOL), ISABELLE (Paulson, 1990; Isabelle), LEGO (Pollack, 1994), NUPRL (Constable *et al.*, 1986; Allen *et al.*, 2000; Nuprl), PVS (Owre *et al.*, 1996; PVS), TPS (Andrews *et al.*, 1996; TPS), and TWELF (Pfenning & Schürmann, 1999). Most applications, however, have dealt only with theoretical algorithms or idealizations of real systems. It is not clear whether the proof methods used in them scale well enough to offer practically useful support for software design.

In this article we will demonstrate that formal logical methods based on expressive type systems are capable of supporting the formal design and implementation

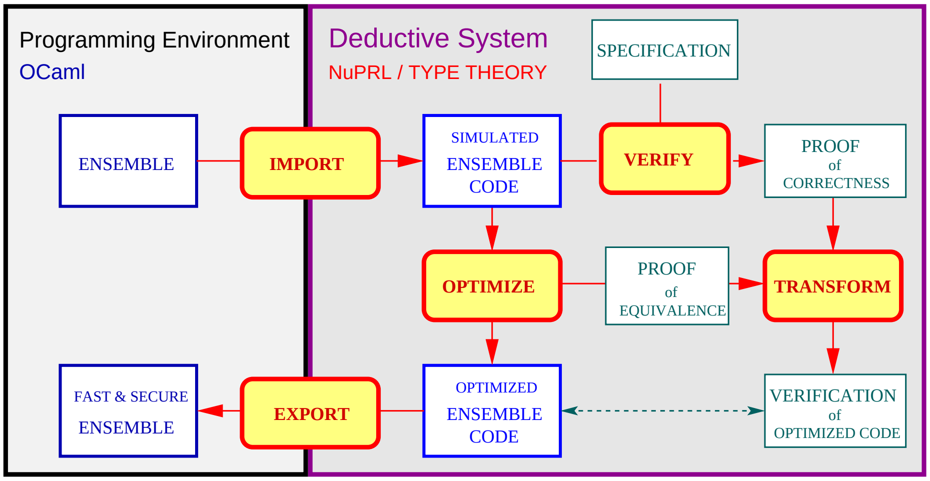


Fig. 1. Linking Ensemble and Nuprl

of large-scale, high-performance network systems. In particular we will show that linking the Ensemble group communication toolkit (Hayden, 1998; Birman *et al.*, 2000) to the Nuprl proof development system (Constable *et al.*, 1986; Allen *et al.*, 2000) provides an infrastructure for the application of logical inference techniques to a real-world system. We call this infrastructure a *logical programming environment* for Ensemble (Kreitz *et al.*, 1998). Within the logical programming environment we have developed logical optimization tools that can substantially increase the performance of Ensemble and as well as mechanisms that support the verification and formal design of Ensemble protocols.

Figure 1 illustrates the methodology of our approach. We link Ensemble and Nuprl by developing tools for importing Ensemble's system code into the Nuprl proof development system and vice versa. These tools convert Ensemble code into terms of Nuprl's logical language and are based on a type-theoretical semantics of Ensemble's implementation language OCaml.

Within Nuprl we then optimize the (represented) code of application configurations of Ensemble by applying semantics-preserving logical transformations and export the result back into the OCaml programming environment. In addition to that we provide a formal proof that the generated code, while being significantly more efficient, has in fact the same functionality as the original one. We can also apply formal reasoning strategies to verify Ensemble protocols and use formal techniques to support the design new communication protocols for Ensemble.

Obviously, our approach is limited to distributed systems with a simple and well-defined design structure. Importing and exporting system code into a formal language like type theory requires the system's implementation language to have a precise mathematical semantics. For formal optimization and verification to become feasible, the system must have components with precisely specified interfaces and well-defined mechanisms for composing them. Fortunately, the Ensemble system and its implementation language OCaml satisfy these requirements.

In the following section we will give a brief account of NUPRL and ENSEMBLE. Section 3 will describe the representation of OCAML programs in NUPRL's type theory as well as the tools for importing and exporting system code. Formal optimization is presented in Section 4, while Section 5 describes research on the verification and formal design of ENSEMBLE protocols. Section 6 addresses related work. We conclude by discussing insights gained from our research as well as future work.

2 Preliminaries

2.1 Nuprl

The NUPRL proof development system (Constable *et al.*, 1986) is a framework for the development of formal mathematical knowledge as well as for the synthesis, verification, and optimization of software.

NUPRL's logical language, summarized in Table 1, is a significant extension of Martin-Löf's *intuitionistic Type Theory* (Martin-Löf, 1984; Constable, 1998) that includes formalizations of fundamental mathematical concepts, an expressive data system, and a functional programming language similar to the core of ML. Each

	Type	*Members and associated noncanonical expressions*
Function Space	$S \to T$, $x{:}S \to T$	$\lambda x.t$, $f\ t$
Product Space	$S \times T$, $x{:}S \times T$	`<`s`,`t`>`, `let <`x`,`y`> =` e `in` u
Disjoint Union	$S+T$	`inl(`s`)`, `inr(`t`)`,
		`case` e `of inl(`x`)` $\mapsto u$ `|` `inr(`y`)` $\mapsto v$
Universes	$\mathbb{U}_j$	— *types of level* j —
Equality Type	$s = t \in T$	`Ax`
Empty Type	`Void`	— *no members* — `any(`x`)`
Atoms	`Atom`	`"`*token*`"`, `if "`a`"="`b`" then` s `else` t
Numbers	$\mathbb{Z}$	`0, 1, -1, 2, -2,...`
		`rec-case` i `of` x`<0` $\mapsto$ `[`f_x`].`s `|` `0` $\mapsto b$ `|` y`>0` $\mapsto$ `[`f_y`].`t
		s`+`t, s`-`t, s`*`t, $s \div t$, s `rem` t,
		`if` i`=`j `then` s `else` t, `if` i`<`j `then` s `else` t
	i`<`j	`Ax`
Lists	S `list`	`[]`, t`::`$list$,
		`rec-case` L `of [] ` $\mapsto b$ `|` x`::`$l \mapsto$ `[`f_l`].`t
Inductive Types	`rectype` $X = T[X]$	— *members defined by unrolling* $T[X]$ —
		`let`* $f(x) = t$ `in` $f(e)$,
Subset	$\{x{:}S \mid P[x]\}$,	— *some members of* S —
Intersection	$\cap x{:}S.T[x]$,	— *members that occur in all* $T[x]$ —
	$x{:}S \cap T[x]$	— *members* x *that occur* S *and* $T[x]$ —
Union	$\cup x{:}S.T[x]$	— *members that occur in some* $T[x]$*, tricky equality*—
Quotient	$x,y : S//E[x,y]$	— *members of* S*, new equality* —
Very Dependent Functions		— *functions whose range types depend on the values*
	$\{f \mid x{:}S \to T[f,x]\}$	*of their inputs and of the functions themselves* —

Table 1. Expressions of NUPRL's Type Theory

Natural numbers	$\mathbb{N}$	$\equiv$ {i:$\mathbb{Z}$|0≤i}
Logical connectives	$\forall\ \exists\ \wedge\ \vee\ \Rightarrow\ \neg$ True False	— *Curry-Howard isomorphism* —
Singleton type	Unit, ()	$\equiv$ 0=0 $\in\mathbb{Z}$, Ax
Top type	Top	$\equiv$ $\cap$x:Void.Void
Booleans	$\mathbb{B}$, tt, ff	$\equiv$ Unit + Unit, inl(()), inr(())
Boolean conditional	if b then s else t	$\equiv$ case b of inl(_) $\mapsto s$ | inr(_) $\mapsto t$
Y combinator	Y	$\equiv$ λf. (λx.f (x x)) (λx.f (x x))
List operations	hd(l), tl(l), l_1@l_2, length(l), map(f;l), rev(l), l[i], l[i..j^-]	

Table 2. Important user-defined Nuprl types and expressions

type comes with notions of (lazy) evaluation and *extensional* equality, which are essential for defining the semantics of expressions. Nuprl's type theory is *open-ended* in the sense that new types may be added to the theory if needed. Recent additions for reasoning about classes and objects include very dependent function types, dependent intersection and records, and a union type (Hickey, 1996; Kopylov, 2000; Constable & Hickey, 2000; Hickey, 2001).

As Nuprl expressions are defined independently of their types, Y combinators and similar constructs may be used within Nuprl terms, which makes it possible to represent all computable functions in type theory. In the course of a formal argument, however, expressions have to be proven to belong to some type. Nuprl proofs are developed in a *top down sequent calculus*: proof goals are refined into subgoals by application of inference rules until they can be handled by axioms or already proven lemmata. For each type there is a a collection of inference rules for reasoning about the formation and equality of types, members, and associated noncanonical expressions, for reasoning about the use of variables of a type, and for reasoning about computation. The latter are crucial for reasoning about values of expressions and for proving properties of expressions that contain non-typeable subexpressions like the Y combinator.

The Nuprl system (Constable *et al.*, 1986; Allen *et al.*, 2000; Nuprl) supports an interactive development of formal mathematical theories and program verifications. It provides a highly visual *proof editor*, a *tactic* mechanism for the development of proof strategies through programmed application of inference rules, *decision procedures* for standard arithmetic and equality reasoning, mechanisms for *extracting programs* from proofs and *evaluating programs*, and an extendable *library* of verified knowledge from various domains. Furthermore, users may use *definitional abstractions* to extend the formal language of type theory in a conservative way (see Table 2 for important user-defined concepts) and *display forms* to customize the outer appearance of terms without changing their internal structure.

The system has been used in increasingly large applications in mathematics and programming, such as constructive versions of Girard's paradox (Howe, 1987), Higman's lemma (Murthy & Russell, 1990), abstract algebra (Jackson, 1994), verifications of a logic synthesis tool (Aagaard & Leeser, 1993) and of the SCI cache coherency protocol (Howe, 1996), and in our current work on communication systems (Kreitz *et al.*, 1998; Hickey *et al.*, 1999; Liu *et al.*, 1999; Bickford *et al.*, 2001c).

In its newest release (Allen *et al.*, 2000), NUPRL features an open, distributed architecture that is organized as a collection of independent communicating processes centered around a persistent knowledge base (or *library*). This enables users to connect external proof tools to NUPRL and to use them simultaneously and asynchronously in complex proofs. Currently, users may invoke the METAPRL proof engine (Hickey & Nogin, 2000; Hickey, 2001; MetaPRL) and the intuitionistic first-order theorem prover JPROVER (Schmitt *et al.*, 2001). Additional proof systems will be connected in the future.

2.2 Ensemble

ENSEMBLE (Hayden, 1998; Ensemble) is a high-performance network protocol architecture that aims at securing critical applications. It is a successor of the widely adopted system ISIS (Birman & van Renesse, 1994) and HORUS (van Renesse *et al.*, 1996) and is designed particularly to support group membership and communication protocols. ENSEMBLE is currently used in the BBN Aqua and Quo platforms, a fault-tolerant test bed at JPL, the Adapt adaptive multimedia middleware system at Lancaster University, a multi-player game by Segasoft, and in the Alier financial database tools.

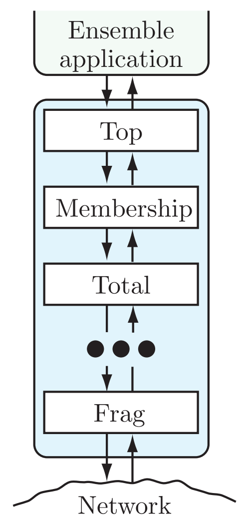


Fig. 2. Ensemble architecture

ENSEMBLE's architecture (Hayden & Rodeh, 2001) is based on the notion of a *protocol stack*. The system is constructed from a library of over sixty *micro-protocol* modules, or *layers*, which implement fragmentation and re-assembly, flow control, message ordering, buffering and retransmission, signing and encryption, group membership, synchronization, and other functionality. Micro-protocols can be stacked in a variety of ways to meet the communication demands of an application. Each module adheres to a common interface consisting of a top-level and a bottom-level part. The top-level interface of a module communicates with the bottom-level interface of the module immediately on top of it. The interface is *event-driven*: modules pass event objects to the adjacent modules. Certain types of events (e.g. send events) travel down, while others (e.g. message delivery events) travel up the stack.

ENSEMBLE is written almost entirely in OCAML (Leroy, 2000). The choice of OCAML instead of C, which was used for implementing ENSEMBLE's predecessor HORUS, significantly reduced the size of micro-protocol code and made it easier to develop and maintain ENSEMBLE modules. The main benefit of OCAML, however, is that it has a precise mathematical semantics, which can be employed for the automated verification, optimization, and generation of ENSEMBLE protocol stacks. These operations can be tedious and error-prone if performed manually. Using the NUPRL proof development system enables us to address both problems: automation makes it possible to re-use common reasoning strategies, and formal reasoning guarantees the correctness of code transformations.

3 Representing OCaml programs in Nuprl

OCaml (Leroy, 2000), is a strongly typed, functional programming language that has been extended by reference cells, exceptions, a module system, and an object calculus. Its functional core is similar to the language of Nuprl's type theory. But it has a different, less rigid syntax and contains many additional features.

To support formal reasoning about systems implemented in OCaml we have to be able to automatically convert OCaml programs into terms of Nuprl's type theory that capture the semantics of these programs. For this purpose we have developed a shallow *type-theoretical semantics* of OCaml that is faithful with respect to the informal semantics given in the OCaml manual (Leroy, 2000) and to the operational semantics generated by the OCaml compiler.

We have "implemented" this formalization using Nuprl's definition mechanism: definitional *abstractions* add new terms to Nuprl's type theory that capture both the structure and the operational semantics of OCaml language constructs, while *display forms* make sure that the outer appearance of these terms is identical to the OCaml construct they represent. For instance, an OCaml function declaration, as we will elaborate in Section 3.1, is represented by the following two objects.

```
ABS     Function{}(p; e)  ≡ λs,env. inr(λe₁. p e₁ e), s
DISP    function p -> e   ≡ Function{}(p; e)
```

The abstraction declares a new abstract term with *operator identifier* `Function`, no *parameters*, and two *subterm arguments* p and e. The abstraction defines the abstract term to be equal to the term on the right hand side. The corresponding display form describes that the new term is to be displayed as `function` p `->` e . Together, abstraction and display form represent *the structure, the semantics, and the syntax* of the OCaml function declaration. In the rest of this paper we will omit the abstract description of new terms and simply write

```
function p -> e   ≡ λs,env. inr(λe₁. p e₁ e), s
```

In addition to the formal representation we have also developed a formal *programming logic* for reasoning about OCaml programs and their evaluation. The programming logic is expressed in the form of inference rules that are implemented as Nuprl tactics and are based on well-formedness theorems of each language construct. This guarantees that the rules are correct with respect to the type-theoretical semantics of OCaml and allows formal reasoning to be performed at the level of OCaml programs instead of the underlying type-theoretical concepts.

Finally, we have created tools that convert OCaml programs into their formal Nuprl representations and store them as objects of Nuprl's library. These tools make the actual OCaml-code of an Ensemble protocol stack available for formal reasoning within Nuprl and help the reasoning system to keep track of modifications in Ensemble's implementation.

In the rest of this section we will describe the formalization of OCaml and the tools that translate between OCaml programs and their formal representations.

3.1 A Type-Theoretical Model for OCaml

The basic language of OCaml is centered around a functional core enhanced by patterns, references, and exceptions. We will briefly discuss aspects of formalizing these components and then describe a type-theoretical model for OCaml. We will illustrate this model by representing OCaml's language core in Table 3 and present the complete formalization of OCaml based on this model in Section 3.3.

OCaml*'s functional core* is a simple applicative language with constants, higher-order functions, local bindings, call-by-value application, and recursive definition. Its *expressions* and *values* are

$$
\begin{array}{llll}
\textit{(Expressions)} & e & ::= & v \mid e_1\ e_2 \mid \texttt{let}\ x\texttt{=}e_1\ \texttt{in}\ e_2 \mid \texttt{let rec}\ x\texttt{=}e_1\ \texttt{in}\ e_2 \\
\textit{(Values)} & v & ::= & c \mid x \mid \texttt{function}\ x\ \texttt{->}\ e
\end{array}
$$

where x is a variable and c a constant. Apart from notational differences this language is similar to the core of Nuprl's type theory. OCaml's function declaration `function` x `->` e has the same semantics as the lambda-abstraction $\lambda x.e$ in type theory, function application in OCaml and Nuprl are the same, a `let`-binding `let` x=e_1 `in` e_2 can be expressed as application of an abstraction $(\lambda x.e_2)\ e_1$, and a recursive definition `let rec` x=e_1 `in` e_2 as $(\lambda x.e_2)(\texttt{Y}(\lambda x.e_1))$. OCaml variables are represented as Nuprl variables while the representation of constants depends on the representation of the corresponding data types.

Patterns are language constructs that enable a programmer to decompose data structures in a convenient way. A local binding of the form `let` p=e_1 `in` e_2 matches the pattern p against the expression e_1 and *binds the variables* of e_2 that occur in p accordingly. In contrast to a simple `let`-binding, the pattern p may contain several variables that are grouped together by data structure constructors, which determine the fragment of e_1 that will be bound to the variables in e_2.

As Nuprl's type theory does not include general pattern matching, OCaml function declarations and `let`-bindings cannot be mapped directly onto corresponding Nuprl constructs anymore. Therefore, we explicitly represent OCaml's runtime *environment* of *variable bindings* and separate expressions from patterns. OCaml *expressions* are functions that evaluate variables and terms in this environment, while *patterns* are functions that update the environment of some expression.

Formally, an *environment* env is represented as a table consisting of pairs of variables and values. Table lookup (env`[`x`]`) and extending the table by a new binding (env`@`$\{x\mapsto v\}$) are defined via predefined standard list and pair operations.

$$
\begin{array}{lcl}
\{\} & \equiv & \texttt{[]} \\
env\texttt{@}\{x\mapsto v\} & \equiv & \texttt{<}x\texttt{,}v\texttt{>}\ \texttt{::}\ env \\
env\texttt{[}x\texttt{]} & \equiv & \texttt{lookup}\ x\ env
\end{array}
$$

An OCaml variable expression x will be represented by a function that expects an environment env as input and looks up the current value of x in that environment. In contrast, a *variable pattern* x_p will be represented by a function that takes two expressions e_1 and e_2 and modifies an environment env for evaluating e_2 by binding x_p to the value of e_1 in env. In this framework, a function definition `function` p `->` e

applies the pattern p to some input expression e_1 and to e, while a `let` binding `let` p=e_1 `in` e_2 is an application of p to e_1 and e_2. Evaluating `let` x_p=e_1 `in` e_2 in the environment env would result in e_2 (env@$\{x_p \mapsto (e_1\ env)\}$).

Reference cells in OCAML are special instances of mutable records, which are associated with generalizations of the familiar operations `ref` e , `!`e , and e_1`:=`e_2. As they may occur in arbitrary OCAML expressions, causing them to have side-effects, we need to extend our formal model by explicitly representing a global state.

We represent OCAML expressions by functions that operate on an environment env and a store s, evaluate a term with respect to env and s, and return both the value and the possibly updated store. Similar to an environment, a *store* s is a table of pairs of *addresses* (natural numbers) and values. The creation of new table entries (`New(`s`)`), store lookup (`!`s`[`$addr$`]`), and updating the contents of a store cell (s`[`$addr \leftarrow v$`]`) are defined via standard list and pair operations.

`[ ]`	$\equiv$	`[]`
`New(`s`)`	$\equiv$	`max (map_fst` s`) + 1`
s`[`$addr \leftarrow v$`]`	$\equiv$	`<`$addr$`,`v`> ::` s
`!`s`[`$addr$`]`	$\equiv$	`lookup` $addr$ s

A reference `ref` e evaluates the expression e with respect to the current store and environment, determines a free address in the updated store s_1, assigns the value v of x to that address, and then returns the address and the modified store s_1`[`$addr \leftarrow v$`]` as result. Dereferencing `!`e means evaluating e and then looking up the resulting address in the (updated) store (`!`s_1`[addr]`). An assignment e_1`:=`e_2 first evaluates e_2 and then e_1. Afterwards it assigns the value of e_2 to the store address determined by e_1. The result of the expression is the `unit` value `()`.

Exceptions are an ML concept for handling failure while ensuring typability of expressions. Exceptions may be *raised* by runtime failures or by explicit calls to the function `raise` and are handled using the expression `try` e_1 `with` p `->` e_2 .

Exceptions can be viewed as alternative result of evaluating an expression and belong to the same type. The exception `Division_by_zero` generated by evaluating `x/0`, for instance, has the type `int`. Formally, OCAML data types are a disjoint union `EXCEPTION +` T of the type of exceptions (represented as atoms) and the type T of the data type's values. Values v and exceptions exn have to be tagged accordingly. For this purpose we use the NUPRL terms `inr(`v`)` and `inl(`exn`)` .

We represent `raise` exn as a function that for each environment env and store s returns `inl(`exn`)` and an unchanged store. `try` e_1 `with` p `->` e_2 is represented by a function that returns the value of e_1 unless the evaluation fails. Otherwise it matches the resulting exception against the pattern p and proceeds with evaluating e_2 in an enriched environment.

As exceptions may occur in arbitrary OCAML expressions, our representation of expressions and patterns must always check whether the result of evaluating an expression is a value or an exception. To simplify the formalization, we introduce an abbreviation `let↓ inr(`v`),`s `=` $eval_1$ `in` $eval_2$ that catches this failure check.

```
let↓ <inr(v),s> = eval₁ in eval₂
                    ≡ let <r,s₁> = eval₁ in
                          case r of   inl(exn) ↦  inl(exn), s
                                    | inr(v) ↦  eval₂
```

The formal model. The combination of the functional core, patterns, references, and exceptions describes all the essential features of the CAML part of OCAML, i.e. the language fragment without the object and module system. The expressions, values, and patterns of this core language are

(Expressions)	e	::=	v | e_1 e_2 | `let` p=e_1 `in` e_2 | `let rec` p=e_1 `in` e_2
(Values)	v	::=	c | x | `function` p `->` e | `ref` | `!` | `:=`
			| `raise` | `try` e_1 `with` p `->` e_2
(Patterns)	p	::=	x_p

where x is a variable, x_p a variable pattern, and c a constant. The type-theoretical formalization of this language core is described by the NUPRL definitions in Table 3. It represents OCAML expressions as elements of the type

`EXPR` $\equiv$ `s:STORE` $\rightarrow$ `env:ENV` $\rightarrow$ `(EXCEPTION + VALUE)` $\times$ `STORE`

where `ENV` is the type of variable binding environments (tables of variable names and values), `STORE` the type of stores (tables of addresses and values), `EXCEPTION` the type of exceptions (atoms), and `VALUE` the type of values (NUPRL's type `Top`). To "lift" a value v to an expression that always evaluates to v, we use the abbreviation

$\lceil v \rceil$ $\equiv$ λ`s,env. inr(`v`), s`

Lifting is necessary to express the call-by-value evaluation order in the representation of function application and exception handling.

OCAML patterns are represented as elements of `EXPR` $\rightarrow$ `EXPR` $\rightarrow$ `EXPR` . The language core contains only variable patterns. The representation of all other patterns of OCAML is discussed in Appendix A.2.

OCAML types are represented in a way that the typing relation in OCAML can be expressed by the built in membership relation of NUPRL's type theory. Thus OCAML types will be represented as subtypes of the type `EXPR`, i.e. as (dependent) function types of the form `s:STORE` $\rightarrow$ `env:ENV` $\rightarrow$ `(EXCEPTION +` T`)` $\times$ `STORE`. The representation of OCAML types is discussed in detail in Appendix A.3

It should be noted that, due to the use of formal abstractions, every OCAML program corresponds to exactly one canonical representation in NUPRL. The type-theoretical semantics of this term, defined in the abstractions, describes the operational semantics of the program. The structure of this term, built from abstract NUPRL terms, describes its abstract syntax tree. The display of the term, defined by the corresponding display forms, is identical to that of the program.

The OCAML program `(function x -> x) x`, for instance, is represented by the NUPRL term `Apply{}(Function{}(PatVar{x}();Var{x}()); Var{x}())`. The semantics of this term, given by the definitions in Table 3, is a term that reduces to λ`s,env. inr(env[`x_v`]),s` , the semantics of (the representation of) the variable `x`. The term is displayed by the NUPRL system as `(function x -> x) x`.

```
e1 e2                    ≡ λs,env. let↓ <inr(v),s1> = e2 s env in
                                   let↓ <inr(f),s2> = e1 s1 env in
                                     (f ⌈v⌉) s2 env
let p = e1 in e2         ≡ p e1 e2
let rec p = e1 in e2     ≡ p (Y (λe. p e e1)) e2
x_v                      ≡ λs,env. inr(env[x_v]),s
function p -> e          ≡ λs,env. inr(λe1. p e1 e), s
ref e                    ≡ λs,env. let↓ <inr(v),s1> = e s env in
                                   let addr = NEW(s1) in
                                     inr(addr), s1[addr←v]
!e                       ≡ λs,env. let↓ <inr(addr),s1> = e s env in
                                   inr(!s1[addr]), s1
e1 := e2                 ≡ λs,env. let↓ <inr(v),s1> = e2 s env in
                                   let↓ <inr(addr),s2> = e1 s1 env in
                                     inr (), s2[addr←v]
raise exn                ≡ λs,env. inl(exn), s
try e1 with p -> e2      ≡ λs,env. let <r,s1> = e1 s env in
                                   case r of inl(exn) ↦ (p ⌈exn⌉ e2) s1 env
                                           | inr(v)   ↦ inr(v), s1
x_p                      ≡ λe1,e2. λs,env. let↓ <inr(v),s1> = e1 s env in
                                   e2 s1 (env@{x_p↦v})
```

Table 3. NUPRL representation of OCAML's language core

3.2 Modules and Objects

Modules and compilation units are *second class* objects of the OCAML programming language. They cannot be used as parts of expressions and have no operational semantics per se. Instead, they provide a means for structuring code, for instance by introducing names for user-defined types and functions and binding them to a particular expression, describing the signature of an abstract data type, providing alternative implementations of the same function, and disambiguating references to names that are defined in multiple modules.

As ENSEMBLE's implementation uses neither functors nor the `with` operator (see §6.10.4 of (Leroy, 2000)) in the specification of its modules, it suffices to represent each module expression as a separate object of NUPRL's library, i.e. as object on the meta-level of NUPRL's type theory. We use meta-level *object generators* to map type and value definitions onto NUPRL abstractions that globally declare the corresponding names, prefixed by the name of the module, as abbreviation for a certain expression or type (see Appendix A.4). Module definitions and the `open` command are instructions that influence the behavior of these object generators and make sure that named references within type-theoretical expressions are linked to the correct abstraction. As a consequence each module expression is accessible as separate NUPRL object and can be reasoned about individually.

We are currently investigating a more general approach that formalizes module expressions *within* type theory instead of its meta-level. In this approach we represent module expressions as functions that update a *global* environment. This allows treating named references as variables that consult the environment during evaluation. The approach is theoretically more satisfactory and also allows representing functors as functions on global environments. However, it is also more complex and its consequences for practical reasoning need to be evaluated.

Objects, methods, classes and inheritance are not used in ENSEMBLE's implementation and currently not represented. Although objects could be understood as generalization of reference cells, the current type theories do not offer sufficient support for expressing the formal semantics of methods and inheritance. Preliminary studies by A. Kopylov indicate that a combination of dependent records and union types may provide a foundation for representing objects in the future.

3.3 Extent of Formalization

As the main purpose of our formalization of OCAML is to provide a foundation for the verification, optimization, and synthesis of ENSEMBLE protocols and stack, we have focused on developing type-theoretical representations of OCAML's functional core and the language constructs that are actually being used in the code of ENSEMBLE. Our formalization was originally based on OCAML-1.07 and later migrated to OCAML-2.02. We are currently working on adding the new features of OCAML-3.0x and making the representation more complete.

Chapter 6 of the OCAML manual (Leroy, 2000) describes the syntax of all OCAML language constructs and their informal semantics. Our type-theoretical representation provides a *formal* semantics of these language constructs through NUPRL *abstractions* and represents their syntax through the corresponding *display forms*. NUPRL *precedence objects* are being used to control the automatic generation of parentheses that follow OCAML's rules for precedences and associativity. Finally *well-formedness theorems* show that the NUPRL representation of a language construct is faithful with respect to OCAML's type system. For instance, NUPRL representations of OCAML expressions must be members of (the NUPRL representations of) the corresponding OCAML types. Our formalization, which we will describe in detail in Appendix A, currently includes the following

- All expressions listed in §6.7, including the constants listed in §6.5, except for bitwise logical and floating point operators, method invocation, object creation, coercion, and object duplication (see Appendix A.1).
- All patterns listed in §6.6, except for float patterns and variant abbreviation patterns (see Appendix A.2).
- All type expressions listed in §6.4, except for closed variant types, object and class types. Labels in function types are not yet supported (see Appendix A.3).
- All type definitions listed in §6.8, except for constraint definitions and optional prefixes for type parameters (see Appendix A.4).
- There is limited support for module specifications, module implementations and compilation units, listed in §6.10–12, as described in Section 3.2.
- Classes, listed in §6.9, are not yet supported.

Our formalization had to take into account that OCaml constructs have a more flexible syntax than the terms of type theory. Record expressions $\{f_1=e_1; .. ; f_n=e_n\}$, for instance, can have arbitrarily many components, while in Nuprl each term must have a fixed number of subterms. Therefore, although we can formalize arbitrarily sized record expressions by Nuprl terms, it is not possible to use a single Nuprl abstraction for this purpose. Instead, we have to *iterate* primitive abstractions to build a formal representation of record expressions, similarly to the way one would build constant list expression by iteratively prepending elements to the empty list, and provide appropriate iteration definitions in the corresponding display forms to make sure that the resulting term is displayed as syntactically correct OCaml record expression (see Appendix A.1 on page 41 for a detailed discussion).

Thus in general, OCaml language constructs do not correspond directly to Nuprl abstractions but to Nuprl terms that are constructed by combining several primitive abstractions. For each construct we have developed a meta-level *term generator*, i.e. an ML function that builds the Nuprl representation of the construct. These term generators are crucial for automatically creating the formal representation of a piece of OCaml code and are used by the tools that import OCaml source code into Nuprl as well as by the tools for synthesizing and optimizing code.

Our representation of the fundamental OCaml language constructs in Nuprl currently includes 220 formal abstractions (to represent OCaml semantics), 230 display forms (to represent OCaml syntax), and 190 wellformedness theorems relating formal abstractions to (OCaml) types. In addition to these we have imported a large fragment of the OCaml libraries listed in §18–28 of (Leroy, 2000) into Nuprl using the automatic tools described in Section 3.5 below. For externally defined library functions this also meant providing "external" implementations, i.e. explicit representations to which the `external` command could link.

3.4 A Programming Logic for OCaml

Formal reasoning about OCaml programs involves two types of arguments: reasoning about program properties (static semantics), and reasoning about the results of program executions (dynamic semantics).

Program properties are usually expressed by a type system. The type of a program can range from being its OCaml data type to a precise specification of its behavior. Our formalization of OCaml enables us to use Nuprl's built-in membership relation for both purposes: the OCaml types of OCaml expressions are represented as the Nuprl types of their Nuprl representation. More specific program specifications can be described using Nuprl's subset constructor.

Reasoning about program executions involves reasoning about the semantic equality of OCaml expressions that are syntactically different. Two OCaml expressions are semantically equal if their representations in Nuprl are. As Nuprl's type theory comes with built-in notions of extensional equality and evaluation of terms, the equality of OCaml expressions can be shown by applying Nuprl's proof rules for reasoning about computation and equality of expressions.

In theory, applying NUPRL's inference rules is sufficient to support formal reasoning about OCAML programs: one would simply have to unfold the abstractions in the formal representations of the programs and types and then reason about the resulting term in type theory. This however would make formal reasoning about OCAML programs impractical and formal arguments inaccessible to programmers, as there is no visible relation between the formal proof and the original program.

To support formal reasoning on the level of OCAML, we have developed a programming logic for OCAML, whose inference rules are *derived* from the formal semantics of expressions and types. These rules preserve the "OCAMLness" of expressions: they always return (representations of) valid OCAML expressions and do not reveal the underlying type-theory. They are implemented as tactics, which in turn are based on formal abstractions, formal lemmata about wellformedness and evaluation, and the proof rules of NUPRL. This ensures that the implemented programming logic is faithful with respect to the type-theoretical semantics of OCAML.

The rules for reasoning about program properties follow the style of reasoning in NUPRL: they are top-down sequent rules that reason about the type of a program expression by decomposing both the expression and the type into smaller fragments such that proofs for the resulting fragment subgoals are sufficient to establish the original goal. The derived rule for reasoning about function application in OCAML, for instance, states that applying an expression f to an expression e is proven to be of OCAML type T if we can prove f to be of type S `->` T and e to be of type S.

```
Δ ⊢ f e ∈ T
BY ApplyMem
  Δ ⊢ f ∈ S -> T
  Δ ⊢ e ∈ S
```

Δ is a placeholder for the sequent's assumptions, which remain unchanged by the rule. Its implementation as NUPRL tactic decomposes a proof goal into the two subgoals described above, provided the goal matches the first line of the rule. A complete list of the derived inference rules for reasoning about properties of OCAML programs is given in (Kreitz, 1997).[1]

Similarly we have derived rules for the symbolic evaluation of OCAML expressions from the formal operational semantics of OCAML expressions. The rule for evaluating an OCAML function application, for instance, states that applying the expression `function` x `->` e to an expression e' can be reduced to the OCAML expression $e[e'/x]$, i.e. the expression e with occurrences of x replaced by e'.

$$\texttt{(function } x \texttt{ -> } e\texttt{)}\ e' \longrightarrow e[e'/x]$$

This rule can only be applied if e' is *free of side-effects*, since side-effects may evaluate differently on the left and right hand side. A tactic that executes the rule through controlled applications of NUPRL rules would not be able to prove the two expressions equal. The function body e, on the other hand, may include side-effects.

[1] Although the formal model for OCAML in (Kreitz, 1997) is simpler than the one described here, the derived rules remain the same. Only their implementation as tactics has changed.

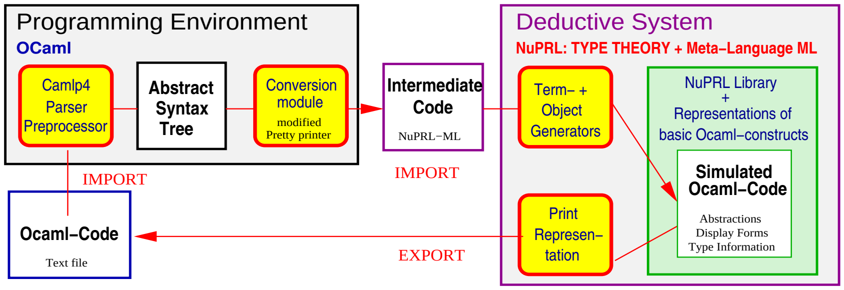


Fig. 3. Importing and Exporting OCaml source code

Appendix A.5 gives a detailed description of the inference rules for reasoning about the computational behavior of OCaml programs. These rules can not only be used for computing the result of executing a program in a specific context, but also for transforming OCaml programs into equivalent ones. As such, they provide a foundation for the formal optimizations that we will discuss in Section 4.

3.5 Import and Export of OCaml Code

The type-theoretical semantics of OCaml and its "implementation" as Nuprl abstractions and display forms provide the foundation for formal reasoning about OCaml programs. However, if we want to reason about real system implementations with ten thousands of lines of code, we have to provide mechanisms that automatically translate the OCaml code into its Nuprl representation and vice versa.

Figure 3 illustrates our mechanisms for importing and exporting OCaml source code. The tools for *importing* OCaml code into Nuprl have to analyze the syntax of the OCaml source code, create the type-theoretical terms that represent it, and store them as objects in Nuprl's library. To ensure faithfulness with respect to the OCaml programming environment we use the Camlp4 parser-preprocessor (de Rauglaudre, 2000), an isolated version of the OCaml-parser, as tool for analyzing program text. Camlp4 parses OCaml source code, generates its *abstract syntax tree*, and then calls an output module for further processing, e.g. pretty-printing or dumping the binary.

To convert the abstract syntax tree into Nuprl objects, we have implemented a new output module for Camlp4 that creates *intermediate code*, which causes Nuprl to build the corresponding abstractions and display forms. The module generates pieces of this code for each node of the syntax tree, distinguishing the various kinds of identifiers, expressions, patterns, types, signature items, and module expressions as specified in Appendix A of (de Rauglaudre, 2000).

The intermediate code is a program in Nuprl's meta-language ML that calls the term and object generators described in Sections 3.3 and 3.2 above to build the Nuprl representations of the OCaml code. It will be passed to the Nuprl system, which generates abstractions and display forms for each function, type, and module declared in the source code. In the process, the object generators also resolve issues that arise when linking the code of several modules, which cannot be addressed by the parser. These are name resolution (i.e. linking a named reference to a specific object in some module), determining whether an identifier represents a variable or a named reference, and resolving overloading of operator names. In addition to that, the object generators also attempt to state and prove a well-formedness theorem for each newly generated object, using its OCaml signature, if available, or a type inference algorithm to determine its type.

To make synthesized OCaml programs available to the OCaml programming environment, we have also implemented a mechanism for *exporting* Nuprl representations of OCaml code. Since we have designed the display forms and precedence objects for each OCaml language construct in a way that they obey OCaml's syntax requirements, the Nuprl system displays and prints terms that represent OCaml programs as syntactically correct OCaml source code. Exporting OCaml code is therefore simply a matter of selecting the code pieces to be exported and printing them into a file. The generated program text can then be compiled, linked, and executed in the OCaml environment without further modifications.

We have used our tools to import a large fragment of the OCaml libraries (44 modules containing about 10,000 lines of code) and the essential modules of Ensemble (79 modules / 40,000 lines) into the Nuprl system. This resulted in the creation of 2,320 formal abstractions and an equal number of display forms and well-formedness theorems. Terms representing user-defined functions reach more than 50KB in size and may grow to more than 1 MB if abstractions are unfolded.

4 Formal Optimization

Ensemble's modular approach to building communication systems has many advantages over monolithic systems. Small components are easier to design, specify, develop, test, verify, and optimize, while application systems may be more readily adapted to new environments and extended at run-time with new components. However, building systems from components usually comes with a performance penalty: the abstraction barriers between the components impose high overheads arising from additional function calls, redundant code, and code that is not used in a particular configuration. In this section we will show how Ensemble's performance can be significantly improved by employing formal logical optimizations that turn Ensemble into one of the fastest reliable multicast systems currently available.

(Hayden, 1998) suggests several optimizations that could be applied to stack-based architectures implemented in functional programming languages.

1. *Avoid garbage collection* during message processing by freeing space allocated for messages after they have been sent or delivered.
2. *Avoid marshaling* of small objects when sending data over the net.
3. *Delay non-critical message processing* by sending and delivering messages before updating a module's state and buffers.
4. Compress the module stack for common sequences of execution, creating a *fast-path* that can be used in the common case.
5. *Compress message headers* that are added by micro-protocols while processing a message that is being sent.

The first three steps do not affect the modular approach as such and can be addressed when implementing the system modules by adding low-level procedures that overwrite the defaults of OCaml's run-time system and by adopting a particular programming style. However, the final two optimizations require generating special code *after* an application system has been configured from the modules.

Generating fast-paths for common cases and compressing headers is far beyond the capabilities of current compiler optimization techniques. Therefore previous work (Abbott & Peterson, 1993; Engler & Kaashoek, 1996; Hayden, 1998) involves significant annotation of the code or hand-optimization, which is a difficult and error-prone process. By using a formal logical tool like Nuprl, we can completely automate these optimization steps and prove that they do not introduce any errors.

4.1 Fast-path Optimization

In most applications of communication systems it is easy to identify common sequences of execution. Data are being sent and received and there is no need for message partitioning, retransmission, buffering, synchronization, etc. This means that only a small fragment of the code of a protocol stack is involved in processing the message and that some micro-protocols are not being activated at all. *Fast-path optimization* aims at improving the system's performance by identifying the code that is actually used in the common case.

To understand fast-path optimization, it is useful to think of a protocol as a function that takes the internal state of the protocol and an input event and produces an updated state and a list of output events. In this view, a protocol stack is a macro-protocol built by composing protocol functions. A protocol can be optimized if we can describe its regular state and common input events. Formally, we use a *Common Case Predicate* (CCP), a Boolean function on the state of a protocol and an input event, for this purpose. CCPs are usually specified by the programmer of a protocol or may be determined from run-time statistics. For example, a CCP may express that the event is a deliver event whose sequence number is equal to the next expected packet to arrive. If the CCP is satisfied for a received message, then the message may be delivered and the index of the next expected packet incremented. Otherwise, the message would have to be buffered.

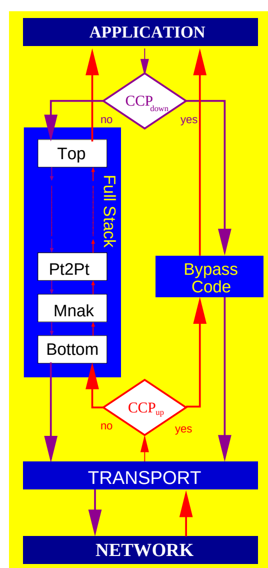


Fig. 4. Placement of bypass code in Ensemble

Analyzing the path of events that satisfy the common case predicate through the code of a protocol stack helps isolating the code that is actually being used. We call this path a *fast-path* through the protocol stack and the resulting code a *bypass*, which will be used to process events in the common case. To decide whether a message can be handled by the bypass code or has to go through the original stack, we use the same CCP that was used to generate the bypass code, as illustrated in Figure 4 (The transport module below the protocol stack provides marshaling of messages). It is necessary to generate efficient code for this CCP, as it will be executed for every event.

In the following sections we will describe how to perform fast-path optimization within the framework of a formal theorem proving environment.

4.2 A Knowledge-based Approach to Optimization

Formally, the optimizations necessary to generate bypass code can be expressed as conditional rewrite steps that simplify and evaluate code fragments in the context of a logical proposition, the common case predicate. To perform these steps, we use the NUPRL proof development system. NUPRL enables us to automatically generate bypass code and to wrap it by a module to be inserted into ENSEMBLE. Additionally, we can produce a formal proof that the generated code is equivalent to the original one with respect to the formal semantics of the programming language.

Experience has shown that purely tactic-based rewrite techniques are not appropriate for optimizing protocol stacks. Although they are very useful for optimizing individual micro-protocols (as described in Section 4.3) they scale badly when protocols are assembled into a stack. The reason for that is that the code for composing protocols must provide for the situation that a protocol's input event may generate several output events to be sent to both adjacent protocols. Tracing the path of an event thus requires an optimization tactic to deal with the formal representation of the entire code of the protocol stack – more than 20,000 lines of code – at once.

Furthermore, although the optimization of individual micro-protocols is largely automated, it occasionally requires expertise about the actual implementation to be provided interactively. In contrast to the programmers who designed the micro-protocols, application designers who configure the components of a communication system to suit a particular application usually do not have this expertise.

Our approach to fast-path optimization takes into account that the *implementation* of the individual micro-protocols is static for each release of ENSEMBLE, while the *configuration* of application stacks is not. This allows us to provide formally verified *knowledge* about fast-path optimizations of each micro-protocol *a priori*, i.e. together with the release of ENSEMBLE, and to implement tactics that use this knowledge to automatically generate fast-path optimizations of application stacks. Formally, we do so by composing optimization theorems as illustrated in Figure 5.

There are two levels of formal optimizations. The first, or *static*, level (Section 4.3) depends solely on the code of the individual micro-protocols and is performed semi-automatically under the guidance of the developer of the micro-protocol and a NUPRL expert, using a small collection of special-purpose tactics. At this level we generate and prove *optimization theorems* about the result of optimizing micro-protocols for four fundamental cases: down- or up-going events (sending or receiving messages) for both point-to-point sending and broadcasting. These theorems may take anything from 2 minutes to an entire afternoon to develop and are included in the optimization tool that is made available to application developers.

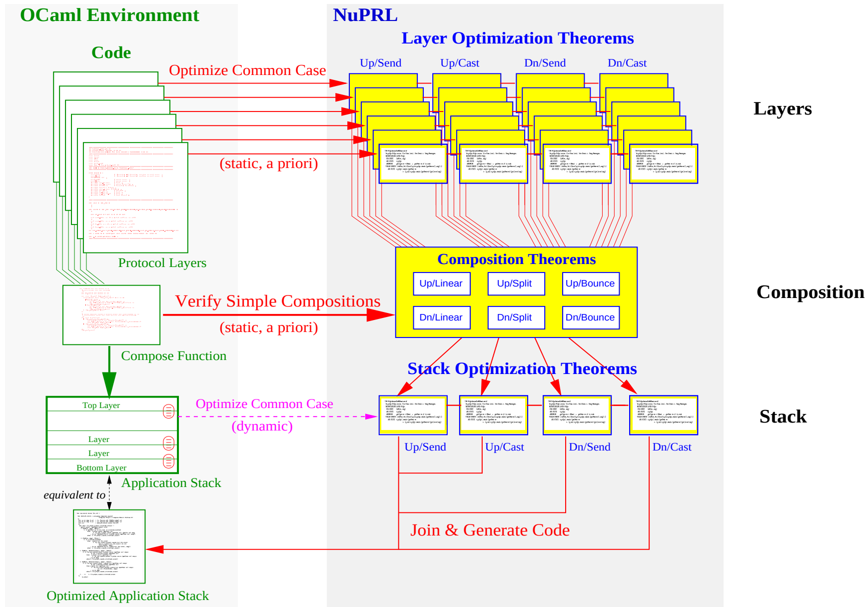


Fig. 5. Optimization methodology: composing optimization theorems.

The second, or *dynamic*, level (Section 4.4) depends on the protocol stack configured by an application developer and cannot be provided a priori. Using *composition theorems* about the effect of applying ENSEMBLE's composition mechanism to common combinations of fast-path optimizations, we generate and prove theorems about the result of optimizing the application stack. Header compression, described in Section 4.5 may be integrated at this stage. We then synthesize bypass code from these theorems and create a module for integrating it into ENSEMBLE (Section 4.6). This step is completely automated and requires only the names of the micro-protocols used in the application stack as input.

It should be noted that optimization is orthogonal to verification. Our formal tools prove that the resulting code is semantically equal to the original protocol stack but do not make any assumptions about the correctness of the stack. In the rest of this section we will explain the technical details of the optimization tool.

4.3 Optimization of Micro-Protocols

The static optimizations of ENSEMBLE micro-protocols are based on an analysis of *possible* fast-paths, or branches in the protocol code, which essentially describes a simple state-event machine. In principle, each branch could be considered a fast-path whose CCP is composed of the predicates in the corresponding conditionals. However, a common case in communication is usually related to “ordinary” messages being sent or received, and not to handling errors, retransmissions, group management, etc. Identifying these paths requires insight into the code, which means that the ENSEMBLE programmer must either annotate the code or provide the information explicitly when initiating the optimization of the micro-protocol.

The optimization of a protocol layer proceeds by a series of code transformations that preserve the semantics of a layer's code under the assumption of a CCP and are based on the following basic mechanisms:

Function inlining and symbolic evaluation simplifies code in the presence of constants or function calls. Logically, this means *rewriting* the code by unfolding definitions and controlled partial evaluation. Both techniques can be expressed in terms of evaluation rules from our programming logic for OCaml (see Section 3.4, which guarantees their correctness with respect to OCaml's type-theoretical semantics).

To automate the application of these rules we have developed an evaluation tactic `Red`, which searches top-down for the first reducible subterm of an OCaml program fragment and reduces it. The search can be restricted by providing a subterm address as additional argument. This allows a user to to leave certain expressions unchanged while focusing on meaningful reductions.

Directed equality substitution such as the application of distributive laws lead to further simplifications of the code. Technically, we apply lemmata from Nuprl's logical library. By adding to each lemma a direction that ensures formulas to become simpler (e.g., indicating whether an equality lemma should be applied from left-to-right or vice versa), we guarantee the termination of this process.

Context-dependent simplifications help in extracting the relevant code from a micro-protocol. They trace the code path of messages that satisfy the CCPs and isolate the corresponding code fragments. Technically, a CCP is expressed as an equality that describes the value of a piece of code in a case split or a conditional. We use this equality to substitute a piece of the code by a value and and then rewrite the result with the above two mechanisms. A tactic `UseAssumption` performs these steps automatically for a given assumption. Its implementation is straightforward, as Nuprl supports equality reasoning and the management of hypotheses.

Tailored transformations take advantage of the fact that micro-protocols in Ensemble are coded according to a certain discipline. This discipline is illustrated in Table 4 by the code of Ensemble's `Pt2pt` micro-protocol, which implements fault-tolerant point-to-point message delivery. Each protocol layer l is built from an initialization function `init` and an event handler `hdlrs`, which describes how input events affect the state of the protocol and what events will be sent to adjacent protocols. The function `hdlrs` is split into five subhandlers for up- and down-going events with full, local, or no message headers. Each of these subhandlers performs a case analysis over type of the input event (sending, broadcasting, group management, error handling, etc.) and triggers state updates and outgoing events accordingly.

Sending events to adjacent micro-protocols is done by calling their respective event handlers rather than sending the events explicitly. This particular programming style makes it possible to create a functional, imperative, or threaded version of Ensemble (with different implementations of event queueing and layer composition) from a single reference implementation by packing the layer's initial state and event handler with a function called `Layer.hdr`. The conversion into functional, imperative, or threaded mode will take place when a protocol stack is built.

```
type header = NoHdr | Data of seqno | Ack of seqno | Nak of seqno * seqno
type state  = {sweep: Time.t; mutable next_sweep: Time.t; ... }
let init _ (ls,vs) = {sweep = Param.time vs.params "pt2pt_sweep"; ... }
let hdlrs s (ls,vs) {up_out=up; upnm_out=upnm;
                     dn_out=dn; dnlm_out=dnlm; dnnm_out=dnnm}
     = let log = Trace.log "PT2PT" ls.name in
         let up_hdlr ev abv hdr = ...
         and uplm_hdlr ev hdr   = ...
         and upnm_hdlr ev       = ...
         and dn_hdlr ev abv     =
              match getType ev with
              | ESend ->
                 let dest = getPeer ev in
                   if dest = ls.rank then failwith "PT2PT: send to myself";
                   let sends = Arraye.get s.sends dest in
                     let seqno = Iq.hi sends in
                       let iov = getIov ev in
                         Iq.add sends iov abv ;
                         dn ev abv (Data seqno)
              | _ -> dn ev abv NoHdr
         and dnnm_hdlr          = dnnm
       in {up_in=up_hdlr; uplm_in=uplm_hdlr; upnm_in=upnm_hdlr;
           dn_in=dn_hdlr; dnnm_in=dnnm_hdlr}
let l args vs = Layer.hdr init hdlrs args vs
let _ = Layer.install "PT2PT" l
```

Table 4. Excerpt from the code of Ensemble's `Pt2pt` micro-protocol.

The coding discipline for Ensemble micro-protocols enabled us to write a tactic `EvaluateCodeStructure` that applies a predetermined series of controlled unfold and evaluation steps, η-reductions, distributive laws, and other "undirected" equalities to isolate the relevant action in the event handler of a protocol. The tactic performs the tedious initial steps of a micro-protocol optimization automatically and is very efficient, since it does not require search.

Optimizations for each micro-protocol proceed in two phases. In the first phase we use tactic-based forward reasoning to isolate the fast-path that corresponds to the CCP and to optimize its code. Optimizations are initiated for four fundamental cases: down- or up-going events for both point-to-point sending and broadcasting. Basic CCPs for these cases are created automatically and the Ensemble programmer may add additional CCPs to describe the common case.

To initialize the optimization of (the event handler of) a protocol layer l, we generate a Nuprl optimization object that contains an expression of the form

$$CCP_l \Rightarrow \ \texttt{let (s}_0\texttt{,hdlr) = convert Functional } l \texttt{ args (ls, vs)} \\ \texttt{in hdlr(}s_l\texttt{, } event\texttt{)}$$

where `convert` is an Ensemble library function that converts the code for the micro-protocol l into a functional protocol stack, which consists of an initial state $\texttt{s}_0$ and an event handler `hdlr`. s_l is the current state of l and *event* an input event, which must be of the form `UpM(ev,hdr)` or `DnM(ev,hdr)`, i.e. up- or down-going events `ev` with a message header `hdr`. The common case predicate CCP_l for the layer l usually characterizes the type of the event `ev` (send or broadcast), the structure of the header `hdr` (full header, no header, or a local header), and the state s_l.

For the sake of clarity we suppress irrelevant formal details in the top-level presentation by introducing a formal abbreviation (i.e. an abstraction and the corresponding display form) for the above expression and write

OPTIMIZE LAYER l FOR EVENT *event* AND STATE s_l ASSUMING CCP_l

A formal optimization begins by unfolding the formal abbreviation and moving the common case predicate into the hypothesis list. Using `EvaluateCodeStructure` we then isolate the case distinction within the relevant event handler of the function `hdlrs` (see Table 4). Afterwards we apply the tactic `UseAssumption` whenever the code to be optimized is a conditional or a case expression that fits one of the assumptions of the CCP, and the tactic `Red` as long as top-level reductions make progress. The process that applies these steps has been completely automated.

Usually the optimization may stop at this point. However, we give the Ensemble programmer an opportunity to invoke additional simplifications with the tactic `Red` before committing the result to Nuprl's library. Often, the (Nuprl representations of) about 250-500 lines of OCaml code will be reduced to a single update of the protocol's state and a single output event to be sent to the next layer.

We illustrate our method by an optimization of Ensemble's `Pt2pt` protocol with respect to point-to-point sending. The basic CCP for this situation states that input events have the form `DnM(ev, hdr)`, where the constructor `DnM` indicates a down-going input event, `ev` is a point-to-point send event (`getType ev = ESend`), and `hdr` is an arbitrary message header. The Ensemble programmer also adds the CCP that applications do not send to themselves (`getPeer ev` $\neq$ `ls.rank`). Formally, we create the following optimization object.

```
OPTIMIZE LAYER Pt2pt
   FOR EVENT   DnM(ev, hdr)
   AND STATE   s_pt2pt
   ASSUMING    getType ev = ESend  ∧  not (getPeer ev = ls.rank)
```

After applying `EvaluateCodeStructure` we reach a choice point, i.e. a case distinction, in the code of the event handler `dn_hdlr` (see Table 4) .

```
1. (getType ev) = ESend
2. not (getPeer ev = ls.rank)
⊢ match getType ev with
     ESend  -> ..... Code hidden ....
   | _      -> ..... Code hidden ....
```

The CCP now appears in the form of two assumptions. Details of the code are temporarily hidden from the display, as they are not relevant at this point and may not fit on the screen. We call `UseAssumption 1`, which leads to an evaluation of the first case of the case expression and eliminates all the other cases from the code.

```
1. (getType ev) = ESend
2. not (getPeer ev = ls.rank)
⊢  if getPeer ev = ls.rank then failwith "PT2PT: send to myself";
   let sends = Arraye.get s.sends (getPeer ev) in
     let seqno = Iq.hi sends in
       let iov = getIov ev in
         Iq.add sends iov hdr ;
         dn ev (Full (Data seqno), hdr)
```

Next, we apply `UseAssumption 2`, which leads to an elimination of the conditional and to subsequent reductions of the `let`-abstractions.

```
1. (getType ev) = ESend
2. not (getPeer ev = ls.rank)
⊢  Iq.add (Arraye.get s_pt2pt.sends (getPeer ev)) (getIov ev) hdr;
   dn ev (Full (Data (Iq.hi (Arraye.get s_pt2pt.sends (getPeer ev))), hdr))
```

No further reductions are meaningful at this point, since the remaining code contains only a single update to the state `s_pt2pt` and a single call to an down-going event handler, which receives the incoming message header, extended by additional data, as input.

In the second phase, we create an *optimization theorem*, which proves that, under the CCP, the bypass code is semantically equal to the protocol from which it was generated. It verifies the correctness of the optimizations and will later be used for the optimization of protocol stacks, described in Section 4.4. To create it, we compose the initial optimization object with its final result into a statement of a formal NUPRL theorem, which requires us to prove a conditional equality of the form

```
⊢ ∀s_l:l.state. ∀Hdr,Args:𝕌. ∀ls:View.local, vs:View.state.
    ∀ev:Event.t. ∀hdr:Hdr. ∀args:Args.
      CCP_l  ⇒  let (s_0,hdlr) = convert Functional l args (ls, vs)
                  in  hdlr(s_l, event)
              = updates; handlers
```

where the left equand is the starting point of the optimization and the right equand its final result, consisting of a series of state updates and calls to event handlers. To make the presentation of this formal theorem more accessible to programmers, we introduce a formal abbreviation that allows us to present statements of the above kind in the NUPRL system as follows.

```
OPTIMIZING LAYER l  FOR EVENT event  AND STATE s_l  ASSUMING CCP_l
YIELDS HANDLERS handlers  AND UPDATES updates
```

To prove the optimization theorem, we use the trace of the formal optimization as proof plan that triggers the application of NUPRL *proof tactics* that perform exactly the same steps on the left hand side of an equation as the rewrite tactics `Red`, `UseAssumption`, and `EvaluateCodeStructure` did on the code of the micro-protocol. As a consequence, the optimization theorem is created and proven automatically, even if the original optimization required considerable interaction. We have written a tactic `CreateOptVerify`, which performs all these steps. Since it is is guaranteed to succeed, it can be triggered as background process when the result of an optimization is being committed. Applying this tactic to the optimization of `Pt2pt` with respect to point-to-point sending, for instance, leads to the optimization theorem presented in Table 5.

Our library currently contains more than 100 optimization theorems for common ENSEMBLE micro-protocols. These theorems were generated from optimization objects that we developed using the tactics described above and CCP information provided by the developers of ENSEMBLE.

```
OPTIMIZING LAYER Pt2pt
   FOR EVENT      DnM(ev, hdr)
   AND STATE      s_pt2pt
   ASSUMING       getType ev = ESend  ∧  not (getPeer ev = ls.rank)
YIELDS HANDLERS   dn ev (Full (Data (Iq.hi
                             (Arraye.get s_pt2pt.sends (getPeer ev))), hdr))
   AND UPDATES    Iq.add (Arraye.get s_pt2pt.sends (getPeer ev)) (getIov ev) hdr

BY InitReconf

GIVEN:  s_pt2pt, Msg, Args, ls, vs, ev, msg, args
9. (getType ev) = ESend
10. not (getPeer ev = ls.rank)
⊢  let (s0,hdlr) = convert Functional Pt2pt.l args (ls, vs)
     in  hdlr(s_pt2pt, DnM (ev, hdr))
 =  Iq.add (Arraye.get s_pt2pt.sends (getPeer ev)) (getIov ev) hdr;
    dn ev (Full (Data (Iq.hi (Arraye.get s_pt2pt.sends (getPeer ev))), hdr))

BY EvaluatingCodeStructure

⊢   match getType ev with
      ESend  -> ..... Code hidden ....
    | _      -> ..... Code hidden ....
 =  Iq.add (Arraye.get s_pt2pt.sends (getPeer ev)) (getIov ev) hdr;
    dn ev (Full (Data (Iq.hi (Arraye.get s_pt2pt.sends (getPeer ev))), hdr))

BY UsingAssumptions [9;10]

⊢   Iq.add (Arraye.get s_pt2pt.sends (getPeer ev)) (getIov ev) hdr;
    dn ev (Full (Data (Iq.hi (Arraye.get s_pt2pt.sends (getPeer ev))), hdr))
 =  Iq.add (Arraye.get s_pt2pt.sends (getPeer ev)) (getIov ev) hdr;
    dn ev (Full (Data (Iq.hi (Arraye.get s_pt2pt.sends (getPeer ev))), hdr))

BY Equality
```

In top-down sequent proofs tactics refine a sequent and write the remaining subgoal sequent(s) below the tactic name, which is recorded next to the BY*. Variable declarations in hypotheses are abbreviated and hypotheses that do not change are not repeated. A proof is complete if there are no more subgoals. In the* NUPRL *system, details of tactic executions may be revealed on demand.*

Table 5. Optimization theorem for `Pt2pt` wrt. point-to-point sending (snapshot)

4.4 Stack Optimization

In contrast to micro-protocol layers, application protocol stacks cannot be optimized a priori, as thousands of possible configurations can be generated with the ENSEMBLE toolkit. Since the application developer has little or no knowledge about ENSEMBLE's code, the process of optimizing an application stack has to be completely automatic. We have developed a tool for optimizing arbitrary protocol stacks that uses formal *composition theorems* to compose optimization theorems for individual micro-protocols into optimization theorems for protocol stacks (c.f. Figure 5).

Composition theorems give an abstract, yet precise description of the effect of applying ENSEMBLE's stack composition mechanism to common combinations of fast-paths, such as *linear traces* (events pass straight through a layer), *bouncing events* (events generate callback events), and *trace splitting* (events cause several events to be emitted from a layer) — both for up- and down-going input events.

The following composition theorem, for instance, expresses the obvious effect of composing *down-going linear* fast-paths: if an event passes straight down through the upper layer and then through the lower one, then it passes straight through the composed layers (`Upper` ||| `Lower`) as well. The state update for the combined layer is the combination of the individual updates.

```
    OPTIMIZING LAYER Upper
       FOR EVENT      DnM(ev,hdr)    AND STATE   s_upper
    YIELDS HANDLERS   dn ev hdr1     AND UPDATES stmt1

∧   OPTIMIZING LAYER Lower
       FOR EVENT      DnM(ev,hdr1)   AND STATE   s_lower
    YIELDS HANDLERS   dn ev hdr2     AND UPDATES stmt2

⇒   OPTIMIZING LAYER Upper|||Lower
       FOR EVENT      DnM(ev,hdr)    AND STATE   (s_upper,s_lower)
    YIELDS HANDLERS   dn ev hdr2     AND UPDATES stmt1; stmt2
```

Again, we have used formal abbreviations to make the presentation of the composition theorem in the NUPRL system easier to comprehend. With all the abbreviations unfolded the theorem would read as follows.

```
⊢ ∀State_l,State_u,Stacks,Msq,Args:𝕌. ∀Lower,Upper:Stacks.
   ∀s_l:State_l. ∀s_u:State_u. ∀args:Args. ∀ls:View.local. ∀vs:View.state.
    ∀ev:Event.t. ∀stmt1,stmt2,dn:EXPR. ∀hdr,hdr1,hdr2:Hdr.
      let (s_0,hdlr) = convert Functional Upper args (ls,vs)
         in  hdlr(s_u, DnM(ev, hdr))
      = stmt1; dn ev hdr1
   ∧  let (s_0,hdlr) = convert Functional Lower args (ls,vs)
         in  hdlr(s_l, DnM(ev, hdr1))
      = stmt2; dn ev hdr2
   ⇒ let (s_0,hdlr) = convert Functional (compose Upper Lower) args (ls,vs)
         in  hdlr((s_u, s_l), DnM(ev, hdr))
      = stmt1;stmt2; dn ev hdr2
```

While the use of formal abbreviations makes the statement of a composition theorem appear almost trivial, its formal proof is complex, as it requires reasoning about the function `compose`, the actual OCAML code of ENSEMBLE's composition mechanism. The implementation of `compose` uses general recursion to handle the traffic of events between the individual layers, which in principle may circulate up and down through a stack several times before they leave.

By proving theorems about the result of applying this code to the most common combinations of fast-paths we lift the optimization process to a higher conceptual level: instead of reasoning about code, we reason about composition as such, using composition theorems as derived inference rules. Optimizing composed protocols is now a *single* reasoning step, while purely tactic-based optimizations would have to apply thousands of simplification steps to the code to achieve the same result.

Like optimization theorems for individual micro-protocols, composition theorems are included in the optimization tool, as they do not depend on a particular application stack. We have proven 12 composition theorems for the most common combinations of fast-paths for up- and down-going events. As a consequence, optimization theorems for protocol stacks can be created and proven automatically.

We have implemented a tactic `CreateOptStack`, which takes the names of the micro-protocols in the application stack as input, creates statements of optimization theorems for the complete stack, proves them correct, and stores the theorems in the NUPRL library. Optimization theorems are created separately for the four

```
OPTIMIZING LAYER  Pt2pt ||| Mnak ||| Bottom
  FOR EVENT  DnM (ev, hdr)
  AND STATE  (s_pt2pt, s_mnak, s_bottom)
  ASSUMING   getType ev = ESend ∧ not(getPeer ev = ls.rank) ∧ s_bottom.enabled
YIELDS HANDLERS dn ev (Full_nohdr (Full_nohdr (Full
                      (Data (Iq.hi (Arraye.get s_pt2pt.sends (getPeer ev))), hdr))))
   AND UPDATES  Iq.add (Arraye.get s_pt2pt.sends (getPeer ev)) (getIov ev) hdr

BY InitStackReconf

GIVEN:  s_pt2pt, s_mnak, s_bottom, Hdr, Args, ls, vs, ev, hdr, args
11. (getType ev) = ESend
12. not (getPeer ev = ls.rank)
13. (s_bottom.enabled)
⊢ OPTIMIZING LAYER Pt2pt ||| Mnak ||| Bottom
     FOR EVENT DnM (ev, hdr)
     AND STATE (s_pt2pt, s_mnak, s_bottom)
  YIELDS HANDLERS dn ev (Full_nohdr (Full_nohdr (Full
                        (Data (Iq.hi (Arraye.get s_pt2pt.sends (getPeer ev))), hdr))))
     AND UPDATES  Iq.add (Arraye.get s_pt2pt.sends (getPeer ev)) (getIov ev) hdr

BY QuoteLayerVerifs

14. OPTIMIZING LAYER Bottom
      FOR EVENT DnM (ev, Full_nohdr (Full
                         (Data (Iq.hi (Arraye.get s_pt2pt.sends (getPeer ev))), hdr)))
      AND STATE s_bottom
    YIELDS HANDLERS dn ev (Full_nohdr (Full_nohdr (Full
                          (Data (Iq.hi (Arraye.get s_pt2pt.sends (getPeer ev))), hdr))))
       AND UPDATES  ()
15. OPTIMIZING LAYER Mnak
      FOR EVENT DnM (ev,Full (Data (Iq.hi (Arraye.get s_pt2pt.sends (getPeer ev))),hdr))
      AND STATE s_mnak
    YIELDS HANDLERS dn ev (Full_nohdr (Full
                          (Data (Iq.hi (Arraye.get s_pt2pt.sends (getPeer ev))), hdr)))
       AND UPDATES  ()
16. OPTIMIZING LAYER Pt2pt
      FOR EVENT DnM (ev, hdr)
      AND STATE s_pt2pt
    YIELDS HANDLERS dn ev (Full
                          (Data (Iq.hi (Arraye.get s_pt2pt.sends (getPeer ev))), hdr))
       AND UPDATES  Iq.add (Arraye.get s_pt2pt.sends (getPeer ev)) (getIov ev) hdr

BY Repeat ComposeReconfigurations

14. OPTIMIZING LAYER Pt2pt ||| Mnak ||| Bottom
      FOR EVENT DnM (ev, hdr)
      AND STATE (s_pt2pt, s_mnak, s_bottom)
    YIELDS HANDLERS dn ev (Full_nohdr (Full_nohdr (Full
                          (Data (Iq.hi (Arraye.get s_pt2pt.sends (getPeer ev))), hdr))))
       AND UPDATES  Iq.add (Arraye.get s_pt2pt.sends (getPeer ev)) (getIov ev) hdr

BY Hypothesis 17
```

Table 6. Generated optimization theorem for the stack `Pt2pt ||| Mnak ||| Bottom`

fundamental cases – down- and up-going events for both point-to-point sending and broadcasting – to allow for the creation of independent bypass code fragments.

The statement of the optimization theorem for a protocol stack has the same basic form as one for micro-protocols. To create it, the tactic composes the statements of the corresponding layer optimization theorems top down. It matches in- and outgoing events according to the statement of the corresponding composition theorem and accumulates the (instantiated) CCPs, states, and updates accordingly.

To prove the theorem, the tactic partially unfolds the formal abbreviations and moving the CCP's into the hypothesis list. It then instantiates the optimization theorems of the micro-protocols in the stack with the actual input event that will enter them. It then applies, step-by-step, the appropriate composition theorems to compose the fast-paths through the micro-protocols into one for the stack and finally shows that this is exactly the result that had to be proven.

The proof is stored in the NUPRL library with only these macro steps visible at the top-level. Thus users inspecting it will first see the basic line of reasoning, which is particularly important for realistic application stacks with 10–30 protocol layers. Further details of the corresponding tactic executions may be revealed on demand.

As an example, we illustrate the optimization of the stack `Pt2pt` ||| `Mnak` ||| `Bottom` with respect to point-to-point sending. To state the optimization theorem, `CreateOptStack` looks up the corresponding optimization theorems of `Pt2pt` (Table 5), `Mnak`, and `Bottom`:

```
OPTIMIZING LAYER Mnak
   FOR EVENT      DnM(ev, hdr)
   AND STATE      s_mnak
   ASSUMING       getType ev = ESend
YIELDS HANDLERS   dn ev (Full_nohdr hdr)
   AND UPDATES    ()
```

```
OPTIMIZING LAYER Bottom
   FOR EVENT      DnM(ev, hdr)
   AND STATE      s_bottom
   ASSUMING       getType ev = ESend
                  ∧ s_bottom.enabled
YIELDS HANDLERS   dn ev (Full_nohdr hdr)
   AND UPDATES    ()
```

Since all three optimizations show a linear behavior, they can be combined according to the composition theorem for down-going linear traces. The input event for the three-layer stack is the input of `Pt2pt`. The layer states are composed into a tuple and the CCP's are accumulated by conjunction. The handler code results from matching the input and output events of adjacent micro-protocols. The resulting update is composed from the individual updates. The generated optimization theorem and its proof are shown in Table 6.

4.5 Header Compression

Optimization theorems do not only describe a fast-path through a protocol stack but also provide the means for an additional optimization that cannot be achieved by partial evaluation or related techniques. They state exactly which headers are added to a typical data message by the sender's stack and how the receiver's stack processes these headers in the respective layers. As most of the header fields are now fixed, we only have to transmit the header fields that may vary. In the stack `Pt2pt` ||| `Mnak` ||| `Bottom`, for instance, point-to-point sending creates the header

```
Full_nohdr (Full_nohdr (Full (Data
                       (Iq.hi(Arraye.get s_pt2pt.sends (getPeer ev)), hdr))))
```

in which only the italicized field contains essential information. Transmitting only this field will reduce the net load and improve the performance of communication.

For this purpose, we generate code for *compressing* and *expanding* headers, and *wrap* the protocol stack with these two functions (using ENSEMBLE's built-in function `wrap_msg`). Both functions are generated automatically by considering the free variables of the events in the optimization theorems. For the stack `Pt2pt` ||| `Mnak` ||| `Bottom` we generate the following functions

```
let compress hdr = match hdr with
    Full_nohdr (Full_nohdr (Full (Data seqno, hdr)))  -> OptSend(seqno, hdr)
  | Full_nohdr (Full (Data (seqno), Full_nohdr hdr)) -> OptCast(seqno, hdr)
  | hdr                                              -> Normal(hdr)
let expand hdr = match hdr with
    OptSend(seqno, hdr) -> Full_nohdr (Full_nohdr (Full (Data seqno, hdr)))
  | OptCast(seqno, hdr) -> Full_nohdr (Full (Data (seqno), Full_nohdr hdr))
  | Normal(hdr)         -> hdr
```

We then optimize the code of the wrapped protocol stack using the same methodology as before. We have provided generic *compression* and *expansion* theorems, which describe the outcome of optimizing a wrapped stack relatively to the result of optimizing a regular stack. We use them to generate optimization theorems for the wrapped stack from those of the regular stack. This step is fully automated.

For point-to-point sending in the stack `Pt2pt` ||| `Mnak` ||| `Bottom`, for instance, combining fast-path optimization with compression leads to the following theorem, which again uses formal abbreviations to make its presentation more accessible.

```
OPTIMIZING LAYER Pt2pt ||| Mnak ||| Bottom  WRAPPED WITH COMPRESSION
   FOR EVENT     DnM(ev, hdr)
   AND STATE     (s_pt2pt, s_mnak, s_bottom)
   ASSUMING      getType ev = ESend  ∧  not (getPeer ev = ls.rank)
                                     ∧  s_bottom.enabled
YIELDS HANDLERS  dn ev (OptSend (Iq.hi
                        (Arraye.get s_pt2pt.sends (getPeer ev))), hdr)
  AND UPDATES    Iq.add (Arraye.get s_pt2pt.sends (getPeer ev))
                        (getIov ev) hdr
```

Integrating compression into the optimization process will always lead to an improvement in the common case, because the optimized code will directly generate (or analyze) events with compressed headers, instead of creating a full header first and compressing it afterwards. Only for the non-common case there will be a slight (but hardly measurable) overhead, as the functions `compress` and `expand` have to be executed explicitly.

4.6 Code Generation

The above optimization steps describe logical operations within the NUPRL system. In a final step, their results are converted into OCAML code that can be compiled and linked to the rest of the communication system. To generate this code, we compose the code fragments from the four optimization theorems into a single program, which also delays state updates until events have been sent or delivered, using the CCPs as conditionals that select either one of the fast-paths or the original stack. The bypass code is wrapped by code fragments that convert it into a module, which then can be compiled and linked to ENSEMBLE without further modifications.

We have developed a tactic `Optimize` that combines stack optimization, header compression, code generation, and exporting the result into the ENSEMBLE source tree into a single operation. Given a list of names of the micro-protocols in the protocol stack it takes less than 30 seconds to generate the optimized code and prove it to be equivalent to the original stack. In the process, it creates 8 stack optimization theorems (with and without compression) as well as 28 abstractions representing the new code module.

Experiments in (Liu *et al.*, 1999) have shown that in common applications with 10 or more layers the optimized code is significantly more efficient than the original ENSEMBLE application stack. Although the synthesized code has to be integrated into the functional version of ENSEMBLE, which typically is about 50% slower than the imperative one, it outperforms the best version of ENSEMBLE by a factor of 3 to 5.

5 Verification and Formal Design

In the previous sections we have focused on reasoning about the code of modular systems and on logic-based tools for optimizing their performance. We will now briefly address the remaining aspects of a formal infrastructure for building reliable, high-performance networks, namely verification and formal design.

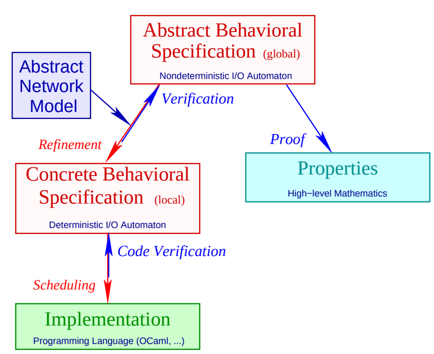
5.1 Specification and Correctness

The goal of specification is to give a precise description of a system and to define and document its features. Specifications can be used to guide the configuration of application systems from modules, to support the design of new modules, and to determine whether an implementation is *correct*.

Specifications range from specifying the *behavior* of a system to specifying its *properties*. Both kinds of specifications are important. Properties describe the system at the highest level while behavioral specifications describe how to implement the properties. Behavioral specifications can be either *concrete* or *abstract*. Abstract specifications are non-deterministic descriptions of a system's global behavior. Concrete specifications give deterministic descriptions of a system's components and can easily be mapped onto *executable code*.

As example consider a FIFO network, which is characterized by the property that *messages are received in the same order in which they were sent*. An abstract behavioral specification would introduce a global queue of events in transit and state that *messages may be appended to the end of the queue and be removed from its beginning*. A concrete behavioral specification would describe a protocol that attaches sequence numbers to messages and require that *incoming messages whose sequence number is too big will be buffered*. At the lowest level we find the implementation of this protocol, for instance as Ensemble's `Pt2pt` module.

The relation between the four levels of specification can be pictured as follows. Properties of an abstract specification are derived by *proof*. A concrete specification is derived from the abstract specification by *refinement*, which involves designing a protocol that implements the abstract requirements with respect to some abstract network model. The implementation is linked to the concrete specification by *scheduling* the order of actions and coding them in a specific programming language.



Introducing several levels of abstraction makes it feasible to prove properties of a system's implementation or to derive an implementation from properties – establishing a direct link between properties and implementation would be much harder. Formal verification and design of networked systems in Nuprl therefore requires a representation of all four level is Nuprl's type theory.

For the highest level, the abstract system properties, we have developed a formal model of communication (Bickford *et al.*, 2001a; Bickford *et al.*, 2001b; Bickford *et al.*, 2001c) that enables us to reason in NUPRL about properties of global traces of events such as reliability, confidentiality, message ordering, etc. Abstract and concrete system specifications are expressed in terms of non-deterministic and deterministic *IO-automata (IOA)* (Lynch, 1996), which are abstractions of the state-event machines implicitly used in the descriptions of network protocols. A type-theoretic representation of IO-automata has been developed in (Hickey *et al.*, 1999; Bickford & Hickey, 1999). The implementation level of ENSEMBLE is the programming language OCAML (Leroy, 2000), whose representation is discussed in Section 3.

A formal verification exploits the above relationship between the four levels. As in the case of fast-path optimizations, a compositional approach is taken: individual micro-protocols are verified independently and the verification of protocol stacks is based on IOA composition, which is proven to preserve all safety properties of the components. A good example of such a proof can be found in (Hickey *et al.*, 1999), which demonstrates the correctness of one of Ensemble's total ordering protocols and located a subtle bug in the original implementation.

5.2 Formal Design

Formal methods can have a large impact when being engaged at the earliest stages of design and implementation. At this stage it is possible to state assumptions and goals that drive the system design and to use proof environments to clarify these goals, to explore ideas, and to detect flaws in the design before it is being coded.

In (Liu *et al.*, 2001; Bickford *et al.*, 2001a; Bickford *et al.*, 2001c) we show how the NUPRL system has contributed to the design and implementation of a verifiably correct adaptive network protocol for ENSEMBLE. The protocol was realized as a hybrid protocol that *switches* between specialized protocols and formally proven correct with the NUPRL system. In the process we have developed a characterization of communication properties that can be preserved by dynamic switching. We have introduced the concept of *meta-properties* to abstractly describe switchable properties and have shown that six meta-properties are sufficient for protocols to work correctly under a switch. We also have characterized a *switch-invariant* that an implementation of the switch has to satisfy to preserve switchable properties.

The verification efforts revealed hidden assumptions that are crucial for the correctness of the implementation and showed limitations for the use of such a generic protocol that might otherwise have gone unnoticed. This demonstrates that engaging proof systems such as NUPRL at the earliest stages of design and implementation adds value to all subsequent stages and creates valuable information needed for the maintenance and evolution of software.

6 Related Work

The CMU Fox project (Biagioni, 1994) uses an extension of Standard ML for building protocol stacks. Its broader goal is to investigate the extent to which high level languages like ML are suitable for systems programming.

Integrated Layer Processing (ILP) (Clark & Tennenhouse, 1990) is an approach to reduce the overhead of layered protocols (Abbott & Peterson, 1993). The Filter Fusion Compiler (FFC) (Proebsting & Watterson, 1996) implements ILP using partial evaluation, but has only been applied to very simple protocols. Furthermore, the code generated by FFC has to be hand-modified to get good performance.

The Esterel compiler (Castelluccia & Dabbous, 1996) is used to convert a protocol specification into a sequential finite automaton, from which efficient C code is generated. Esterel was used to specify and implement a large subset of the TCP protocol, but it does not scale easily to arbitrary protocol stacks. Furthermore it cannot formally ensure the correctness of the optimization or the protocol itself.

In operating system research there is related work on locating and optimizing common paths. Synthesis (Massalin, 1992) uses a run-time code generator to optimize the most frequently used kernel routines. (Pu *et al.*, 1995) describes work on optimizing Synthetix kernel functions by reducing the length of common paths.

HOL-ML (VanInwegen & Gunter, 1994; VanInwegen, 1996) is an encoding of a subset of SML and its dynamic semantics in HOL (Gordon & Melham, 1993). It formalizes the abstract syntax of SML expressions on the object level of HOL and defines the static and dynamic semantics by a collection of explicitly introduced inference rules. The main emphasis of HOL-ML was proving *properties of the programming language* within the framework of a formal theorem prover.

Filliâtre (Filliâtre, 1998; Filliâtre, 2002) developed a calculus for reasoning about functional programs with references in Coq (Dowek & et. al, 1991; Coq). The calculus is based on an ML-like model programming language that translates imperative programs into the functional calculus of inductive constructions using a memory model similar to ours. The focus of this work is exploring Floyd-Hoare style reasoning for a richer programming language within the framework of type theory.

The LOOP project (van den Berg *et al.*, 2000; van den Berg & Jacobs, 2001; Jacobs & Poll, 2001) provides a tool for translating Java classes into the formal theories of PVS (Owre *et al.*, 1996; PVS) and Isabelle/HOL (Paulson, 1990; Isabelle). It formalizes a subset of Java in a type-theoretical model of untyped memory cells (similar to our model of store), which are used for storing Java objects and arrays. The main emphasis of the project is the formal verification of Java programs.

Our representation of OCaml in Nuprl provides an infrastructure for reasoning about large-scale programs and for applying semantics-preserving optimizations. Nuprl abstractions and display forms enable us to represent the abstract syntax of OCaml programs, their operational semantics, and the original syntax in which they were written. Inference rules for reasoning about the static and dynamic semantics of OCaml are *derived* from this representation and operate entirely on the level of the programming language instead of revealing the underlying logic. This makes formal reasoning accessible to programmers and allows exporting the result of formal transformations back into the programming environment.

A variety of formal systems has been used for verifying and synthesizing hardware and software systems. *Model checking* (McMillan, 1993; Manna & Pnueli, 1995; Clarke *et al.*, 1999) has been very successful in the verification of hardware and finite state software systems. Numerous systems (Cleaveland & et. al, 1994; Dill, 1996; Holzmann, 1997) have been used in a variety of case studies. But so far they are restricted to finite state systems and of limited value for reasoning about the complete code of real distributed systems. *Deductive model checking* (Finkbeiner *et al.*, 1998; Sipma *et al.*, 1999) combines model checking with automated deduction for the verification of infinite-state and real-time systems. Although this approach is very promising, it is not yet applicable to distributed systems.

There have been numerous applications of ISABELLE (Paulson, 1990; Paulson, 1998; Paulson, 1999), PVS (Lincoln & Rushby, 1993; Owre *et al.*, 1996; Rushby, 1994; Rushby, 1997; Rushby, 1999), and ACL2 (Kaufmann *et al.*, 2000; ACL2) to verifying abstract communication protocols. Few projects, however, deal with the actual code of real-world systems.

The KIDS system (Smith, 1991) and its successor SPECWARE (Srinivas & Jüllig, 1995) support the modular construction of formal specifications and their refinement into executable code. These systems have been successful in practical applications (Smith & Parra, 1993; Gomes *et al.*, 1996; Blaine *et al.*, 1998; Westfold & Smith, 2001) but are currently limited to the development of sequential algorithms.

7 Conclusion

We have described an embedding of the ENSEMBLE group communication toolkit to the NUPRL proof development system that is based on a type-theoretical semantics of ENSEMBLE's implementation language OCAML. The formal link between ENSEMBLE and NUPRL provides an infrastructure for the application of logical inference techniques to the actual code of a modular, real-world system. Using this infrastructure we have shown how to build logic-based optimization tools that can significantly improve the performance of the already optimized ENSEMBLE system in concrete applications and are guaranteed not to introduce any errors. Our results show that logical methods for program synthesis, verification, and optimization can be made to scale effectively to large software systems.

This article extends preliminary work reported in (Kreitz, 1997; Kreitz *et al.*, 1998; Kreitz, 1999), which now has matured into pushbutton technology and is based on an advanced semantical model for OCAML, which allows for a type-theoretical representation of a larger fragment of the programming language.

Although we chose a limited application domain – all ENSEMBLE configurations are stacks of micro-protocols, which can be considered state-event machines – we believe that our approach could generalize and scale to more general configuration and component types. We believe that the following ingredients were key to the success of our approach:

1. Using small and simple system components, which are easier to reason about.
2. Using a well-defined configuration operation on components.
3. Using a mostly functional implementation of components in a language with a formal semantics, which allows formal manipulations.
4. Using and continuously expanding a tactical proof system for a rich specification language such as NUPRL, which makes verification and optimization techniques scale and more accessible to system developers.
5. Using several layers of formal abstraction, libraries of verified formal knowledge, and compositional reasoning, which makes formal techniques independent from a particular application domain.
6. Using a collaboration between systems and theorem proving groups, as the joint expertise is required for making formal methods apply to real systems.

We believe that it may be possible to use our approach in other complex systems such as file systems, atomic transaction protocols, optimizing compilers, and perhaps eventually an entire operating system kernel, as the formal techniques described here can be applied to other modular systems whose components and composition mechanisms can be described semantically.

We also hope to elaborate our optimization technique into one that would allow us to detect common combinations at run-time, and generate the optimized code dynamically, using layer optimization theorems for all possible bypass paths. We can then make use of Ensemble's support for dynamically loading layers and switching protocol stacks on the fly (van Renesse *et al.*, 1999).

We also intend to extend our work on formalizing the semantics of programming languages. Our methods would have a wider impact if we could express the type theoretical semantics of OCAML's classes and object and also target other back-end languages such as JAVA, using ideas developed for the LOOP project (van den Berg *et al.*, 2000) and preliminary work in (Attalli *et al.*, 1998; Naumov, 1998). This may, however, require expanding our logical foundations, as type theories do not yet offer sufficient support for objects, methods, classes, and inheritance.

Finally we will continue our research on the verification and formal design of distributed systems. We plan to refine and extend our formal models to include reasoning about embedded systems with bandwidth and resource limitations, and time constraints.

Acknowledgements

The work reported in this paper would not have been possible without the help and inspiration of colleagues – Mark Bickford, Ken Birman, Robert Constable, Richard Eaton, Mark Hayden, Jason Hickey, Xiaoming Liu, Lori Lorigo, and Robbert Van Renesse – from both the systems and theorem proving groups at Cornell.

This work is supported in part by ARPA/RADC grants F30602-98-2-0198 and F30602-99-1-0532, and ARPA/ONR grant N00014-01-0765.

References

Aagaard, M., & Leeser, M. (1993). Verifying a logic synthesis tool in Nuprl. *Pages 72–83 of:* Bochmann, G., & Probst, D. (eds), *Workshop on Computer-Aided Verification*. LNCS 663. Springer Verlag.

Abbott, M., & Peterson, L. (1993). Increasing network throughput by integrating protocol layers. *IEEE/ACM Transactions on Networking*, **1**(5), 600–610.

ACL2. *ACL2 home page*. `http://www.cs.utexas.edu/users/moore/acl2`.

ALFA. *ALFA home page*. `http://www.cs.chalmers.se/~hallgren/Alfa`.

Allen, Stuart, Constable, Robert, Eaton, Richard, Kreitz, Christoph, & Lorigo, Lori. (2000). The Nuprl open logical environment. *Pages 170–176 of:* McAllester, D. (ed), *17th Conference on Automated Deduction*. LNAI 1831. Springer Verlag.

Altenkirch, Thorsten, Gaspes, Veronica, Nordström, Bengt, & von Sydow, Björn. (1994). *A user's guide to ALF*. University of Göteborg.

Andrews, Peter B., Bishop, Matthew, Issar, Sunil, Nesmith, Dan, Pfenning, Frank, & Xi, Hongwei. (1996). TPS: A theorem proving system for classical type theory. *Journal of automated reasoning*, **16**(3), 321–353.

Attalli, Isabelle, Caromel, Denise, & Russo, Marjorie. (1998). A formal executable semantics for Java. *OOPSLA'98 workshop on the formal underpinnings of Java*.

Biagioni, E. (1994). A structured TCP in Standard ML. *Pages 36–45 of: ACM SIGCOMM conference*.

Bickford, Mark, & Hickey, Jason. (1999). Predicate transformers for infinite-state automata in Nuprl type theory. *Irish formal methods workshop*.

Bickford, Mark, Kreitz, Christoph, van Renesse, Robbert, & Constable, Robert. (2001a). An experiment in formal design using meta-properties. *Pages 100–107 of:* Lala, J., Maughan, D., McCollum, C., & Witten, B. (eds), *Darpa Information Survivability Conference and Exposition II*, vol. II. IEEE Computer Society Press.

Bickford, Mark, Kreitz, Christoph, & van Renesse, Robbert. (2001b). *Formally verifying hybrid protocols with the* NUPRL *logical programming environment*. Tech. rept. Cornell CS:2001-1839. Cornell University. Department of Computer Science.

Bickford, Mark, Kreitz, Christoph, van Renesse, Robbert, & Liu, Xiaoming. (2001c). Proving hybrid protocols correct. *Pages 105–120 of:* Boulton, Richard, & Jackson, Paul (eds), *14th International Conference on Theorem Proving in Higher Order Logics*. LNCS 2152. Springer Verlag.

Birman, Ken, Constable, Robert, Hayden, Mark, Hickey, Jason, Kreitz, Christoph, van Renesse, Robbert, Rodeh, Ohad, & Vogels, Werner. (2000). The Horus and Ensemble projects: Accomplishments and limitations. *Pages 149–160 of: Darpa Information Survivability Conference and Exposition*. IEEE Computer Society Press.

Birman, K.P., & van Renesse, Robbert. (1994). *Reliable Distributed Computing with the Isis Toolkit*. IEEE Computer Society Press.

Blaine, Lee, Gilham, Li-Mei, Liu, Junbo, Smith, Douglas R., & Westfold, Stephen. (1998). Planware – domain-specific synthesis of high-performance schedulers. *Pages 270–280 of: 13th Automated Software Engineering Conference*. IEEE Computer Society Press.

Castelluccia, C., & Dabbous, W. (1996). Generating efficient protocol code from an abstract specification. *Pages 60–72 of: ACM SIGCOMM conference*.

Clark, D., & Tennenhouse, D. (1990). Architectural consideration for a new generation of protocols. *Pages 200–208 of: ACM SIGCOMM conference*.

Clarke, E. M., Grumberg, O., & Peled, D. (1999). *Model checking*. MIT Press.

Cleaveland, R., & et. al. (1994). The concurrency factory – practical tools for the specification, simulation, verification, and implementation of concurrent systems. *Pages 75–90* *of:* Blelloch, G.E., Chandy, K.M., & Jagannathan, S. (eds), *Specification of parallel algorithms*.

Constable, Robert L. (1998). Types in logic, mathematics, and programming. *Pages 684–786 of:* Buss, S. R. (ed), *Handbook of Proof Theory*. Elsevier Science Publishers.

Constable, Robert L., & Hickey, Jason. (2000). Nuprl's Class Theory and its Applications. *Pages 91–116 of:* Bauer, Friedrich L., & Steinbrueggen, Ralf (eds), *Foundations of secure computation*. NATO ASI Series, Series F: Computer & System Sciences. IOS Press.

Constable, Robert L., Allen, Stuart F., Bromley, H. Mark, Cleaveland, W. Rance, Cremer, J. F., Harper, Robert W., Howe, Douglas J., Knoblock, Todd B., Mendler, Nax Paul, Panangaden, Prakash, Sasaki, Jim T., & Smith, Scott F. (1986). *Implementing mathematics with the Nuprl proof development system*. Prentice Hall.

Coq. *Coq home page*. `http://pauillac.inria.fr/coq/coq-eng.html`.

de Rauglaudre, Daniel. (2000). *Camlp4 version 3.00*. Institut National de Recherche en Informatique et en Automatique.

Dill, David. (1996). The Murphi verification system. *Pages 390–393 of:* Alur, R., & Henzinger, T. (eds), *Computer Aided Verification*. LNCS 1102. Springer Verlag.

Dowek, G., & et. al. (1991). *The Coq proof assistant user's guide*. Institut National de Recherche en Informatique et en Automatique. Report RR 134.

Engler, D., & Kaashoek, M. (1996). DPF: Fast, flexible message demultiplexing using dynamic code generation. *Pages 53–59 of: ACM SIGCOMM conference*.

Ensemble. *Ensemble home page*.
`http://www.cs.cornell.edu/Info/Projects/Ensemble/index.htm`.

Filliâtre, J.-C. (1998). Proof of imperative programs in type theory. *International Workshop, Types '98*, LNCS 1657. Springer Verlag.

Filliâtre, J.-C. (2002). Verification of non-functional programs using interpretations in type theory. *Journal of Functional Programming*.

Finkbeiner, B., Manna, Z., & Sipma, H. (1998). Deductive verification of modular systems. *Pages 239–275 of: Compositionality: The Significant Difference*. LNCS 1536. Springer.

Gomes, Carla P., Smith, Douglas R., & Westfold, Stephen J. (1996). Synthesis of schedulers for planned shutdowns of power plants. *Pages 12–20 of: Eleventh Knowledge-Based Software Engineering Conference*. IEEE Computer Society Press.

Gordon, Michael, & Melham, T. (1993). *Introduction to HOL: a theorem proving environment for higher-order logic*. Cambridge University Press.

Hayden, Mark. (1998). *The Ensemble system*. Ph.D. thesis, Cornell University. Department of Computer Science.

Hayden, Mark, & Rodeh, Ohad. (2001). *Ensemble reference manual*.
`http://www.cs.cornell.edu/Info/Projects/Ensemble/ref.pdf`.

Hickey, J., Lynch, N., & van Renesse, R. (1999). Specifications and proofs for Ensemble layers. *Pages 119–133 of:* Cleaveland, R. (ed), *5^{th} International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. LNCS 1579. Springer.

Hickey, Jason. (1996). Formal objects in type theory using very dependent types. *Foundations of Object Oriented Languages 3*. Williams College.

Hickey, Jason. (2001). *The MetaPRL logical programming environment*. Ph.D. thesis, Cornell University. Department of Computer Science.

Hickey, Jason, & Nogin, Aleksey. (2000). Fast tactic-based theorem proving. *Pages 252–266 of:* Harrison, J., & Aagaard, M. (eds), *13th International Conference on Theorem Proving in Higher Order Logics*. LNCS 1869. Springer Verlag.

HOL. *HOL home page*. `http://www.cl.cam.ac.uk/Research/HVG/HOL`.

Holzmann, G. J. (1997). The model checker SPIN. *IEEE Transactions on Software Engineering*, **23**(5), 279–295.

Howe, D.J. (1996). Importing mathematics from HOL into Nuprl. *Pages 267–282 of:* von Wright, J., Grundy, J., & Harrison, J. (eds), *THeorem Proving in Higher Order Logics*. LNCS 1125. Springer Verlag.

Howe, Douglas J. (1987). The computational behaviour of Girard's paradox. *Pages 205–214 of:* Gries, David (ed), *Second Annual Symposium on Logic in Computer Science*. IEEE Computer Society Press.

Isabelle. *Isabelle home page*. `http://www.cl.cam.ac.uk/Research/HVG/Isabelle`.

Jackson, Paul B. (1994). Exploring abstract algebra in constructive type theory. Bundy, Alan (ed), *12th Conference on Automated Deduction*. LNAI 814. Springer Verlag.

Jacobs, B., & Poll, E. (2001). A logic for the Java modeling language JML. *Pages 284–299 of:* Hussmann, H. (ed), *Fundamental Approaches to Software Engineering*. LNCS 2029. Springer Verlag.

Kaufmann, Matt, Manolios, Panagiotis, & Moore, J Strother. (2000). *Computer-Aided Reasoning: ACL2 case studies*. Kluwer Academic Publishers.

Kopylov, Alexei. (2000). *Dependent intersection: A new way of defining records in type theory*. Tech. rept. TR 2000-1809. Cornell University. Department of Computer Science.

Kreitz, Christoph. 1997 (June). *Formal reasoning about communication systems I: Embedding ML into type theory*. Tech. rept. TR97-1637. Cornell University. Department of Computer Science.

Kreitz, Christoph. (1999). Automated fast-track reconfiguration of group communication systems. *Pages 104–118 of:* Cleaveland, R. (ed), *5th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. LNCS 1579. Springer.

Kreitz, Christoph, Hayden, Mark, & Hickey, Jason. (1998). A proof environment for the development of group communication systems. *Pages 317–331 of:* Kirchner, C., & Kirchner, H. (eds), *15th Conference on Automated Deduction*. LNAI 1421. Springer.

Leino, K. Rustan M. (2000). Extended static checking: A ten-year perspective. *Pages 157–175 of:* Wilhelm, R. (ed), *Informatics: 10 years back, 10 years ahead*. LNCS 2000. Springer Verlag.

Leroy, Xavier. (2000). *The Objective Caml system release 3.00*. Institut National de Recherche en Informatique et en Automatique.

Lincoln, Patrick, & Rushby, John. (1993). A formally verified algorithm for interactive consistency under a hybrid fault model. *Pages 402–411 of: 23rd Fault-Tolerant Computing Symposium*.

Liu, Xiaoming, Kreitz, Christoph, van Renesse, Robbert, Hickey, Jason, Hayden, Mark, Birman, Kenneth, & Constable, Robert. (1999). Building reliable, high-performance communication systems from components. *Pages 80–92 of: 17th ACM Symposium on Operating Systems Principles*. Operating Systems Review, vol. 34, no. 5.

Liu, Xiaoming, van Renesse, Robbert, Bickford, Mark, Kreitz, Christoph, & Constable, Robert. (2001). Protocol switching: Exploiting meta-properties. *Pages 37–42 of:* Rodrigues, Luis, & Raynal, Michel (eds), *International Workshop on Applied Reliable Group Communication*. IEEE Computer Society Press.

Lynch, Nancy. (1996). *Distributed algorithms*. Morgan Kaufmann.

Manna, Z., & Pnueli, A. (1995). *Temporal verification of reactive systems*. Springer Verlag.

Martin-Löf, Per. (1984). *Intuitionistic Type Theory*. Studies in Proof Theory Lecture Notes, vol. 1. Napoli: Bibliopolis.

Massalin, H. (1992). *Synthesis: An efficient implementation of fundamental operating system services*. Ph.D. thesis, Computer Science Department, Columbia University.

McMillan, K. L. (1993). *Symbolic model checking*. Kluwer Academic Publishers.

MetaPRL. *Metaprl home page*. `http://metaprl.org`.

Murthy, Chetan R., & Russell, James R. (1990). A constructive proof of Higman's lemma. *Pages 257–267 of:* Mitchell, John C. (ed), *Fifth Annual Symposium on Logic in Computer Science*. IEEE Computer Society Press.

Naumov, Pavel. (1998). *Formalizing reference types in Nuprl*. Ph.D. thesis, Cornell University. Department of Computer Science.

Nuprl. *Nuprl home page*. `http://www.nuprl.org`.

Owre, S., Rajan, S., Rushby, J. M., Shankar, N., & Srivas, M. K. (1996). PVS: Combining specification, proof checking and model checking. *Pages 411–414 of:* Alur, Rajeev, & Henzinger, Thomas A. (eds), *Computer-Aided Verification*. LNCS 1102. Springer Verlag.

Paulson, L. (1999). Inductive analysis of the internet protocol TLS. *ACM Transactions on Computer and System Security*, **2**(3), 332–351.

Paulson, Lawrence C. (1990). Isabelle: The next 700 theorem provers. *Pages 361–386 of:* Odifreddi, Piergiorgio (ed), *Logic and Computer Science*. Academic Press.

Paulson, Lawrence C. (1998). The inductive approach to verifying cryptographic protocols. *Journal of Computer Security*, **6**, 85–128.

Pfenning, Frank, & Schürmann, Carsten. (1999). Twelf—a meta-logical framework for deductive systems. *Pages 202–206 of:* Ganzinger, H. (ed), *16^{th} Conference on Automated Deduction*. LNAI 1632. Springer Verlag.

Pollack, Robert. (1994). *The theory of LEGO – a proof checker for the extendend calculus of constructions*. Ph.D. thesis, University of Edinburgh.

Proebsting, T., & Watterson, S. (1996). Filter fusion. *Pages 119–130 of: 23^{th} ACM Symposium on Principles of Programming Languages*.

Pu, C., Autrey, T., Black, A., Consel, C., Cowan, C., Inouye, J., Kethana, L., Walpole, J., & Zhang, K. (1995). Optimistic incremental specialization. *Pages 314–321 of: 15th ACM Symposium on Operating Systems Principles*.

PVS. *PVS home page*. `http://pvs.csl.sri.com`.

Rushby, John. (1994). A formally verified algorithm for clock synchronization under a hybrid fault model. *Pages 304–313 of: 13^{th} ACM Symposium on Principles of Distributed Computing*.

Rushby, John. (1997). Systematic formal verification for fault-tolerant time-triggered algorithms. *Pages 191–210 of:* Meadows, Catherine, & Sanders, William (eds), *Dependable Computing for Critical Applications: 6*. IEEE Computer Society Press.

Rushby, John. (1999). Systematic formal verification for fault-tolerant time-triggered algorithms. *IEEE Transactions on Software Engineering*, **25**(5), 651–660.

Schmitt, Stephan, Lorigo, Lori, Kreitz, Christoph, & Nogin, Alexey. (2001). JProver: Integrating connection-based theorem proving into interactive proof assistants. *Pages 421–426 of:* Gore, R., Leitsch, A., & Nipkow, T. (eds), *International Joint Conference on Automated Reasoning*. LNAI 2083. Springer Verlag.

Schneider, Fred B., Morrisett, Greg, & Harper, Robert. (2000). A language-based approach to security. *Pages 86–101 of:* Wilhelm, R. (ed), *Informatics: 10 years back, 10 years ahead*. LNCS 2000. Springer Verlag.

Sipma, H., Uribe, T., & Manna, Z. (1999). Deductive model checking. *Formal methods in system design*, **15**, 49–74.

Smith, Douglas R. (1991). KIDS — a knowledge-based software development system. *Pages 483–514 of:* Lowry, Michael R., & McCartney, Robert D. (eds), *Automating software design*. AAAI Press / The MIT Press.

Smith, Douglas R., & Parra, Eduardo A. (1993). Transformational approach to transportation scheduling. *Pages 60–68 of: 8th Knowledge-Based Software Engineering Conference.*

Srinivas, Y. V., & Jüllig, Richard. (1995). SPECWARE: Formal Support for composing software. *International Conference on the Mathematics of Program Construction.*

TPS. *TPS home page.* `http://gtps.math.cmu.edu/tps.html`.

van den Berg, J., & Jacobs, B. (2001). The LOOP compiler for Java and JML. *Pages 299–312 of:* Margaria, T., & Yi, W. (eds), *7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems.* LNCS 2031. Springer Verlag.

van den Berg, J., Huisman, M., Jacobs, B., & Poll, E. (2000). A type-theoretic memory model for verification of sequential Java programs. *Pages 1–21 of:* Bert, D., Choppy, C., & Mosses, P.D. (eds), *Recent Trends in Algebraic Development Techniques.* LNCS 1827. Springer Verlag.

van Renesse, R., Birman, K., & Maffeis, S. (1996). Horus: A flexible group communication system. *Communications of the ACM*, **39**(4), 76–83.

van Renesse, Robbert, Birman, Ken, Hayden, Mark, Vaysburg, Alexey, & Karr, David. (1999). Building adaptive systems using Ensemble. *Software: Practice and experience*, **28**(9), 963–979.

VanInwegen, Myra. (1996). *The machine-assisted proof of programming language properties.* Ph.D. thesis, University of Pennsylvania.

VanInwegen, Myra, & Gunter, Elsa. (1994). HOL-ML. *Pages 61–73 of: Higher Order Logic Theorem Proving and its Applications.* LNCS 780. Springer Verlag.

Westfold, S. J., & Smith, D. R. (2001). Synthesis of efficient constraint satisfaction programs. *Knowledge Engineering Reviews.*

A The Type-Theoretical Formalization of OCaml

The type-theoretical formalization of OCaml's semantics is based on the model described in Section 3.1. It represents OCaml expressions as functions that operate on a store s and an environment env, and return a value (or an exception) and a possibly updated store. Expressions may include patterns, which are represented as functions that take two expressions e_1 and e_2 and modify the environment of e_2 according to the result of matching e_1 against the pattern template. OCaml types are represented as (dependent) function types, which enables us to express OCaml typings by the membership relation of Nuprl's type theory.

Our embedding of OCaml into type theory is *shallow*: instead of describing the type of all possible OCaml expressions and defining an evaluation function for these terms, we describe OCaml expressions by Nuprl terms that directly represent their operational semantics. All tactics for manipulating OCaml expressions are based on derived inference rules that preserve the "OCaml-ness" of Nuprl terms, which makes sure that the internal representation will not be revealed.

Our formalization only covers OCaml programs that are *type-correct*, as these are the only expressions that actually have a semantics. It does not handle compile-time errors like detecting an attempt to match an integer against a list, but only run-time errors, which will result in raising exceptions. Our tools for importing OCaml code into Nuprl (Section 3.5) make sure that only programs accepted by OCaml's type checker will be translated into a type-theoretical representation.

Although the OCaml manual (Leroy, 2000) does not specify the evaluation order of subexpressions in OCaml programs, the evaluation order of a formal representation has to be fixed in order to guarantee a deterministic behavior of programs in formal reasoning. To ensure faithfulness with respect to the OCaml compiler our formalization usually encodes a right-most depth-first strategy.

In addition to the basic type theory of Nuprl (Table 1) our formalization of a type theoretical model of OCaml utilizes a variety of user-defined operations on lists, booleans (see Table 2), tables, numbers, etc. that are included in Nuprl's standard libraries, the definitions given in Section 3.1, and the following notions.

```
IDENT      ≡  Atom
LABEL      ≡  Atom
VALUE      ≡  Top
ENV        ≡  (IDENT × VALUE) List
ADDR       ≡  ℕ
STORE      ≡  (ADDR × VALUE) List
EXCEPTION  ≡  Atom
EXPR       ≡  s:STORE → env:ENV → (EXCEPTION + VALUE) × STORE
```

A.1 Expressions

The syntax and meaning of OCaml expressions is described in §6.7 of the OCaml manual (Leroy, 2000). As described above, we represent OCaml expressions as elements of the type `EXPR`, i.e. as functions of the form `λs,env.` *value*`,`*s'*, where *value* is the result of evaluating the expression and *s'* the updated store.

x	≡ λs,env. inr(env[x]),s
raise e	≡ λ↓[v,s_1/e,env]. inl(v), s
i	≡ ⌈i⌉
'chr'	≡ ⌈"chr"⌉
()	≡ ⌈Ax⌉
false	≡ ⌈ff⌉
true	≡ ⌈tt⌉
if e then e_1 else e_2	≡ λ↓[v,s_1/e,env]. (if v then e_1 else e_2) s_1 env
if e then e_1	≡ if e then e_1 else ()
[]	≡ ⌈[]⌉
e_1::e_2	≡ λ↓[v_2,v_1,s_1/e_2,e_1,env]. inr(⌈v_1⌉::⌈v_2⌉), s_1
[e_1; ..; e_n]	≡ e_1::...::e_n::[]
$cconstr$	≡ ⌈<"$cconstr$",()>⌉
$ncconstr$ e	≡ λ↓[v,s_1/e,env]. inr(<"$ncconstr$",⌈v⌉>, s_1
'tag	≡ ⌈<"tag",()>⌉
'tag e	≡ λ↓[v,s_1/e,env]. inr(<"tag",⌈v⌉>, s_1
(e_1,e_2)	≡ λ↓[v_2,v_1,s_1/e_2,e_1,env]. inr(<⌈v_1⌉,⌈v_2⌉>), s_1
e_1 e_2	≡ λ↓[v_2,v_1,s_1/e_2,e_1,env]. (v_1 ⌈v_2⌉) s_1 env
{}	≡ ⌈λfield.-1⌉
{f_1=e_1}⊗e_2	≡ λ↓[v_2,v_1,s_1/e_2,e_1,env]. let addr = if v_2 f_1 < 0 then NEW(s_1) else v_2 f_1 in inr(λf. if f = f_1 then $addr_1$ else v_2 f), s_1[addr←v_1]
{e with f_1=e_1; ..; f_n=e_n}	≡ {"f_1"=e_1}⊗..⊗{"f_n"=e_n}⊗e
{f_1=e_1; ..; f_n=e_n}	≡ {{} with f_1=e_1; ..; f_n=e_n}
e.f	≡ λ↓[v,s_1/e,env]. inr(!s_1[v "f"]), s_1
e_1.f <- e_2	≡ {"f_1"=e_1}⊗e_2; ()
ref e	≡ {contents=e}
!e	≡ e.contents
e_1 := e_2	≡ e_1.contents <- e_2
[| e_0; ..; e_n |]	≡ ⌈ref e_0::...::ref e_n::[]⌉
e_1.(e_2)	≡ λ↓[v_2,v_1,s_1/e_2,e_1,env]. !(v_1[v_2]) s_1 env
e_1.(e_2) <- e_3	≡ λ↓[v_3,v_2,v_1,s_1/e_3,e_2,e_1,env]. (v_1[v_2] := ⌈v_3⌉) s_1 env
"c_0..c_n"	≡ [|'c_0';..;'c_n'|]
e_1.[e_2]	≡ e_1.(e_2)
e_1.[e_2] <- e_3	≡ e_1.(e_2) <- e_3
e_1; e_2	≡ λ↓[v,s_1/e,env]. e_2 s_1 env
while e do e_1 done	≡ Y(λwhile. if e then (e_1; while))
e[v/x]	≡ λs,env. e s (env@{x↦v})
for i = e_1 to e_2 do e_3 done	≡ λ↓[v_2,v_1,s_1/e_2,e_1,env]. (rec-case v_2-v_1 of n<0 ↦ [loop]. () 0 ↦ e_3[v_1/i] n>0 ↦ [loop]. loop; e_3[v_l+n/i]) s_1 env
for i = e_1 downto e_2 do e_3 done	≡ λ↓[v_2,v_1,s_1/e_2,e_1,env]. (rec-case v_2-v_1 of n<0 ↦ [loop]. loop; e_3[v_l+n/i] 0 ↦ e_3[v_1/i] n>0 ↦ [loop]. ()) s_1 env

Table A 1. NUPRL representation of OCAML expressions (I)

e when e_1	≡	if e_1 then e else ↑
p when e_1 -> e	≡	λv. p v (e when e_1)
p -> e	≡	p when true -> e
p when e_1 -> e $\mid$ *matching*	≡	λv.λs,env. let <match,s_1> = p v (e when e_1) s env in case match of inl(exn) ↦ *matching* v s_1 env $\mid$ inr(val) ↦ inr(val), s_1
p -> e $\mid$ *matching*	≡	p when true -> e $\mid$ *matching*
$\overline{matching}$	≡	λe. λs,env. let↓ <inr(v),s_1> = e s env in *matching* v s_1 env
match e with *matching*	≡	$\overline{matching}$ e
function *matching*	≡	⌈$\overline{matching}$⌉
fun p_1...p_n when e_1 -> e	≡	function p_1 -> ... function p_n when e_1 -> e
try e with *matching*	≡	λs,env. let <result,s_1> = e s env in case result of inl(exn) ↦ *matching* exn s_1 env $\mid$ inr(v) ↦ inr(v), s_1
$\overline{pat}$	≡	λe_1,e_2. λs,env. let↓ <inr(v),s_1> = e_1 s env in *pat* v e_2 s_1 env
p = e	≡	$\overline{p}$ e
p = e and *binding*	≡	λe'. *binding* (p e e')
x p_1..p_n = e	≡	x = fun p_1..p_n -> e
x p_1..p_n = e and *binding*	≡	x = fun p_1..p_n -> e and *binding*
rec p = e	≡	$\overline{p}$ (Y (λe'. $\overline{p}$ e' e))
rec p_1=e_1 and ... p_n=e_n	≡	rec p_1,..,p_n = e_1,..,e_n
rec x p_1..p_n = e	≡	rec x = fun p_1..p_n -> e
let *bindings* in e	≡	*bindings* e
(e)	≡	e
begin e end	≡	e
(e:T)	≡	e

Table A 2. NUPRL representation of OCAML expressions (II)

Tables A 1 and A 2 give a complete description of the formal representation of OCAML expressions in NUPRL. In the tables we use the following abbreviations.

⌈v⌉	≡	λs,env. inr(v), s
↑	≡	λs,env. inl("Match_failure"), s
λ↓[v,s_1/e, env]. e'	≡	λs,env. let↓ <inr(v),s_1> = e s env in e'
λ↓[v_2,v_1,s_1/e_2,e_1, env]. e'	≡	λs,env. let↓ <inr(v_2),s_2> = e_2 s env in let↓ <inr(v_1),s_1> = e_1 s_2 env in e'
λ↓[v_3,v_2,v_1,s_1/e_3,e_2,e_1, env]. e'	≡	λs,env. let↓ <inr(v_3),s_3> = e_3 s env in let↓ <inr(v_2),s_2> = e_2 s_3 env in let↓ <inr(v_1),s_1> = e_1 s_2 env in e'

The terms {}, {f_1=e_1}⊗e_2, and e[v/x] do not belong to the language of OCAML. They are used only for describing the semantics of record expressions and for loops.

The formalization does not yet include expressions for float constants, bitwise logical and floating point operators, labels in matchings that may affect function evaluation, and operations on objects (new *obj*, e#*method*, (e:>T), (e:T_1:>T_2), {v_1=e_1,..,v_n=e_n}). Expressions of the language core described in Table 3, such as function p -> e and let p = e_1 in e_2, are subsumed by more general versions.

`ref` e is now an abbreviation for a mutable singleton record. The representation of the operators listed in §6.7.4 of the manual is not included in the tables, as they are defined explicitly in the OCaml library module `pervasives.ml`.

It should be noted that there is some overlap between the native syntax of OCaml and Nuprl's type theory. The list prepend operation, for instance, is written as e_1`::`e_2 in both formal languages, and the decomposition of pairs in Nuprl is written as `let <`x`,`y`>=`e_1 `in` e_2 while the application of a pair pattern in an OCaml let-binding is written as `let (`x`,`y`)=`e_1 `in` e_2 . Usually, it is clear from the context which formal construct is being used. For instance, the representation of constant list expressions is based on (the representation of) OCaml's list append operation, while the representation of constant array expressions is based on Nuprl's list append and lifting. For Nuprl's proof system, the distinction is obvious, since the terms for Nuprl's and OCaml's list append operations have different operator identifiers and the similarity is only a matter of term display.

Our tables use a dot-notation to describe the formalization of OCaml constructs that do not have a fixed size, such as constant list, array, and record expressions, and multiple matchings and bindings. Since formal definitions in Nuprl require terms to have a fixed number of subterms, the actual representation of these constructs has to be based on *iterated abstractions*. For instance, the representation of a record expression $\{f_1$`=`e_1`; ..;` f_n`=`$e_n\}$ is built from the representation of the empty record $\{\}$ and n applications of the record composition operator $\{f$`=`$e\}\otimes e'$. The display form for the latter has to make sure that the resulting term is displayed as syntactically correct record expression and not as $\{f_1$`=`e_1`;` $\{$`..;` $\{f_n$`=`$e_n\}$`..`$\}\}$.

In some rare cases, such as the definition of mutually recursive let bindings, it is not possible to use an iteration of basic abstractions to represent the OCaml language construct: to build `rec` p_1`,`p_2 `=` e_1`,`e_2 from `rec` p_1 `=` e_1 and `rec` p_2 `=` e_2 we would have to distribute the second abstraction over two subterms of the first. Therefore, our implementation provide a series of abstractions for each fixed number of mutually recursive bindings.

To simplify the representation of records, we represent all their components as mutable reference cells, even if they are not declared mutable. Attempts to assign a new value to a non-mutable component will be caught by the OCaml type checker, while otherwise the outward behavior of mutable and non-mutable components is identical. For the same reason, we simply represent parenthesized expressions with type constraints `(`e`:`T`)` by the expression e , as the type constraint is checked at compile time and has no influence on the operational semantics.

A.2 Patterns

Patterns, described in §6.6 of the OCaml manual, are templates that allow selecting data structures of a given shape and binding identifiers to components of this structure. As discussed in Section 3.1, we represent patterns as functions that modify an expression e by updating its environment according to the result of matching a value v against the template. These functions decompose the values of expressions that are built using the language constructs listed in Table A 1.

$\_$	$\equiv$	λv,e. e
x	$\equiv$	λv,e. e[v/x]
p as x	$\equiv$	λv,e. (p v e)[v/x]
i	$\equiv$	λv,e. if i=v then e else $\uparrow$
'chr'	$\equiv$	λv,e. if "chr"=v then e else $\uparrow$
"str"	$\equiv$	λv,e. if str $=_{str}$ v then e else $\uparrow$
true	$\equiv$	λv,e. if v then e else $\uparrow$
false	$\equiv$	λv,e. if v then $\uparrow$ else e
[]	$\equiv$	λv,e. rec-case v of [] $\mapsto$ e $\mid$ v_{hd}::v_{tl} $\mapsto$ $\uparrow$
p_1::p_2	$\equiv$	λv,e. rec-case v of [] $\mapsto$ $\uparrow$ $\mid$ v_{hd}::v_{tl} $\mapsto$ p_1 v_{hd} (p_2 v_{tl} e)
[p_1; ..; p_n]	$\equiv$	p_1::p_2::...::p_n::[]
$cconstr$	$\equiv$	λv,e. let $\langle v_1,v_2\rangle$=v in if v_1="$cconstr$" then e else $\uparrow$
$ncconstr$ p	$\equiv$	λv,e. let $\langle v_1,v_2\rangle$=v in if v_1="$ncconstr$" then p v_2 e else $\uparrow$
\`tag	$\equiv$	λv,e. let $\langle v_1,v_2\rangle$=v in if v_1="tag" then e else $\uparrow$
\`tag p	$\equiv$	λv,e. let $\langle v_1,v_2\rangle$=v in if v_1="tag" then p v_2 e else $\uparrow$
p_1, p_2	$\equiv$	λv,e. let $\langle v_1,v_2\rangle$=v in p_1 v_1 (p_2 v_2 e)
{f_1=p_1; ..; f_n=p_n}	$\equiv$	λv,e. p_1 !s[v "f_1"] (..(p_n !s[v "f_n"] e)..)
[|p_0; ..; p_n|]	$\equiv$	λv,e. p_0 !s[v[0] "contents"] (..(p_n !s[v[n] "contents"] e)..)
(p)	$\equiv$	p
(p:T)	$\equiv$	p
()	$\equiv$	($\_$:$unit$)
p_1 | p_2	$\equiv$	λv,e. (p_1 v e) ? (p_2 v e)

Table A 3. NUPRL representation of OCAML patterns

Table A 3 gives a complete description of the formal representation of OCAML patterns in NUPRL. In the table we use the following abbreviation.

$$e_1 \;?\; e_2 \quad\equiv\quad \lambda s,env.\ \texttt{let}\ \langle result,s_1\rangle = e_1\ s\ env\ \texttt{in}\ \texttt{case}\ result\ \texttt{of}\ inl(exn) \mapsto e_2\ s_1\ env \mid inr(v) \mapsto inr(v),\ s_1$$

The formalization does not yet include patterns for float constants and for type constructors abbreviating polymorphic variants (# *typeconstr*).

In the NUPRL implementation, the variable-sized patterns for constant list, array, and record expressions are represented by an iterated application of basic abstractions, similar to the way the corresponding expressions are represented. Parenthesized patterns with type constraints (p:T) are represented by the pattern p, as the type constraint is checked at compile time.

Since patterns match templates against values, they have to be *lifted* to functions on expressions instead of values in order to become applicable within bindings and matchings. Applying a lifted pattern $\overline{\text{pat}}$ (see Table A 2 for a formal definition) to an expression e_1 first evaluates e_1 and then applies the pattern to the resulting value.

A.3 Type Expressions

Type expressions, described in §6.4 and $6.8 of the OCAML manual, are used to denote the data types of OCAML as well as type constraints within patterns and

'id	≡ id
int	≡ ⌈int⌉
char	≡ ⌈Atom⌉
string	≡ char array
bool	≡ ⌈$\mathbb{B}$⌉
unit	≡ ⌈Unit⌉
array	≡ λT.⌈REF(T) List⌉
list	≡ λT.⌈T List⌉
exn	≡ ⌈EXCEPTION⌉
(T)	≡ T
T_1->T_2	≡ ⌈$T_1 \rightarrow T_2$⌉
T_1 * T_2	≡ ⌈$T_1 \times T_2$⌉
T typeconstr	≡ typeconstr T
(T_1,..,T_n) typeconstr	≡ typeconstr <T_1,..,T_n>
T as 'id	≡ rectype id = T
T_1[T as 'id/z]	≡ rectype id = $T_1[T/z]$
{mutable f_1:T_1}	≡ ⌈f:FIELD → (if f = f_1 then REF(T_1) else EXPR)⌉
{f_1:T_1}	≡ {mutable f_1:T_1}
{$fielddecl_1$;..;$fielddecl_n$}	≡ {$fielddecl_1$}∩..∩{$fielddecl_n$}
cconstr	≡ ⌈c:CONSTR × if c="cconstr" then unit else EXPR⌉
ncconstr of T	≡ ⌈c:CONSTR × if c="ncconstr" then T else EXPR⌉
$constr_1 \mid \ldots \mid constr_n$	≡ $constr_1$ ∩..∩ $constr_n$
\`tag	≡ ⌈tg:TAG × (if tg = "tag" then unit else EXPR)⌉
\`tag of T	≡ ⌈tg:TAG × (if tg = "tag" then T else Top)⌉
[> $variant_1 \mid \ldots \mid variant_n$]	≡ $variant_1$ ∩..∩ $variant_n$
[$variant_1 \mid \ldots \mid variant_n$]	≡ [> $variant_1 \mid \ldots \mid variant_n$]

Table A 4. NUPRL representation of OCAML types

expressions. Our formalization of OCAML type expressions aims at representing the typing relation in OCAML by the built-in membership relation of NUPRL's type theory. Thus OCAML types will be represented as (dependent) function types of the form s:STORE → env:ENV → (EXCEPTION + T) × STORE .

Table A 4 gives a complete description of the formal representation of OCAML type expressions, including record and variant (constructor) types, in NUPRL. In the table we use the following abbreviations.

⌈T⌉	≡	s:STORE → env:ENV → (EXCEPTION + T) × STORE
REF(T)	≡	s:STORE → env:ENV → ({i:ADDR $\mid$ ⌈!s[i]⌉ ∈ T} + VALUE) × STORE
FIELD	≡	LABEL
CONSTR	≡	LABEL
TAG	≡	Atom

The formalization does not yet include the types constants float, *id* option, and ('id_1, 'id_2, 'id_3) format, which are defined explicitly in OCAML's library module pervasives.ml, closed variant types [< $variant_1 \mid .. \mid variant_n$], object types (<..>, <$m_1$:$T_1$;..;$m_n$:$T_n$;..>), and class types (# *class*, T # *class*, (T_1,..,T_n) # *class*). Labels in function types are not yet supported.

Type variables usually occur only in type definitions and type constraints. In the former, they are linked to the data type parameters of a type constructor (see Appendix A.4), while in the latter they represent unspecified types that can be instantiated to satisfy the constraint. As type constraints are checked at compile time and ignored at runtime, we represent OCaml type variables as Nuprl variables.

Although type constructors are usually *defined* by users, OCaml has a few predefined constructors such as `list` and `array`. Note that type constructor application in OCaml uses postfix notation. The type `exn` of exceptions describes the result type of error messages that are returned when an exception is "raised". Formally, it is a type like any other OCaml type.

Record types are defined as dependent function types that assign different types to different values of a field. A record type declaration $\{f_1{:}\,T_1;\ ..;\ f_n{:}\,T_n\}$ is represented by the type

$$\texttt{f:FIELD} \rightarrow \texttt{if f} = f_1 \texttt{ then } T_1 \texttt{ else ... else if f} = f_n \texttt{ then } T_n \texttt{ else Top}$$

As record type declarations are variable-sized, we need to build their formal representation an iterated application of basic abstractions. We use Nuprl's intersection type constructor for this purpose, since (the representation of) $\{f_1{:}\,T_1; f_2{:}\,T_2\}$ has the same members as $\{f_1{:}\,T_1\} \cap \{f_2{:}\,T_2\}$. As explained in Appendix A.1, all component types are represented as mutable types, even if they are not explicitly declared mutable. Note, that in OCaml record types and variant constructor types always have to be defined types, i.e. they may only occur on the top-level of type definitions. Polymorphic variant types may occur in arbitrary type expressions.

Building the representation of complex recursive types ($T_1[T\ \texttt{as}\ \texttt{'}id]$) involves a meta-level construction that detects the occurrence of $T\ \texttt{as}\ \texttt{'}id$ in T_1, replaces it by T and wraps the whole expression with Nuprl's constructor for inductive types.

A.4 Type and Value Definitions

Type and value definitions, described in §6.8 and $6.11 of the OCaml manual, bind type constructors and value names to data types and expressions. They do not have an object level semantics in OCaml, but have to be considered meta-level operations that link names to actual types and expressions.

As a consequence, type and value definitions are not represented by object-level terms of type theory. Instead, they are mapped onto definition objects of the Nuprl system, i.e. meta-level objects that bind new abstract terms to type theoretical expressions. Nuprl abstractions represent the links between user-defined type and expression variables and their definitions, while display forms make sure that the syntactical appearance of the abstract term is identical to the original OCaml code.

Our mechanisms for importing OCaml code into Nuprl (Section 3.5) make sure that the abstraction objects corresponding to user-defined types and values allow identifying both the name chosen by the user and the module within which it was defined. References to these names within a piece of code will be disambiguated at "compile time" and mapped onto the appropriate abstract terms. Thus evaluating user-defined OCaml types and values simply means unfolding the corresponding

`type` $typeconstr$ `=` T	$\equiv$ $typeconstr \equiv$ `Y` $(\lambda typeconstr.T)$
`type` $'id\ typeconstr$ `=` T	$\equiv$ `type` $typeconstr$ `=` $\lambda id.T$
`type` $('id_1,..,'id_n)\ typeconstr$ `=` T	
	$\equiv$ `type` $typeconstr$ `=` λ`types.` `let` $<id_1,..,id_n>$ `= types in` T
`let` id `=` e	$\equiv$ $id \equiv e$
`let` $id\ p_1..p_n$ `=` e	$\equiv$ `let` id `= fun` $p_1..p_n$ `->` e
`let rec` id `=` e	$\equiv$ $id \equiv$ `Y` $(\lambda id.e)$
`let rec` $id\ p_1..p_n$ `=` e	$\equiv$ `let rec` id `= fun` $p_1..p_n$ `->` e

Table A 5. Nuprl representation of OCaml type and value definitions

Nuprl abstraction. As an additional feature, this approach enables Nuprl users to look up definitions of OCaml functions by clicking on the abstract term.

Table A 5 describes the formal definitions that are needed to build a formal representation of OCaml's type and value definitions. Note that these definitions map OCaml definitions to (the contents of) Nuprl definition objects, as indicated by the "$\equiv$" symbol on the right hand side of a definition.

The formalization accounts for the fact that type definitions may be recursive but does not yet include type constraints and optional prefixes for type parameters, which indicate whether a type constructor is to be co- or contravariant with respect to that parameter. Furthermore, as all OCaml definitions are mapped onto individual definition objects in Nuprl, multiple top level bindings as in `type` $typedef_1$ `and ...` $typedef_n$ and `let [rec]` $binding_1$ `and ...` $binding_n$ will be separated into individual definitions when the code is imported into Nuprl.

A.5 Partial evaluation of OCaml expressions

Symbolic computation rules for OCaml expressions support the partial evaluation of OCaml programs. As the semantics of OCaml expressions and patterns clearly describes the dynamic behavior of OCaml expressions, the computation rules for OCaml can be implemented as *derived inference rules* using Nuprl tactics and computation rules. A critical issue in this implementation is the preservation of the "OCamlness" of expressions: the tactics always have to return a (representation of a) valid OCaml expression and must not reveal the underlying type-theoretical description. Another issue is the potential presence of reference cells in expressions: the tactics can rewrite an expression into another one only if both expressions have the same values and the same side-effects.

Many of our computation rules OCaml that we describe in this section therefore require certain subexpressions to be *free of side-effects*, which roughly means that they do not contain assignments of the form $e_1.f$ `<-` e_2, e_1 `:=` e_2, $e_1.(e_2)$ `<-` e_3, or $e_1.[e_2]$ `<-` e_3. Function definitions in `function`, `fun`, and `let` expressions are always free of side-effects, even if the function body is not. Function applications, however, are free of side-effects only if the function body is.

In Nuprl the test for being free of side-effects is performed by meta-level tactics. A practical difficulty is the occurrence of user-defined functions: unfolding their

Redex		**Contractum**	
$-i$, i+j, i-j, $i*j$, i/j, i mod j	$\longrightarrow$	*as usual*	
i=j, i<>j, i<j, i<=j, i>j, i>=j	$\longrightarrow$	*as usual*	
if true then e_1 else e_2	$\longrightarrow$	e_1	
if false then e_1 else e_2	$\longrightarrow$	e_2	
e::[e_1;..;e_n]	$\longrightarrow$	[e::e_1;..;e_n]	
{f_1=e_1; ..; f_n=e_n}.f_i	$\longrightarrow$	e_i	1
{f_1=e_1; ..; f_n=e_n}.f_i<-e	$\longrightarrow$	{f_1=e_1; ..; f_i=e; ..; f_n=e_n}	2
$[\vert\, e_0;\ ..;\ e_n \vert]$.($i$)	$\longrightarrow$	e_i	1
$[\vert\, e_0;\ ..;\ e_n \vert]$.($i$)<-$e$	$\longrightarrow$	$[\vert\, e_0;\ ..;\ e;\ ..;\ e_n \vert]$	2
"c_0..c_n".[i]	$\longrightarrow$	'c_i'	
"c_0..c_n".[i]<-'chr'	$\longrightarrow$	"c_0..chr..c_n"	
e; e_2	$\longrightarrow$	e_2	3
while e do e_1 done	$\longrightarrow$	if e then (e_1; while e do e_1 done)	
for i=e_1 to e_2 do e_3 done	$\longrightarrow$	if e_1<=e_2 then ($e_3[e_1/i]$; for i=e_1+1 to e_2 do e_3 done)	4
match e with p_1 -> e_1' when e_1 ... p_n -> e_n' when e_n	$\longrightarrow$	e_i'$\lfloor e/p_i \rfloor$	2 3 5
(function $matching$) e	$\longrightarrow$	match e with $matching$	
(fun p_1..p_n when e' -> e) e_1..e_n	$\longrightarrow$	$e\lfloor e_1/p_1 \rfloor$..$\lfloor e_n/p_n \rfloor$	6
let p_1 = e_1 and .. p_n = e_n in e	$\longrightarrow$	$e\lfloor e_1/p_1 \rfloor$..$\lfloor e_n/p_n \rfloor$	2
let x p_1..p_n = e in e'	$\longrightarrow$	let x = fun p_1..p_n -> e in e'	
let rec x_1=e_1 and .. x_n=e_n in e	$\longrightarrow$	let x_1=e_1[let rec x_1=e_1 in x_1 / x_1] and ... x_n=e_n[let rec x_n=e_n in x_n / x_n] in e	2
let rec x p_1..p_n = e in e'	$\longrightarrow$	let rec x = fun p_1..p_n -> e in e'	
id	$\longrightarrow$	e	7
id e_1..e_n	$\longrightarrow$	$e\lfloor e_1/p_1 \rfloor$..$\lfloor e_n/p_n \rfloor$	2 8

[1]*: e_j free of side-effects for $j \neq i$.* [2]*: e_j free of side-effects for all j.* [3]*: e free of side-effects.*
[4]*: e_1, e_2 free of side-effects.* [5]*: i is the first j with $e_j\lfloor e/p_j \rfloor \neq \uparrow$ and e_j'$\lfloor e/p_j \rfloor$=true .*
[6]*: e'$\lfloor e_1/p_1 \rfloor$..$\lfloor e_n/p_n \rfloor$ = true.* [7]*: let id=e user-defined .* [8]*: let rec id p_1..p_n=e user-defined.*

Table A 6. (Derived) computation rules for OCaml

definitions and checking them for side-effects would make the test extremely expensive, as this often requires unfolding further definitions. Therefore the test assumes the bodies of other user-defined functions to have side-effects. If needed, the definitions of other functions can be unfolded explicitly by tailored evaluation tactics. This approach prevents the computation rule from being applied incorrectly, i.e. to expression where it cannot produce a formal justification for the evaluation step.

Using side-effects within expressions other than the bodies of functions is considered poor programming style and usually avoided, as this makes it difficult to understand the program's behavior. For this reason, our symbolic computation rules for OCaml are applicable to almost all "real" programs, despite their restriction to expressions that are free of side-effects.

Table A 6 describes the current set of derived computation rules for partially evaluating OCaml programs within the framework of the Nuprl system. These

$e\lfloor e'/\_\rfloor$	$\mapsto\ e$	
$e\lfloor e'/x\rfloor$	$\mapsto\ e[e'/x]$	
$e\lfloor e'/p\ \texttt{as}\ x\rfloor$	$\mapsto\ e\lfloor e'/p\rfloor\,[e'/x]$	
$e\lfloor i/i\rfloor$	$\mapsto\ e$	
$e\lfloor\text{'}chr\text{'}/\text{'}chr\text{'}\rfloor$	$\mapsto\ e$	
$e\lfloor\text{"}c_0..c_n/\text{"}c_0..c_n\text{"}\rfloor$	$\mapsto\ e$	
$e\lfloor\texttt{true}/\texttt{true}\rfloor$	$\mapsto\ e$	
$e\lfloor\texttt{false}/\texttt{false}\rfloor$	$\mapsto\ e$	
$e\lfloor[\,]/[\,]\rfloor$	$\mapsto\ e$	
$e\lfloor e_1{::}e_2\ /\ p_1{::}p_2\rfloor$	$\mapsto\ e\lfloor e_1/p_1\rfloor\,\lfloor e_2/p_2\rfloor$	
$e\lfloor[e_1;..;e_n]\ /\ p_1{::}p_2\rfloor$	$\mapsto\ e\lfloor e_1/p_1\rfloor\,\lfloor[e_2;..;e_n]/p_2\rfloor$	
$e\lfloor[e_1;..;e_n]\ /\ [p_1;..;p_n]\rfloor$	$\mapsto\ e\lfloor e_1/p_1\rfloor..\lfloor e_n/p_n\rfloor$	
$e\lfloor cconstr/cconstr\rfloor$	$\mapsto\ e$	
$e\lfloor ncconstr\ e'\ /\ ncconstr\ p\rfloor$	$\mapsto\ e\lfloor e'/p\rfloor$	
$e\lfloor\text{‘}tag\ /\ \text{‘}tag\rfloor$	$\mapsto\ e$	
$e\lfloor\text{‘}tag\ e'\ /\ \text{‘}tag\ p\rfloor$	$\mapsto\ e\lfloor e'/p\rfloor$	
$e\lfloor e_1,\, e_2\ /\ p_1,\, p_2\rfloor$	$\mapsto\ e\lfloor e_1/p_1\rfloor\,\lfloor e_2/p_2\rfloor$	
$e\lfloor\{f_1{=}e_1;..;e_n{=}e_n\}/\{f_{i1}{=}p_1;..;f_{ik}{=}p_k\}\rfloor$	$\mapsto\ e\lfloor e_{i1}/p_1\rfloor..\lfloor e_{ik}/p_k\rfloor$	*if* $k\leq n$
$e\lfloor[\mid e_0;..;e_n\mid]\ /\ [\mid p_0;..;p_n\mid]\rfloor$	$\mapsto\ e\lfloor e_0/p_0\rfloor..\lfloor e_n/p_n\rfloor$	
$e\lfloor e'/(p)\rfloor$	$\mapsto\ e\lfloor e'/p\rfloor$	
$e\lfloor e'/(p{:}T)\rfloor$	$\mapsto\ e\lfloor e'/p\rfloor$	
$e\lfloor e'/()\rfloor$	$\mapsto\ e$	
$e\lfloor e'/p_1\mid p_2\rfloor$	$\mapsto\ e\lfloor e'/p_1\rfloor$	*if* $e\lfloor e'/p_1\rfloor\neq\uparrow$
$e\lfloor e'/p_1\mid p_2\rfloor$	$\mapsto\ e\lfloor e'/p_1\rfloor$	*if* $e\lfloor e'/p_1\rfloor=\uparrow$

If e' *is in canonical form and none of the above rules applies then* $e\lfloor e'/p\rfloor\ \mapsto\ \uparrow$

Table A 7. (Derived) rules for symbolic pattern evaluation

rules can be applied to arbitrary sub-terms of an OCaml expression as long as their proviso (see the bottom of the table) is satisfied. The evaluation tactic `Red`, described in Section 4.3, uses search to find the first applicable evaluation rules. Other tactics attempt to reduce a specific sub-term of a proof goal.

In computation rules that involve pattern matching the notation $e\lfloor e'/p\rfloor$ describes a substitution where the free variables of e that occur in the pattern p are instantiated by the subexpressions of e' which result from matching the template p against e'. For p being a variable pattern x, this substitution is the same as the usual term substitution $e[e'/x]$. For all other templates it is the result of applying the pattern evaluation rules described in Table A 7, which are derived from the type theoretical semantics of (lifted) patterns in Table A 3.

Patterns can only be matched against expressions that are in canonical form and free of side-effects: function applications, conditionals, and other expressions occurring in Table A 6 have to be evaluated before symbolic pattern matching can be applied to them. Pattern matching fails (i.e. returns the expression $\uparrow$) if none of the rules in Table A 7 applies.