Embedding UNITY into the Calculus of Inductive Constructions

Mark-Oliver Stehr
University of Hamburg
Department of Computer Science
Theoretical Foundations of Computer Science Group

September 1998

Mark-Oliver Stehr
Universität Hamburg
Fachbereich Informatik
Arbeitsbereich Theoretische Grundlagen der Informatik (TGI)
Vogt-Kölln-Str. 30
22527 Hamburg
EMail: stehr@informatik.uni-hamburg.de

ABSTRACT

We realize a simple approach employing a logical type theory for the specification and verification of concurrent programs and systems. Type theories have been sucessfully applied in the field of functional programming to ensure partial or even total correctness. Here we present an embedding of a notion of concurrent programs and systems into the Calculus of Inductive Constructions. The embedding is not only simple and natural but also allows us to take full advantage of the underlying type theory. As a basis we use the idea of the UNITY method developed by Chandy and Misra to represent programs and systems as unbounded nondeterministic iterative transformations. As a specification and verification language we employ a variant of UNITY temporal logic in the expressive higher order logic framework which is provided by the propositions-as-types interpretation. As a first step and a starting point for applications we have developed a verified, general-purpose temporal logic library for the COQ proof assistant which can be used to specify and verify concurrent programs and systems. The concrete application of the library is demonstrated by a small example. Actually, the library is open for extensions and will be further developed in the future guided by the experience obtained from case studies.

KURZFASSUNG

Wir realisieren einen einfachen Ansatz, eine logische Typtheorie zur Spezifikation und Verifikation von nebenläufigen Programmen und Systemen zu nutzen. Bisher wurden Typtheorien mit Erfolg im Bereich funktionaler Programme zur Sicherstellung partieller oder vollständiger Korrektheit eingesetzt. Hier stellen wir eine Einbettung von nebenläufigen Programmen und Systemen in den Calculus of Inductive Constructions vor, die nicht nur einfach und natürlich ist, sondern es uns erlaubt, die Typtheorie in voller Allgemeinheit zu nutzen. Als Basis verwenden wir die zugrundeliegende Idee der UNITY Methode, entwickelt von Chandy und Misra, Programme und Systeme als unbeschränkt nichtdeterministische iterative Transformationen darzustellen. Als Spezifikations- und Verifikationssprache verwenden wir eine Variante der temporalen UNITY-Logik in dem ausdrucksstarken Rahmen der Logik höherer Ordnung, die uns die Typtheorie nach der Propositions-as-Types-Interpretation zur Verfügung stellt. Als erster Schritt und Ausgangspunkt für Anwendungen wurde eine verifizierte Temporal-Logik Bibliothek für den Beweisassistenten COQ entwickelt, die zur Spezifikation und Verifikation von nebenläufigen Programmen und Systemen eingesetzt werden kann. Der konkrete Einsatz der Bibliothek wird an einem kleinen Beispiel vorgeführt. Die Bibliothek ist offen für Erweiterungen und wird entsprechend der Erfahrung aus zukünftigen Fallstudien weiterentwickelt.

Contents

PREFACE

The present work shows how a notion of concurrent system/program together with a UNITY-style temporal logic can be embedded into a logical type theory, namely the Calculus of Inductive Constructions. A major goal of the present work was to preserve the elegance and simplicity of the original UNITY method including the independence of a particular execution semantics and still provide a rigorous formalization. The formalization has been carried out using the proof assistent COQ, which implements an extension of the Calculus of Inductive Constructions. The development has the form of a verified library which can be used for the specification and verification of a quite general class of concurrent programs and systems, namely those which can be specified by functional transition systems with a finite number of transition functions. Concerning the temporal logic we feel that in particular the use of operators from New UNITY in the present development has led to a clearer presentation of the theory while being fully compatible with the original UNITY approach.

The material can be seen as an introduction to formal system/program verification using type theory, but since its structure reflects the formal development it can also serve as a manual for those who would like to apply and extend the verification library presented in this work. Readers interested in the full development including formalized proofs of all theorems presented here can access the author's home page at `http://www.informatik.uni-hamburg.de/TGI` to obtain a copy of the library. It might also be of interest here that a generalization of the library which goes considerably beyond the original UNITY approach is in preparation.

The development has been used in a project/seminar on formal verification organized by the author at the University of Hamburg in summer 1998. The present work has benefited much from the helpful comments and suggestions of Fabian Nilius, Daniel Moldt, Michael Koehler, Stefan Reich and Wolfram Roisch which are gratefully acknowledged.

Chapter 1

Introduction

The UNITY method has been developed in the late 80ies by Chandy and Misra [5]. The formal techniques of UNITY comprise a programming language and a temporal logic. A UNITY program is a set of multiple assignment statements executed by fair interleaving. More abstractly UNITY programs can be conceived as *Unbounded Nondeterministic Iterative Transformations* on a set of global states. UNITY programs can be seen as state-based descriptions of rather general discrete systems, covering imperative programs, parallel programs, concurrent systems and asynchronous circuits. Many non-trivial examples and case studies are contained in [5]. More complex applications are reported in [22] and [39].

The UNITY specification and verification language is a variant of linear time temporal logic which can deal efficiently with those assertions that are most important in practice, namely assertions of the form p `is invariant` and $p \leadsto q$ (pronounced "p leads to q"). Assume a fixed program/system and a notion of execution. Then the assertion p `is invariant` is a safety assertion meaning that the state-dependent predicate p holds at all states during every execution of the program/system. $p \leadsto q$ on the other hand is a liveness assertion expressing that for all executions of the program/system if p is true at some moment during the execution then q will be true in the present or in the future of this moment. The importance of these two kinds of assertions has already been observed by Lamport in [23]. A high-level specification of a program/system usually involves assertions of these kinds and the verification of a program/system tries to establish such assertions, usually proceeding via several intermediate assertions. To support this activity UNITY offers further temporal operators such as p `is stable`, p `unless` q and p `ensures` q and a rich collection of useful proof rules.

Type theory on the other hand is a theory about typed functions. In a narrower sense type theories deal with various typed lambda calculi [2]. Basically, a type theory deals with judgements of the form $M : A$, meaning that a term M is of type A. Typed lambda calculi are not only the theoretical basis of functional programming langugages but they are also used as typed expression langugages for higher-order logics. Logical type theories such as the Calculus of Constructions [12] exploit type theory in an even more essential and fundamental way: Type theory is not only used as a functional programming language but it also provides an internal higher-order logic which can be exploited for applications such as formalized mathematics, theorem proving and specification and verification of functional programs. The Calculus of Inductive Constructions which provides the formal basis for the present work is an extension of the Calculus of Constructions. It has been implemented in the COQ proof assistent [19].

The aim of this work is to show how a logical type theory can be applied to the specification and verification of concurrent and nondeterministic programs/systems that usually exhibit a considerably richer behavior than functions. It turns out that the UNITY view of programs/systems as unbounded nondeterministic iterative transformations is a very useful abstraction to provide an embedding of a general notion of program/system into type theory [19].

In the present work the UNITY temporal logic will be defined within the Calculus of Inductive Constructions and all important proofs rules are rigorously derived as formal theorems. In this way we do not only obtain a fully formalized and more abstract version of the UNITY method which can be applied to a class of functional transition systems, but we also benefit for the underlying type theory which provides an expressive functional and logical framework. The last aspect is of particular advantage, since it allows us to reason about functional components, about the UNITY logic or just about general mathematical results needed for a particular verification task without leaving the verification framework.

The formalization has been carried out using the COQ proof assistent [19] and it has been structured in such a way that it constitutes an application-independent, verified library which can be specialized for concrete programs/systems. As a simple example, which demonstrates the instantiation of the library, we will verify a solution to the well-known common meeting time problem.

Chapter 2

Techniques of UNITY

This chapter provides a brief introduction to UNITY [5], covering its programming language and its temporal logic. After an overview of the UNITY programming language we will introduce functional transition systems as a more abstract way to present UNITY programs.

Functional transition systems just capture the original UNITY idea, namely the view of programs as unbounded nondeterministic iterative transformations (cf. [5]). Moreover functional transition systems are perfectly suited for an embedding into type theory since any overhead associated with the syntax of particular programming language is avoided. So we use the Calculus of Inductive Constructions directly as a polymorphic, higher order programming and specification language.

The temporal logic we employ is close to UNITY temporal logic but its presentation is based on operators from New UNITY [28][4]. These operators will also be useful for verification purposes since they close the gap between the Floyd-Hoare assertional method and UNITY logic.

2.1 UNITY Programming Language

In the following we give a simplified overview of the UNITY programming language. In [5] a precise specification of the language is avoided. This admits some freedom to adopt a language suited for the particular application domain to be addressed. As explained in [5] the UNITY programming language is a typed language and programs are given by

- a *variable declaration*,
- an *initialization condition*,
- a finite **set** of *conditional multiple assignment statements*.

UNITY programs are equipped with an *interleaving semantics*: A *state* is an assignment of values to variables. An execution sequence is an infinite sequence of states such that

- it starts with a state satisfying the initialization condition,
- it is obtained by the *fair* and *nondeterministic* selection and execution of statements.

Concurrency of independent statements is modeled by executing them in an arbitrary order (interleaving semantics). The semantics can be defined more formally using the concepts of functional transition systems and state transition sequences which are introduced subsequently.

2.2 Functional Transition Systems

From a more abstract point of view UNITY programs can be seen as a means for specifying functional transition systems. However, we introduce a further simplification of the model: We do not distinguish a particular set of initial states. Instead we assume that all states are possible initial states. Hence, there is no need to specify this set in the following definition.

A *functional transition system* consists of

- a set of *states*,
- a set of *transitions*,
- a *state transition function* associating to each transition a mapping from states to states, also called the *effect of a transition.*

If state transition functions are restricted to recursive functions we obtain an *operational transition system.*

A UNITY program specifies an operational transition system as follows: Each conditional multiple assignment statement of the program corresponds to a transition. The associated state transition function describes the effect of the statement on the state. If the condition of a statement is not satisfied the effect of the corresponding transition is the identity function on states.

For a functional transition system the execution semantics is given by fair state transition sequences:

- A *state transition sequence* is a pair (s, w) where s and w are infinite sequences over states and transitions, respectively, and for each index i the state s_{i+1} is obtained from s_i by the state transition function associated to w_i.
- A state transition sequence (s, w) is *fair* iff w contains each transition infinitely often.

Notice that functional transition systems are *deterministic* in the sense that the present state and a transition uniquely determines the successor state.

Notice also that we do not require state transition sequences to start at a distinguished set of initial states. Reasoning about initial states will become a matter of temporal logic as explained below.

2.3 UNITY Temporal Logic

UNITY employs a temporal logic comprising temporal operators `unless`, `invariant`, `ensures` and `leads to`. All of them are binary operators used to formulate assertions on *state predicates*, i.e. predicates depending on the state. Although UNITY temporal logic should be classified as a variant of linear time temporal logic, it is not possible to define all its operators in terms of standard linear time logic [27].

In the present work we will follow a more recent presentation of UNITY logic called New UNITY in [28]. In addition to `unless` and `ensures` which are the basic operators in [5] two new operators `co` and `transient` are introduced, since they provide a more elementary basis to define the other operators. The operators `co` and `transient` have also been used in [4].

Unfortunately, an ambiguity in [5] has led to some confusion about the meanings of the UNITY operators (see [37], [35] and [21] for a discussion of this issue). We feel, however, that there is no need for major changes of the UNITY methodology if the so-called substitution axiom is dropped and the traditional quantification over all states (instead of the subset of reachable states) is maintained in the definitions of assertions. Another solution is to keep the substitution axiom and restrict quantification to reachable states. This, however, leads to difficulties with compositionality. It is interesting to note that both solutions can be unified if all states are considered reachable, i.e. by choosing a trivial initial condition which is satisfied for all states. Speaking about more restricted initial conditions will become an issue of the temporal logic.

The problematic substitution axiom will be replaced by theorems about relativized temporal operators similar to the approach in [37]. However, in contrast to [37] we do not introduce new subscripted temporal operators but we prefer to view relativized operators just as ordinary UNITY operators of a particular kind. As a consequence the modifications w.r.t. the original UNITY approach are kept to a minimum.

Technically, this means that the set of initial states does not play any role in the definitions of temporal operators. Instead, initial conditions will be made explicit in the temporal assertions if this is desired in a particular context (cf. Chapter 5). The irrelevance of initial states also justifies the use of arbitrary but fair state transition sequences as the execution semantics.

The following temporal assertions are properties of state transition sequences. An assertion holds for a functional state transition system (or a UNITY program) iff it holds for every fair state transition sequence of the system.

- p `co` q:
  If p holds in an arbitrary state then q holds in the successor state.

- p `is stable`:
  Once p holds it remains true.

- p `unless` q:
  If p holds then p remains true at least as long as q does not hold.

- p `leads to` q:
  If p holds then q will eventually become true.

There are two auxiliary operators which are usually not employed in high-level specifications. Instead they are used to establish `leads to` assertions during the verification phase:

- `transient` p:
  p cannot hold permanently because there is a single transition which falsifies p once it is true.

- p `ensures` q:
  p `unless` q and if p holds then q will eventually become true either because q holds without any further transition or because of a single transition which falsifies p.

An important requirement in the definition of `transient` and `ensures` is that the single transition which is responsible for the progress is the same for all state transition sequences.

In addition we will use two particular kinds of assertions: The `inductively invariant` assertion and the `invariant` assertion. Both of them depend on an additional state predicate which will usually be a set of states considered to be initial in a particular context:

- p `is inductively invariant w.r.t.` q:
  q implies p and p `is stable`.
- p `is invariant w.r.t.` q:
  p holds in every state reachable from some state satisfying q.

It is clear that every inductive invariant w.r.t. q is an invariant w.r.t. q. The converse is not true, since an invariant is an assertion about reachable states only and does not imply stability for arbitrary states. On the other hand every invariant is implied by some inductive invariant, namely the *strongest invariant* which characterizes the set of reachable states [20].

The strongest invariant [24] is also useful in a high-level temporal specification where the behavior of a system is specified only for reachable states. Hence, we treat the strongest invariant as a state predicate which can be used in any specification. Typically, it will be used in connection with relativized assertions.

Non-inductive invariants are not useful for compositional reasoning since the set of reachable states may change under composition. Inductive invariants are instead compositional since they are assertions about arbitrary states. A discussion of these issues and their relation to the UNITY substitution axiom can be found in [21]. Instead of assuming the substitution axiom, we will carefully distinguish the two classes of invariants mentioned above.

It is noteworthy that conditional assertions of the form $R \Rightarrow G$ are admitted in UNITY for assertions R and G. Informally, a conditional assertion $R \Rightarrow G$ has the following meaning: If R is satisfied (for *all* fair state transition sequences) it follows that G is satisfied (again for *all* fair state transition sequences). Conditional assertions are useful for rely-guarantee style compositional reasoning.

Chapter 3

Type Theory and Logic

Type theories provide a foundation not only for typed programming languages but also for interactive mathematical proof assistants. We use the notion type theory to refer to different kinds of typed lambda calculi, where types are associated to lambda terms. Typed lambda calculi are pure functional programming languages. here we are interested in type theories that are extensions of the simply typed lambda calculus [7] by more expressive types. A unified presentation can be found in [2] and a general introduction into type theory is given in [40].

For the present work we are not only using type theory as an underlying theory of typed expressions but we will also exploit the logical contents of type theory which is given by the propositions-as-types interpretation to be explained below. This is different from the use of type theory as the underlying expression language of some (higher-order) logic which is the theoretical basis for proof assistants such as HOL [16], IMPS [13], PVS [29] and the generic proof assistant Isabelle [33].

Here we will employ the expressive logical type theory Calculus of Inductive Constructions (CIC) as a functional programming language and simultaneously as a logical specification and verification language. The Calculus of Inductive Constructions is an extension of the Calculus of Constructions (CC). So we will give a brief overview of the Calculus of Constructions first.

3.1 Calculus of Constructions

The Calculus of Constructions (see [11], [9], [12] and [10]) extends simply typed lambda calculus by explicit polymorphism, type operators and dependent function types. The Calculus of Constructions is an impredicative constructive type theory which is strongly normalizing implying that all representable functions are total and computable. It has an embedded intuitionistic higher-order logic according to the propositions-as-types interpretation. Type checking is decidable and classical models are possible.

Formally terms and types of the Calculus of Constructions are not distinguished. As usual terms which differ only in their bound variables are identified. Terms are of the following forms, where M, N and A are arbitrary terms:

x	Variable
`Prop`	impredicative universe
`Type`	auxiliary universe
(MN)	function application
$[x : A]$ M	lambda abstraction (variable binder)
$(x : A)$ M	dependent function type (variable binder)

The Calculus of Constructions has the following typing rules (cf. [2]) for judgements $\Gamma \triangleright M : A$ meaning that M is of type A in the context Γ. Here a context is a finite sequence of declarations $[x_i : A_i]$ which declares x_i to be of type A_i. $x \in \Gamma$ means that x is declared in Γ. $\Gamma\Gamma'$ denotes concatenation of contexts Γ and Γ'. U, U_1, U_2 range over universes $\{\texttt{Prop}, \texttt{Type}\}$. $B[x := N]$ denotes the usual substitution of free occurrences of x by N in B.

$$\triangleright \texttt{Prop} : \texttt{Type} \tag{Ax}$$

$$\Gamma \triangleright x : A \; if [x : A] \in \Gamma \tag{Var}$$

$$\frac{\Gamma \triangleright A : U \quad \Gamma\Gamma' \triangleright M : B}{\Gamma[x : A]\Gamma' \triangleright M : B} if x \notin \Gamma, \Gamma' \tag{Weak}$$

$$\frac{\Gamma \triangleright A : U_1 \quad \Gamma[x : A] \triangleright B : U_2}{\Gamma \triangleright (x : A)B : U_2} \tag{Form}$$

$$\frac{\Gamma[x : A] \triangleright M : B \quad \Gamma \triangleright (x : A)B : U}{\Gamma \triangleright [x : A]M : (x : A)B} \tag{Intro}$$

$$\frac{\Gamma \triangleright M : (x : A)B \quad \Gamma \triangleright N : A}{\Gamma \triangleright (MN) : B[x := N]} \tag{Elim}$$

$$\frac{\Gamma \triangleright A : B \quad \Gamma \triangleright B' : U \quad B \cong B'}{\Gamma \triangleright A : B'} \tag{Conv}$$

A non-dependent function type $A \to B$ abbreviates $(x : A)B$ for some x which does not occur in B.

The first rule Ax just declares `Prop` as a universe of types which is itself contained in the "technical" universe `Type`. The rules Var and Weak are the standard context rules for exploiting variable declarations and extending the context (weakening). The type formation rule Form is the only rule which can be used to form dependent function types $(x : A)B$ (and also standard function types $A \to B$ with the abbreviation above). In particular, it states that `Prop` is impredicative, i.e. closed under formation of (dependent) function types. The introduction rule Intro allows us to introduce elements of dependent function types, namely lambda-abstractions. The elimination rule Elim explains how an element of a (dependent) function type, i.e. a (dependent) function, can be used: It can be applied to an argument with suitable type and the type of the application may depend on the argument itself. Last but not least, the conversion rules Conv stats that types related by β-conversion $\cong$ are not distinguished. The variant of the Calculus of Constructions we are interested in (since it provides the basis for the Calculus of Inductive Constructions) has only β-conversion as computational/judgemental equality (cf. [2]).

3.2 Propositions as Types

Under the propositions-as-types interpretation a type assertion

$$M : A$$

can be interpreted as a logical proposition together with its proof:

$$M \text{ is a proof of } A$$

Under this interpretation basic type constructors can be equipped with a logical meaning:

$A \to B$ (function type)	$A \Rightarrow B$ (implication)
$(x : A)B$ (dependent function type)	$\forall x \in A.B$ (universal quantification)

The basis for this correspondence is the BHK-interpretation of intuitionistic logic: A proof of an implication $A \Rightarrow B$ is a constructive method for translating a proof of A into a proof of B. A proof of a universal quantification $\forall x \in A.B$ is a constructive method providing a proof of B for each x in A. In constructive type theory a constructive method is just a function which is total and computable, as mentioned above. It is easy to see that the typing rules given above are sound under the propositions-as-type interpretation.

Since other logical constants and operators can be *defined* inside the theory, the Calculus of Constructions contains an intuitionistic higher-order logic with an explicit representation of proofs as objects. As a consequence typed functional programming and theorem proving are completely identified. Proving a theorem means writing a program which produces an appropriate proof. The fact that a program produces a proof can be verified by type checking only.

3.3 Calculus of Inductive Constructions

The Calculus of Inductive Constructions (see [19] and [32]) generalizes the Calculus of Constructions by an infinite universe hierarchy `Type(i)` with two independent impredicative universes `Prop` and `Set` at the bottom, terminating recursion and inductive types such that all features mentioned before are preserved.

The logical connectives can be defined inductively on logical formulas (also called propositions) of the universe `Prop`.

$$\begin{aligned}
\texttt{False} &= \ldots : \texttt{Prop}\\
\texttt{True} &= \ldots : \texttt{Prop}\\
\texttt{not} &= [A : \texttt{Prop}](A \to \texttt{False}) : \texttt{Prop} \to \texttt{Prop}\\
\texttt{and} &= \ldots : \texttt{Prop} \to \texttt{Prop} \to \texttt{Prop}\\
\texttt{or} &= \ldots : \texttt{Prop} \to \texttt{Prop} \to \texttt{Prop}\\
\texttt{Ex} &= \ldots : (A : Set)(A \to \texttt{Prop}) \to \texttt{Prop}\\
\texttt{eq} &= \ldots : (A : Set)A \to A \to \texttt{Prop}
\end{aligned}$$

In the COQ proof assistant `(not A)`, `(and A B)`, `(or A B)` and `(eq ? M N)` are also written as `~A`, `A /\ B`, `A \/ B` and `M = N`, respectively.

It is noteworthy that the Calculus of Constructions admits classical models, where `Prop` is interpreted as a two element set containing a nonempty set (the interpretation of `True`) and an empty set (the interpretation of `False`).

The universe `Set` is intended as a universe of (computational) data types such as natural numbers `nat` and booleans `bool` which are given by their usual inductive definitions. `Set` is closed under dependent function types over arbitrary types, but we will not exploit this kind of impredicativity.

For an inductive definition the user has to specify only the constructors for the data type to be defined. A suitable induction principle and an operator for primitive recursion is generated according to a general scheme. For more information on inductive types in the Calculus of Inductive Constructions we refer to [19].

3.4 The COQ Proof Assistant

The proof assistant COQ [19] is a stable, well-structured and extensible system implemented in the functional programming language Object-CAML, an object-oriented dialect of ML. COQ allows interactive theorem proving and program construction in the Calculus of Inductive Constructions and provides additionally:

- built-in tactics for simple theorem proving steps,
- a rich mathematical library,
- universe polymorphism (avoids to use of explicit universe levels),
- extensible grammars for syntactic sugaring,
- coercions guided by a coercion hierarchy,
- tools for program extraction from constructive proofs,
- user friendly recursive definitions,
- generation of natural language from proof terms, and
- extensibility by user-defined tactics.

For the sake of clarity we have avoided using features such as extensible grammars and coercions, but they will certainly be a useful and important means for coping with more complex verification tasks in the future. In particular, user-defined tactics are an important feature to make a verification library practical. The theoretical aspects of the Calculus of Inductive Constructions and also the rich collection of features and extensions provided by the implementation have motivated the choice of the COQ proof assistant as a framework for the present development.

Chapter 4

A Verified Temporal Logic Library

In general a type theoretic embedding of a notion of concurrent system/program is one way to obtain a flexible and expressive formal basis for system/program specification and verification. There are a number of good reasons why in particular an embedding of UNITY into the Calculus of Inductive Constructions (CIC) is worth to investigate:

- UNITY is a methodology proven to be useful in many small but non-trivial case studies. It is a good compromise between simplicity and expressiveness.
- As a constructive type theory the CIC has a built-in notion of computation, but due to termination and confluence this is not directly applicable to concurrent systems in general. One solution is to represent concurrent systems in the type theory, e.g. as functional transition systems.
- CIC is an expressive type theory which is missing in the UNITY language. Inductive definitions allow powerful data types. Parameterization by elements and types allows highly generic UNITY programs.
- CIC contains a higher-order logic useful for reasoning e.g. about state predicates. As a consequence proof rules become theorems in higher-order logic which can be verified themselves. Also conditional assertions can be easily expressed.
- CIC is a functional programming language which can be used to formulate UNITY statements. CIC is strongly normalizing, i.e. functional programs are terminating, which is the most essential property of atomic statements. As a consequence termination of atomic statements is proved by type checking alone. Moreover the execution of single statements and finite sequences of statements is possible in CIC. As a consequence the apparent gap between execution and verification disappears.
- A formal embedding leads to a more precise presentation and a better understanding of UNITY, e.g. program variables and meta variables are clearly distinguished, proofs are completely formalized, detailed proofs become feasible with tool support.
- The UNITY methodology does not support proving general mathematical theorems (algebra, graph theory,...) which are often necessary in correctness proofs. In CIC on the other hand these theorems can be formulated and proved.

- A particular strength of CIC is the theory-inherent support for reasoning about inductively defined objects. Hence CIC is an ideal tool to reason about UNITY temporal logic in a semantics-independent way, where the main liveness operator `leads to` is inductively defined in terms of `ensures`.

- Last but not least CIC allows us to investigate advantages and potential problems arising from the use of a decidable type theory for system/program verification (cf. [26]).

4.1 Related Work

There are a number of approaches to formalize UNITY in different logics and proof assistants:

- "Mechanically verifying concurrent programs with the Boyer-Moore Prover" [15],
- "A proof environment for concurrent programs" using the B inference engine [3],
- "Formal verification of concurrent programs" using using the Larch prover [6],
- "Program verification using HOL-UNITY" in the HOL proof assistent [1],
- "A modular coding of UNITY in COQ" [17].

Whereas the first three references employ (fragments of) first order logic, the last two approaches exploit the possibilites offered by higher order logics. Only the last reference uses an expressive logical type theory such as CIC. The last work is also the one which is most closely related to ours. However there are some important differences explained briefly in the following.

The focus of the presentation in [17] is on correctness and completeness proofs w.r.t. to an interleaving semantics and the formalization of an interesting concept of environments called contexts which can be seen an internal solution to the problems associated with the substitution axiom and compositionality. Our approach on the other hand is guided by New UNITY [28] and is less foundational and more pragmatic: We have set up a verified library of temporal proof rules maintaining a good compromise between effort and formal security. We have not formalized a particular execution semantics. In this aspect the formalization [17] can be seen as complementary to our work. Instead we have formally derived all important proof rules of [5] and many other important results from the standard definitions of the temporal operators in a semantic-independent way.

Another more technical difference w.r.t. [17] is that we interpret assertions not over reachable states but over all states as in [1]. Instead of introducing an explicit concept of environment, we do not distinguish between the system transitions and environment transitions when defining the temporal operators. All temporal operators are parameterized, i.e. they can explicitly refer to an arbitrary component of the full transition system. Also initial conditions appear only as parameters of some temporal operators. Reasoning on restricted parts of the state is supported by relativized assertions. So the solution we adopt to deal with the substiution axiom and compositionality can be classified as an external one.

It is convenient that rely-guarantee style reasoning depending on transition safety guarantees is directly supported by contexts in [17]. In this aspect the approach is similar to [8]. Nevertheless, we adopt a different approach, namely the original one of [5], where rely-guarantee style reasoning is seen as an application of reasoning with conditional assertions. Hence arbitrary formulas (including liveness assertions) can serve as rely and guarantee assertions. In contrast to more standard temporal logics but in the tradition of the assertional Floyd-Hoare-method [14][18], temporal operators of UNITY are defined in terms of the structure of the transition system (system-based definitions) and they are *not* defined relative to individual executions. It follows that expressiveness is rather limited and conditional assertions are simple and useful but they are not powerful enough to express those very general kinds of rely-guarantee style reasoning proposed in [34].

As a consequence of treating all transitions uniformly, our approach does not admit different fairness requirements for different kinds of transitions. In [17] uniform fairness of system transitions is assumed but fairness of environment transitions is not enforced, which is a reasonable decision also taken in [8]. Since fairness is an important issue in system verification we plan to generalize UNITY in this aspect considerably and extend the library by a more flexible notion of fairness in the future. Actually, the aim to explore generalizations of different aspects of UNITY has been one of the motivations for the present formal development which can be seen as a basis for such experiments.

In the remainder of this chapter we will first present the formal development of a general UNITY library within an abstract framework which does not make any application-specific assumptions. Each of the following sections describes a COQ module. In the subsequent chapter we explain the use of the library by means of a particular example, namely the well-known "Common Meeting Time Problem".

4.2 An Abstract Frame

This section describes the abstract frame for the UNITY library contained in the module `abstract.v`. This module is the abstract root of the UNITY library as it refers to all other library modules. It declares the type of program variables and operations intended to define their associated types. Also the type of program transitions is declared here. This frame is abstract in the sense that it contains only a few general assumptions necessary to prove the theorems in the library which do not depend on a particular application or program. For a specific program this module will be replaced by a more concrete one defining precisely the variables, their types and the concrete transitions of the specific program.

`unity_init.v` is included here to import other libraries the UNITY library depends on.

```
Load unity_init.
```

`u_var` is declared as the type of program variables. For a particular UNITY program this type will contains the names of variables occurring in the program. It is important not to confuse these variables with the variables of the UNITY logic which will directly be represented as variables of CIC.

```
Variable u_var:Set.
```

`u_vars` is a natural number intended to fix the number of variables. `u_ith_var` is intended as an enumeration of these variables. Actually, we do not require injectivity so `u_vars` is the number of variable indices and there might be less variables than specified by `u_vars`.

```
Variable u_vars : nat.
Variable u_ith_var : nat->u_var.
```

Since we need an induction principle for variables we assume that every variable `v` will have an index w.r.t. to the enumeration `u_ith_var` which is less than `u_vars`. Clearly this ensures that `u_var` has only a finite number of variables.

```
Hypothesis u_var_finite : (v:u_var)
  (Ex [i:nat] (lt i u_vars) /\ (u_ith_var i)=v).
```

Moreover we assume that equality on variables is decidable. `u_var_eq_dec` is an operation which yields an object of a disjoint union type which is a tagged proof of `(v=v')` or `~(v=v')`.

```
Hypothesis u_var_eq_dec:(v,v':u_var){(v=v')}+{~(v=v')}.
```

`u_type_name` is the type of type names used by variables of a program. Only those type names contained in `u_type_name` can be used in a program.

```
Variable u_type_name:Set.
```

The only assumption we need about `u_type_name` is that equality on type names should be decidable.

```
Hypothesis u_type_name_eq_dec:(n,n':u_type_name){(n=n')}+{~(n=n')}.
```

The operation `u_type` associates a CIC-type with each type name. In this way arbitrary CIC-types can be used in programs.

```
Variable u_type:u_type_name->Set.
```

The operation `u_var_type_name` associates a type name (and therefore also a type) to each variable.

```
Variable u_var_type_name:u_var->u_type_name.
```

The module `unity_states.v` which will be explained below defines the notion of typed state space. There `u_st` will be defined as the type of states. A state is essentially a type correct assignment of values to variables.

```
Load unity_states.
```

Now the type of transitions `u_tr` (or more precisely transition names) is introduced. Each program statement corresponds to precisely one transition.

```
Variable u_tr:Set.
```

Again equality on transition names is required to be decidable.

```
Hypothesis u_tr_eq_dec:(t,t':u_tr){(t=t')}+{~(t=t')}.
```

To represent the effect of a transition on the state we declare an operation `u_effect` which associates with a transition `t` and a state `s` the successor state `(u_effect t s)`. Notice that transitions are deterministic in the sense that after specifying their name and the current state the successor state is uniquely determined. Notice also that there may be extensionally equivalent transitions, i.e. transitions with the same effect, having different names.

```
Variable u_effect : u_tr->u_st->u_st.
```

The following modules will be discussed in more detail in the subsequent sections.

`unity_program.v` defines the type of a program as a list of transitions. Also operations like `u_prog_contains` to check if a program contains a specific transition and `u_prog_union` for the composition of programs by the union of their transitions are provided.

```
Load unity_program.
```

`unity_logic.v` defines the central type `u_prop` of state dependent propositions representing predicates depending on the state. It provides a collection of operators lifted from propositions `Prop` to state predicates `u_prop`.

```
Load unity_logic.
```

The (in)dependence of a state property from a variable is formalized in `unity_depend` and some theorems are proved there.

```
Load unity_depend.
```

`unity_safety.v` defines temporal operators for specifying safety assertions, in particular `u_co`, `u_unless`, `u_stable` and `u_invariant`. Moreover a variety of theorems relating logical operators and temporal safety operators as well as the program itself are contained in this module.

```
Load unity_safety.
```

`unity_progress.v` contains definitions of temporal operators used to specify progress assertions. These operators are `u_transient`, `u_ensures` and `u_leads_to`. Also a collection of theorems relating safety and progress assertions are provided here.

```
Load unity_liveness.
```

`unity_union.v` provides rudimentary support for compositional reasoning. The idea is to derive temporal properties of the composed program from temporal properties of its constituents.

```
Load unity_union.
```

4.3 Mathematical Libraries

The purpose of this module `unity_init.v` is to include some libraries, in particular those providing data types and support for boolean values, arithmetic, polymorphic lists, dependent equality and well-founded relations.

```
Require Bool.
Require Arith.
Require Compare_dec.
Require Peano_dec.
Require Minus.
Require nat_suppl.
Require PolyList.
Require Eqdep.
Require Relations.
Require Wf.
Require Wf_nat.
```

`Bool` defines the type of boolean values and some operations on this type.

`Arith.v` contains definitions and facts about the type `nat` of natural numbers. `Peano_dec` and `Compare_dec` provide decidability properties of equality and ordering. `nat_suppl` contains a set of auxiliary theorems about `nat` needed in some proofs.

Dependent equality `Eqdep.v` will be used to define equality of states in `unity_states.v`. Compared with the standard equality it is more flexible concerning the argument types.

Well-founded relations defined in `Wf.v` will play a central role in `unity_progress.v` where proof rules for well-founded induction are derived. To obtain proof rules for induction over natural numbers as a special case we need the fact that the natural order on `nat` is well-founded. This fact is proved in `Wf_nat.v`.

Except for `nat_suppl.v` all libraries are part of the COQ distribution.

4.4 Typed State Space

The following module `unity_states.v` defines the type of states `u_st`. It also provides an equality on states and operations which are used to change to state.

We will define a state as an assignment of values to variables. This is not the only possible way to organize the state space but it is convenient and natural in a functional language. Moreover it closely corresponds to the notion of state space in the UNITY approach [5] and is also used in other approaches such as [27].

In constrast to the approach in [1] our state space is strictly typed in the sense that every variable has a well defined type and type correctness of the values of a variable will be enforced by the type theory.

In the module `abstract.v` we already declared a type of variables `u_var` and a type of type names `u_type_name`. In both cases we will use the standard equality which has been assumed to be decidable by `u_var_eq_dec` and `u_type_name_eq_dec`, respectively. Often we do not distinguish type names and types in informal explanations. Notice, however, that we have decidable equality on type names but not on types.

Here the operation `u_var_type` is introduced which is only a convenient abbreviation associating to each variable its type directly.

```
Definition u_var_type := [v:u_var] (u_type (u_var_type_name v)).
```

`u_st` is the type of states. A state associates a value with every variable. The value has to be of the variable's type.

```
Definition u_st := (v:u_var)(u_var_type v).
```

Notice that the dependent function space allows us to define the state space in the most natural way, which also corresponds to the usual set theoretic definition. In contrast to set theory, the fact that an expression denotes a state can be verified automatically instead of givig rise to a proof obligation.

To verify equality of states it will be necessary to verify equality of the associated values for all variables. The following predicate verifies whether two such values `x` and `y` are equal. `(u_val_eq n x n' x')` means `x` of type `n` equals `x'` of type `n'`.

```
Definition u_val_eq :=
  [n:u_type_name][x:(u_type n)][n':u_type_name][y:(u_type n')]
  (eq_dep u_type_name u_type n x n' y).
```

Obviously, the types of `x` and `y` which will be extracted from variables, say `vx` and `vy`, will depend on the type associated to `vx` and `vy`, respectively. That the variable names `vx` and `vy` will always be equal in the sequel is not visible to the typechecker at this level. So we cannot use the standard polymorphic equality which requires that convertibility of the argument types can be inferred. Instead the dependent equality `eq_dep` from the COQ library is used to compare values whose types are not convertible. However, one should be aware of the fact that the dependent equality module `Eqdep.v` is not a conservative extension of CIC since the axiom `eq_rec_eq` is assumed there.

Standard equality implies value equality since dependent equality `eq_dep` is (inductively) defined to be reflexive if both sides are instantiated by the same type.

```
Theorem u_eq_imp_val_eq:(n:u_type_name)
  (x,y:(u_type n))(x=y)->(u_val_eq n x n y).
```

The theorem `u_val_eq_imp_eq` is the converse of the previous implication. It just corresponds to the lemma `eq_dep_eq` from `Eqdep.v` specialized for our setting.

```
Theorem u_val_eq_imp_eq:(n:u_type_name)
  (x,y:(u_type n))(u_val_eq n x n y)->(x=y).
```

In the presence of an equality of type names `n` and `n'` it should be possible to cast an element `x` of `n` to an equal element of `n'`. The operation `(u_cast n n' e x)` converts `x` of type `n` to type `n'` given an equality proof `e:n=n'`.

```
Definition u_cast := [n,n':u_type_name][e:(n=n')][x:(u_type n)]
  (eq_rec u_type_name n u_type x n' e).
```

Now using the axiom `eq_rec_eq` assumed in `Eqdep.v` we can easily prove the theorem `u_cast_thm` stating that type casting between identical types does not change the value.

```
Theorem u_cast_thm : (n:u_type_name)(e:(n=n))(x:(u_type n))
  ((u_cast n n e x)=x).
```

```
Lemma eq_lemma :
  (T:Set)(P:T->T->Prop)((x:T)(P x x))->((x,y:T)(x=y)->(P x y)).
```

The previous trivial lemma will be used to prove the following theorem stating that type casting between equal types `n` and `n'` respects value equality.

```
Theorem u_cast_resp_eq:(n,n':u_type_name)(x:(u_type n))(e:(n=n'))
    (u_val_eq n' (u_cast n n' e x) n x).
```

Here we assume extensionality for states `u_st`: Two states are identified if they coincide for all variables. This means that a state is fully characterized by the contents of its variables.

```
Axiom u_st_extensionality : (s,s':u_st)((v:u_var)((s v)=(s' v)))->(s=s').
```

Notice that this is a non-conservative extension of the intensional type theory. Indeed this is the second non-conservative extension we use (Remember the axiom assumed in `Eqdep.v`). This axiom could be avoided by defining an appropriate equivalence on `u_st`. This however would complicate proofs and require additional theorems stating that this equivalence is respected by operations working on states. For convenience we have chosen the axiomatic solution.

A more abstract treatment of values and states which is not realized in the present version of the library should allow an arbitrary equivalence on subsets of values to admit variable types which are (subsets of) inductive data types with additional equations, e.g. multisets. Then this (dependent) equivalence on values should be lifted to an equivalence on states. A type theory with subtypes and quotient types would support this directly. An alternative for CIC (or type theories with Σ-types in general) is to use setoids, i.e. types equipped with a (partial) equivalence relations.

The next lemma expresses the trivial fact that equal variables have equal types. It is needed for the following definition.

```
Lemma u_var_eq_to_type_name_eq : (v,v':u_var)
  (v=v')->(u_var_type_name v)=(u_var_type_name v').
```

Subsequently the assignment operation is introduced as the only operation used to modify a state. Given a state `s`, a variable `v` and a new value `x` compatible to the type of `v` the term `(u_assign s v x)` denotes the new state obtained from `s` by assigning the value of `x` to `v`.

```
Definition u_assign := [s:u_st][v:u_var][x:(u_var_type v)]
  [v':u_var] <u_var_type v'>Case (u_var_eq_dec v v') of
  (* v=v' *) [p:(v=v')] (u_cast (u_var_type_name v) (u_var_type_name v')
                                (u_var_eq_to_type_name_eq v v' p) x)
  (* ~(v=v') *) [_](s v')
  end.
```

Observe that the operation deciding equality of variables `(u_var_eq_dec v v')` and the type casting operation `u_cast` are used in an essential way to make `u_assign` properly typed, i.e. resulting in a state from `u_st`. As explained above `u_cast` needs an equality proof on types as a justification for performing type casting. This is obtained by the previous lemma `u_var_eq_to_type_name_eq` from the equality proof on variables `p:(v=v')` provided by `u_var_eq_dec`.

The remaining theorems of this module are basic properties of the assignment operation which are required for most applications.

First it is clear that assigning the contents of a variable to itself does not change the state. Of course, extensionality `u_st_extensionality` is a major ingredient for the proof.

```
Theorem u_assign_unchanged : (s:u_st)(v:u_var)((u_assign s v (s v))=s).
```

The next theorem states: A variable `v` which has just received the value `x` by assignment will contain that value `x`.

```
Theorem u_assign_changed : (s:u_st)(v:u_var)(x:(u_var_type v))
  (((u_assign s v x) v)=x).
```

On the other hand, modifying a variable `v` does not change the value of any variable `v'` distinct from `v`.

```
Theorem u_assign_indep : (s:u_st)(v,v':u_var)(x:(u_var_type v))
  ~(v=v')->((u_assign s v x) v')=(s v').
```

After repeated assignment of a value to a variable `v` the contents of `v` is the value of the last assignment.

```
Theorem u_assign_twice : (s:u_st)(v:u_var)(x,y:(u_var_type v))
  (u_assign (u_assign s v x) v y)=(u_assign s v y).
```

And finally we prove that the order of assigning values to two distinct variables does not matter. In such a situation the two assignments commute.

```
Theorem u_assign_commutes : (s:u_st)
  (v,v':u_var)(x:(u_var_type v))(y:(u_var_type v'))
  ~(v=v')->(u_assign (u_assign s v x) v' y)=(u_assign (u_assign s v' y) v x).
```

4.5 Programs as Functional Transition Systems

This is the module `unity_program.v` which defines the type `u_prog` of programs and some related operations. The general idea is to find a definition which faithfully captures the concept of a UNITY program but on the other hand it should be general enough to represent other models for describing concurrent systems. Of course it should also nicely integrate into the type theory we use.

From the viewpoint of type theory the most natural concept is that of an operation which can be effectively executed. Type theory is a theory about functions. To exploit the power and elegance of type theory for the verification of functional programs it seems to be advantageous to define a system as a collection of communicating functional components. Each of these components could be specified and verified within type theory using standard techniques. So the system model we have chosen will be specified by a set of functions communicating over a set of variables.

The most simple way realizing the idea above is to conceive a program as a finite set of transitions. Transitions are elements of type `u_tr` which has been already introduced in `abstract.v`. In order to compute with finite sets we use lists as a concrete representation.

```
Definition u_prog := (list u_tr).
```

As declared in `abstract.v` there is an operation `u_effect:u_tr->u_st->u_st` which associates with each transition `t:u_tr` an operation `(u_effect t)` which describes the effect of `t` on states. If a state `s'` is obtained by applying `(u_effect t)` to a state `s` we also say that `s'` is obtained from `s` by the occurrence of `t`.

To recapitulate, the description of a concrete program or system consists of two parts: A description of the state space and a description of possible transitions. The state space is defined by a set of typed variables as described in `abstract.v` and `unity_states.v`. The transitions are specified by a set of transition names and an operation associating to each transition name its effect. If we add a set of possible initial states we have just formalized a class of functional transition systems as they have been introduced in Section 2.2.

Although the present module does not specify the semantics of a program formally, the intention is that programs are executed in accordance with the UNITY execution model: An fair state transition sequence (s, w) is an infinite sequence of states s and an infinite sequence of transitions w such that: (1) For each index i the state s_{i+1} is obtained from s_i by the occurrence of transition t_i. (2) Each transition occurs in w infinitely often.

For many theorems provided by the temporal logic part of the library we need the precondition that the program in consideration is not empty. For this purpose we define the following predicate. `In` is the membership predicate for polymorphic lists from `PolyList.v`.

```
Definition u_prog_not_empty := [pr:u_prog] (Ex [t:u_tr] (In t pr)).
```

In the remainder of this module two operations on programs are defined and their specification is verified.

The first one checks if a program contains a specific transition. This is possible since we assumed a decision operation `u_tr_eq_dec` for equality on `u_tr`. It will be convenient in proofs that `prog_contains` is not only a predicate but computes a boolean result.

```
Fixpoint prog_contains [pr:(list u_tr)] : u_tr-> bool :=
  [stmt:u_tr] Case pr of
    (* (nil u_tr) *) false
    (* (cons h t) *) [h:u_tr][t:(list u_tr)]
      Case (u_tr_eq_dec stmt h) of
        (* (stmt=h) *) [_] true
        (* ~(stmt=h) *) [_] (prog_contains t stmt)
      end
  end.
```

Of course the behavior we expect from prog_contains is given by the following specification.

```
Theorem prog_contains_spec : (pr:u_prog)(t:u_tr)
  ((prog_contains pr t)=true)<->(In t pr).
```

In other words u_prog_contains is a decision procedure for (In u_tr).

```
Theorem u_prog_contains : (pr:u_prog)(t:u_tr){(In t pr)}+{~(In t pr)}.
```

Another operation on programs is the union of the corresponding sets of transitions. It is realized by concatenating lists but avoiding repetition of transitions. An application of the union operation, namely composition of programs, will be discussed in Section 4.10.

```
Fixpoint u_prog_union [pr:(list u_tr)] : (list u_tr)->(list u_tr) :=
[pr':(list u_tr)] Case pr of
  (* (nil u_tr) *) pr'
  (* (cons h t) *) [h:u_tr][t:(list u_tr)]
    Case (u_prog_contains pr' h) of
      (* (prog_contains pr' h) *) [_](u_prog_union t pr')
      (* ~(prog_contains pr' h) *) [_](u_prog_union t (cons h pr'))
    end
  end.
```

Finally we prove that we have indeed realized a union of sets in this way.

```
Theorem u_prog_union_spec : (pr,pr':(list u_tr))(t:u_tr)
  ((In t pr) \/ (In t pr')) <-> (In t (u_prog_union pr pr')).
```

For convenience we split this theorem in two implications:

```
Theorem u_prog_union_spec_1 : (pr,pr':(list u_tr))(t:u_tr)
  ((In t pr) \/ (In t pr')) -> (In t (u_prog_union pr pr')).

Theorem u_prog_union_spec_2 : (pr,pr':(list u_tr))(t:u_tr)
  (In t (u_prog_union pr pr')) -> ((In t pr) \/ (In t pr')).
```

The union operation will only be relevant for compositional reasoning supported by proof rules to be defined in the module unity_union.v.

Before proceeding with the explanation of further modules some general remarks seem to be in order.

The choice of representing the effect of a transition directly as an operation of the type theory leads to a restriction on those effects which can be described in the pure CIC. It is known that CIC allows only the representation of primitive recursive higher order functions. This class is sufficiently rich in practice although the use of primitive recursion is not always convenient. Fortunately, some user-friendly definition schemes are available in COQ.

One should also be aware of the fact that a transition should specify an atomic execution step. This is well in harmony with the fact that all operations in CIC are total and therefore terminating. So termination proofs are implicit on the level of an atomic step. However, on the level of programs executions might or might not terminate relative to a variety of criteria. As for termination, determinism is ensured for each step (if the transition is known). Again it does not hold on the level of programs where determinism might or might not appear in several variants.

Finally a limitation of the preset formalization should be mentioned: Due to the choice of lists of statements as a representation of programs it is not possible for programs to contain an infinite number of statements. Strictly also the UNITY language does not admit this, but it allows the use of a statement scheme instead of a single statement. Such a scheme is intended to represent a finite family of statements as part of the program in a uniform way. The use of schemes is convenient from a modeling and verification point of view and necessary for dealing with other models of concurrent systems. For a future version of the library we plan to represent programs as sets of statements which may be infinite and therefore more general than the UNITY approach.

4.6 State Predicates

The module unity_logic.v introduces the type u_prop for propositions depending on states, also called state predicates. It also provides simple but frequently used operators and theorems.

In CIC propositions are elements of type Prop. Those propositions admitting a proof are called provable or inhabitated. The later refers to the proposition-as-types interpretation where the elements of a proposition are its proofs.

```
Definition u_prop := (u_st->Prop).
```

After defining the type of state predicates u_prop the usual logical operators are lifted from propositions to this new type by pointwise application.

```
Definition u_true := [p:u_st] True.
Definition u_false := [p:u_st] False.
Definition u_or := [p,q:u_prop][s:u_st] (p s) \/ (q s).
Definition u_or3 := [p,q,r:u_prop][s:u_st] (p s) \/ (q s) \/ (r s).
Definition u_and := [p,q:u_prop][s:u_st] (p s) /\ (q s).
Definition u_and3 := [p,q,r:u_prop][s:u_st] (p s) /\ (q s) /\ (r s).
```

```
Definition u_imp := [p,q:u_prop][s:u_st] (p s) -> (q s).
Definition u_iff := [p,q:u_prop][s:u_st] (p s) <-> (q s).
Definition u_not := [p:u_prop][s:u_st] ~(p s).
Definition u_ex := [T:Set][P:T->u_prop] [s:u_st] (Ex [t:T] ((P t) s)).
Definition u_all := [T:Set][P:T->u_prop] [s:u_st] (t:T)((P t) s).
Definition u_all_nat := [b:nat][P:nat->u_prop][s:u_st]
                        ((i:nat)(lt i b)->((P i) s)).

Definition u_everywhere := [p:u_prop] (s:u_st)(p s).
```

An exception is the operator `u_everywhere` which does not yield a state predicate but a (state independent) proposition. Therefore `(u_everywhere p)` for some `p:u_prop` is also said to be an assertion.

Theorems containing statements about logical operators on propositions `Prop` have obvious counterparts on the level of state predicates `u_prop`. Some of them are given in the following. Of course their proofs are absolutely trivial.

```
Theorem u_imp_p_p : (p:u_prop)(u_everywhere (u_imp p p)).

Theorem u_imp_false : (p:u_prop)(u_everywhere (u_imp u_false p)).

Theorem u_imp_true : (p:u_prop)(u_everywhere (u_imp p u_true)).

Theorem u_iff_imp : (p,q:u_prop)
  (u_everywhere (u_iff p q))->(u_everywhere (u_imp p q)).

Theorem u_iff_refl : (p:u_prop)(u_everywhere (u_iff p p)).

Theorem u_iff_sym : (p,q:u_prop)
  (u_everywhere (u_iff p q))->(u_everywhere (u_iff q p)).

Theorem u_iff_trans : (p,q,r:u_prop)
  (u_everywhere (u_iff p q))->(u_everywhere (u_iff q r))->
  (u_everywhere (u_iff p r)).

Theorem u_imp_and_l : (p,q:u_prop)(u_everywhere (u_imp (u_and p q) p)).

Theorem u_imp_and_r : (p,q:u_prop)(u_everywhere (u_imp (u_and p q) q)).

Theorem u_imp_or_l : (p,q:u_prop)(u_everywhere (u_imp p (u_or p q))).

Theorem u_imp_or_r : (p,q:u_prop)(u_everywhere (u_imp q (u_or p q))).
```

The fact that the logic of pure CIC is intuitionistic favors the use of constructive arguments. In principle, there is not need to employ classical logic, since it is known that classical logic can be embedded into intuitionistic logic. In practice, however the use of this embedding is not very convenient. An intuitionistic development of UNITY temporal logic seems to be possible if the definitions of the temporal operators are slightly modified w.r.t. the original presentation in [5]. However, in the present work we have chosen to stay close to the original definitions of UNITY operators. Consequently, we decided to employ classical logic whenever necessary to derive standard temporal proof rules. This avoids a further level of difficulty also in view of the fact that the benefits of inituitionistic temporal logic for practical verification problems are questionable.

Formally, classical logic is obtained by adding the single axiom `classic` given below, expressing the principle "tertium non datur". The assumption of this classical axiom can be justified by the existence of classical models of CIC. This axiom will only be employed to prove theorems which do not admit an intuitionistic proof.

```
Axiom classic : (p:Prop)(p \/ ~p).
```

This axiom has the following counterpart on the level of state predicates:

```
Theorem u_classic : (p:u_prop)(u_everywhere (u_or p (u_not p))).
```

4.7 Dependence on Variables

This section deals with the module `unity_depend.v` which formalizes dependence and independence of state predicates on program variables.

First, we introduce some technical tools for reasoning inductively about program variables. In order to refer to variables in a uniform way which is independent of the set of variables of a concrete program and their names we use the enumeration of variables provided by `u_ith_var` and `u_vars`.

The next theorem allows to infer that a property holds for all variables if it holds for all variables up to the maximum index. It uses the enumeration `u_ith_var` declared in `abstract.v`. Subsequently it will be useful to reduce proofs of properties of variables to induction over natural numbers.

```
Theorem u_var_index_thm : (P:u_var->Prop)
  ((i:nat)(lt i u_vars)->(P (u_ith_var i)))->(v:u_var)(P v).
```

The following auxiliary operation is used to merge two states in the following sense: Given an index `n` and states `s` and `s'` the term `(u_equalize_n_vars n s s')` returns a new state which is identical to `s` with the exception that the contents of all variables up to index `n` is copied from `s'` to `s`.

```
Fixpoint u_equalize_n_vars [n:nat] : u_st->u_st->u_st :=
  [s,s':u_st] Case n of
    (* O *) s
```

```
    (* (S p) *) [p:nat] (u_assign (u_equalize_n_vars p s s')
                                  (u_ith_var p) (s' (u_ith_var p)))
  end.
```

The informal specification is formulated in the next lemma.

```
Lemma u_equalize_1 : (s,s':u_st)(n:nat)(i:nat)(lt i n)->
((u_equalize_n_vars n s s') (u_ith_var i))=(s' (u_ith_var i)).
```

Applying `u_equalize_n_vars` to the number of variable indices `u_vars` gives an effective procedure for cloning a state. By extensionality on states we indeed arrive at an identical state.

```
Theorem u_equalize_works : (s,s':u_st) (u_equalize_n_vars u_vars s s')=s'.
```

When reasoning about programs temporal proof rules sometimes require preconditions stating the independence of certain state predicates on particular variables. To express the most simple form of such conditions the following definition of independence will be useful. Given a state predicate `p` and a variable `v` the proposition `(u_independent p v)` states that `p` does not depend on `v`.

```
Definition u_independent := [p:u_prop][v:u_var]
  (s:u_st)(x,y:(u_var_type v)) (p (u_assign s v x)) -> (p (u_assign s v y)).
```

An immediate consequence of this definition is that if `p` is independent of `v` then the fact that `p` holds at `s` is preserved under arbitrary modifications of `s` at `v`.

```
Theorem  u_independent_imp : (p:u_prop)(v:u_var)(u_independent p v)->
  (s:u_st)(p s)->(x:(u_var_type v))(p (u_assign s v x)).
```

Using induction and the previous theorem we can prove a rather technical lemma: If a state predicate `p` is independent of all variables (up to some index) and `p` holds at some state `s` we can modify all these variables, e.g. using their values from another state `s'`, without changing the validity of `p`.

```
Lemma u_independent_and_equalize : (p:u_prop)
  ((v:u_var) (u_independent p v))->
  (s,s':u_st)(i:nat)(le i u_vars)->
  (p s)->(p (u_equalize_n_vars i s s')).
```

Using this lemma it is easy to prove that if a state predicate is independent of all variables it cannot depend on the state.

```
Theorem u_fully_independent : (p:u_prop)
  ((v:u_var)(u_independent p v))->(s,s':u_st)(p s)->(p s').
```

In addition to the capability of expressing independence via `u_independent` we would also like to express preconditions stating that a predicate may depend on a particular variable but it definitely does not depend on further ones. This is achieved by `(u_may_depend_on p v)` given a state predicate `p` and a single variable `v` on which it may depend.

```
Definition u_may_depend_on := [p:u_prop][v:u_var]
  (m:(u_var_type v))(s,s':u_st) (p (u_assign s v m))->(p (u_assign s' v m)).
```

Given a predicate `p` there is a natural way to define a new predicate independent of an arbitrary variable `v`. The new predicate `[s:u_st](p (u_assign s v m))` behaves as `p` except for the fact that it does not use the contents of `v` but a constant `m` instead.

```
Lemma u_make_independent : (p:u_prop)(v:u_var)(m:(u_var_type v))
  (u_independent [s:u_st](p (u_assign s v m)) v).
```

The obvious relation between `u_independent` and `u_may_depend_on` is verified in the following theorem. It `p` is independent of all variables except for `x` it follows that `p` may depend on `x`. Observe that this does not exclude the case that `p` is independent of `x`.

```
Theorem u_dependent : (x:u_var)(p:u_prop)
  ((v:u_var)~(v=x)->(u_independent p v)) -> (u_may_depend_on p x).
```

A useful abbreviation for a term used already in the definition of `u_make_independent` is the following: Given a state predicate `p`, a variable `v` and a value `x` of its type, `(u_with p v x)` denotes the new predicate behaving as `p` but using `x` instead of the contents of `v`.

```
Definition u_with := [p:u_prop][v:u_var][x:(u_var_type v)]
  [s:u_st](p (u_assign s v x)).
```

The following is just the theorem `u_independent` rewritten using `u_with`.

```
Theorem u_indep_with : (p:u_prop)(v:u_var)(m:(u_var_type v))
  (u_independent (u_with p v m) v).
```

Finally we prove a theorem stating that the independence of each variable is preserved by `u_with`.

```
Theorem u_indep_imp_indep_with : (p:u_prop)(v,v':u_var)(m:(u_var_type v))
  (u_independent p v')->(u_independent (u_with p v m) v').
```

At the current state of the verification library the only theorem which exploits the present module is the elimination theorem called `u_elimination` contained in `unity_safety.v`. The elimination theorem will be used in the example in Chapter 5.

4.8 Safety Assertions

The module `unity_safety.v` contains the definitions of temporal logic operators `u_co`, `u_unless`, `u_stable`, `u_invariant` and `u_invariant` used for formulating assertions of safety properties. As in the proposal [28] for New UNITY `u_co` is chosen as the basic safety operator and the other operators are defined on top of it. This is not a major deviation from [5] but rather a more convenient way to structure the definitions.

So we start with the definition of the constraint operator `u_co`. For a program `pr` and two predicates `p` and `q` the assertion `(u_co pr p q)` is pronounced "`p` constrains `q`". It asserts that if `p` holds at some state then `q` holds at the successor state no matter which transition occurs. Formally this is written as a recursive definition which are part of CIC. The definition is recursive over the program `pr` which is represented as a list. The resulting formula is a conjunction of subformulas `((s:u_st)(p s) -> (q ((u_effect h) s)))`, one for each transition `h` appearing in `pr`.

```
Fixpoint u_co [pr:u_prog] : u_prop->u_prop->Prop :=
  [p,q:u_prop] Cases pr of
                 nil => True
               | (cons h t) => ((s:u_st)(p s)->(q ((u_effect h) s))) /\
                               (u_co t p q)
               end.
```

The recursive definition is useful if the program `pr` is given as a concrete list of statements. In this case the computational capabilities of type theory can be used to evaluate `(u_co u_pr)`.

If we are faced with an abstract program `pr` on the other hand, it is more convenient to use the following characterization of `u_co` instead of using the recursive definition directly.

```
Theorem u_co_characterization : (pr:u_prog)(p,q:u_prop)
  (u_co pr p q) <->
  ((t:u_tr)(s:u_st)(In t pr)->(p s)->(q ((u_effect t) s))).
```

The previous characterization of `u_co` consists of two implications, a necessary condition `u_nc_co` and a sufficient condition `u_sc_co`. Both of these will be widely used in proofs.

```
Theorem u_nc_co : (pr:u_prog)(p,q:u_prop)
  (u_co pr p q) ->
  ((t:u_tr)(s:u_st)(In t pr)->(p s)->(q ((u_effect t) s))).

Theorem u_sc_co : (pr:u_prog)(p,q:u_prop)
  ((t:u_tr)(s:u_st)(In t pr)->(p s)->(q ((u_effect t) s)))->
  (u_co pr p q).
```

An important remark concerning the difference between `co` and `u_co` seems to be appropriate at this point and also holds correspondingly for other types of temporal assertions introduced later: The first operator `co` has been defined in terms of state transition sequences. This *sequence-based definition* has been given informally in Section 2.3. Formally we work with the alternative *system-based* definition u_co given above. Instead of using state transition sequences generated by a system, u_co is defined in terms of the system itself.

As explained in the beginning the aim of this work is *not* to formalize the sequence-based definition and to prove the coincidence between sequence-based and system-based definitions of certain temporal operators. This has been done successfully in [30] [31] and in the formalization [17].[1] In the present work the emphasis is on developing a verified library of temporal operators and sound, pragmatic proof rules which are of immediate use for specification and verification purposes. The use of system-based definitions allows us to infer temporal properties directly from state transition systems without any need to reason about state transition sequences. An advantage of this approach is that the formalization is more economic and the temporal logic together with all proof rules remains independent of a particular execution semantics.

Now it is time to define further operators in terms of u_co. Notice that all operators are relative to a program pr specified as the first argument. In general pr may also be a component of the program to be verified, a possibility which can be exploited in compositional verification (cf. 4.10). The other arguments are state predicates, usually written using small letters p, q and r.

```
Definition u_unless := [pr:u_prog][p,q:u_prop]
  (u_co pr (u_and p (u_not q)) (u_or p q)).
```

The assertion (u_unless pr p q) states that p holds unless q holds, or more precisely: If p holds at some state and q does not hold then at the next step p remains true or q becomes true. It is important to see that progress does not have to take place, i.e. the assertion does not exclude the possibility that p remains true forever and q never holds. In [5] u_unless was taken as the basic operator to express safety assertions. Here we use u_co, also in the following definition.

It is also noteworthy that the preceeding definition of (u_unless pr p q) would not be equivalent to (u_co pr (u_and p (u_not q)) (u_not (u_and (u_not p) (u_not q)))) in intuitionistic logic. However, since we assumed the classical axiom classic we have an equivalence here.

The assertion (u_stable pr p) asserts that p is stable, i.e. once p becomes true it will hold forever.

```
Definition u_stable := [pr:u_prog][p:u_prop] (u_co pr p p).
```

A stronger assertion is (u_ind_invariant pr initially p) stating that p is an inductive invariant of the program pr with initial condition initially. In addition to stability of p it is required that p is implies by the initial condition. The initial condition initially has to be specified explicitly, since it is not a fixed part of the program pr as explained in the introduction.

```
Definition u_ind_invariant := [pr:u_prog][initially:u_prop][p:u_prop]
  (u_everywhere (u_imp initially p)) /\ (u_stable pr p).
```

[1]Actually, the proof is immediate for safety assertions but more difficult for liveness assertions.

From a program augmented with an initial condition the set of reachable states contains precisely those states reachable from some state satisfying the initial condition. In general, we may have an invariant I, i.e. a property holding at all reachable states, which is not an inductive invariant. It is, however, easy to see that we can find an inductive invariant (e.g. the characteristic predicate of the set of reachable states) which implies I. Hence, in order to state such invariants we can use `u_invariant` as defined below.

```
Definition u_invariant := [pr:u_prog][initially:u_prop][p:u_prop]
  (ExT [inv:u_prop] (u_ind_invariant pr initially inv) /\
                    (u_everywhere (u_imp inv p))).
```

The following state predicate `(u_si pr initially)`, called the strongest invariant [24], characterizes exactly the set of reachable states. Hence it is useful for temporal specifications, in particular in connection with relativized assertions to be introduced later.

```
Definition u_si := [pr:u_prog][initially:u_prop][s:u_st]
  (p:u_prop)(u_ind_invariant pr initially p)->(p s).
```

This concludes the list of temporal operators used to formulate safety assertions. In the remainder of this module a collection of theorems about these operators is proved. This includes also various relations between these operators. Most proof rules contained in [5] are among these theorems. Also many proof rules for `u_co` are proved here, among them also some rules contained in [28], in particular the elimination theorem.

We start with a combined left-strengthening/right-weakening rule `u_co_implication` for `u_co`. For convenience we add separate rules `u_co_implication_r` and `u_co_implication_l` for left-hand-side strengthening and right-hand-side weakening.

```
Theorem u_co_implication : (pr:u_prog)(p,p',q,q':u_prop)
  (u_everywhere (u_imp p p')) ->
  (u_everywhere (u_imp q q')) ->
  (u_co pr p' q) -> (u_co pr p q').

Theorem u_co_implication_r : (pr:u_prog)(p,q,r:u_prop)
  (u_everywhere (u_imp q r)) ->
  (u_co pr p q) -> (u_co pr p r).

Theorem u_co_implication_l : (pr:u_prog)(p,q,r:u_prop)
  (u_everywhere (u_imp p q)) ->
  (u_co pr q r) -> (u_co pr p r).
```

Of course the following right-hand-side and left-hand-side substitution rules are implied by the previous two theorems.

```
Theorem u_co_subst_r : (pr:u_prog)(p,q,q':u_prop)
  (u_everywhere (u_iff q q'))->(u_co pr p q)->(u_co pr p q').
```

```
Theorem u_co_subst_l : (pr:u_prog)(p,p',q:u_prop)
  (u_everywhere (u_iff p p'))->(u_co pr p q)->(u_co pr p' q).
```

Such substitution rules express that within an assertion (here `u_co`) it should be possible to substitute state predicates by equivalent ones. Here state predicates are said to be equivalent iff they are equivalent for all states. In an intensional type theory like CIC logical equivalence does not imply equality in general. So substitution theorems for logical equivalent formulas and state predicates have to be proved explicitly as above. An alternative solution is a non-conservative extension of CIC by adding a logical extensionality axiom.

Two `u_co` assertions can be combined to a new one by taking the conjunction of their left-hand sides and their right-hand sides, respectively.

```
Theorem u_co_conjunction : (pr:u_prog)(p,p',q,q':u_prop)
  (u_co pr p q) -> (u_co pr p' q') -> (u_co pr (u_and p p') (u_and q q')).
```

The substitution principle for `u_co` can be used to obtain a substitution principle for `u_stable` and `u_ind_invariant`.

```
Theorem u_stable_subst : (pr:u_prog)(p,q:u_prop)
  (u_everywhere (u_iff p q)) ->
  (u_stable pr p) -> (u_stable pr q).

Theorem u_ind_invariant_subst : (pr:u_prog)(r,r',p,q:u_prop)
  (u_everywhere (u_imp r' r))->(u_everywhere (u_iff p q)) ->
  (u_ind_invariant pr r p) -> (u_ind_invariant pr r' q).
```

The rule `u_co_conjunction` can be specialized for `u_stable` as follows.

```
Theorem u_co_stable_conjunction : (pr:u_prog)(p,q,r:u_prop)
  (u_stable pr r) -> (u_co pr p q) ->
  (u_co pr (u_and p r) (u_and q r)).
```

The following two rules are useful to derive *relativized assertions*, i.e. assertions which do not make statements about all states but only about a subset of states usually specified by some inductive invariant. Formally, we conceive relativized assertions as ordinary assertions with some additional restriction expressed by conjunction. For instance, `(u_co pr (u_and r p) (u_and r q)` and `(u_stable pr (u_and r p))` are relativations of `(u_co pr p q)` and `(u_stable pr p)` w.r.t. some inductive invariant `r`. The following two theorems show that we can exploit `r` not only on the left hand side but also as an assumption on the right-hand side if we know that `r` is stable.

```
Theorem u_rel_co : (pr:u_prog)(p,q,r:u_prop)
  (u_stable pr r)->(u_co pr (u_and r p) (u_imp r q))->
  (u_co pr (u_and r p) (u_and r q)).

Theorem u_rel_stable : (pr:u_prog)(p,r:u_prop)
  (u_stable pr r)->(u_co pr (u_and r p) (u_imp r p))->
  (u_stable pr (u_and r p)).
```

Rules for further relatived assertions will be given later. In some sense they allow us to use inductive invariants as assumptions on both sides of a relativized assertion, but in a more disciplined way than postulated by the substitution axiom (cf. [5]) which is not assumed here. Relativized assertions as they are introduced in the present work do correspond closely to the relativized assertions introduced in [37] (and corrected in [35]) as one solution to the dilemma with the substitution axiom. However, instead of introducing new families of temporal operators indexed by inductive invariants, we prefer use the original UNITY operators to express relativized assertions.

Above we have introduced the strongest invariant operator `u_si`. Remember that `(u_si pr initially)` is a predicate characterizing the set of states reachable from some state statisfying `initially`. It is interesting to note that the use of `u_si` makes it unecessary to formalize reachable states directly, just as there will be no need to formalize the operational sequence-based semantics. In the following some basic facts about the relationship between `u_si`, `u_ind_inv` and `u_invariant` will be given.

```
Theorem u_ind_inv_is_inv : (pr:u_prog)(initially:u_prop)(p:u_prop)
  (u_ind_invariant pr initially p)->(u_invariant pr initially p).

Theorem u_si_is_indinv : (pr:u_prog)(initially:u_prop)
  (u_ind_invariant pr initially (u_si pr initially)).

Theorem u_si_is_inv : (pr:u_prog)(initially:u_prop)
  (u_invariant pr initially (u_si pr initially)).

Theorem u_indinv_imp_si : (pr:u_prog)(initially:u_prop)(p:u_prop)
  (u_ind_invariant pr initially p)->
  (u_everywhere (u_imp (u_si pr initially) p)).

Theorem u_inv_imp_si : (pr:u_prog)(initially:u_prop)(p:u_prop)
  (u_invariant pr initially p)->
  (u_everywhere (u_imp (u_si pr initially) p)).
```

The next two results combined in Theorem `u_si_char` provide a characterization of `u_invariant` in terms of the strongest invariant given by `u_si`. With the additional fact that `u_si` characterizes the set of reachable states, the result says that `(u_invariant pr initially p)` holds iff `p` is true at all reachable states.

```
Theorem u_si_imp_inv : (pr:u_prog)(initially:u_prop)(p:u_prop)
  (u_everywhere (u_imp (u_si pr initially) p))->
  (u_invariant pr initially p).

Theorem u_inv_char : (pr:u_prog)(initially:u_prop)(p:u_prop)
  (u_invariant pr initially p)<->
  (u_everywhere (u_imp (u_si pr initially) p)).
```

The following theorem specializes u_rel_stable to relativized inductive invariants. Here it is necessary to require that r is not only stable but an inductive invariant. The theorem is particularly useful to prove and strengthen invariants incrementally. Notice, that r can not only be exploited in the proof of the left-hand side of the u_co premise but also to establish that p holds initially.

```
Theorem u_rel_inv : (pr:u_prog)(initially,p,r:u_prop)
  (u_ind_invariant pr initially r)->
  (u_everywhere (u_imp (u_and initially r) p))->
  (u_co pr (u_and r p) (u_imp r p))->
  (u_ind_invariant pr initially (u_and r p)).
```

Using u_co_stable_conjunction and u_co_implication we can prove a conjunction rule for u_stable and u_unless stating that a stable state predicate r can be added by conjunction to both sides of an existing u_unless assertion.

```
Theorem u_unless_stable_conjunction : (pr:u_prog)(p,q,r:u_prop)
  (u_stable pr r) -> (u_unless pr p q) ->
  (u_unless pr (u_and p r) (u_and q r)).
```

Extending the list of proof rules for relativized assertions u_rel_co, u_rel_stable and u_rel_inv we add another rule for relativized u_unless assertions.

```
Theorem u_rel_unless : (pr:u_prog)(p,q,r:u_prop)
  (u_stable pr r)->(u_co pr (u_and3 r p (u_not q)) (u_imp r (u_or p q)))->
  (u_unless pr (u_and r p) (u_and r q)).
```

The rule u_co_conjunction allows by its successive application the conjunction of finitely many u_co assertions. The following theorem u_co_big_conjunction provides a conjunction rule for an arbitrary family of u_co assertions indexed by a type T which may be infinite in general. Notice that (u_all T P) is the conjunction over the family P of state predicates indexed by T. Throughout the work, families of state predicates will be written using capital letters P and Q.

```
Theorem u_co_big_conjunction : (T:Set)(pr:u_prog)(P,Q:T->u_prop)
  ((t:T)(u_co pr (P t) (Q t)))->(u_co pr (u_all ? P) (u_all ? Q)).
```

As for the conjunction of u_co assertions there are rules for disjunction. Again we have a finitary version and a general version for an arbitrary family of premises. Below (u_ex T P) is the disjunction over the family P of state predicates indexed by T.

```
Theorem u_co_big_disjunction : (T:Set)(pr:u_prog)(P,Q:T->u_prop)
  ((t:T)(u_co pr (P t) (Q t)))->(u_co pr (u_ex ? P) (u_ex ? Q)).
```

```
Theorem u_co_disjunction : (pr:u_prog)(p,p',q,q':u_prop)
 (u_co pr p q) -> (u_co pr p' q') -> (u_co pr (u_or p p') (u_or q q')).
```

Remember that u_false and u_true are state propositions which are true and false for all states, respectively. Here we use them to state two trivial assertions involving u_co and u_unless.

```
Theorem u_co_false : (pr:u_prog)(p:u_prop)(u_co pr u_false p).
```

```
Theorem u_co_true : (pr:u_prog)(p:u_prop)(u_co pr p u_true).
```

```
Theorem u_unless_true : (pr:u_prog)(p:u_prop)(u_unless pr p u_true).
```

As for u_co and u_stable there is a substitution principle for u_unless. Again we prove rules for left-hand-side and right-hand-side substitution separately.

```
Theorem u_unless_subst_r : (pr:u_prog)(p,q,q':u_prop)
  (u_everywhere (u_iff q q'))->(u_unless pr p q)->(u_unless pr p q').
```

```
Theorem u_unless_subst_l : (pr:u_prog)(p,p',q:u_prop)
  (u_everywhere (u_iff p p'))->(u_unless pr p q)->(u_unless pr p' q).
```

The operator u_unless is reflexive on state predicates as stated in the following theorem.

```
Theorem u_unless_refl : (pr:u_prog)(p:u_prop)(u_unless pr p p).
```

Moreover u_unless includes the implication between state predicates.

```
Theorem u_unless_imp : (pr:u_prog)(p,q:u_prop)
  (u_everywhere (u_imp p q))->(u_unless pr p q).
```

There is an equivalence between (u_stable pr p) and (u_unless pr p u_false) as proved by the following two implications. In [5] this equivalence is taken to define u_stable.

```
Theorem u_stable_imp_unless : (pr:u_prog)(p,q:u_prop)
  (u_stable pr p)->(u_unless pr p q).
```

```
Theorem u_unless_imp_stable : (pr:u_prog)(p:u_prop)
  (u_unless pr p u_false)->(u_stable pr p).
```

Using the general disjunction rule u_co_big_disjunction for u_co we can prove a similar rule for u_unless.

```
Theorem u_unless_big_disjunction : (pr:u_prog)(u_prog_not_empty pr)->
  (T:Set)(P,Q:T->u_prop)
  ((i:T)(u_unless pr (P i) (Q i)))->
  (u_unless pr (u_ex ? P) (u_ex ? Q)).
```

The following rule allows weakening of the right-hand side of u_unless. It is known as the consequence weakening rule.

```
Theorem u_unless_implication_r : (pr:u_prog)(p,q,r:u_prop)
  (u_everywhere (u_imp q r)) ->
  (u_unless pr p q) -> (u_unless pr p r).
```

There is also a conjunction rule for `u_unless`. Notice that in contrast to the conjunction rule for `u_co` we have a disjunction of three predicates on the right-hand side depending on whether progress takes place in the second, in the first or in both premises.

```
Theorem u_unless_conjunction : (pr:u_prog)(p,p',q,q':u_prop)
  (u_unless pr p q) -> (u_unless pr p' q') ->
  (u_unless pr (u_and p p')
     (u_or3 (u_and p q') (u_and p' q) (u_and q q'))).
```

Weakening the previous conjunction rule using `u_unless_implication_r` yields the simple conjunction rule.

```
Theorem u_unless_simple_conjunction : (pr:u_prog)(p,p',q,q':u_prop)
  (u_unless pr p q) -> (u_unless pr p' q') ->
  (u_unless pr (u_and p p') (u_or q q')).
```

As a special case of `u_unless_simple_conjunction` we obtain the following left-hand side conjunction rule for `u_unless`, which is generalized by induction to an finite `nat`-indexed family in `u_unless_left_big_conjunction`.

```
Theorem u_unless_left_conjunction : (pr:u_prog)(p,p',q:u_prop)
  (u_unless pr p q) -> (u_unless pr p' q) ->
  (u_unless pr (u_and p p') q).
```

```
Theorem u_unless_left_big_conjunction : (pr:u_prog)(u_prog_not_empty pr)->
  (b:nat)(Q:nat->u_prop)(r:u_prop)
  ((i:nat)(lt i b)->(u_unless pr (Q i) r))->
  (u_unless pr (u_all_nat b Q) r).
```

Similar to `u_unless_simple_conjunction` there is a simple disjunction rule for `u_unless` formulated in the following theorem. It is interesting to note that this is the first rule which relies on the use of the classical axiom `classic`.

```
Theorem u_unless_simple_disjunction : (pr:u_prog)(p,p',q,q':u_prop)
  (u_unless pr p q) -> (u_unless pr p' q') ->
  (u_unless pr (u_or p p') (u_or q q')).
```

Similar to `u_unless_conjunction` we can combine two `u_unless` assertions in a disjunctive fashion using the rule `u_unless_disjunction`. The proof is done by applying `u_co_disjunction` and `u_co_implication`.

```
Theorem u_unless_disjunction : (pr:u_prog)(p,p',q,q':u_prop)
  (u_unless pr p q) -> (u_unless pr p' q') ->
  (u_unless pr (u_or p p')
     (u_or3 (u_and (u_not p) q') (u_and (u_not p') q) (u_and q q'))).
```

Without the classical axiom `classic` it is not possible to make use of the left-hand side of the conclusion which is `(u_and (u_or p p') (u_not (u_or3 (u_and (u_not p) q') (u_and (u_not p') q) (u_and q q'))))` if we expand the definition of `u_unless`. Consider a state satisfying `p` and `p'` (as a special case of `p` or `p'`). The only information we can infer in this case is that `q` and `q'` cannot hold simultaneously, but we do not know which one does not hold. So although classically we have `(u_and p (u_not q))` or `(u_and p' (u_not q'))` this does not hold intuitionistically. It follows that without `classic` the `u_unless` premises cannot be applied to infer anything about the next state.

`u_unless_disjunction` is used to prove the following cancelation rule: If we can leave a state satisfying `p` only via a state satisfyin `q` and a state satisfying `q` only via a state statisfyin `r`, it follows that a state satisfying `(u_or p q)` can only be left via `r`.

```
Theorem u_unless_cancelation : (pr:u_prog)(p,q,r:u_prop)
  (u_unless pr p q) -> (u_unless pr q r) -> (u_unless pr (u_or p q) r).
```

The remainder of this module contains a formalized proof of the elimination theorem stated in [28]. This part makes use of the module `unity_depend.v` which provides support for dealing with dependence of predicates on program variables. For instance, `u_with` and `u_independent` are defined there.

We need two lemmas for the proof of the theorem. The first one roughly states that substituting `x` in `p` by its own value `(s x)=m` yields a predicate equivalent to `p`. The existential quantifier `u_ex` makes the auxiliary variable `m` local to the expression.

```
Lemma u_elimination_1 : (x:u_var)(p:u_prop)(u_everywhere
  (u_iff (u_ex ? [m:(u_var_type x)](u_and (u_with p x m) [s:u_st](s x)=m)) p)).
```

The second lemma will be proved using the following theorem which will also be useful for other purposes. If a predicate `p` is independent of all variables then it must be stable, since it cannot extract any information from the state.

```
Theorem u_indep_imp_stable : (pr:u_prog)(p:u_prop)
  ((v:u_var)(u_independent p v))->(u_stable pr p).
```

The second lemma states: Given a predicate `p` which may depend only on a variable `x`, the predicate `(u_with p x m)` substituting any constant `m` for `x` is stable. The proof involves `u_indep_imp_stable` and `u_indep_imp_indep_with`.

```
Lemma u_elimination_2 : (pr:u_prog)(x:u_var)(p:u_prop)
  ((v:u_var)~(v=x)->(u_independent p v)) ->
  (m:(u_var_type x))(u_stable pr (u_with p x m)).
```

Now we are prepared to prove the elimination theorem. First we convey the intuitive idea: Assume we have a predicate p which depends on no other variable than x. We would like to conclude that if p holds at some state then at the successor state there is some m such that (u_with p x m), meaning that p holds with m substituted for x. So far this simply states that if p was true in the past there must have been some value m for x where p was true. Notice that we do not know anything else about m.

Now assume we have the additional information that once the variable x contains m the predicate (q m) holds in the next state. Then we can strengthen our conclusion: If p holds at some state then at the successor state there is some m such that (u_with p x m) and also (q m) holds. This is just the statement of the following elimination theorem.

```
Theorem u_elimination :
  (pr:u_prog)(x:u_var)(p:u_prop)(q:(u_var_type x)->u_prop)
  ((v:u_var)~(v=x)->(u_independent p v)) ->
  ((m:(u_var_type x))(u_co pr [s:u_st]((s x)=m) (q m))) ->
  (u_co pr p (u_ex ? [m:(u_var_type x)](u_and (u_with p x m) (q m)))).
```

The proof uses u_co_big_disjunction, u_co_conjunction and the previous two lemmas.

4.9 Liveness Assertions

Liveness assertions u_ensures and u_leads_to are build on the top of safety assertions. This is done in the module unity_liveness.v which is discussed in this section. It is remarkable that according to the approach of New UNITY the only basic liveness assertion which is additionally necessary is the unary u_transient assertion, which expresses that a state predicate will eventually be falsified by a single transition once it it true.

More precisely, (u_transient pr p) is satisfied iff the program pr contains a transition such that for every state satisfying p the successor state obtained by executing t invalidates p.

```
Definition u_transient := [pr:u_prog][p:u_prop]
  (Ex [t:u_tr](In t pr)/\((s:u_st)(p s)->~(p ((u_effect t) s)))).
```

An apparent defect of this definition (which is a formalization of the original one from [28]) is that the assertion is never satisfied if the program is empty. For instance, (u_transient (nil u_tr) u_false) is not satisfied. To exclude empty programs the following theorems presuppose (u_prog_not_empty pr).[2] We will assume it implicitly in the discussion of all remaining theorems.

```
Theorem u_transient_false : (pr:u_prog)
  (u_prog_not_empty pr)->(u_transient pr u_false).
```

An obvious rule for u_transient allows us to replace the state predicate by a stronger one.

[2] In [28] it is assumed that every program contains at least the skip statement, which does not change the state. This is possible but not enforced in our approach. The advantage of adding skip is that temporal assertions become invariant under stuttering (cf. [25]).

```
Theorem u_transient_implication : (pr:u_prog)(p,q:u_prop)
  (u_transient pr p)->(u_everywhere (u_imp q p))->(u_transient pr q).
```

This can be easily stated in a different form:

```
Theorem u_transient_strengthen : (pr:u_prog)(p,q:u_prop)
  (u_transient pr p)->(u_transient pr (u_and p q)).
```

The following rule allows us to combine a `stable` and a `transient` assertion. Intuitively, stability of a predicate and the property of being transient are contradictory in nontrivial cases. The only possibility which avoids a contradiction is that the state predicate is not satisfied in any state as it is stated in the conclusion of the rule.

```
Theorem u_stable_and_transient : (pr:u_prog)(p:u_prop)
  (u_stable pr p) -> (u_transient pr p) -> (u_everywhere (u_not p)).
```

The following rule allows substitution of state predicates by equivalent ones under the transient assertion. It extends our list of substitution rules `u_co_subst_r`, `u_co_subst_l`, `u_stable_subst`, `u_unless_subst_r` and `u_unless_subst_l`.

```
Theorem u_transient_subst : (pr:u_prog)(p,q:u_prop)
  (u_everywhere (u_iff p q)) ->
  (u_transient pr p) -> (u_transient pr q).
```

To derive relativized `u_transient` assertions the following rule can be used:

```
Theorem u_rel_transient : (pr:u_prog)(u_prog_not_empty pr)->(p,r:u_prop)
  (u_stable pr r)->
  (Ex [t:u_tr](In t pr)/\
    ((s:u_st)(r s)->(p s)->(r ((u_effect t) s))->~(p ((u_effect t) s))))->
  (u_transient pr (u_and r p)).
```

The `u_ensures` assertion is essentially `u_unless` equipped with liveness. More precisely, `(u_ensures pr p q)` combines two assertions: (1) The safety assertion `(u_unless pr p q)`, if `p` holds it cannot cease to hold before `q` holds. (2) The liveness assertion that `q` will hold eventually after `p` holds and that is accomplished without a transition or by a single one. Remember the definition of `(u_unless pr p q)` as `(u_co pr (u_and p (u_not q)) (u_or p q))` stating that `(u_and p (u_not q))` implies that `(u_or p q)` holds in the next state. To ensure liveness we add `(u_transient pr (u_and p (u_not q)))` expressing that `(u_and p (u_not q))` cannot hold permanently. This excludes the possibility that `p` holds forever and that `q` never becomes true.

```
Definition u_ensures := [pr:u_prog][p,q:u_prop]
  (u_unless pr p q) /\ (u_transient pr (u_and p (u_not q))).
```

From `u_unless_true` and `u_transient_false` we obtain immediately:

```
Theorem u_ensures_true : (pr:u_prog)(u_prog_not_empty pr)->
  (p:u_prop)(u_ensures pr p u_true).
```

As for all other assertions we need substitution rules which are proved again without assuming extensionality.

```
Theorem u_ensures_subst_r : (pr:u_prog)(p,q,q':u_prop)
  (u_everywhere (u_iff q q'))->(u_ensures pr p q)->(u_ensures pr p q').
```

```
Theorem u_ensures_subst_l : (pr:u_prog)(p,p',q:u_prop)
  (u_everywhere (u_iff p p'))->(u_ensures pr p q)->(u_ensures pr p' q).
```

Setting `q := p` in the definition of `u_ensures` we obtain the following reflexivity rule from `u_unless_refl` and `u_transient_false`.

```
Theorem u_ensures_refl : (pr:u_prog)(u_prog_not_empty pr)->
  (p:u_prop)(u_ensures pr p p).
```

More generally, any implication between state predicates can be cast into a `u_ensures` assertion.

```
Theorem u_ensures_imp : (pr:u_prog)(u_prog_not_empty pr)->
  (p,q:u_prop)(u_everywhere (u_imp p q)) -> (u_ensures pr p q).
```

In an assertion `(u_ensures pr p u_false)` the state predicate `p` can never be true, otherwise `u_false` would have to become true eventually which is impossible.

```
Theorem u_ensures_false : (pr:u_prog)(u_prog_not_empty pr)->
  (p:u_prop)(u_ensures pr p u_false)->(u_everywhere (u_not p)).
```

The next rule allows weakening on the right-hand side of a `u_ensures` assertion. The safety part is proved by `u_unless_implication_r`.

```
Theorem u_ensures_implication_r : (pr:u_prog)(p,q,r:u_prop)
  (u_everywhere (u_imp q r)) ->
  (u_ensures pr p q) -> (u_ensures pr p r).
```

The following conjunction rule for `u_ensures` has a form similar to the `u_unless` conjunction rule. Notice that only one premise has to be a liveness assertion in order to infer liveness.

```
Theorem u_ensures_conjunction : (pr:u_prog)(p,p',q,q':u_prop)
  (u_unless pr p q) -> (u_ensures pr p' q') ->
  (u_ensures pr (u_and p p')
     (u_or3 (u_and p q') (u_and p' q) (u_and q q'))).
```

As for the other operators there is also a rule to prove relativized `u_ensures` assertions. Notice that the relativized assertions in the premises can be further refined using `u_rel_unless` and `u_rel_transient`.

```
Theorem u_rel_ensures : (pr:u_prog)(u_prog_not_empty pr)->(p,q,r:u_prop)
  (u_stable pr r)->
  (u_unless pr (u_and r p) (u_imp r q))->
  (u_transient pr (u_and r (u_and p (u_not q))))->
  (u_ensures pr (u_and r p) (u_and r q)).
```

u_leads_to can be defined inductively on top of u_ensures. Remember that the intended meaning of (u_leads_to pr p q) is that if p holds in an arbitrary state then q will hold eventually in the future. This is obviously a weaker property than (u_ensures pr p q), justifying the first implication in the inductive definition below. (u_ensures pr p q) admits a situation where p is satisfied that p becomes false before q becomes true. It is easy to see that transitivity is a property of u_leads_to which does not hold for u_ensures in general, justifying the second implication below. Finally, we require the disjunction property: If (u_leads_to pr (P i) q) for a family P of state predicates then we can set p equivalent to the disjunction of the family P and conclude (u_leads_to pr p q). The justification is straightforward: Give a state s satisfying the disjunction for the family P there must be some index i such that (P i) is true at s. now we can use the premise (u_leads_to pr (P i) q) to infer that q will hold eventually.

```
Inductive u_leads_to [pr:u_prog] : u_prop->u_prop->Prop :=
  u_leads_to_basis : (p,q:u_prop)
  (u_ensures pr p q) -> (u_leads_to pr p q)
| u_leads_to_transitivity : (p,q,r:u_prop)
  (u_leads_to pr p q) -> (u_leads_to pr q r) -> (u_leads_to pr p r)
| u_leads_to_disjunction : (T:Set)(P:T->u_prop)(p,q:u_prop)
  ((i:T)(u_leads_to pr (P i) q)) -> (u_everywhere (u_iff p (u_ex ? P))) ->
  (u_leads_to pr p q).
```

The use of an equivalence instead of an equality in u_leads_to_disjunction is important in the framework of an intensional logic, otherwise certain u_leads_to assertions and also a substitution rule could not be proved.

In summary the inductive definition defines u_leads_to as the smallest predicate satisfying the three properties u_leads_to_basis, u_leads_to_transitivity and u_leads_to_disjunction. We have seen that using the state transition sequence semantics all these properties can be justified. In other words the system-based definition u_leads_to is sound w.r.t. to the sequence-based definition leads_to of Section 2.3. Of course, soundness is the most important feature of a verified library like the one we are aiming at. This is the reason why we introduce only a few well understood temporal assertions by system-based definitions which are obviously sound w.r.t. the sequence-based definitions.

Although the definitions for temporal assertions, including the one for u_leads_to, are clearly sound, it is not obvious at all that every sequence-based leads_to assertion which is satisfied can actually be proved using the three properties in the system-based definition above. In other words, there may be some doubt about the completeness of u_leads_to. Here we can say that the definition of u_leads_to is indeed complete. A mathematical proof is contained in [30][31] and a formal proof has been carried out in [17]. However, it should also be stressed that the practical relevance of completeness results is questionable: For instance, an incomplete system may be very useful in practice if verifiability of the full class of assertions is not required. On the other hand, a complete system is not necessarily efficient in terms of proof complexity: So it might happen that a (theoretically) complete system is practically incomplete, simply because the effort to prove certain properties is far beyond any realistic assumptions.

Now let us continue with proof rules for `u_leads_to` assertions. The following rule follows from `u_ensures_true`.

```
Theorem u_leads_to_true : (pr:u_prog)(u_prog_not_empty pr)->
  (p:u_prop)(u_leads_to pr p u_true).
```

Next we observe that the more standard form of `u_leads_to_disjunction` is obtained as a special case.

```
Theorem u_leads_to_disjunction_2 :
  (pr:u_prog)(T:Set)(P:T->u_prop)(q:u_prop)
  ((i:T)(u_leads_to pr (P i) q)) -> (u_leads_to pr (u_ex ? P) q).
```

Due to the careful inductive definition `u_leads_to` enjoys substitution rules for both sides.

```
Theorem u_leads_to_subst_r : (pr:u_prog)(p,q,q':u_prop)
  (u_everywhere (u_iff q q'))->(u_leads_to pr p q)->(u_leads_to pr p q').

Theorem u_leads_to_subst_l : (pr:u_prog)(p,p',q:u_prop)
  (u_everywhere (u_iff p p'))->(u_leads_to pr p q)->(u_leads_to pr p' q).
```

The following inductive definition of `u_lt_pre` is very similar to the definition of `u_leads_to`. The only difference is that `(u_leads_to pr)` is an inductively defined *binary relation* on state predicates, whereas `(u_lt_pre pr r)` is an inductively defined *set* of state predicates. The intention is that it contains those state predicates which are `u_leads_to` preconditions of some fixed state predicate `r`.

```
Inductive u_lt_pre [pr:u_prog;r:u_prop] : u_prop->Prop :=
  u_lt_pre_basis : (p:u_prop)
  (u_ensures pr p r) -> (u_lt_pre pr r p)
| u_lt_pre_transitivity : (p,q:u_prop)
  (u_ensures pr p q) -> (u_lt_pre pr r q) -> (u_lt_pre pr r p)
| u_lt_pre_disjunction : (T:Set)(P:T->u_prop)
  ((i:T)(u_lt_pre pr r (P i))) ->
  (p:u_prop)(u_everywhere (u_iff p (u_ex ? P))) ->
  (u_lt_pre pr r p).
```

So `(u_lt_pre pr q p)` is equivalent to `(u_leads_to pr p q)` as it is proved in the following two theorems. There are good reasons to set up an alternative definition: The second definition of an inductive set is structurally simpler. It shows that that the original definition of `u_leads_to` can be cast into a form which is a parameterized inductive definition (`pr` and `r` are parameters) of a set of state predicates. So the full power of binary inductive definitions is not necessary. Formally, the definitions give rise to different induction principles. So depending on the current goal we may prefer one definition or the other.

```
Theorem u_lt_pre_imp_leads_to : (pr:u_prog)(p,q:u_prop)
  (u_lt_pre pr q p) -> (u_leads_to pr p q).

Theorem u_leads_to_imp_precond : (pr:u_prog)(p,q:u_prop)
  (u_leads_to pr p q) -> (u_lt_pre pr q p).
```

Reflexivity of `u_leads_to` follows from reflexivity of `u_ensures`.

```
Theorem u_leads_to_refl : (pr:u_prog)(u_prog_not_empty pr)->
  (p:u_prop)(u_leads_to pr p p).
```

An implication can be cast into a `u_ensures` assertion by `u_ensures_imp` and further by the inductive definition into a `u_leads_to` assertion.

```
Theorem u_leads_to_imp : (pr:u_prog)(u_prog_not_empty pr)->
  (p,q:u_prop)(u_everywhere (u_imp p q)) -> (u_leads_to pr p q).
```

As a special case of `u_leads_to_imp` we can obtain:

```
Theorem u_leads_to_false : (pr:u_prog)(u_prog_not_empty pr)->
  (p:u_prop)(u_leads_to pr u_false p).
```

Using `u_leads_to_imp` and `u_leads_to_transitivity` an assertion `u_leads_to` can be strengthened on the left-hand side and weakened on the right-hand side as stated by the following two theorems.

```
Theorem u_leads_to_implication_l : (pr:u_prog)(u_prog_not_empty pr)->
  (p,q,r:u_prop)(u_everywhere (u_imp p q)) ->
  (u_leads_to pr q r) -> (u_leads_to pr p r).

Theorem u_leads_to_implication_r : (pr:u_prog)(u_prog_not_empty pr)->
  (p,q,r:u_prop)(u_everywhere (u_imp q r)) ->
  (u_leads_to pr p q) -> (u_leads_to pr p r).
```

Sometimes it is convenient to have strengthening and weakening available as a single proof rule:

```
Theorem u_leads_to_implication : (pr:u_prog)(u_prog_not_empty pr)->
  (p,p',q,q':u_prop)
  (u_everywhere (u_imp p p'))->
  (u_everywhere (u_imp q q'))->
  (u_leads_to pr p' q)-> (u_leads_to pr p q').
```

Right-hand side weakening can also be written with `u_or` and left-hand side strengthening can be written using `u_and`.

```
Theorem u_leads_to_weakening : (pr:u_prog)(u_prog_not_empty pr)->
  (p,q,r:u_prop)(u_leads_to pr p q)->(u_leads_to pr p (u_or q r)).
```

```
Theorem u_leads_to_strengthening : (pr:u_prog)(u_prog_not_empty pr)->
  (p,q,r:u_prop)(u_leads_to pr p q)->(u_leads_to pr (u_and p r) q).
```

As special cases of `u_leads_to_disjunction_2` we obtain `u_leads_to_binary_disjunction` and `u_leads_to_ternary_disjunction`. These special cases are needed much more often than the general disjunction rule.

```
Theorem u_leads_to_binary_disjunction :
  (pr:u_prog)(u_prog_not_empty pr)->(p,p',q:u_prop)
  (u_leads_to pr p q)->(u_leads_to pr p' q)->
  (u_leads_to pr (u_or p p') q).
```

```
Theorem u_leads_to_ternary_disjunction :
  (pr:u_prog)(u_prog_not_empty pr)->(p,p',p'',q:u_prop)
  (u_leads_to pr p q)->(u_leads_to pr p' q)->(u_leads_to pr p'' q)->
  (u_leads_to pr (u_or3 p p' p'') q).
```

We can also prove a more general form of `u_leads_to_disjunction` than required in the inductive definition. Actually we have a general disjunction rule for arbitrary families of `u_leads_to` assertions. Remember that the existential formulas `(u_ex ? P)` and `(u_ex ? Q)` can be seen as big disjunctions over families `P` and `Q`, respectively.

```
Theorem u_leads_to_big_disjunction : (pr:u_prog)(u_prog_not_empty pr)->
  (T:Set)(P,Q:T->u_prop)
  ((i:T)(u_leads_to pr (P i) (Q i)))->
  (u_leads_to pr (u_ex ? P) (u_ex ? Q)).
```

Using an index set `T` with two elements we obtain the small disjunction rule as a special case.

```
Theorem u_leads_to_small_disjunction :
  (pr:u_prog)(u_prog_not_empty pr) -> (p,q,p',q':u_prop)
  (u_leads_to pr p p') -> (u_leads_to pr q q')->
  (u_leads_to pr (u_or p q) (u_or p' q')).
```

A state predicate `p` with `(u_leads_to pr p u_false)` cannot be true in any state, otherwise a contradiction arises, since `u_false` would have to become true eventually.

```
Theorem u_leads_to_impossible : (pr:u_prog)(u_prog_not_empty pr)->
  (p:u_prop)(u_leads_to pr p u_false)->(u_everywhere (u_not p)).
```

The following cancelation rule is very useful in practice: Assume (1) `p` leads to `q` or `b` and (2) `b` leads to `r`. Then `p` leads either to a state satisfying `q` or if this is not the case it must lead to a state satisfying `b` which in turn leads to `r`. So we can conclude that `p` leads to `q` or `r`.

```
Theorem u_leads_to_cancelation : (pr:u_prog)(u_prog_not_empty pr) ->
  (p,q,b,r:u_prop)(u_leads_to pr p (u_or q b))->(u_leads_to pr b r)->
  (u_leads_to pr p (u_or q r)).
```

A very powerful rule is the following progress-safety-progress rule `u_leads_to_psp`, which combines a liveness assertion and a safety assertion and allows us to infer a new liveness assertion. The formal proof is rather tedious but the intuitive idea is easy: Assume (1) `p` leads to `q` and that (2) `r`, once it is true, cannot become false before `b` becomes true. Now there are two possibilities in a state statisfying `p` and `r`: Either `b` remains false forever or it becomes true eventually. In the first case `p` and `r` leads to `q` and `r` is justified by the fact that `r` remains true. In the second case it is obvious that `p` and `r` leads to `b`.

The proof of the progess-safety-progress rule `u_leads_to_psp` is by induction over the `u_leads_to` premise. First we prove the base case, i.e. the progress-safety-progress rule for `u_ensures`.

```
Theorem u_ensures_psp : (pr:u_prog)(u_prog_not_empty pr)->
  (p,q,r,b:u_prop)(u_ensures pr p q)->(u_unless pr r b)->
  (u_ensures pr (u_and p r) (u_or (u_and q r) b)).
```

```
Theorem u_leads_to_psp : (pr:u_prog)(u_prog_not_empty pr)->
  (p,q,r,b:u_prop)(u_leads_to pr p q)->(u_unless pr r b)->
  (u_leads_to pr (u_and p r) (u_or (u_and q r) b)).
```

Another way to combine safety and liveness is the following: A `u_leads_to` assertion can be strengthened on both sides by a state predicate which is stable. Notice that, whereas strengthening on the left-hand side is always possible, strengthening on the right-hand side only would be incorrect.

```
Theorem u_leads_to_stable_conjunction : (pr:u_prog)(u_prog_not_empty pr)->
  (p,q,r:u_prop)(u_stable pr r)->(u_leads_to pr p q)->
  (u_leads_to pr (u_and p r) (u_and q r)).
```

Using `u_leads_to_stable_conjunction` we can prove `u_rel_leads_to` which completes out list of proof rules for relativized assertions.

```
Theorem u_rel_leads_to : (pr:u_prog)(u_prog_not_empty pr)->(p,q,r:u_prop)
  (u_stable pr r)->
  (u_leads_to pr (u_and r p) (u_imp r q))->
  (u_leads_to pr (u_and r p) (u_and r q)).
```

Often we have to reason about `u_leads_to` assertions in the presence of invariants. Remember that `(u_invariant pr p r)` states that `r` is an inductive invariant of program `pr` with initial condition `p`. More precisely, it says `p` implies `r` and `r` is stable. By the previous theorem we have `(u_leads_to pr (u_and p r) (u_and q r))` and weakening the left-hand side yields `(u_leads_to p (u_and q r))`.

```
Theorem u_ind_invariant_imp_leads_to : (pr:u_prog)(p,q,r:u_prop)
  (u_prog_not_empty pr)->(u_ind_invariant pr p r)->
  (u_leads_to pr p q)->(u_leads_to pr p (u_and q r)).
```

In addition to the characterization of `u_leads_to` by `u_lt_pre` there is another characterization given by the following inductive definition of `u_strong_leads_to`. The difference w.r.t. the inductive definition of `u_leads_to` is that transitivity is expressed by `(u_ensures pr p q)->(u_strong_leads_to pr q r) -> (u_strong_leads_to pr p r)` instead of `(u_leads_to pr p q)->(u_leads_to pr q r) -> (u_leads_to pr p r)`. The advantage of `u_strong_leads_to` is that a stronger induction principle is available due to the stronger premise `(u_ensures pr p q)` instead of `(u_leads_to pr p q)`.

```
Inductive u_strong_leads_to [pr:u_prog] : u_prop->u_prop->Prop :=
  u_strong_leads_to_basis : (p,q:u_prop)
  (u_ensures pr p q) -> (u_strong_leads_to pr p q)
| u_strong_leads_to_transitivity : (p,q,r:u_prop)
  (u_ensures pr p q) -> (u_strong_leads_to pr q r) ->
                        (u_strong_leads_to pr p r)
| u_strong_leads_to_disjunction : (T:Set)(P:T->u_prop)(p,q:u_prop)
  ((i:T)(u_strong_leads_to pr (P i) q)) ->
  (u_everywhere (u_iff p (u_ex ? P))) ->
  (u_strong_leads_to pr p q).
```

The equivalence between `u_strong_leads_to` and `u_leads_to` is stated in the following two theorems.

```
Theorem u_strong_leads_to_imp_leads_to :
  (pr:u_prog)(u_prog_not_empty pr)->(p,q:u_prop)
  (u_strong_leads_to pr p q)->(u_leads_to pr p q).
```

```
Theorem u_leads_to_imp_strong_leads_to :
  (pr:u_prog)(u_prog_not_empty pr)->(p,q:u_prop)
  (u_leads_to pr p q)->(u_strong_leads_to pr p q).
```

Via this equivalence the strong induction principle for `u_strong_leads_to` is inherited by `u_leads_to`. It is formulated in the following theorem.

```
Theorem u_leads_to_strong_induction :
  (IH:u_prog->u_prop->u_prop->Prop)(pr:u_prog)
  (u_prog_not_empty pr)->
  ((p,q:u_prop)(u_ensures pr p q)->(IH pr p q))->
  ((p,q,r:u_prop)
   ((u_ensures pr p q)->(u_leads_to pr q r)->
    (IH pr p q)->(IH pr q r)->(IH pr p r)))->
  ((T:Set)(P:T->u_prop)(p,q:u_prop)
   ((i:T)(u_leads_to pr (P i) q))->((i:T)(IH pr (P i) q))->
   (u_everywhere (u_iff p (u_ex ? P))) -> (IH pr p q))->
  ((p,q:u_prop)
   (u_leads_to pr p q)->(IH pr p q)).
```

With the previous induction principle we can prove certain statements of the form `((p,q:u_prop)(u_leads_to pr p q)->(IH pr p q))`. Later, however, we will be in a situation where we have to prove a statement with two `u_leads_to` premises of the form `((p,q,p',q':u_prop)(u_leads_to pr p q)->(u_leads_to pr p' q')->(IH pr p q p' q'))`. For this purpose we could employ the following double induction principle. The proof is done by nested applications of `u_leads_to_strong_induction`.

```
Theorem u_leads_to_double_induction :
  (IH:u_prog->u_prop->u_prop->u_prop->u_prop->Prop)(pr:u_prog)
  (u_prog_not_empty pr)->
  ((p,q,p',q':u_prop)
   (u_ensures pr p q)->(u_leads_to pr p' q')->(IH pr p q p' q'))->
  ((p,q,p',q':u_prop)
   (u_leads_to pr p q)->(u_ensures pr p' q')->(IH pr p q p' q'))->
  ((p,q,r,p',q',r':u_prop)
   ((u_ensures pr p q)->(u_leads_to pr q r)->
    (u_ensures pr p' q')->(u_leads_to pr q' r')->
    (IH pr p q p' q')->
    (IH pr p q q' r')->(IH pr q r p' q')->
    (IH pr q r q' r')->
    (IH pr p r q' r')->(IH pr q r p' r')->
    (IH pr p r p' r')))->
  ((T:Set)(P':T->u_prop)(p,q,p',q':u_prop)
    (u_leads_to pr p q)->
    ((i:T)(u_leads_to pr (P' i) q'))->
    ((i:T)(IH pr p q (P' i) q'))->
    (u_everywhere (u_iff p' (u_ex ? P')))->
    (IH pr p q p' q'))->
  ((T:Set)(P:T->u_prop)(p,q,p',q':u_prop)
    ((i:T)(u_leads_to pr (P i) q))->
    (u_leads_to pr p' q')->
    ((i:T)(IH pr (P i) q p' q'))->
    (u_everywhere (u_iff p (u_ex ? P)))->
    (IH pr p q p' q'))->
  ((p,q,p',q':u_prop)
   (u_leads_to pr p q)->(u_leads_to pr p' q')->(IH pr p q p' q')).
```

The double induction principle is more general than necessary if the formula `(IH pr p q p' q')` is logically symmetric w.r.t. to replacement of `(p,q)` by `(p',q')`. Actually, this will be the case in our application which is the proof of the completion theorem. In the presence of such a symmetry `u_leads_to_double_induction` can be simplified to `u_leads_to_symmetric_double_induction` which reduces the number of premises to obtain the same goal.

```
Theorem u_leads_to_symmetric_double_induction :
  (IH:u_prog->u_prop->u_prop->u_prop->u_prop->Prop)
  (pr:u_prog)(u_prog_not_empty pr)->
```

```
((p,q,p',q':u_prop)(IH pr p q p' q')->(IH pr p' q' p q))->
((p,q,p',q':u_prop)
 (u_ensures pr p q)->(u_leads_to pr p' q')->(IH pr p q p' q'))->
((p,q,r,p',q',r':u_prop)
 ((u_ensures pr p q)->(u_leads_to pr q r)->
  (u_ensures pr p' q')->(u_leads_to pr q' r')->
  (IH pr p q p' q')->
  (IH pr p q q' r')->(IH pr q r p' q')->
  (IH pr q r q' r')->
  (IH pr p r q' r')->(IH pr q r p' r')->
  (IH pr p r p' r')))->
((T:Set)(P:T->u_prop)(p,q,p',q':u_prop)
  ((i:T)(u_leads_to pr (P i) q))->
  (u_leads_to pr p' q')->
  ((i:T)(IH pr (P i) q p' q'))->
  (u_everywhere (u_iff p (u_ex ? P)))->
  (IH pr p q p' q'))->
((p,q,p',q':u_prop)
 (u_leads_to pr p q)->(u_leads_to pr p' q')->(IH pr p q p' q')).
```

Our next major goal is the to prove the general UNITY completion rule given by Theorem `u_leads_to_completion`. We will proceed using various intermediate statements, but first we will give an informal explanation of the completion rule which is very important in practice: Basically the rule combines several `u_leads_to` assertions into a single one. Of course, we cannot expect a conjunction rule of a family of `u_leads_to` assertions since the right-hand sides do not necessarily become true simultaneously. However, under some additional restrictions we can approximate this idea: Assume a finite family of assertions `(u_leads_to pr (P i) (Q i))` and assume we know that `u_unless pr (Q i) r`, i.e. the states satisfying `(Q i)` can only be left via `r`. Then we can conclude that `(u_all T P)` leads to `(u_all T Q)` or `r` holds. The disjunction on the right-hand side corresponds to the two possible cases: Either all `(Q i)` will simultaneously become true at some time or if this is not the case at least one of the state predicates `(Q i)` will become true and then false again. But in this case `r` will become `true`. Actually, the following theorem is a little bit more general, as it assumes `(u_leads_to pr (P i) (u_or (Q i) r))`, because a sutuation where `r` becomes true leads directly to our goal.

The proof in [5] is rather informal, since it relies on induction over the length of proofs which are themselves not formalized (as syntactic objects). Consequently, the length of proofs remains a rather vague notion, in particular in view of the infinitary inductive definition of `leads_to`. Here we adopted a different approach based on structural induction which is well supported by CIC: We will employ a very general double induction principle for `u_leads_to`, namely `u_leads_to_symmetric_double_induction`, which has been proved before. Actually it has been isolated since it can be of use for other purposes. Using this induction principle we prove `u_leads_to_simple_conjunction`, which is also proposed as an intermediate step in [5]. From this we obtain the binary completion theorem `u_leads_to_simple_completion`. Finally, we prove the general completion theorem `u_leads_to_completion` by induction via the intermediate statement `u_leads_to_completion_nat`.

First we prove the base case of u_leads_to_simple_conjunction where both u_leads_to premises are actually u_ensures assertions.

```
Lemma u_leads_to_simple_conjunction_basis : (pr:u_prog)(u_prog_not_empty pr)->
  (p,q,p',q',r:u_prop)
  (u_ensures pr p q)->
  (u_leads_to pr p' q')->
  (u_unless pr q r)->
  (u_unless pr q' r)->
  (u_leads_to pr (u_and p p') (u_or (u_and q q') r)).
```

Next we prove u_leads_to_simple_conjunction which is obviously symmetric w.r.t. to replacement of (p,q) by (p',q'). So we can immediately apply u_leads_to_symmetric_double_induction. The base case has been treated by the previous lemma. The remaining induction steps are contained in the proof of the theorem.

```
Theorem u_leads_to_simple_conjunction : (pr:u_prog)(u_prog_not_empty pr)->
  (p,q,p',q',r:u_prop)
  (u_leads_to pr p q)->
  (u_leads_to pr p' q')->
  (u_unless pr q r)->
  (u_unless pr q' r)->
  (u_leads_to pr (u_and p p') (u_or (u_and q q') r)).
```

By applying u_unless_simple_disjunction and u_leads_to_simple_conjunction to the previous result we obtain the binary version of the completion theorem which has two u_leads_to premisses.

```
Theorem u_leads_to_simple_completion : (pr:u_prog)(u_prog_not_empty pr)->
  (p,q,p',q',r:u_prop)
  (u_leads_to pr p (u_or q r))->
  (u_leads_to pr p' (u_or q' r))->
  (u_unless pr q r)->
  (u_unless pr q' r)->
  (u_leads_to pr (u_and p p') (u_or (u_and q q') r)).
```

The general completion theorem u_leads_to_completion which has an arbitrary finite family of u_leads_to premises can be reduced to the following special case where the family is indexed by an initial interval of the natural numbers. The proof is done by induction over b using the previous result for the binary case.

```
Theorem u_leads_to_completion_nat : (pr:u_prog)(u_prog_not_empty pr)->
  (b:nat)(P,Q:nat->u_prop)(r:u_prop)
  ((i:nat)(lt i b)->(u_leads_to pr (P i) (u_or (Q i) r)))->
  ((i:nat)(lt i b)->(u_unless pr (Q i) r))->
  (u_leads_to pr (u_all_nat b P) (u_or (u_all_nat b Q) r)).
```

To formulate the next theorem we need some auxiliary definitions concerning finiteness of types. (card_le n X), (card_ge n X) and (card_eq n X) are propositions stating that the cardinality of a type X is less or equal, greater or equal or equal to n. Finiteness of X is expressed by (finite X) and can be naturally defined by requiring that the cardinality of X is smaller than some natural number.

```
Definition card_le := [n:nat][X:Set]
  (Ex [f:nat->X] ((x:X) (Ex [i:nat] (lt i n) /\ ((f i) = x)))).
```

```
Definition card_ge := [n:nat][X:Set]
  (Ex [f:X->nat] ((i:nat) (lt i n) -> (Ex [x:X] ((f x) = i)))).
```

```
Definition card_eq := [n:nat][X:Set]
  (Ex [f:nat->X] (Ex [g:X->nat] ((x:X) ((f (g x)) = x) /\ (lt (g x) n)))).
```

```
Definition finite := [X:Set] (Ex [n:nat] (card_le n X)).
```

Now the proof of the general completion theorem u_leads_to_completion follows by exploiting the definition of (finite T).

```
Theorem u_leads_to_completion : (pr:u_prog)(u_prog_not_empty pr)->
  (T:Set)(finite T)->(P,Q:T->u_prop)(r:u_prop)
  ((i:T)(u_leads_to pr (P i) (u_or (Q i) r)))->
  ((i:T)(u_unless pr (Q i) r))->
  (u_leads_to pr (u_all T P) (u_or (u_all T Q) r)).
```

Finally we turn to another family of proof rules addressing the fact that and important method to prove u_leads_to assertions is induction. Here we will discuss very general proof rules for well-founded induction. Later we will give a specialization to induction over natural numbers. It should be kept in mind, however, that many other well-founded orderings are useful for proving liveness assertions. This leads to further specializations and is clearly an important point for future extensions of the library.

We start with a general well-founded induction rule which assumes that the induction is carried out over a state function variant mapping states into some domain. We assume a well-founded ordering less on this domain which does not even have to be total. The predicate well_founded is defined in the mathematical library. It states that the ordering does not admit any infinite decreasing chain. The goal of the subsequent proof rule is that p leads to q. To guarantee this property we use the following condition as premise: If p holds and the value of variant equals m then there are two possibilities: (1) We will eventually reach a state satisfying q. In this case the conclusion is justified. (2) We will eventually reach a state satisfying p again, but now with the value of variant being decreased w.r.t. m. It is clear that we have a kind of loop where choice (2) can be taken only finitely often due to the well-foundedness condition. So choice (1) has to be taken eventually and q will be satisfied then.

```
Theorem u_leads_to_induction : (pr:u_prog)(u_prog_not_empty pr)->
  (T:Set)(less:T->T->Prop)(variant:u_st->T)
  (well_founded T less)->
  (p,q:u_prop)
  ((m:T)(u_leads_to pr (u_and p [s:u_st] (variant s) = m)
                       (u_or (u_and p [s:u_st] (less (variant s) m)) q)))->
  (u_leads_to pr p q).
```

The following induction rule is a specialization of the preceding one: Instead of a general state function `variant` we use a program variable `M`.

```
Theorem u_leads_to_var_induction : (pr:u_prog)(u_prog_not_empty pr)->
  (M:u_var)(less:(u_var_type M)->(u_var_type M)->Prop)
  (well_founded (u_var_type M) less)->
  (p,q:u_prop)
  ((m:(u_var_type M))(u_leads_to pr (u_and p [s:u_st] (s M) = m)
                      (u_or (u_and p [s:u_st] (less (s M) m)) q)))->
  (u_leads_to pr p q).
```

Now we substitute `less` by `lt`, the usual well-founded strict total order on `nat`, and obtain the following induction rule for natural numbers.

```
Theorem u_leads_to_nat_induction : (pr:u_prog)(u_prog_not_empty pr)->
  (variant:u_st->nat)
  (p,q:u_prop)
  ((m:nat)(u_leads_to pr (u_and p [s:u_st] (variant s) = m)
                         (u_or (u_and p [s:u_st] (lt (variant s) m)) q)))->
  (u_leads_to pr p q).
```

In supplement to the previous induction principle where the value decreases during the execution we provide a reverse induction principle where the variant increases. Of course, to ensure that the conclusion holds there must be some upper bound for the variant which is specified by `maximum`. In the proof a new decreasing variant `[s:u_st](minus maximum (variant s))` is defined and the previous induction rule for natural numbers is employed.

```
Theorem u_leads_to_rev_nat_induction :
  (pr:u_prog)(u_prog_not_empty pr)->
  (variant:u_st->nat)(maximum:nat)(p,q:u_prop)
  (u_everywhere (u_imp p [s:u_st](le (variant s) maximum)))->
  ((m:nat)(le m maximum)->
     (u_leads_to pr (u_and p [s:u_st] m = (variant s))
                    (u_or (u_and p [s:u_st] (lt m (variant s))) q)))->
  (u_leads_to pr p q).
```

4.10 Composition by Union

Compositionality is important to achieve modularity of correctness proofs. This is in particular of interest for large systems composed of verified subsystems. Compositional verification means that in order to prove a system property we can rely on verified specifications of its modules and we do not need to look further into their structure. The natural composition operation on programs or functional transition systems is the union, i.e. the transitions of the composed systems are obtained by union of transitions of its modules. For simplicity, we assume a global state space `u_st` and a global space of transition names `u_tr`. So composition becomes union of programs as defined in Section 4.5. An interesting feature of UNITY logic is that almost all of its operators enjoy elementary compositionality properties which are presented in this section.

```
Theorem u_co_union : (pr,pr':u_prog)(p,q:u_prop)
  (u_co pr p q) -> (u_co pr' p q) ->
  (u_co (u_prog_union pr pr') p q).

Theorem u_stable_union : (pr,pr':u_prog)(p:u_prop)
  (u_stable pr p) -> (u_stable pr' p) ->
  (u_stable (u_prog_union pr pr') p).

Theorem u_indinv_stable_union : (pr,pr':u_prog)(initially:u_prop)(p:u_prop)
  (u_ind_invariant pr initially p) -> (u_stable pr' p) ->
  (u_ind_invariant (u_prog_union pr pr') initially p).

Theorem u_stable_indinv_union : (pr,pr':u_prog)(initially:u_prop)(p:u_prop)
  (u_stable pr' p) -> (u_ind_invariant pr initially p) ->
  (u_ind_invariant (u_prog_union pr pr') initially p).

Theorem u_unless_union : (pr,pr':u_prog)(p,q:u_prop)
  (u_unless pr p q) -> (u_unless pr' p q) ->
  (u_unless (u_prog_union pr pr') p q).

Theorem u_unless_stable_union : (pr,pr':u_prog)(p,q:u_prop)
  (u_unless pr p q) -> (u_stable pr' p) ->
  (u_unless (u_prog_union pr pr') p q).

Theorem u_transient_union : (pr,pr':u_prog)(p:u_prop)
  ((u_transient pr p) \/ (u_transient pr' p)) ->
  (u_transient (u_prog_union pr pr') p).

Theorem u_transient_union_l : (pr,pr':u_prog)(p:u_prop)
  (u_transient pr p) ->
  (u_transient (u_prog_union pr pr') p).

Theorem u_transient_union_r : (pr,pr':u_prog)(p:u_prop)
  (u_transient pr' p) ->
  (u_transient (u_prog_union pr pr') p).
```

```
Theorem u_ensures_unless_union : (pr,pr':u_prog)(p,q:u_prop)
  (u_ensures pr p q) -> (u_unless pr' p q) ->
  (u_ensures (u_prog_union pr pr') p q).

Theorem u_unless_ensures_union : (pr,pr':u_prog)(p,q:u_prop)
  (u_unless pr p q) -> (u_ensures pr' p q) ->
  (u_ensures (u_prog_union pr pr') p q).

Theorem u_ensures_stable_union : (pr,pr':u_prog)(p,q:u_prop)
  (u_ensures pr p q) -> (u_stable pr' p) ->
  (u_ensures (u_prog_union pr pr') p q).

Theorem u_stable_ensures_union : (pr,pr':u_prog)(p,q:u_prop)
  (u_stable pr p) -> (u_ensures pr' p q) ->
  (u_ensures (u_prog_union pr pr') p q).
```

It is remarkable that the only operators which are apparently not compositional (in the direct sense) are `u_invariant` and `u_leads_to`. The most direct solution is just to work on the level of compositional operators like `u_ind_invariant` and `u_ensures` as far as possible. Finally programs and properties are composed and `u_invariant` and `u_leads_to` assertions can be derived.

For the verification library the addition of more advanced composition rules is planned in the future. As shown in [8] there are possibilities of compositional reasoning with `u_leads_to` in a rely-guarantee style. Conditional assertions which can be easily expressed in higher-order logic are a natural means to support such kind of reasoning without modifying the temporal logic itself.

Chapter 5

An Example Application

In this chapter we demonstrate the use of the library by an intuitively simple but formally still interesting and non-trivial example. We describe a completely formal specification and verification of the "Common Meeting Time Problem" which can be stated as follows.

5.1 The Common Meeting Time Problem

Modeling time by natural numbers the problem is to find the earliest possible common meeting time for two persons. We assume that a least common meeting time exists. For each of the two persons we assume a function, say `f` and `g`, respectively, such that `f(time)` and `g(time)` is the earliest availability time not earlier than `time`.

An easy solution is provided by the following UNITY program:

```
Program CMT
declare time : nat
initially time = 0
assign time := f(time)
     | time := g(time)
end
```

Notice that in general the fairness requirement that each statement is executed infinitely often is sufficient to gurantee that this algorithm will find the common meeting time eventually.

5.2 A Concrete Frame

Formally we replace the abstract frame of the module `abstract.v` explained in Section 4.2 by a concrete frame, which starts with the assumptions stated in the informal explanation of the problem and the program given above. In this sense we can say that our UNITY library is instantiated for a particular problem. The module discussed in the following is called `cmt.v`.

```
Load unity_init.
```

It follows from the informal problem specification that we can assume two functions f and g on natural numbers which are ascending and monotone. Below le is the less-or-equal predicate on natural numbers as defined in the mathematical library.

```
Variables f,g : nat->nat.
Axiom f_is_ascending : (m:nat)(le m (f m)).
Axiom g_is_ascending : (m:nat)(le m (g m)).
Axiom f_is_monotone : (m,n:nat)(le m n)->(le (f m) (f n)).
Axiom g_is_monotone : (m,n:nat)(le m n)->(le (g m) (g n)).
```

The concept of common meeting time is formulated as a common fixpoint of f and g.

```
Definition common := [t:nat] (f t)=t /\ (g t)=t.
```

Next we assume that a smallest common meeting time exists, as our informal specification states. The smallest common meeting time itself is assumed to be cmt. This variable is useful for specification of verification purposes, but, of course, it should not be used by the program.

```
Variable cmt : nat.
Axiom cmt_exists : (common cmt).
Axiom cmt_smallest : (m:nat)(common m)->(le cmt m).
```

According to the problem specification these are all assumptions the program can rely on. Next, we describe the concrete embedding of the UNITY program seen as a functional transition system in type theory. Here, we carry out the transformation manually, but it should be obvious that a mechanization of this process could easily be realized.

The program contains exactly one program variable time. It is represented by a variable name u_time which is the only element of the following inductive type u_var. For simple programs like ours the inductive type may be a simple enumeration type, but there are also occasions where the full power of inductive types can be exploited, e.g. to represent families of program variables.

```
Inductive u_var : Set := u_time : u_var.
```

Here we repeat that there is only one variable, formally u_vars is defined as 1, and we provide an enumeration function u_ith_var of u_var as required by the abstract frame. Also we have to provide an easy proof of finiteness for u_var.

```
Definition u_vars := (S O).

Definition u_ith_var := [i:nat] u_time.

Theorem u_var_finite : (v:u_var)
  (Ex [i:nat] (lt i u_vars) /\ (u_ith_var i)=v).
```

The abstract frame also requires a decision function for equality on program variables. The following theorem provides this trivial decision function together with a proof of its correctness.

```
Theorem u_var_eq_dec : (v,v':u_var){(v=v')}+{~(v=v')}.
```

In addition to a variable name we need names for all types occurring in the program. Again we have only one type name `u_nat` intended to represent the type `nat` of natural numbers. Decidability of type name equality is again trivial.

```
Inductive u_type_name : Set := u_nat : u_type_name.
```

```
Theorem u_type_name_eq_dec:(n,n':u_type_name){(n=n')}+{~(n=n')}.
```

Now we have translated all names used in the program into type theoretic entities: `f` and `g` become variables of the type theory, `time` and `nat` are names represented in type theory by `u_time` and `u_nat`, respectively.

As a next step, the type name `u_nat` must be associated with a CIC type, namely `nat`. This is done by the following function `u_type` mapping `u_nat` to `nat`.

```
Definition u_type := [n:u_type_name]
  Cases n of u_nat => nat end.
```

We have not yet declared the types of program variables. This is done by the following function `u_var_type_name` which maps `u_time` to its associated type name `u_nat`.

```
Definition u_var_type_name := [v:u_var]
  Cases v of u_time => u_nat end.
```

Now, after all variables together with their types have been fixed the state space `u_st` is defined. This is done by the module `unity_states.v` again following the scheme provided by the abstract frame.

```
Load unity_states.
```

After setting up the state space we have to represent the active part of the UNITY program. Again we start by introducing names as objects of the type theory. `u_tr` is an inductive type containing two transitions (more precisely transition names) `update_f` and `update_g` intended to represent the two statements the UNITY program can execute.

```
Inductive u_tr : Set :=
  update_f : u_tr | update_g : u_tr.
```

As usual for names we show decidability:

```
Lemma u_tr_eq_dec:(t,t':u_tr){(t=t')}+{~(t=t')}.
```

That the transitions introduced above are indeed related to the body of the UNITY program is enforced by the following definition of the state transition function `u_effect` which specifies the effect of each transition on the state. Typically it will take the form of a case analysis over all transitions.

```
Definition u_effect := [t:u_tr]
  Cases t of
          update_f => [s:u_st] (u_assign s u_time (f (s u_time)))
        | update_g => [s:u_st] (u_assign s u_time (g (s u_time)))
  end.
```

Now the type of programs is introduced in `unity_prog.v` and finally the full UNITY program is represented as a list of transitions. In our example there are only two transitions: `update_f` and `update_g`.

```
Load unity_program.

Definition u_pr := (cons update_f (cons update_g (nil u_tr))).
```

On the top of our concrete program specification we include the remainder of the general library which is thereby instantiated for our particular application. More precisely, it is instantiated for our particular choice of `u_st` and `u_tr`. However, as temporal operators are parameterized by the program, we have the freedom to reason about other (sub-)programs or composed programs on the same state space and on the same set of transitions.

```
Load unity_logic.
Load unity_depend.
Load unity_safety.
Load unity_liveness.
```

5.3 Verification of Safety Assertions

So far a concrete frame has been set up which can be used to formulate and prove concrete properties for our specific example. As usual in mathematics and verification the proof is structured into a couple of lemmas. The overall strategy is to establish safety assertions first and then on top of this liveness assertions are proved. The three theorems we prove will correspond exactly to our informal specification of a solution to the problem.

We start with a lemma which is used to prove the next statement `CMT_S2`. It states that if `time` has a value `m` then at the successor state `time` will not increase above `(f m)` or it will not increase above `(g m)`. The proof is easy by unfolding `u_co` and `u_pr`. Then the condition is verified for both transitions `update_f` and `update_g`.

```
Lemma CMT_S1 : (m:nat)(u_co u_pr ([s:u_st] (s u_time)=m)
                    ([s:u_st] (le (s u_time) (f m)) \/ (le (s u_time) (g m)))).
```

Following [28] we use the elimination rule on the program variable `u_time`. See Section 4.8 for the intuitive explanation and our formalization of the elimination rule. We instantiate `x`, `p` and `q` in `u_elimination` by `u_time`, `[s:u_st](common n)->(le (s u_time) n)` and `[m:nat][s:u_st](le (s u_time) (f m))\/(le (s u_time) (g m))`, respectively. Independence of `p` from other variables than `x` is trivially satisfied, as `u_time` is the only program variable. The remaining `u_co` premise is just the preceding lemma. So we obtain the following `u_co` assertion from the conclusion of `u_elimination`.

```
Lemma CMT_S2 : (n:nat)(u_co u_pr
  [s:u_st]((common n)->(le (s u_time) n))
  (u_ex ? [m:nat] [s:u_st]((common n)->(le m n))/\
                        ((le (s u_time) (f m))\/(le (s u_time) (g m))))).
```

The next lemma states that `time` cannot skip any common meeting time. The proof starts with the `u_co` assertions of the preceding lemma and applies right-hand side weakening to obtain the `u_stable` assertion.

```
Lemma CMT_S3 : (n:nat)(u_stable u_pr
           ([s:u_st] (common n) -> (le (s u_time) n))).
```

Together with the fact that `([s:u_st] (common n) -> (le (s u_time) n)))` holds initially, i.e. it is implied by `[s:u_st](s u_time)=O` we establish an inductive invariant: `time` stays below any common meeting time and in particular below the smallest one `cmt`.

```
Lemma CMT_S4 : (n:nat)(u_ind_invariant u_pr [s:u_st](s u_time)=O
           ([s:u_st] (common n) -> (le (s u_time) n))).
```

Substituting `cmt` for `n` we obtain a first major correctness property of our program: `time` cannot increase beyond the smallest common meeting time `cmt`.

```
Theorem CMT_S5 : (u_ind_invariant u_pr
  [s:u_st](s u_time)=O [s:u_st](le (s u_time) cmt)).
```

The next theorem states that `time` can only increase but never decrease. Together with the previous theorem this establishes *partial correctness* of the program: `time` can only increase but not beyond `cmt`. The proof is done by unfolding `u_co` and `u_pr` and then by considering each of the two transitions `update_f` and `update_g` together with the axioms `f_is_ascending` and `g_is_ascending`, respectively.

```
Theorem CMT_S6 : (m:nat)
  (u_co u_pr [s:u_st]m=(s u_time) [s:u_st](le m (s u_time))).
```

5.4 Verification of Liveness Assertions

In order to establish *total correctness* we have to prove the liveness assertion stating that the smallest common meeting time `cmt` is reached eventually. This will be done in this section.

First we observe that our program is not empty, a precondition needed for some temporal proof rules of liveness assertions.

```
Lemma u_pr_ne : (u_prog_not_empty u_pr).
```

The final liveness property we wish to prove is `(u_leads_to u_pr ([s:u_st](s u_time)=0) ([s:u_st](s u_time)=cmt))` as formulated in Theorem `CMT_L15`. To this end we employ (reverse) induction on natural numbers according to Theorem `u_leads_to_rev_nat_induction`. We have to supply a variant which increases up to some maximum value. The choice is obvious: Our variant is simply the value of `time` and the maximum value which can be reached is the smallest common meeting time `cmt`.

```
Definition variant := [s:u_st](s u_time).
```

First we prove that the variant cannot skip the smallest common meeting time, which can be done easily using `CMT_S3` with `n` instantiated by `cmt`.

```
Lemma CMT_S7 : (u_stable u_pr [s:u_st](le (variant s) cmt)).
```

In the following we prepare the application of the reverse induction rule `u_leads_to_rev_nat_induction` which will be used later to prove Lemma `CMT_L14` which in turn immediately implies the final liveness property `CMT_L15`. The `u_leads_to` assertion in the premise of the reverse induction rule is proved in Lemma `CMT_L13` directly as a `u_ensures` assertion. The safety part of this assertion is stated in the following `u_unless` assertion: If `time` is not beyond `cmt` then there are only two possibilities to leave such a state: Either we reach `cmt` or `time` increases but not beyond `cmt`. The proof unfolds `u_unless` and applies `u_co_stable_conjunction` to `CMT_S7` and `CMT_S6`. Then `u_co_implication` is used to obtain the goal.

```
Lemma CMT_S8 : (m:nat)
  (u_unless u_pr
     (u_and [s:u_st](le (s u_time) cmt) [s:u_st]m=(s u_time))
     (u_or
       (u_and [s:u_st](le (s u_time) cmt) [s:u_st](lt m (s u_time)))
       [s:u_st](s u_time)=cmt)).
```

The safety part of the `u_ensures` assertion yielding `CMT_L13` is covered by `CMT_S8`. The liveness part will be provided by the `u_transient` assertion in Lemma `CMT_L12`, which will be proved from the following lemma `CMT_L10` via an intermediate lemma `CMT_S10`. Lemma `CMT_L10` states: If `time` is not a common meeting time then `time` will eventually change. The proof is done by unfolding `u_transient`. Then two cases are considered: If `m` equals `(f m)` then `update_g` is responsible for progress, otherwise `update_f` is the progress witness.

```
Lemma CMT_S9 : (m:nat) (u_transient u_pr
  [s:u_st](not (common (s u_time))) /\ m=(s u_time)).
```

Using `u_transient_implication` and the fact that `cmt` is the smallest common meeting time we obtain another `u_transient` assertion:

```
Lemma CMT_S10 : (m:nat) (u_transient u_pr
  [s:u_st]((lt (s u_time) cmt) /\ m=(s u_time))).
```

The previous assertion implies the next, again by u_transient_implication and some case analysis on time.

```
Lemma CMT_L12 : (m:nat)(u_transient u_pr
   (u_and (u_and [s:u_st](le (s u_time) cmt) [s:u_st]m=(s u_time))
     (u_not
       (u_or
         (u_and [s:u_st](le (s u_time) cmt) [s:u_st](lt m (s u_time)))
         [s:u_st](s u_time)=cmt)))).
```

Now we can combine safety (CMT_S8) and liveness (CMT_L12) lemmas to obtain a u_ensures assertion which can be directly used to prove the following u_leads_to assertion by u_leads_to_basis (see the inductive definition of u_leads_to).

```
Lemma CMT_L13 : (m:nat)
  (u_leads_to u_pr
    (u_and [s:u_st](le (s u_time) cmt) [s:u_st]m=(variant s))
    (u_or
      (u_and [s:u_st](le (s u_time) cmt) [s:u_st](lt m (variant s)))
      [s:u_st](s u_time)=cmt)).
```

Now we can prove CMT_L14 directly using the reverse induction rule u_leads_to_rev_nat_induction. We instantiate maximum by cmt and variant by variant as defined above. The u_leads_to premise is given by CMT_L13. Informally we have the following assertion: If time is not beyond the smallest meeting time cmt then it will reach cmt sometime in the future.

```
Lemma CMT_L14 : (u_leads_to u_pr
  ([s:u_st](le (s u_time) cmt)) ([s:u_st](s u_time)=cmt)).
```

Of course, this left-hand side of u_leads_to covers also the initial condition of our original program. So we obtain by our final liveness result by weakening the left-hand side using u_leads_to_implication_l:

```
Theorem CMT_L15 : (u_leads_to u_pr
  ([s:u_st](s u_time)=O) ([s:u_st](s u_time)=cmt)).
```

Together with Theorems CMT_S5 and CMT_S6 the last theorem provides the liveness assertion needed for total correctness of our program.

Chapter 6

Conclusions and Outlook

For verification of functional programs logical type theories are certainly the most advanced and uniform formalisms. Concurrency and nondeterminism, however, are issues that are not directly supported by type theories, although many practical applications require a means to deal with them.

We have proposed a simple approach combining a logical type theory (including applications such as higher order functional programming and theorem proving) and a notion of concurrent program/system by means of an embedding. The embedding generalizes the UNITY idea to functional transition systems which are used here as a model of concurrent programs and systems. Functional transition systems encourage the description of programs/systems as interacting functional subsystems communication via a common state space. The UNITY programming language appears as a particular syntax for a subclass of functional transition systems. But the approach we propose is not restricted to UNITY: Many other formalisms for concurrent systems can be equipped with a semantics as functional transition systems. The embedding exploits the capabilities of the Calculus of Inductive Constructions in the context of concurrent programs and systems: We can not only profit from the expressive type system but we can also exploit strong normalization of the Calculus of Inductive Construction to guarantee termination of atomic statements/transitions.

The approach we propose also exploits the internal higher order logic of the Calculus of Inductive Constructions as a formal basis for a temporal logic in the style of (New) UNITY. This increases logical expressiveness and allows us to carry out specification and verification in a higher order framework which also supports general mathematical reasoning. Of course, the last point is of major importance as already demonstrated by our simple example which requires straightforward mathematical reasoning about natural numbers. In more realistic examples we could use sophisticated inductive reasoning about complex structures, since inductive types are a strong feature of CIC. Although one should expect that intentional intuitionistic logic may be too restrictive due to the lack of indirect proof schemes, it might be surprising that we were able to prove many proof rules without the use of classical axioms. On the other hand there are important rules like `u_unless_disjunction` and `u_unless_cancelation` which do not hold intuitionistically (without changing the basic definitions). However, it is interesting to observe that the only place where the classical axiom `classic` is used directly is the proof of `u_unless_disjunction`. However, the proofs of other rules like `u_unless_cancelation` and `u_leads_to_completion` make use of `classic` indirectly.

In the future it is planned to carry out larger cases studies, e.g. the verification of a distributed network algorithm. Guided by the new experiences the library should be enhanced and extended. An obvious necessity is the extension of the library to programs consisting of infinite sets of statements, which is excluded in the UNITY method. Also the issue of fairness deserves more attention, since the unconditional fairness inherited from UNITY is too weak for certain purposes. Stronger fairness assumptions could be integrated by modifying the definition of `u_ensures` (cf. [36]). Last but not least the issue of compositionality has to be addressed more seriously. For instance, this could be done following the ideas of [8] but also the integration of ideas from [34], [38] and [25] should be considered.

Bibliography

[1] F. Andersen, K. D. Petersen, and J. S. Pettersson. Program verification using HOL-UNITY. In J. J. Joyce and C. J. H. Segar, editors, *Higher Order Logic Theorem Proving and Its Applications. 6th International Workshop, HUG '93 Proceedings*. Springer-Verlag, 1994.

[2] H. P. Barendregt. Lambda-calculi with types. In S. Abramsky, D. M. Gabbay, and T.S.E. Maibaum, editors, *Background: Computational Structures*, volume 2 of *Handbook of Logic in Computer Science*. Oxford: Clarendon Press, 1992.

[3] N. Brown and D. Mery. A proof environment for concurrent programs. In J.C.P. Woodcock and P.G. Larsen, editors, *FME'93: Industrial-Strength Formal Methods, First International Symposium of Formal Methods Europe, Odense, Denmark, April 1993, Proceedings*, Lecture Notes in Computer Science 670, pages 196–215. Springer-Verlag, 1993.

[4] K. M. Chandy. Properties of concurrent programs. *Formal Aspects of Computing*, 6(6):607–619, 1994.

[5] K. M. Chandy and J. Misra. *Parallel Program Design. A Foundation.* Addison-Wesley, 1988.

[6] B. Chetali. Formal verification of concurrent programs using the larch prover. In U. H. Engberg, K. G. Larsen, and A. Skou, editors, *Proceedings of the Workshop on Tools and Algorithms for the Construction and Analysis of Systems*. BRICS Univ. Aarhus; Aarhus, Denmark, 1995.

[7] A. Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5(1), 1940.

[8] P. Collette and E. Knapp. A foundation for modular reasoning about safety and progress properties of state-based concurrent programs. *Theoretical Computer Science*, 183:253 – 279, 1997.

[9] T. Coquand. *Une theorie des constructions.* PhD thesis, Universite de Paris, 1985. These de troisieme cycle.

[10] T. Coquand. Metamathematical investigations of a calculus of constructions. In P. Odifreddi, editor, *Logic and Computer Science*, number 31 in APIC Studies in Data Processing. Academic Press Limited, London, 1990.

[11] T. Coquand and G. Huet. Constructions: A higher order proof system for mechanizing mathematics. In *Proc. EUROCAL'85*, Lecture Notes in Computer Science 203. Springer-Verlag, 1985.

[12] T. Coquand and G. Huet. The calculus of constructions. *Information and Computation*, 76(2/3):95–120, 1988.

[13] William M. Farmer, Joshua D. Guttman, and F. Javier Thayer. IMPS: An interactive mathematical proof system. *Journal of Automated Reasoning*, 11(2):213–248, October 1993.

[14] R.W. Floyd. Assigning meanings to programs. In *Proceedings of Symposia in Applied Mathematics XIX*, pages 19–32. American Mathematical Society, 1967.

[15] D. M. Goldschlag. Mechanically verifying concurrent programs with the Boyer-Moore prover. *IEEE Transactions on Software Engineering*, 16(9), 1990.

[16] M. J. C. Gordon and T. F. Melham. *HOL: A Proof Generating System for Higher-Order Logic*. Cambridge University Press, 1992.

[17] B. Heyd and P. Cregut. A modular coding of UNITY in Coq. In J. von Wright, J. Grundy, and J. Harrison, editors, *Theorem Proving in Higher Order Logics. 9th International Conference, TPHOLs'96.* Springer-Verlag, 1996.

[18] C.A.R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–583, 1969.

[19] G. Huet, C. Paulin-Mohring, et al. The Coq Proof Assistent Reference Manual, Version 6.2, Coq Project. Technical report, INRIA, 1998. http://pauillac.inria.fr/coq.

[20] R. M. Keller. Formal verification of parallel programs. *Communications of the ACM*, 19(7):371–384, July 1976.

[21] E. Kindler. Invariants, composition, and substitution. *Acta Informatica*, 32(4):299–312, 1995.

[22] E. Knapp. Derivation of concurrent programs. *Sci. Comput. Programming*, 19:1–23, 1992.

[23] L. Lamport. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering*, SE-3(2):126–143, 1977.

[24] L. Lamport. Win and sin: predicate transformers for concurrency. *ACM Transcations on Programming Languages and Systems*, 12:396–428, 1990.

[25] L. Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872 – 923, May 1994.

[26] Leslie Lamport and Lawrence C. Paulson. Schould your specification language be typed? Technical Report 147, DEC, May 1997.

[27] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems.* Springer-Verlag, 1992.

[28] J. Misra. New UNITY. Manuscript available via `http://www.cs.utexas.edu/users/psp/newunity.html`, 1994.

[29] S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. In Deepak Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, Lecture Notes in Artificial Intelligence 607, pages 748–752, Saratoga, NY, June 1992. Springer-Verlag.

[30] J. Pachl. A simple proof of a completeness result for leads-to in the UNITY logic. *Information Processing Letters*, 41, 1992.

[31] J. Pachl. A simple proof of a completeness result for leads-to in the UNITY logic (corrigendum). *Information Processing Letters*, 44, 1992.

[32] C. Paulin-Mohring. Inductive Definitions in the system Coq – Rules and Properties. In M. Bezem and J . F. Groote, editors, *Typed Lambda Calculi and Applications, International Conference on Typed Lambda Calculi and Applications, TLCA 93*, Lecture Notes in Computer Science 664. Springer Varlag, 1993.

[33] L. C. Paulson. Isabelle: the next 700 theorem provers. In P. Odifreddi, editor, *Logic and Computer Science*, number 31 in APIC Studies in Data Processing. Academic Press Limited, London, 1990.

[34] A. Pnueli. In transition from global to modular temporal reasoning about programs. In K. R. Apt, editor, *Logics and Models of Concurrent Systems*, volume 13 of *Series F: Computer and System Science*, pages 123–144. Springer-Verlag, 1985.

[35] I. S. W. B. Prasetya. Error in the UNITY substitution rule for subscripted operators. *Formal Aspects of Computing*, 6:466–470, 1994.

[36] J. R. Rao. *Extensions of the UNITY methodology: compositionality, fairness and probability in parallelism.* Lecture Notes in Computer Science 908. Springer-Verlag, 1995.

[37] B. A. Sanders. Eliminating the substutution axiom from UNITY logic. *Formal Aspects of Computing*, 3:189–205, 1991.

[38] N. Shankar. A lazy approach to compositional verification. Technical Report SRI-CSL-93-08, SRI International, December 1993.

[39] M.G. Staskauskas. Formal derivation of concurrent programs: an example from industry. *IEEE Trans. on Software Eng.*, 19:503–528, 1993.

[40] S. Thompson. *Type Theory and Functional Programming.* International Computer Science Series. Addison-Wesley Publishing Group, 1991.