A Modern Isabelle-based Prover for VDM's Logic of Partial Functions

George Karabotsos

A Thesis

in

The Department

of

Computer Science

Presented in Partial Fulfillment of the Requirements
for the Degree of Master of Computer Science at
Concordia University
Montreal, Quebec, Canada

August 2005

Your file Votre référence
ISBN: 0-494-10289-6
Our file Notre référence
ISBN: 0-494-10289-6

ABSTRACT

A Modern Isabelle-based Prover for VDM's Logic of Partial Functions

George Karabotsos

Specification and verification tools, often, employ theorem proving technology. This technology encompasses an underlying logic encoded in a theorem prover. VDM-LPF, first developed by Sten Agerholm and Jacob Frost in 1996, encodes VDM's Logic of Partial Functions (LPF) by extending the generic theorem prover Isabelle. Because of Isabelle's powerful syntax facilities, terms in this logic are written using a subset of the VDM-SL specification language.

In this thesis we provide the reader with a detailed description of the steps we undertake to revive the VDM-LPF logic by upgrading it to the latest Isabelle version, namely Isabelle2005, and by using its latest Intelligible Semi-Automated Reasoning (Isar) theory syntax. We first provide a description of the Isabelle theorem prover, its generic design, and the unique facilities it provides for extending itself and developing new object-logics. We then explain in detail the original VDM-LPF system as developed for the Isabelle94-8. We describe all the necessary modifications that were needed to perform the upgrade. Finally, a number of proofs are performed, using the upgraded VDM-LPF system, that form a case-study.

Acknowledgments

I want to extend my gratitude to all people who helped me with my research and the writing of this thesis. First and foremost I would like to thank my supervisor Prof. Patrice Chalin for introducing me in this very interesting research area. I am deeply indebted to him for his help, stimulating suggestions, and encouragement throughout the research and writing of this thesis.

I thank all the members of our Dependable Software Research Group (DSRG), for their help, support and valuable hints. I would like to especially thank Daniel Sinnig for reviewing my thesis and for his very constructive remarks.

I would like to thank the members of the Isabelle community for always being there to answer my many Isabelle questions, during my long and fairly treacherous trip of learning the Isabelle theorem prover. I would like to especially thank Dr. Laurence Paulson for not only providing me with such an excellent tool, but also for always helping in answering these questions.

Last but not least, I would like to thank my family and Panagiota for their continuing support and love during this past year and throughout my life.

Table of Contents

List of Figures

1 Introduction

Program verification is the process of ensuring that a software program performs as stated by its specifications. To perform program verification in a rigorous manner the specifications must be written in a specification language that is supported by an appropriate logical framework. Specifications written using such a specification language allow for the automation of program verification through the use of theorem-proving technology. That is, they allow the generation of proof obligations based on these specifications and then using a theorem prover, equipped with the correct logic, to validate these obligations. In this sense, program verification proves that the software system performs as expected.

VDM-LPF is the result of the experiments Sten Agerholm and Jacob Frost performed in creating a program verification system for VDM. VDM-LPF encodes VDM's Logic of Partial Functions (LPF) using the Isabelle generic theorem proving environment. This encoding is comprised of syntax, a proof system, and a set of proof tactics. The syntax is a subset of VDM's specification language, which is known as VDM-SL. The proof system is defined axiomatically. It includes theories for propositional and predicate logic as well as theories for advanced topics, such as sequences, maps, and types. Proof tactics are employed in the process of deriving new theorems. Isabelle provides with a comprehensive set of basic tactics. Moreover, Agerholm and Frost developed a set of automatic proof tactics, specifically for VDM-LPF. These tactics facilitate the proving process by making proofs compact.

The motivation behind our involvement with VDM-LPF is its underlying logic. The logical foundation of VDM-LPF is the Logic of Partial Functions. The need to better understand and experiment with such a logic arises from the belief that a non-classical logic, such a LPF, is more appropriate for reasoning about software systems. Such a belief alone is not enough. We are required to understand and experiment with a non-classical logic, such as LPF. This understanding is a essential for two reasons: (1) to visualize how much more complex such a logic is and (2) to understand why most software reasoning systems utilize a classical two-value logic. The difference between a two-value and a three value logic is the manner they handle partial functions.

1.1 Contribution

The main contribution of this thesis is a realization of support for VDM's Logic of Partial Functions (LPF) in one of the most popular theorem provers, Isabelle (2005 edition). Rather than starting from scratch we chose to work from Isabelle/VDM-LPF, a realization of LPF written for

Isabelle about a decade ago. This thesis explains the step-by-step approach that we followed in order to revive Isabelle/VDM-LPF. This revival was accomplished by rewriting theory files to make use of the *Intelligible Semi-Automated Reasoning* (Isar) theory syntax. All aspects of the theories have changed in various degrees thus making our upgrading task a challenge. In particular, we successfully imported and were able to reuse the built-in proof search tactics specifically developed for VDM-LPF. Finally, as a form of validation, we present a case study in the use of our VDM-LPF to conduct proofs.

1.2 Thesis Organization

This thesis is organized into seven chapters and four appendices. Chapter 1 introduces the reader to our work by providing a brief background, our motivation to pursue this topic and the main contribution of our work.

Chapter 2 reviews the background of our work and a presents a small sample of tools that make use of theorem proving technology that is related to our work. Additionally, both classical and non-classical logics are examined, concentrating on the manner they handle partial functions. Two specification languages, namely VDM and JML, are briefly discussed. Next we have a brief look at computer-aided theorem proving, where we mainly concentrate on two popular theorem provers, namely PVS and Isabelle – in particular, we examine their support for Higher-Order Logic (HOL). Finally, two tools that make use of theorem proving technology are examined.

Chapter 3 presents a detailed view on the Isabelle theorem prover. We present Isabelle's evolution, architecture, and generic design. We further discuss Isabelle's proof system and the two types of theory development syntax. Finally, we examine the process for creating a new object logic.

Chapter 4, presents the VDM-LPF logic as developed by Sten Agerholm and Jacob Frost. We present the three different components comprising VDM-LPF – syntax, proof system, and the proof search tactics are presented.

Chapter 5 presents the main contribution of this thesis. We describe the four step process of: (1) obtaining the original VDM-LPF and initial obstacles that had to be resolved, (2) the successful interpretation of the VDM-LPF theories on an older version of Isabelle, (3) the first migration to the latest Isabelle version, (4) the final upgrade that makes use of the Isar theory and proof language.

Chapter 6 portrays a set of proofs we have performed using the upgraded VDM-LPF object logic. These proofs include propositional and predicate logic, as well as a proof involving VDM's specification language (VDM-SL) conditional statements.

Chapter 7 contains our conclusions and discusses the lessons learned from this exercise. Additionally, ideas about future VDM-LPF enhancements are presented.

Appendix A lists the contents of the original `Prop.thy` theory file as developed by Agerholm and Frost. Appendix B, contain the `LPF_Prover.ML` file, which holds the implementation of the built in proof search tactics. Appendix C contains the syntax of the meta-logic as displayed by the `print_syntax` Isabelle command. Finally, Appendix D holds all the upgraded theory files of VDM-LPF.

2 Background and Related Work

The purpose of this chapter is twofold. First, we provide the reader with the background material necessary to understand later chapters. Second, we present a summary of related work ; this will better help define the overall context of our work. The material covered includes specification languages, logics, theorem provers, and tools that make use of theorem proving technology.

These topics are necessary because they are related in the specification and verification field of study. Specification and verification tools are developed because of the need to build quality software products. These tools use theorem proving technology to check specifications and their associated code. These specifications are written following the language and semantics of a specification language which in turn is based on a underlying logic.

2.1 Specification Languages

The research reported in this thesis is being done in the context of a larger research project on dependable software engineering which seeks to advance practical, state-of-the-art rigorous development techniques and tools. At the heart of these techniques is the notion of writing contracts for software modules so that the software will be:

- better documented, and hence more easily understood and maintained;
- of higher quality since automated tools can be used to, e.g., automatically generate test cases from specifications, or better still, statically verify that modules accurately implement their specifications.

Hence, although this thesis is actually about a theorem proving support for a logic, the original motivation for this logic comes from its presence as the formal foundation of a specification language.

A specification language is a formal language used to model and describe the behavior of software systems. Two important classes of specification languages (which are *not* mutually exclusive) are model-based specification languages (SLs) and Behavioral Interface Specification Languages (BISL). VDM, B [Abrial96] and Z [Spivey92] are examples of the former while the Larch family of languages and the more recent Java Modeling Language (JML) [LBR05] and Spec# [Barnett+04] are examples of the latter. Model based specification languages allow developers to accurately document a model of the software system. A BISL describes the software system by describing the interfaces and behavior of specific modules of the system. As such BISL are tailored to specific programming languages (e.g. JML is tailored for Java) and usually a BISL will not be appropriate for a different language.

Next we provide a slightly more detailed overview of VDM and JML. We do so for VDM because it is founded on LPF—the main topic of this thesis—and for JML because it is one of the specification languages being actively worked on by the members of our research group. Furthermore, it is becoming apparent that an LPF-like logical foundation will be most appropriate for JML as well [Chalin05, Chalin05-1].

2.1.1 Vienna Development Method (VDM)

The Vienna Development Method (VDM) is more than just a specification language [Jones90]. VDM is a formal development method mainly comprised of a specification language (VDM-SL), data reification techniques, and operation decomposition techniques. A number of VDM tools are available to developers with the most notable ones being IFAD's VDMTools; although somewhat dated, a research effort know as Overture is seeking to create a public domain (Eclipse-based) version of the VDM (actually VDM++) support tools.

VDM-SL is the specification language underlying VDM. A typical VDM specifications consists of a data model, a set of operations on the data model, and proof obligations. The data model abstracts the main data types of the software system and can also be used to represent its state. The operations are defined to describe the behavior of the system and may also change the state as a side-effect. These operations are described in terms of inputs and outputs of the data model [Jones90, Bicarregui+94]. The proof obligations are logical formulas capturing basic "well-formedness" properties of the data model and their related operations—e.g., that an operation's post-condition be well-defined in all situations that satisfy its precondition. The Logic of Partial functions (LPF) [BCJ84, Cheng86, CJ91] is the logic that underlies VDM-SL and, in particular, in which the proof obligations are written. LPF will be described in more detail starting in Section 2.2.2.

Data reification is a process that allows a data model to be formally related to a more "refined" data model. This process can be repeated. The goal of this process is to successively refine an abstract data model until it reaches a level of detail that can be mapped directly into an implementation language [Jones90, Bicarregui+94].

Operation decomposition is to operations as data reification is to the data model. It is, again, an iterative process whereby operations, initially depicted using mathematical concepts, are rewritten (decomposed) so that they increasingly incorporate programming language constructs, such as loops, conditional statements, etc. [Jones90, Bicarregui+94] In other words the operations are moving from an abstract representation into one that more closely resembles an implementation.

2.1.2 Java Modeling Language (JML)

JML is a specification language that falls under the class of Behavioral Interface Specification Languages. As its name implies, it is a specification language tailored for use with the Java™ programming language. Its main purpose is to allow the interfaces of Java modules to be accurately specified and in a way that captures both module signatures as well as component constraints and behaviors. JML specifications are most often embedded directly with Java code. Like VDM, there are a number of tools available for processing and checking JML, most notably ESC/JAVA2 and LOOP which allow partial and, respectively, total verification of JML annotated Java modules; we will have a closer look on these two tools later in this chapter.

A very simple Java class with a JML method specification is given in Figure 1. The first two Java statements are used to declare a class named `Tester` and a method named `addOne`. The same statements are used in JML to declare the interfaces for the fore-mentioned class and method.

Module behavior is also specified using specially embedded comments in the Java code. The comments are annotated with an `@` character. The keywords `requires` and `ensures` head assertions that are used to define a method's pre- and post-conditions. These conditions are specified using augmented Java expressions with predicate logic quantifiers, such as `forall` and `exists`. These expressions are logical statements that describe the behavior of the module. JML's underlying logic is (currently) a classical two-valued logic where partial functions are modeled as underspecified total functions. The differences between JML's and VDM's logical foundations are explained next.

```
public class Tester {
  public static int addOne(int i)
{
    //@ behavior {
    //@     requires i > 0;
    //@     ensures \result == i +
1;
    //@ }
    return(i+1)
  }
}
```

Figure 1: JML Sample Specification

2.2 Logics

Logic comes from the ancient Greek work λόγος (logos), which came to mean reason [WP]. Two main branches can be identified: philosophical and mathematical logic. Of particular interest to us are two kinds of mathematical logic: classical two-valued and non-classical three-valued. The main difference between the two is the manner they handle undefined terms that result from the application of partial functions. A partial function is a function that it is not defined for all

possible values in its domain; when applied to arguments outside its domain, it does not yield a defined value.

In most formulations of classical logic, all functions are total functions – total functions are functions that are defined for all possible values in its domain. However, since partial functions arise naturally in mathematics and computer science (e.g. division) techniques have been devised so that they can "deal with" partial functions as well. Unfortunately, the manner in which this is done is, more often than not, inappropriate for reasoning about programs because of its complexity in dealing with issues such as error conditions, non-termination, undefinedness, etc. [Hahnle05]. But why is classical two-valued logic almost always used for program reasoning? The answer lies in the desire to preserve well known properties of classical logic that do not hold in a non-classical logic.

$$\overline{p \vee \sim p}$$

Figure 2: Law of Excluded Middle

Non-classical three-valued logics extend (or "lift") all value sets with an extra value used to represent undefinedness. With such an approach, partial functions can be properly represented. It would seem that using a non-classical logic in program reasoning is the right approach. Unfortunately this approach also has its share of problems. The problems arise from the complexity in the semantics of the different operators, relational and logical alike, because of the introduction of this special "undefined" value. Moreover, fundamental properties such as the law of the excluded middle (given in Figure 2) no longer hold.

2.2.1 Classical Two-Valued Logic

A number of methods have been devised so that a classical two-value logic can deal with partial functions. Restricting the domain set of a function, viewing functions as relations, and underspecification are just a representative sample of these methods.[CJ91]

When dealing with partial functions in a classical logic, one can attempt to find a more restrictive domain of that function while at the same time leaving the function unchanged. By finding such a domain we are transforming this function from partial to total. Consequently, there is no need to worry about undefined values. To illustrate consider the partial function [CJ91]:

```
zero: Z => Z
zero(i) ≙ if i = 0 then i else zero(i-1)
```

This function accepts an integer and it subtracts one from it until it is reduced to zero. However, this function is undefined for all the negative values because it will never terminate. Changing the function so that its domain and range is the set of natural numbers (`zero: N => N`) will result in no possible application of this function resulting in undefined values. Not all partial functions

are this simple. For most, finding such a restrictive set can be difficult. Even when such a set is found, validating properties involving such a function maybe a very challenging task [CJ91].

Another solution is to avoid function application altogether. This can be achieved by modeling a function as a relation (or graph). When a function is to be applied to a given argument, a test is performed to determine whether the (domain, range) pair is in the graph. This means that reasoning about a function expression such as `f(x) = y` takes the `(x,y) ε f` form. Thus classical two valued logic is sufficient to represent such functions. [CJ91] Again there are issues with this approach – the natural universally understood function application is been replaced by an alternate notation. Such a solution has been implemented in the Z specification language and its derivatives, such as B [Hahnle05].

Finally, underspecification is another method used to model partial functions in a two-valued logic. Underspecification accepts the fact that function application may cause an undefined result, however, instead of using a third undefined value an unknown or unspecified value from the range of the function is returned [Chalin05]. Lets consider the quintessential example, integer division by 0. If such an application occurs the division will result in an unspecified integer value. Practically speaking that unspecified value is always the same – for example in Isabelle/HOL division by zero always gives zero. Furthermore, Isabelle provides a polymorphic constant, `arbitrary::'a`, that can be used with partial functions [NPW98]. JML makes use of underspecification in order to work around partial functions.

2.2.2 Non-Classical Three-Valued Logic

Three valued logics have been introduced for various reasons, but most importantly to enable a more natural (or direct) form of reasoning about partial functions. However, due to their nature such logics have some serious drawbacks because of the complexity in the semantics and because classical logic fundamental properties no longer hold.

The complexities in the semantics are introduced because of the third truth value. Relational and logical operators are no longer treated in the classical manner. As a first example of this we note that three-valued logics come equipped with two kinds of equality: strong and weak equality – see Figure 3. In this table v1 and v2 are the values we test for equality while U denotes the undefined value. The meaning of logical operators is even more complex. A choice of strong Kleene (known also as monotonic and non-strict), weak

Strong	v1	v2	U
v1	T	F	F
v2	F	T	F
U	F	F	T

Weak	v1	v2	U
v1	T	F	U
v2	F	T	U
U	U	U	U

Figure 3: Strong and Weak Equality Truth Tables

Kleene (known as Bochvar and strict), and conditional (known also as McCarthy) logical operators are available [Hahnle05, Cheng86]. Figure 4 shows the truth tables for disjunction.

Strong Kleene			
OR	T	F	U
T	T	T	T
F	T	F	U
U	T	U	U

Weak Kleene			
OR	T	F	U
T	T	T	U
F	T	F	U
U	U	U	U

Conditional			
OR	T	F	U
T	T	T	T
F	T	F	U
U	U	U	U

Figure 4: Truth Tables for Strong and Weak Kleane, and Conditional Disjunction

Fundamental properties from classical logic no longer hold. The law of the excluded middle is one such property. If conditional logical operators are used the commutative property no longer holds either.

Ideally in computing we would want a logic that is three-valued while maintaining all properties of classical logic. However, this cannot be achieved. Consequently the next best thing will be a three-valued logic that preserves as many as possible of the properties found in classical logic.

LPF is a non-classical three-valued first order predicate logic designed for reasoning about languages with partial functions. This logic is used in VDM. Propositional logic uses the strong Kleene connectives which preserve the commutative and distributive properties while predicates range over defined values only. This non-classical logic inherits all properties of classical two-valued logic, except for the law of the excluded middle. This is because LPF also deals with undefined values. To alleviate this, a new operator is defined in LPF, namely δ, to test for definedness. The definition is as follows:

$$\delta e = e \vee \sim e$$

Having this operator defined, the law of the excluded middle will hold in LPF if it is modified in the following manner:

$$\frac{\delta e}{e \vee \sim e}$$

In a similar way, any classical logic tautology that no longer holds in LPF can be made valid by use of this operator.

2.3 Theorem Provers

Like any formal language, a logic is most useful to a software engineer or a mathematician if supporting tools are available that allow formal proofs to be carried out. The intent of this section is to provide the reader with a brief introduction to theorem provers. As such our goal is to give the reader a basic idea of what a theorem prover is and what it feels like to write theories and perform computer aided proofs. This background is needed since the main contribution of this thesis involves such a theorem prover.

A theorem prover is a software system that is able to reason about theorems. These theorem provers come equipped with a language where new definitions can be expressed and a logic where properties of these new definitions can be established or refuted. This language more often than not, resembles a functional programming language. However, the aim is not to write programs but to design abstract models of systems and validate them [NPW98]. Consequently one can view a theorem prover as a specification and verification tool.

There are two main categories of theorem provers, those that are fully-automated and those that can interactively validate theorems. Instances of the former are Simplify [STP], Harvey [Ranise+03] and Twelf [Appel2] while instances of the latter include HOL [GM93], Isabelle [Paulson, NPW98, Wenzel] and PVS [OSRS1, OSRS2, OSRS3]. It should be noted that proving via interactive theorem provers, as the ones mentioned above, can be automated to a certain extent via, e.g., tactics and decision procedures.

We will concentrate on just two of these theorem provers, namely Isabelle and PVS, for the following reasons. First, PVS and Isabelle are two of the most widely used theorem provers. Second, interactive provers are more appropriate for reasoning about software systems. This is because the structure of such proofs is known in advance and they can be tedious and very large [PST2]. Finally, VDM-LPF, the object logic this thesis is concerned with, was developed using Isabelle.

2.3.1 PVS

PVS is a specification and verification system. It is actively developed by SRI International which is an independent nonprofit research institute. PVS has an active user community including prominent organizations such as NASA. PVS mainly offers a specification language based on a higher-order logic, and a prover supporting this language. It also offers a user interface in the form of an emacs-customizable mode.

The PVS specification language is particularly rich. It allows users to define types, functions, and theorems. These definitions are organized into theory files. Functions can be recursive and pattern matching on arguments can be achieved via the `CASES ENDCASES` block statement. Base types for Boolean, the natural, integer, and real numbers exist. Tuples and records can also be defined. New types can be defined via the `DATATYPE` statement. Moreover, PVS allows for definition of subtypes. Subtypes can be defined using logical expressions. For example, the following:

```
NonZeroNats : TYPE = {n:nat | n /= 0}
```

defines a subtype (subset) of the natural numbers that excludes zero. Such a subtype is desirable in expressions that may involve division by zero, where the result is undefined.

The theorem prover is based on a classical typed higher-order logic (HOL). Theorems can be specified via the `THEOREM` or (its synonym) `LEMMA` statements followed by a formula. Formulas are expressions of the Boolean type. A large number of methods are available that can be applied in order to resolve the proof, methods such as `induct` and `grind`. The `induct` method is used to perform proof by induction while `grind` is a general purpose method that attempts to resolve the proof automatically.

Figure 5 portrays a simple theory named `sum` which defines a recursive function of the same name and a theorem to be proved. The function recursively implements the summation of a natural number. What the theorem says is that the recursive definition of the summation is equivalent to its closed form. The first thing to do is to *parse* and *typecheck* the theory.

```
sum: THEORY
  BEGIN

    n: VAR nat

    sum(n): RECURSIVE nat =
      (IF n = 0 THEN 0 ELSE n + sum(n-1) ENDIF)
    MEASURE (LAMBDA n: n)

    closed_form: THEOREM sum(n) = n * (n + 1)/2

  END sum
```

Figure 5: PVS sum Theory

The next step will be to attempt to prove the theorem. This is achieved by placing the cursor anywhere in the line containing the `closed_form` theorem and using the PVS `prove` command. The emacs screen splits and we enter the PVS's proof mode. In this mode we can apply different PVS methods in order to advance and hopefully resolve our theorem. Proofs are presented in a form of sequent calculus [OSRS3]. Figure 6 shows the PVS's proof mode and the successful resolution of the `closed_form` theorem. In gray you can see the user's input. We use induction and then we apply the brute force grind method twice to resolve the base and the induction step cases. PVS clearly displays the successful resolution of this theorem by the `Q.E.D.` line.

```
closed_form :

  |-------
{1}   FORALL (n: nat): sum(n) = n * (n + 1) / 2

Rule? (induct n)
Inducting on n on formula 1,
this yields  2 subgoals:
closed_form.1 :

  |-------
{1}   sum(0) = 0 * (0 + 1) / 2

Rule? (grind)
sum rewrites sum(0)
  to 0
Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of closed_form.1.

closed_form.2 :

  |-------
{1}   FORALL j:
        sum(j) = j * (j + 1) / 2 IMPLIES
         sum(j + 1) = (j + 1) * (j + 1 + 1) / 2

Rule? (grind)
sum rewrites sum(1 + j)
  to 1 + j + sum(j)
Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of closed_form.2.

Q.E.D.
```

Figure 6: PVS close_form Proof

2.3.2 Isabelle/HOL

In this section we will provide the reader with a brief introduction to Isabelle. We will concentrate on an instance of Isabelle called Isabelle/HOL which is similar, from a user's point of view, to PVS. In Chapter 3 we give a detailed account of Isabelle, its evolution and its generic design, which is required material for the understanding of the main contribution of this thesis.

Although Isabelle, like PVS, can be used as a specification and verification system it is more often described as a generic theorem proving environment. That is, it allows users to create and experiment with the design of new specification languages and the underlying logics. This last point is what sets Isabelle apart from other theorem provers, such as PVS. While PVS is equipped to reason within the "limits" of a classical typed HOL, Isabelle in addition to HOL provides instances for first-order logic (FOL), Logic of Computable Functions (LCF) and Zermelo-Fraenkel set theory (ZF). The last two logics are implemented as extensions to HOL and FOL respectively.

Isabelle/HOL is the instance of Isabelle for HOL. It is very similar to what PVS offers. It offers a specification language where specifications can be expressed and a logic where these specifications can be reasoned about. Isabelle comes with a user interface in the form of a third-party project, namely ProofGeneral [Aspinal, Aspinal2], which is also an emacs based interface (an Eclipse based interface is under development).

Isabelle/HOL's specification language is comprised of types, functions, and theorems. It follows the functional programming paradigm and it is very similar to ML. ML is Isabelle's implementation language and they share many similarities. Functions can be recursive and make full use of pattern matching. Isabelle/HOL offers all the usual base types (bool, nat, int, real) and a polymorphic typing system where tuples, records, type synonyms, and new types can be defined. Theorems make use of logical and relational operators to formulate Boolean terms. λ-expressions are also available in the language.

The underlying logic is a classical two-valued typed high order logic. Partial functions are implemented as total underspecified functions, by making use of the polymorphic `arbitrary` constant. Theorems are written using the `theorem` and `lemma` statements. Similar to PVS methods for induction (`induct`) and brute force (`auto`) resolution are supplied.

For an example of an Isabelle/HOL theory consider the sum theory on Figure 7. This theory declares the exact same function and `closed_form` theorem as the PVS theory displayed earlier. The definitions are enclosed in `theory end` statement blocks. We also see that the `Main` theory is included in the current context. The `Main` theory is the top level theory of the HOL logic and needs to be included if the HOL theorems and definitions are to be used. Next the recursive function `sum` is defined. The definition clearly shows how one can use pattern matching in Isabelle. Finally, the `closed_form` theorem along with its proof can be seen in the `lemma done` block. As before, induction followed by the brute force method `auto` is applied. Unlike the PVS however, we need to apply an additional method, namely `arith`. This is because the `auto` method cannot deal with complicated arithmetic expressions.

```
theory sum = Main:

 consts
   sum :: "nat => nat"
 primrec
   "sum 0       = 0"
   "sum (Suc n) = (Suc n) + (sum n)"

 lemma closed_form: "sum(n::nat) = (n * (n + 1)) div 2"
  apply(induct n)
  apply(auto)
  apply(arith)
 done

end
```

Figure 7: Isabelle/HOL sum Theory

2.4 Tools use Theorem Provers

In this section we briefly describe two tools that make use of theorem proving technology. The first one is ESC/Java2 which performs extended static checking of JML annotated Java programs. The second one is the LOOP tool that performs program verification, also, of JML annotated Java programs.

2.4.1 ESC/Java2

ESC/Java2 is been developed by David Coq and Joe Kiniry [ESCJ2]. It uses a subset of the JML specification language to perform extended static checking of the Java code. This checking is performed at compile time. ESC/Java2 is intentionally designed not to find nor report all possible errors in order to make it easier to use. In addition to static type-checking it reports of the potential presence of runtime exceptions. An interesting aspect of ESC/Java2 is the use of a theorem prover to prove that JML specifications are correct [Burdy+05].

2.4.2 LOOP

The LOOP tool is a program verification tool, developed at the University of Nijmegen [JP03]. It can be used to perform full program verification of JML annotated Javacard applications. It is a compiler that takes a Java program annotated with JML specifications and generates PVS theories that expression verification conditions are generated from a specialized Hoare logic. These theories are then fed into PVS for semi-automated proof under the guidance of developers.

3 Isabelle

Isabelle is essentially a generic theorem proving framework that ships with predefined logics built upon this framework, such as High Order Logic (HOL) and First Order Logic (FOL) among others. In fact, what differentiates Isabelle from other theorem provers is this ability to support different calculi [Paulson]. Isabelle is currently a joint project between Lawrence C. Paulson (University of Cambridge, UK) and Tobias Nipkow (Technical University of Munich, Germany).

In this chapter we provide a more in-depth look of the Isabelle theorem prover. Our aim is to provide the necessary background information necessary for understanding the main contribution of this thesis. We will look at Isabelle's history and evolution, its flexible architecture, we will see the two different flavors of Isabelle theories, and finally the facilities it provides to define new object logics.

3.1 Historical Background

In 1972, while at Stanford University, Robin Milner created LCF, a proof checker for Dana Scott's Logic for Computable Functions. This version came to be known as the Stanford LCF. By 1977 Robin Milner had moved to the University of Edinburgh, where he remained until 1995. There he created a new version of LCF, namely Edinburgh LCF. This was the first version of LCF where the proof commands could be programmed and extended. This was done by using the ML programming language which was also designed and developed by Milner and his team for this purpose. By 1981, further development of LCF was split between the University of Edinburgh and the University of Cambridge. Larry Paulson joined Cambridge university in that same year and helped in the development effort of LCF [Gordon97, Paulson90].

By 1985 Paulson started working on Isabelle. His goals were to build a generic theorem prover implemented in its entirety in ML, making use of "Sokolowski's (1987) technique of solving unknowns in goals by unification" [Paulson90], and to experiment with "de Bruijn's (1972) treatment of bound variables" [Paulson90]. By 1986 the first version of Isabelle was released. Isabelle86 was available with calculi for Constructive Type Theory (CTT), first-order logic (FOL), and Zermelo-Fraenkel set theory (ZF).

A problem with the 1986 version was that it did not support natural deduction style proofs. By 1989 a new meta-logic, namely Isabelle/Pure, was developed. This meta-logic incorporated a simply typed λ-calculus as well as a subset of high order logic with implication, equality, and the universal quantifier with their usual meaning. Through this meta-logic new calculi (also known as object logics) were created [Paulson90].

By 1993 Tobias Nipkow of the Technical University of Munich was part of the Isabelle development team. Isabelle93 featured with the addition of a number of new logics, with the most notable being a high-order logic (HOL). It should be noted that Nipkow may have been working on Isabelle before 1993, however, the fore-mentioned date is the first reference we have found of his involvement.

In 1999 the last major change was introduced to Isabelle. Marcus Wenzel, created a new language for expressing theories and proofs, namely Isar - *Intelligible semi-automated reasoning.* This change eliminated the need for ML styled proofs and merged the proof and theory files, collectively known as old-style theories, in what we call today a new-style or Isar theory. Currently Isabelle/Isar is made available via the Emacs editor. Efforts are underway to provide an Eclipse-based interface as well (for those who do not find Emacs convivial enough).

3.2 Architecture

Isabelle is implemented in Standard ML and it provides a number of formal calculi to users. It provides them with a (meta-)logical base, namely Isabelle/Pure, that is expressive enough and contains enough deductive machinery to allow users to extend it and create new calculi. Some of these tools include Isar [Wenzel], the Simplifier [Paulson], and the Classical Reasoner [Paulson]. Calculi such as First Order Logic (Isabelle/FOL) and High Order Logic (Isabelle/HOL) have been created by extending Isabelle/Pure. Further calculi can be created by extending FOL or HOL. The Logic of Computable Functions (Isabelle/HOLCF) and Zermelo-Fraenkel set theory (Isabelle/ZF) are examples of such logics, extending HOL and FOL respectively. Figure 8 shows the overall organization of the main Isabelle components – the logics displayed here (second and third row) are just a representative sample, albeit the most complete of the existing ones. Proof General is an (X)Emacs front-end for a number of theorem provers, including Isabelle. It provides for interactive development of theorem files and it is superior to the only other alternative available, a shell interface.

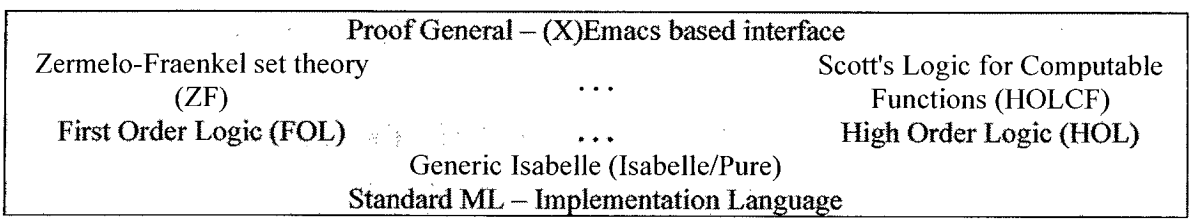

Figure 8: Overall organization of main Isabelle/Isar components

3.3 Proofs 101

Isabelle's proof system is based on the style known as natural deduction [NPW98]. Natural deduction is the reasoning method that resembles human reasoning patterns [NPW98]. That is, proofs are developed incrementally, and eventually satisfied, by using inference rules. The general format of an inference rule is:

$$\frac{A1 \quad \ldots \quad An}{C}$$

Where `A1` and `An` are predicates (called assumptions or hypotheses) that are necessary to infer the conclusion `C` (according to the given inference rule).

In general, there are two main types of inference rules: introduction and elimination. The former introduces a logical symbol while the later eliminates (an instance of) it from their conclusions. To illustrate, consider the conjunction inference rules from Figure 9. The first one says that from `P` and `Q` we can infer `P /\ Q` – i.e. it introduces conjunction. The remaining two are elimination rules and they say that from `P /\ Q` we can infer either one of `P` and `Q`. In other words, the introduction rule shows how we can prove a `P /\ Q` conjunction while the elimination rules show what can we prove from a `P /\ Q` conjunction.

$$\frac{P \quad Q}{P \wedge Q} \qquad \frac{P \wedge Q}{P} \qquad \frac{P \wedge Q}{Q}$$

Figure 9: Conjunction Inference Rules

In general these inference rules are or can be made available in Isabelle. Isabelle provides facilities where such rules can be expressed. Figure 10 shows a possible Isabelle representation of the conjunction inference rules seen above. The `==>` symbol is Isabelle's meta-implication symbol that is also used to separate assumptions from the conclusion of a rule. The `[|` and `|]` symbols are used to enclose multiple assumptions while a semicolon, `;`, character separates them. The propositions `P` and `Q` are prefixed by a `?` character. This is because these propositions may be instantiated by other propositions during the proof process. For example, consider the `A /\ B` proposition. If such a proposition is one of the goals (*subgoals* is the Isabelle terminology for such goals) to prove and we apply the conjunction introduction rule, the `?P` and `?Q` will be instantiated as `A` and `B` respectively. This kind of variables in Isabelle are known as *schematic* variables and the process of replacing such variables is called *unification* [NPW98].

```
[|?P; ?Q|] ==> ?P /\ ?Q | ?P /\ ?Q ==> ?P | ?P /\ ?Q ==> ?Q
```

Figure 10: Conjunction Inference Rules – Isabelle Representation

We have seen how inference rules can be represented in Isabelle. Next we illustrate how such rules can be used during a proof. Lets start by explaining how this is done with introduction rules. The general case for applying the `[|A1; ; An|] ==> A` introduction rule to a `[|C1; ;`

Cm|] ==> C subgoal is to first unify A and C, remove the current subgoal and replace it with n subgoals [|C1; ; Cm|] ==> A1 [|C1; ; Cm|] ==> An. Consider A /\ B ==> B /\ A as the current subgoal. Applying the [|?P; ?Q|] ==> ?P /\ ?Q rule entails the unification of ?P /\ ?Q with B /\ A. This means that all instances of ?P and ?Q, throughout the rule, will be replaced by B and A respectively. Consequently we have progressed our proof state and we are now presented by two new subgoals:

```
1. A /\ B ==> B
2. A /\ B ==> A
```

Elimination rules are similar to introduction rules. Additionally however, the first premise of the inference rule is unified with a matching assumption from the current subgoal , that assumption is then removed. The general case for applying the [|A1; ; An|] ==> A elimination rule to a [|C1; ; Cm|] ==> C subgoal is to first unify A and C, then unify A1 with one of Ci, remove the current subgoal and replace it with n-1 subgoals [|C1; ; Cm-1|] ==> A1, [|C1; ; Cm-1|] ==> An-1. For the purposes of an example we use a different elimination rule for conjunctions, not presented before:

```
[|?P /\ ?Q; [|?P;?Q|] ==> ?R|] ==>?R
```

It specifies that if we know that a conjunction holds and we also know that and arbitrary proposition can be inferred from the propositions of the conjunction, then we can infer that arbitrary proposition. Lets apply this new elimination rule to the A /\ B ==> B subgoal (the first subgoal from the previous example). Like before, unification on the conclusions of the rule and the subgoal is performed, where occurrences of ?R, throughout the rule, are replaced by B. Unification on the first assumption of the elimination rule is performed where ?P and ?Q are replaced by A and B respectively. The unified assumption is eliminated and the subgoal is replaced by the remaining assumption ([|?P;?Q|] ==> $R) of the elimination rule:

```
[|A;B|] ==> B
```

Isabelle provides the rule and erule methods for applying introduction and elimination rules. These two methods are the main methods used in what is called backward or goal oriented reasoning. They are classified this way because they decompose the current subgoal and replace it with one or more subgoals based on that decomposition. Proofs in Isabelle are mainly performed in such a backward reasoning style. Additionally, Isabelle provides with the frule and drule (where f and d stand for forward and destruction) methods that allow for forward reasoning, where the subgoal is not decomposed but complemented by additional assumptions and/or subgoals.

In forward reasoning (i.e. using the `frule` method) the general case for applying a `[| A1; ; An|] ==> A` rule to a `[|C1; ;Cm|] ==> C` subgoal is first to unify `A1` with one of `Ci` and then to create n subgoals `[|C1; ;Cn|] ==> A2, ,[|C1; ;Cn|] ==> An`, and `[|C1; ,;Cn; A|] ==> C`. Performing rules using the `drule` method is identical to `frule`, however, the `Ci` unified assumption is removed.

3.4 Theories

Interacting with Isabelle, for most users, means that they will need to write theories. A theory is comprised of types, declarations, and proofs. There are two styles of theory files, one is referred to as *old-style* or *classic Isabelle* while the other as *new-style* or *Isar*.

In the following sections we will explain the two different styles of theories. Our medium for this explanation will be the definition of a simple object logic for propositions. Emphasis in this section will be placed on the differences between these theory styles and not on Isabelle's facilities for creating a new object logic - these facilities are explained in more detail in Section 3.5.

3.4.1 Old Style Theories

Two different types of files are needed to hold old styled theories: a theory file and a proof file. The theory file with extension `thy` contains the types, declarations, and axioms of the theory. The proof file is an ML file, with extension `ML`, that contains the proofs.

To illustrate the development of a theory in classic Isabelle we define a subset of propositional logic with negation, disjunction and conjunction. The old-style theory files are provided in the following two subsections. The contents of these files closely resemble the definitions found in VDM-LPF. This is intentionally done in order to facilitate the reader's understanding of VDM-LPF when it will be covered in the Chapter 4.

3.4.1.1 Theory File

A theory declaration has the following general format:

```
T = B1 + B2 + ... + Bn +
types, declarations
end
```

where `T` is the theory name, while `B1`, `B2`, and `Bn` are theories to be included in the current theory. The end of the theory is denoted by the `end` keyword. Figure 11 shows the contents of the `TestProp.thy` file. This file contains the types, definitions, and axioms of our theory. We can

tell that the theory name is `TestProp` and we include the definitions of the `Pure` theory – in other words we are extending the `Pure` theory.

The `types` and `arities` statements are used to declare logical types. Since this theory defines a subset of propositional logic we need to define a type that will denote propositions. The type is called *o* (reminiscent of Church's type *o* [Church40]) and is defined as follows:

```
types
  o
arities
  o :: logic
```

The `consts` keyword is used to declare logical constants with their associated type, as well as an optional infix syntax annotation. Declaring elements as constants is a prerequisite before such elements are allowed to appear in axioms, definitions, or proofs.

```
consts
  true  :: o
  false :: o
  neg   :: o => o       ("(~ _)"     [50])
  disj  :: o => o => o ("(_ | _)"   [30,31] 30)
  conj  :: o => o => o ("(_ & _)"   [40,41] 40)
```

Here we define constants for the truth values (`true` and `false`) and for the negation (`neg`), disjunction (`disj`), and conjunction (`conj`) logical connectives. The `::` character sequence separates the constant by its type. For example, both `true` and `false` constants are simple propositions, while `neg`, `disj`, and `conj` constants form compound propositions and their types are augmented by the `=>` symbol, which denotes the type of functions from `o` to `o`. Negation takes a proposition and returns a (compound) proposition. For disjunction and conjunction it states that they take two propositions and they return a compound proposition. An alternate syntax for the type of a constant can be formed as `conj :: [o,o] => o`, where `[o,o]` abbreviates par `o => o`. The `TRUEPROP` constant is a special type of constant which is used when defining new object logics – we defer its explanation until the next section.

These logical connectives are complemented by syntax annotations. These syntax annotations are a short form syntax. They have two main components, a pretty printing string and a set of priorities. The pretty printing string is enclosed within double quotes and parenthesis, where underscore characters denote the argument positions while the symbols `~`, `|`, and `&`, denote synonyms for the `neg`, `disj`, and `conj` constants. Consequently, the following compound proposition, `conj(A,B)` is pretty-printed to `A & B`. A set of optional priorities for each of the propositions and the connective as a whole can also be specified. This allows us to determine the association of the connective. As expected both `conj` and `disj` are set to associate to the right

since the right proposition has a higher priority. Moreover, `conj` has a higher priority than `disj` – thus, the following proposition, `A | B & C` is interpreted as `A | (B & C)`.

Next in our theory is a set of axioms, declared via the `rules` keyword, that establish some of the basic properties of our simple subset of propositional logic. We first establish one of the simplest axioms, `true_intr "true"`, which states that `"true"` is always true. Next we establish two properties of negation:

```
not_not_intr "P ==> ~ ~ P"
not_not_dest "~ ~ P ==> P"
```

which state that a doubly negated proposition is equal to its self. Recall that the `==>` symbol is the meta-implication symbol and is used to separate assumptions from the conclusion of axioms and rules. Finally, a number of properties for disjunction are defined:

```
or_intr_left  "P ==> P | Q"
or_intr_right "Q ==> P | Q"
or_elim       "[| P | Q; P ==> R; Q ==> R |] ==> R "
not_or_intr   "[| ~ P; ~ Q |] ==> ~ (P | Q)"
not_or_elim   "[| ~ (P | Q); [| ~ P; ~ Q |] ==> R |] ==> R"
```

The first two state that assuming that at least one proposition holds in a disjunction then we can conclude that the disjunction holds. The `or_elim` states if we know a disjunction to be true then we have to prove the conclusion twice – once for each proposition of the disjunction. Finally, the last two define properties of propositions containing negation and disjunction.

The last set of statements in our test theory is a set of definitions (or abbreviations), performed via the `defs` statement:

```
false_def "false   == ~ true"
conj_def  "p & q   == ~ (~ p | ~ q)"
```

This definitional approach is preferable because we do not have to write axioms to define properties for falsity and conjunction. It is always preferable to keep the number of axioms small since it can help reduce the risk of introducing contradictions. [NPW98]

```
TestProp = Pure +

types
  o

arities
  o :: logic

consts
  TRUEPROP :: o => prop ("_" 5)

consts
  true  :: o
  false :: o
  neg   :: o => o       ("(~ _)"      [50] 50)
  disj  :: o => o => o ("(_ | _)"    [30,31] 30)
  conj  :: o => o => o ("(_ & _)"    [40,41] 40)

rules
  true_intr    "true"

  not_not_intr "P ==> ~ ~ P"
  not_not_dest "~ ~ P ==> P"

  or_intr_left  "P ==> P | Q"
  or_intr_right "Q ==> P | Q"
  or_elim       "[| P | Q; P ==> R; Q ==> R |] ==> R "
  not_or_intr   "[| ~ P; ~ Q |] ==> ~ (P | Q)"
  not_or_elim   "[| ~ (P | Q); [| ~ P; ~ Q |] ==> R |] ==> R"

defs
  false_def "false   == ~ true"
  conj_def  "p & q   == ~ (~ p | ~ q)"

end
```

Figure 11: Old Style Theory File

3.4.1.2 Proof File

Proofs in old-styled theories are performed by using Isabelle's ML modules. Figure 12 shows the proof file that contains two proofs we performed based on the simple propositional logic we defined earlier.

The first command in the proof file is the `open TestProp;`, which is the ML command for *opening* ML structures. ML structures is the method to group ML declarations and associate them with a name.[Paulson96, Milner+97] By opening a structure all the declarations can be referenced without using name qualification. When Isabelle is asked to interpret an old-style theory it creates an ML file containing a structure of the same name. This structure contains the ML representation of the types and declarations of the theory. By opening the `TestProp` structure we can refer to the declarations, such as the axiom `true_intr` by just writing its name – the alternative is to write `TestProp.true_intr`.

```
open TestProp;

val [p1] = goalw TestProp.thy [] "P | Q ==> Q | P";
by (cut_facts_tac [p1] 1);
be or_elim 1;
br or_intr_right 1;
ba 1;
br or_intr_left 1;
ba 1;
qed "or_comm";

val [p1] = goalw TestProp.thy [conj_def] "P & Q ==> Q & P";
by (cut_facts_tac [p1] 1);
be not_or_elim 1;
bd not_not_dest 1;
bd not_not_dest 1;
br not_or_intr 1;
br not_not_intr 1;
ba 1;
br not_not_intr 1;
ba 1;
qed "and_comm";
```

Figure 12: Old Style ML Proof File

Proving that disjunction, as defined in our simple logic, commutes is the first proof we perform. The following statement shows how we initiate the proof:

```
val [p1] = goal TestProp.thy "P | Q ==> Q | P";
```

where `TestProp.thy` denotes the theory this proof belongs to, followed by the string `P | Q ==> Q | P` which is the theorem that expresses the commutative property of disjunction. It states that given the assumption that `P | Q` holds, then we can conclude that `Q | P` also holds. Optionally, the ML `goal` function may return a list containing the assumptions of the theorem – in this particular case only one assumption is present and is assigned in the ML variable `p1`.

Isabelle responds to the execution of the `goal` statement by displaying the proof state:

```
Level 0 (1 subgoal)
Q | P
 1. Q | P
```

where `Level 0` indicates the number of rules we have applied thus far, `1 subgoal` indicates the number of subgoals present at this level, `Q | P` displays the original theorem to be proved, and `1. Q | P` initiates the sequence of subgoals that we need to prove before we satisfy the original theorem.

The first tactic we use is the `cut_facts_tac`. With this tactic we are able to introduce to our first subgoal the assumption that `P | Q` holds. The new proof state is as follows:

```
Level 1 (1 subgoal)
Q | P
 1. P | Q ==> Q | P
```

Isabelle, again, informs us of the level, the number of subgoals, the theorem to be proved, and the subgoals we need to satisfy. In this case we can see that our assumption is part of the subgoal. We next proceed by applying the `or_elim` axiom as an elimination rule:

```
be or_elim 1;
```

This results in the following updated proof state:

```
Level 2 (2 subgoals)
Q | P
 1. P ==> Q | P
 2. Q ==> Q | P
```

This has transformed our subgoal into two subgoals, albeit with a simpler assumption. We operate next on the first subgoal and we use the `or_intr_right` axiom as an introduction rule. The new proof state now is as follows:

Level 3 (2 subgoals)

```
Q | P
 1. P ==> P
 2. Q ==> Q | P
```

The first subgoal now can be easily satisfied by assumption – when a subgoal is satisfied it is eliminated from the proof state. Consequently the new proof state is:

```
Level 4 (1 subgoal)
Q | P
 1. Q ==> Q | P
```

The remaining subgoal is satisfied in a similar manner. The only difference is that we apply the `or_intr_left` as an introduction rule in this case, resulting in the following proof state:

```
Level 5 (1 subgoal)
Q | P
 1. Q ==> Q
```

This subgoal is also satisfied by assumption, resulting in the completion of this proof. The following proof state indicates that the commutative property of disjunction has been proven:

```
Level 6
Q | P
No subgoals!
```

The proof now can be stored for later use. This is achieved by the `qed` statement. Once stored, this theorem can be invoked and used to further prove propositional theorems at any time via its name. It can be applied in the same manner axioms are applied. The `qed "or_comm"` statement results in the following output:

```
val or_comm = "?P | ?Q ==> ?Q | ?P" : Thm.thm
```

The propositions `P` and `Q` used in the original theorem have now changed to `?P` and `?Q`. Isabelle variables prefixed by a `?` character are referred to as *schematic variables*. Such variables are free variables that can be instantiated arbitrarily by appropriate terms.

The other proof, seen on Figure 12, establishes the commutative property for conjunctions. The same principles apply in this proof as with the previous one. The only difference is that in this proof we automatically expand the `conj_def` definition, through the use of `goalw` statement. This results in the following initial proof state:

```
Level 0 (1 subgoal)
Q & P
 1. ~ (~ Q | ~ P)
```

The subgoal has been rewritten in terms of negation and disjunction. This is done because all properties we have established are specified in such terms. We can now proceed to complete this proof using the existing properties. Eventually, we will have enough properties for conjunction established that such rewriting will be unnecessary. Remaining explanations on the `and_comm` proof are omitted because of their similarity to the previous proof.

3.4.2 New Style (Isar) Theories

With the introduction of Isar, a new language for writting theories and proofs is available. Isar eliminated the ML proof files and merged them with the theory ones. New statements, such as `lemma` and `theorem`, are introduced at the theory level to replace the ML functions such as `goal` and `goalw`. Furthemore, a number of syntactic changes were made. Figure 13 shows the `TestProp` theory written in the Isar language. Both these theories are equivalent, they differ only in the syntax. In what follows we will highlight these syntax differences.

A theory is now contained withing a `theory ... end` block and has the following general format:

```
theory T = B1 + ... + Bn:
types,declarations,proofs
end
```

where `T`, `B1`, and `Bn` have the same meaning as defined in the old-style.

The type `o` for propositions is declared now by a single statement, namely `typedecl o`. This statement has the same effect as the `types` and `arities` statements together.

The constant definitions are identical to the ones from the old-style theory. The only difference is the need to have the types of these constants enclosed within double quotes. The double quotes

can be removed when a simple type is involved. As such the double quotes for the type of the `true` and `false` constants are not necessary but were included for the sake of consistency.

Axioms are now defined using the very appropriate `axioms` keyword. Moreover, a colon character is used to separate the name of the axiom by the formula.

Definitions, for the most part, are exactly the same as the ones from the old-styled proofs. Like axioms, the names of the definitions and the actual definitions are seperated by a colon character.

The biggest change was the one involving proofs. With the introduction of Isar the proving syntax underwent a substantial change. Proofs now co-exist with the other definitions in the theory file and have the following general case:

```
lemma name: "ϕ"
apply(method)
...
done|oops
```

where *name* is the name of this lemma, ϕ is the formula to be proven, *method* denotes the different proof methods of applying axioms or previously proven lemmas. Finally, `done` denotes the completion of a successful proof while `oops` has the opposite effect.

Let trace the Isar `or_comm` proof in a similar way as it was done for the corresponding old-style proof. We will see how the ML tactics have been replaced by Isar methods, as well as, what the differences are, in terms of Isabelle's response to each of these methods. Isabelle displays the following proof state in response to the `lemma or_comm: "P | Q ==> Q | P"` statement:

```
proof (prove): step 0

fixed variables: P, Q

goal (lemma (or_comm), 1 subgoal):
1. P | Q ==> Q | P
```

`level` has now been replaced by the `step` label, however, their functionality remains the same. The `fixed variables: P, Q` displays the variables that are present in this theorem. Finally, Isabelle displays the list of subgoals for this proof.

The rules and the order they are applied are identical to the ones from the old-style proof. The only difference is that in this case there is no need to perform the Isar equivalent method for `cut_facts_tac` because the premise is retained. After the application of the `or_elim` rule the new proof state changes the current subgoal into two subgoals:

```
proof (prove): step 1

fixed variables: P, Q

goal (lemma (or_comm), 2 subgoals):
 1. P ==> Q | P
 2. Q ==> Q | P
```

The output shows that this is the first step and we need to resolve two subgoals. In Isar, when more than one subgoal is present, methods operate on the first one. Applying the `or_intr_right` rule, as before, will result in the following proof state:

```
proof (prove): step 2

fixed variables: P, Q

goal (lemma (or_comm), 2 subgoals):
 1. P ==> P
 2. Q ==> Q | P
```

Resolving the first subgoal is easily done by assumption and it modifies the proof state to:

```
proof (prove): step 3

fixed variables: P, Q

goal (lemma (or_comm), 1 subgoal):
 1. Q ==> Q | P
```

Next, the last subgoal is simplified by applying the `or_intr_left` rule:

```
proof (prove): step 4

fixed variables: P, Q

goal (lemma (or_comm), 1 subgoal):
 1. Q ==> Q
```

The proof is complete by assumption and Isabelle will display the following:

```
proof (prove): step 5

fixed variables: P, Q

goal (lemma (or_comm)):
No subgoals!
```

Where `No subgoals!` denotes that the proof was completed successfully. Finally the `done` command will end the proof and store it for later use. The output of the `done` command is similar to the old-style `qed` one:

```
lemma or_comm: ?P | ?Q ==> ?Q | ?P
```

The `and_comm` proof is also performed in the Isar theory. However, it is not explained in detail since it adds no further detail. As before, the only difference is the expansion of the `conj_def` definition. In Isar this expansion is performed by the `unfold` method.

```
Theory TestProp = Pure:

typedecl o

judgment TRUEPROP :: "o => prop" ("_" 5)

consts
  true  :: "o"
  false :: "o"
  neg   :: "o => o"       ("(~ _)"     [50])1
  disj  :: "o => o => o" ("(_ | _)"   [30,31] 30)
  conj  :: "o => o => o" ("(_ & _)"   [40,41] 40)

axioms
  true_intr:    "true"

  not_not_intr:  "P ==> ~ ~ P"
  not_not_dest:  "~ ~ P ==> P"

  or_intr_left:  "P ==> P | Q"
  or_intr_right: "Q ==> P | Q"
  or_elim:       "[| P | Q; P ==> R; Q ==> R |] ==> R "
  not_or_intr:   "[| ~ P; ~ Q |] ==> ~ (P | Q)"
  not_or_elim:   "[| ~ (P | Q); [| ~ P; ~ Q |] ==> R |] ==> R"

defs
  false_def: "false   == ~ true"
  conj_def:  "p & q   == ~ (~ p | ~ q)"

lemma or_comm: "P | Q ==> Q | P"
 apply(erule or_elim)
  apply(rule or_intr_right)
  apply(assumption)
 apply(rule or_intr_left)
 apply(assumption)
done

lemma and_comm: "P & Q ==> Q & P"
 apply(unfold conj_def)
 apply(erule not_or_elim)
 apply(drule not_not_dest)
 apply(drule not_not_dest)
 apply(rule not_or_intr)
  apply(rule not_not_intr)
  apply(assumption)
 apply(rule not_not_intr)
 apply(assumption)
done

End
```

Figure 13: Isar Theory File

3.5 Defining a New Object Logic

The previous section demonstrated how a new object logic can be defined in Isabelle. However, parts of the definition, such as the purpose of the TRUEPROP constant, and other components, such as syntax, the simplifier, and classical reasoner packages of Isabelle, were not described in that section. Consequently, this section portrays the process of extending Isabelle with a new object

logic. We will describe the different steps, explain what purpose they serve, and the tools Isabelle provides to help us in this creation process.

As mentioned earlier on, there are two ways to extend Isabelle – first by extending an already defined object logic, such as HOL, and second by extending generic Isabelle. Generic Isabelle (also known as Isabelle/Pure or as just Pure) is the bare-bones Isabelle that contains its meta-logic. This meta-logic is a subset of high-order logic based on typed λ-calculus and it is expressive enough to define constructs and proof rules of new logics.

In general the process of creating a new object logic entails the following steps:

- Declare the abstract syntax
- Declare the elementary and derived inference rules
- Declare the concrete syntax
- Instantiate the generic automatic proof tools
- Manually code special proof procedures

These steps are just a guideline – most new objects logics perform all these steps, however, some may not be necessary. Only the first two are absolutely mandatory for defining a new object logic. However, the remaining steps make the difference between a usable and an unusable logic.

3.5.1 meta-logic

The meta-logic is contained within the `Pure` theory. More precisely, the meta-logic is contained within the `ProtoPure` theory. `Pure` and its closely related `CPure` inherit the definitions of `ProtoPure` and extend it with the syntax of prefix function application. Pure's function application follows the tupled paradigm while CPure's follows the curried one [Paulson, NPW98].

The meta-logic is the subset of higher-order logic with implication (==>), equality(==), and universal quantification (!!) based on Church's typed λ-calculus. There are two main classes of terms in this meta logic: *logic* which denotes all logical types and *prop* which denotes formulae of the meta-logic. Extending the meta-logic, essentially means extending these two classes. Appendix C displays the Pure syntax where all elements of the meta-logic can be viewed, including the *logic* and *prop* classes. The remaining components are not discussed but a detailed explanation can be seen in [Paulson].

All logical types extend the *logic* class of the meta-logic. The *logic* class is "equipped with the syntax of types, identifiers, variables, parenthesis, λ-abstraction and application" [Paulson]. Statements, such as `typedecl o`, allow terms of type `o` to inherit the above syntactic elements and their meaning. Consequently, terms of these new logical types can be expressed using such a syntax – for example, `(true)`, `(%x. true & x)true`.

In each new object logic there is a statement that denotes the truth judgment for the logic. Statements, such as `judgment TRUEPROP :: "o => prop"`, specify that terms of type `o` can be used as formulae. Essentially, such judgment statements extend the class *prop* of the meta-logic and allow for such terms to be used as axioms and theorems [Paulson].

3.5.2 Declare the Abstract Syntax

After declaring the types and establishing the truth judgment of the logic, the next step is to declare the abstract syntax that Isabelle can understand and reason about. Such abstract syntax is defined by creating logical constants via the `consts` statement. For example, consider the following set of constants as seen in the simple propositional object logic example, presented on Section 3.4:

```
consts
  true  :: "o"
  false :: "o"
  neg   :: "o => o"
  disj  :: "o => o => o"
  conj  :: "o => o => o"
```

Note that we have removed the syntax annotations. This is because such annotations are part of the concrete syntax and not the abstract one. Such constant definitions allow us to express propositions in the abstract syntax. Propositions such as, `conj(P,Q)`, which denote the conjunction of the `P` and `Q` propositions.

3.5.3 Declare the Concrete Syntax

At first glance short and simple propositions, expressed in abstract syntax, could be considered as being sufficient. However, if we consider longer proposition such as:

```
neg(conj(P, disj(Q,neg(R))))
```

it becomes apparent that abstract syntax is not easy to understand and work with. If we imagine that such a statement is presented to a mathematician (i.e. the logic's target user) who is unaware of Isabelle's abstract syntax, we should not be surprised if such a presentation will create confusion.

Fortunately, Isabelle provides us with a number of ways to develop and decorate new object logics with a concrete syntax that closely resembles the syntax of the object logic. We have

already seen one such way in the form of syntax annotations. Syntax annotations are appropriate when the format of the abstract and concrete syntax closely resemble each other. In the case of our simple propositional logic example, the formats are closely matched, thus adding to the previous consts declarations to

```
consts
  true  :: "o"
  false :: "o"
  neg   :: "o => o"       ("(~ _)"      [50])
  disj  :: "o => o => o" ("(_ | _)"   [30,31] 30)
  conj  :: "o => o => o" ("(_ & _)"   [40,41] 40)
```

creates syntax annotations that will allow us to write the previously mentioned proposition into the more readable

```
~ ( P & ( Q | ~ R ) )
```

In the case where syntax annotations are not sufficient to bridge the gap between abstract and concrete syntax, Isabelle additionally provides us with macros or syntax translations and translation functions. For illustration purposes we will extend our simple `TestProp` theory with the implication connective, namely `=>`. Considering implication can be defined in terms of negation and disjunction, we can define it using a `defs` statement, similar to the manner conjunction is defined. However, we will instead opt for declaring it as a macro since that will further our goal of illustrating macros. The syntax and translation statements below declare the implication connective as a macro:

```
syntax
  impl  :: "o => o => o" ("(_ => _)"  [20,21] 20)
translations
  "p => q" == "~ p | q"
```

`syntax` statements are similar to `consts`, in the way they are declared.. However, `syntax` statements declare purely syntactic constants. Such statements are used to enrich the syntactic elements of a theory and they are not reasoned about by Isabelle. Consequently, such syntactic constants must be rewritten into logical terms for reasoning.

Such rewriting is performed via the `translations` statements. In our example translation, we inform Isabelle that, whenever propositions with the implication connective are encountered, they are first to be translated into their equivalent negation and disjunction form, then reason about them, and finally translate them back using the implication connective. Isabelle's parsing and printing process can be seen on Figure 14 and it will be explained in greater detail later on in this section.

Thus far we have seen syntax annotations and macros for creating a user-friendly syntax. The last method is that of translation functions. The translation function method is the most complex

and the most powerful one. It involves knowledge of Isabelle's parsing and printing process (as shown on Figure 14), knowledge of Isabelle's ML internal data structures, namely `term` and `ast` (as shown in Figure 15), and it requires knowledge of the ML programming language. We will neither attempt to be complete nor show an example of a translation function in this section; we will see a translation function during our explanation of VDM-LPF in Chapter 4. Instead we will describe Isabelle's parsing and printing process and point out the locations where translation functions can affect this process.

During parsing and printing of strings, there are places where users can affect the results of that process. When a string is written in Isabelle, the lexer breaks it into tokens and passes it to the parser where a parse tree is generated. That parse tree is then transformed into a tree represented by Isabelle's internal datatype `ast`. The user can affect this transformation by writing appropriate ML routines, known as parse AST translations. The AST is then further changed by applying the defined macros. This new AST is then translated into terms which are probably with incorrect types, since type-inference occurs later on in the process. Users can also affect this transformation by writing ML functions, which are known as parse translation functions. The last step of the parsing process is to transform those terms into well-typed terms which then Isabelle can reason about.

The printing process is the mirror image of the parsing process described above. The first transformation is the one that changes the well-typed terms into a new AST – user defined ML functions known as print translation functions can affect this transformation. Next the AST is modified by the application of the defined macros – this time the macros are changing the logical constants into the syntactic ones. Finally a new string is formed from the AST. This is the last place where a user can supply ML functions, known as print AST translation functions, to affect this transformation.

Isabelle's translation function apparatus is powerful and complex at the same time. However, when it comes to syntactic issues such as the one faced by the VDM-LPF creators, translation functions offer an elegant solution. It is the complexity in writing translation functions that prompted Larry Paulson to introduce translation functions by the following quote: "This chapter is intended for experienced Isabelle users who need to define macros or code their own translation functions" [Paulson].

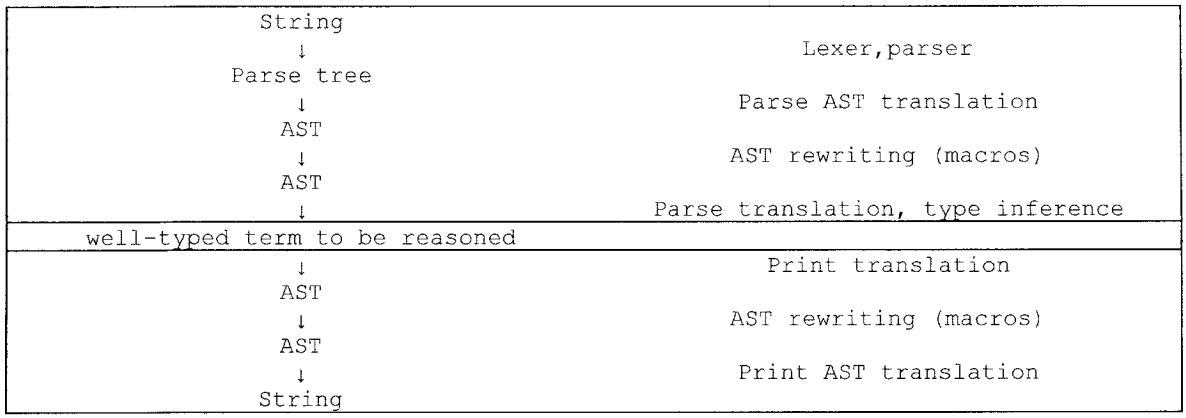

Figure 14: Isabelle's Parsing & Printing Process

```
datatype ast =
Constant of string |
Variable of string |
Appl of ast list

datatype term =
Const of string * typ |
Free of string * typ |
Var of indexname * typ |
Bound of int |
Abs of string * typ * term |
$ of term * term
```

Figure 15: Isabelle's Internal Data Structures Pertinent to Syntax Translation

3.5.4 Declare Inference Rules

Inference rules define the proof system of a new logic. First a set of elementary rules is defined in the form of axioms. These axioms are then used to further develop the proof system by deriving rules as shown in the previous section. We have seen how this is done through our `TestProp` example. We will see further examples when we describe VDM-LPF.

3.5.5 Instantiate the Automatic Proof Tools

The automatic proof tools are mainly the Simplifier and the generic Classical Reasoner packages. These packages are ML modules and may be instantiated to be used with new object logics. They are suited for classical object logics and require certain theorems to be present, theorems such as equality substitution, modus-ponens, etc. When instantiated for an appropriate object logic they provides a suite of automatic tactics, such as `Blast_tac`, `auto`, `force`, `simp`, `simp_all`, etc. [Paulson]. These tactics allow for automating large parts of proofs. For a more detailed discussion of the Simplifier and the Classical Reasoner refer to [Paulson].

3.5.6 Code Proof Procedures

In cases where the automatic proof tools are not appropriate, Isabelle provides with ML modules where it is possible to write proof procedures and allow for a certain degree of automation of the proving process. This creation of custom proof procedures will be seen in detail when we explain VDM-LPF in Chapter 4.

4 VDM-LPF Object Logic in Isabelle94-8

This chapter will present the definition of a new object logic, called VDM-LPF, as created by the original authors, Sten Agerholm and Jacob Frost [AF96]. As such, all code segments presented in this chapter are in their original form, which follows Isabelle's old-style theory syntax. The reader is assumed to have basic knowledge of Isabelle as is provided by Chapter 3. Additional material on the fore-mentioned subjects can be found in [Paulson]. Furthermore, we aim to present and explain what is deemed to be important towards understanding what VDM-LPF is and how it was defined. It is not our intention to provide a full presentation.

Sten Agerholm and Jacob Frost first developed the VDM-LPF logic in 1996 in an attempt to mechanize proof support for VDM-SL. Their long term goal was to develop an industrial proof support tool that could be integrated with IFAD's VDM toolbox [AF96]. They started by experimenting with a proof engine. To develop such an engine they used the Isabelle theorem proving environment by extending generic Isabelle with the Logic of Partial Functions (LPF). The result of these experiments is what has been called Isabelle/VDM-LPF or simply VDM-LPF.

The definition of VDM-LPF consists of syntax, a proof system, and a set of proof tactics. The syntax is further partitioned into abstract and concrete so to facilitate users by allowing them to write their VDM proof obligations in a familiar language. The proof system is defined based upon an elementary set of rules defined as axioms and a set of syntactic definitions. These axioms and definitions are then used to expand this system by deriving additional, often more complex, rules. Finally, a set of built in proof tactics provided by Isabelle as well as a set of search proof tactics, developed specifically for VDM-LPF, comprise the proof tactics system.

4.1 Syntax

VDM-LPF syntax is partitioned into abstract syntax and concrete syntax. The abstract syntax is what Isabelle "understands" and is able to reason about. The concrete syntax (a subset of VDM-SL) is what users, of the logic, utilize to write their specifications and proof obligations. Internally, concrete syntax terms are translated to the corresponding abstract syntax terms. Once reasoning is completed, abstract syntax terms are translated back to the equivalent concrete syntax terms and then are displayed as output. This is possible through Isabelle's powerful and flexible syntax translation facilities. However, notice that the exercise of creating the concrete syntax is not necessary, since one can use Isabelle by directly employing its abstract syntax. Nevertheless, allowing VDM users to create specifications and proof obligations in a familiar syntax makes the system more usable.

This is best illustrated by means of an example. Below you can see an example of a nested conditional statement, making use of predicate and propositional logic, specified in abstract syntax:

```
if'(forall'(A,(%x. P)), and'(p,q), if'(exists'(A,(%x. P)), or'(p,q),q))
```

This same statement expressed in the concrete syntax is:

```
if (forall x:A & P(x)) then (p and q) elseif (exists x:A & P(x)) then (p or q) else q
```

Both of these statements are equivalent, however, it is obvious that the concrete syntax is more intuitive to users, even ones unfamiliar with VDM-SL.

What follows below is a detailed explanation of the abstract and concrete syntaxes of VDM-LPF.

4.1.1 Abstract Syntax

The abstract syntax contains only two phrase classes, one for expressions and one for types. Thus, Isabelle's typed λ-calculus is extended with two new types `ex` and `ty` [AF96]. These are the only new types introduced for the purpose of representing the object logic VDM-LPF. Definitions and terms belong to either `ex` or `ty`, while formulas belong to just `ex`. Figure 16 shows how the `ex` type is declared in Isabelle. The `types` command declares a new type constructor `ex` while `arities` adds a new arity to the constructor. It is through this `arities` statement that we extend Isabelle's λ-calculus with the new type `ex`. The last statement of Figure 16 informs Isabelle that formulas will be terms of type `ex`. The `ty` type is declared in a similar manner, without the `Trueprop` constant definition, since formulas are of type `ex`.

```
types
ex

arities
ex :: logic

consts
Trueprop :: ex => prop ('(_)' 5)
```

Figure 16: Declaration of ex Type

Figure 17 shows a sample of the abstract syntax declared as Isabelle constants. These declare constants for universal and existential quantification, for the subtraction function of natural numbers, for the type of natural numbers, for equality, for the successor function of the natural numbers, and for the natural number zero. It should be noted that all natural numbers in VDM-LPF are internally denoted in terms of the successor function and zero. Digits as we know them still exist but they are translated in successor, zero terms for internal use.

```
forall' :: [ty,ex => ex] => ex
exists' :: [ty,ex => ex] => ex
sub'    :: [ex,ex] => ex
natty'  :: ty
eq'     :: [ex,ex] => ex
succ'   :: ex => ex
zero'   :: ex

forall'(natty',%x. exists'(natty',%y. eq'(y,succ'(x))))
```

Figure 17: Sample Abstract Syntax Definitions & Sample Terms

As an example, consider the following term of Boolean type which expresses the predicate that every natural number has a successor:

```
forall'(natty',%x. exists'(natty',%y. eq'(y,succ'(x))))
```

4.1.2 Concrete Syntax.

Concrete syntax, on the other hand, is useful for presenting a familiar notation to users; a notation that closely matches VDM-SL's syntax. Being able to create VDM-SL specifications and proof obligations in VDM-LPF, not only enhances usability of the tool, but also the interoperability between VDM tools – which was one of the stated goals of the original authors. To achieve this, the original authors make use of Isabelle's syntax annotations, syntax translations (called macros), and translation functions.

4.1.2.1 Syntax Annotations

In situations where the abstract and concrete syntax structures closely match each other, syntax annotation is an easy and effective way to create the association between the two. Figure 18 shows both the constant definitions and their associated syntax annotations for subtraction and equality. Outer parenthesis open and close a syntax annotation block. Such a block contains:

```
sub'    :: [ex,ex] => ex    ("(_ -/ _)" [410,411] 410)
eq'     :: [ex,ex] => ex    ("(_ =/ _)" [310,311] 310)
```

Figure 18: Sample of Syntax Annotations

- A template that denotes the concrete syntax: in the subtraction case, the inner parenthesis denote a pretty printing block where _ is an argument position and / is a possible line break. Thus, this pretty printing block denotes an argument, followed by a space, followed by the subtraction symbol -, followed by a space or a line break, and finally followed by a second argument.
- A set of optional priorities: again in the subtraction case, the priorities are provided and we see that the subtraction is associated to the right, because the right argument has a higher priority.

- Finally an optional priority for the whole construct may be provided. Missing priorities default to the lowest possible one, i.e. to priority 0.

As you may have already perceived, syntax annotations are merely decorations of the abstract syntax. They are however very powerful and can be confusing to new users, prompting the Isabelle creators to warn that novices will be better off to avoid them altogether and keep writing their terms using abstract syntax [PNW98].

4.1.2.2 Syntax Translations

When the structures of the abstract and concrete syntaxes differ, syntax translations, also known as macros, are used to bridge the gap. Unlike macros known from programming languages, Isabelle's macros work both ways; as they can be used to deal with both parsing and printing [Paulson]. This is easily specified by the operator used in the translation statement; these operators are == for both parse and print, => for parse only, and <= for print only

Figure 19 presents all the components necessary for the syntax translations of the universal and existential quantifiers. The need for syntax translations is evident because quantifiers in VDM-SL can bind more than one variable (e.g. `forall x:nat, y:nat & P`). To achieve this, the `tbinds` and `tbind`, syntax only, types are declared. Syntax only types is another mechanism, Isabelle provides, for facilitating the creation of concrete syntax. Notice that no `arities` for these types are declared; this is because it is the `arities` declarations that creates logical types. These syntactic types will be replaced by appropriate logical ones through translations.

The definitions under the `syntax` commands are similar to constant declarations, with the only exception being that they are not reasoned by Isabelle. Their only purpose is to extend Isabelle's syntax. The first set of syntax definitions handle the multiple variable bindings of the universal quantifiers, while the second one declares the syntactic equivalent definitions for `forall` and `exists`.

The next step in the process is to relate these syntactic elements to logical ones so that they can be reasoned about. This is done through Isabelle's `translations` statements. In the `translations` section of Figure 19, the first translation of both `forall` and `exists` unfolds quantifications of the form `forall x1:τ1, x2:τ2, , xn:τn & P` to `forall x1:τ1 & forall x2:τ2 & & forall xn:τn & P`. The second translation statement replaces unfolded concrete syntax statements with their abstract syntax counterparts. Thus, unfolded statements such as the one above will look like `forall'(τ1,%x1. forall'(τ2,%x2. forall'(τn,%xn. P) ))` in the abstract syntax.

```
Types tbinds tbind

syntax
""       :: tbind => tbinds                ("_")
tbindsn  :: [tbind,tbinds] => tbinds       ("(_,/ _)")
tbind    :: [idt,ty] => tbind      ("(_ :/ _)")

consts
forall' :: [ty,ex => ex] => ex
exists' :: [ty,ex => ex] => ex

syntax
forall  :: [tbinds,ex] => ex     ("(2forall/ _ &/ _)" [100,100] 100)
exists  :: [tbinds,ex] => ex     ("(2exists/ _ &/ _)" [100,100] 100)

translations
"forall_ (tbindsn_ tb tbs) e" == "forall_ tb (forall_ tbs e)"
"forall_ (tbind_ x A) e" => "forall' A (%x. e)"
"exists_ (tbindsn_ tb tbs) e" == "exists_ tb (exists_ tbs e)"
"exists_ (tbind_ x A) e" => "exists' A (%x. e)"
```

Figure 19: Syntax Translation for Universal & Existential Quantifiers.

4.1.2.3 Translation Functions

Translation functions are also used in VDM-LPF. They deal with abstract to concrete syntax transformations and vice-versa, that neither syntax annotations nor syntax translations can deal with. They are the most powerful method for dealing with differences between abstract and concrete syntax and the most complicated. Writing translation functions is complicated because it requires knowledge of Isabelle's parsing and printing process, Isabelle's internal data structures, and the ML programming language. For more information regarding this process and its related internal data structures please visit Section 3.5.3.

From Figure 19 we can see that there are two translations that are declared as parse only. This is necessary since these translation statements cannot handle the printing of terms with quantifiers and multiple variable bindings in an appropriate manner. If these translations were changed to include printing, then only the outer most binding variable would be translated, while the rest would be left unchanged. Consequently a print translation function is provided to handle this situation. Figure 20 shows the ML functions and the way these functions are attached to the function translation apparatus. `quan_tr'` is the translation function while `eta_exp` is a helper function. In just this small code segment one can see elements of the Isabelle `term` built in data type: elements such as `Abs`, `Const`, `op $` which denote terms of constants, λ-abstractions, and function applications respectively. `print_translation` is a built in variable that creates the association between logical and syntactic elements. Thus, when a logical element is encountered the call to the associated function is performed. In this particular example when `forall'` and `exists'` logical elements are encountered, a call to `quant_tr'` with either `"forall_"` or

"exists_" syntax only elements, as the first argument, is performed. Moreover, all the terms on the left side of the original forall' and exists' elements are passed as arguments to the quant_tr' function. This function will then use all these data to generate the corresponding concrete syntax term.

```
fun eta_exp (e as Abs(_,_,_)) = e
| eta_exp e =Abs("x",dummyT,e$(Bound 0));

fun quan_tr' r [A,e] =
let val Abs(x,_,e') = eta_exp e
val (x',e'') = variant_abs(x,dummyT,e')
in
Const(r,dummyT)$(Const("tbind_",dummyT)$Free(x',dummyT)$A)$e''
end;

val print_translation = [("forall'",quan_tr' "forall_"),("exists'",quan_tr'
"exists_")];
```

Figure 20: Sample of a Print Translation Function

4.2 Proof System

The proof system was copied from [Bicarregui+94]. It is defined axiomatically, which means that axioms are used to define the properties of basic elements for the logic. The system is further extended through a set of definitions and a set of derived rules, built upon the foundations laid by the previously defined axioms. In addition to rules and definitions for propositional and predicate LPF, [Bicarregui+94] also defines a number of rules for data types such as natural numbers, sets, sequences, maps, and booleans [AF96]. These rules follow the natural deduction style which is particularly well supported in Isabelle.

What follows is an explanation of how the proof system as seen in the previously mentioned book can be written as Isabelle theories.

4.2.1 Axioms

Encoding those rules from the book in Isabelle was a fairly straight forward process. In [Bicarregui+94] axioms are presented using the Hilbert-style system, where hypothesis and conclusion are separated by a horizontal line. Axioms in both the book and Isabelle are used to state undisputed facts that are universally accepted and that require no further reasoning to check their validity; in other words axioms are rules that we know are true. Some sample axioms can be seen in Figure 21. Also note that the naming conventions suggest that rule names usually end in

an I or an E. I stands for introduction while E for elimination. Introduction rules operate on the conclusion of a goal while elimination rules on the hypothesis.

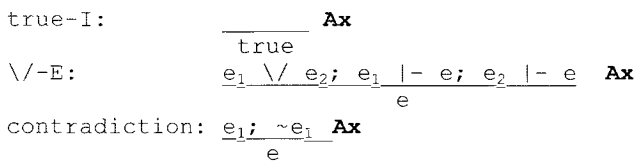

Figure 21: Axioms Definitions as Found in the [Bicarregui+94] Book

Rewriting the axioms found in the fore-mentioned book in Isabelle is fairly easy. Axioms are always written in the theory file under the rules section. Hypothesis and conclusion are separated by the meta-implication symbol, `==>`. When more than one hypothesis is present they are enclosed within the `[|` and `|]` symbols and are separated by a `;` character. When there is a need to denote an arbitrary element in either conclusion and or hypothesis, then the meta-universal quantification symbol, ||, is used. Isabelle requires a set of logical constants to be defined for the new logical connectives. It is worth noting that although these constants are declared through `consts` statements, all their properties are defined through axioms. Thus the axioms mentioned above and the newly introduced logical constants are expressed, in Isabelle, as shown in Figure 22.

```
Consts
true' :: ex    ("true")
or' :: [ex,ex] => ex    ("(_ or\ _)" [230.231] 230)
not' :: ex => ex    ("(2 not \_)" [250] 250)
rules
true-intr "true"
or-elim    "[| P or Q; P ==> R; Q ==> R |] ==> R"
contr      "[|P;  not P|] ==> Q"
```

Figure 22: Isabelle Axiom Definitions

4.2.2 Definitions

Sometimes it is not necessary nor appropriate to extend a theory by means of axioms. In the case of propositional LPF, it is sufficient to introduce all necessary properties of truthfulness, negation, and disjunction by axioms. Introducing further propositional connectives via axioms is unnecessary because connectives such as disjunction and implication can be defined in terms of disjunction and negation while bi-implication can be defined in terms of conjunction and implication Thus, syntactic definitions are a simple way for expressing the remaining propositional connectives. The remaining connectives for propositional LPF can be seen in Figure 23 as syntactic definitions.

```
e1 /\ e2 == ~(~e1 \/ ~e2)
false == ~true
e1 => e2 == ~e1 \/ e2
e1 <=> e2 == (e1 => e2) /\ (e2 => e1)
```

Figure 23: Definitions of Propositional LPF from the [Bicarregui+94] Book

These syntactic definitions can be easily written in Isabelle via `consts` and `defs` statements as shown in Figure 24.

```
Consts
false'          :: ex                    ("false")
iff'            :: [ex,ex] => ex         ("(_ <=>/ _)" [210,211] 210)
imp'            :: [ex,ex] => ex         ("(_ =>/ _)" [221,220] 220)
and'            :: [ex,ex] => ex         ("(_ and/ _)" [240,241] 240)
defs
false_def
"false == not true"
imp_def
"P => Q == not P or Q"
and_def
"P and Q == not (not P or not Q)"
iff_def
"P <=> Q == (P => Q) and (Q => P)"
```

Figure 24: Isabelle Definitions of Propositional LPF

4.2.3 Derived Rules

Derived rules are denoted in exactly the same manner as axioms with the only exception being the **Ax** notation is missing. Unlike axioms, derived rules are premises whose validity needs to be verified. This is done using axioms, syntactic definitions, and/or previously derived rules. It is worth noting that derived rules and definitions are not necessary to perform proofs. Proofs can be performed using axioms only. However, such proofs are often lengthy and not very intuitive.

$$\sim\text{-}\backslash/\text{-I-deM}: \frac{\sim e_1 \;\backslash/\; \sim e_2}{\sim(e_1 \;/\backslash\; e_2)}$$

$$/\backslash\text{-comm}: \frac{e_1 \;/\backslash\; e_2}{e_2 \;/\backslash\; e_1}$$

$$\Rightarrow\text{-trans} \; \frac{e_1 \Rightarrow e_2;\; e_2 \Rightarrow e_3}{e_1 \Rightarrow e_2}$$

Figure 25: Propositional LPF Derived Rules from [Bicarregui+94]

Figure 25 presents a small sample of propositional LPF derived rules. The first is one of the deMorgan rules, the second denotes the commutative property for conjunctions, and the last defines the transitive property of implication.

Rewriting rules in Isabelle is identical to rewriting axioms. The only difference is that derived rules reside in the ML proof files. The rules above can be seen rewritten in Isabelle in Figure 26. The !! is the ascii representation of the meta-universal quantifier. Regarding their proofs, refer to the proof tactics section.

```
~~\/-I-deM: !!e1. not e1 and not e2 ==> not (e1 or
e2)
/\-comm: !!e1. e1 and e2 ==> e2 and e1
=>-trans: !!e1. [| e1 => e2; e2 => e3 |] ==> e1 =>
e3
```

Figure 26: Isabelle Propositional LPF Derived Rules

4.3 Proof Tactics

The set of proof tactics falls under two categories, the simple or built-in ones and the proof search tactics designed and developed specifically for VDM-LPF by Sten Agerholm and Jacob Frost.

4.3.1 Built in Tactics

The built-in proof tactics are the standard resolution tactics provided by Isabelle, such as `resolve_tac`, `eresolve_tac`, `assume_tac`, etc., and their short form `br`, `be`, `ba`, etc., respectively. These tactics make use of existing rules and definitions to derive new rules.

In order to illustrate the process we will perform a proof using exclusively built in tactics, axioms, and definitions. The rules from Figure 26 will be used for the sake of consistency. Figure 27 shows the proof as performed using the interactive Isabelle interface. Shaded text represents user input while the remaining text presents Isabelle's output. The proof is initiated by the statement:

```
goalw Prop.thy [and_def] "!!P. not P and not Q ==> not (P or Q)";
```

Where `"!!P. not P and not Q ==> not (P or Q)"` is the rule to be proved and `[and_def]` is a definition to be expanded. Isabelle responds by displaying one subgoal, labeled "1.", with the rule rewritten with the right hand side of the `and_def` definition.

The next step consists of finding a suitable rule to apply. By consulting the propositional LPF axioms, documented in Appendix A, we can identify two such possible axioms that may be applied: `not_or_elim` and `nor_or_intr`. We can choose either one. However, we choose to start with `not_or_elim` simply because the hypothesis seems to be slightly more complex than the conclusion; thus simplifying it first seems to be the right route. The rule is applied as seen below:

```
be not_or_elim 1;
```

Again we choose to simplify the assumptions in order to remove some of the nested `not` connectives. This is easily done via the `not_not_dest` axiom. This rule is applied twice so that both assumptions are simplified:

```
bd not_not_dest 1;
```

Next we turn our attention to the conclusion of the subgoal in question. This time we apply the `not_or_intr` axiom that is identical to our stated goal:

```
br not_or_intr 1;
```

This time Isabelle displays two sub-goals where, in each one of them, the conclusion is identical to one in the assumptions. Thus, both goals can be resolved by applying the assumption tactic twice:

```
ba 1;
```

Finally the desired output, `No subgoals!`, is displayed by Isabelle. The new rule is stored through the `qed` statement in the mnemonic name `not_or_intr_dem`. This rule can be further used through this name to help in deriving additional rules.

```
> goalw Prop.thy [and_def] "!!P. not P and not Q ==> not (P or Q)";
Level 0
!!P.  not P and  not Q ==>  not (P or Q)
1. !!P.  not ( not  not P or  not  not Q) ==>  not (P or Q)
val it = [] : thm list
> be not_or_elim 1;
Level 1
!!P.  not P and  not Q ==>  not (P or Q)
1. !!P. [|  not  not  not P;  not  not  not Q |] ==>  not (P or Q)
val it = () : unit
> bd not_not_dest 1;
Level 2
!!P.  not P and  not Q ==>  not (P or Q)
1. !!P. [|  not  not  not Q;  not P |] ==>  not (P or Q)
val it = () : unit
> bd not_not_dest 1;
Level 3
!!P.  not P and  not Q ==>  not (P or Q)
1. !!P. [|  not P;  not Q |] ==>  not (P or Q)
val it = () : unit
> br not_or_intr 1;
Level 4
!!P.  not P and  not Q ==>  not (P or Q)
1. !!P. [|  not P;  not Q |] ==>  not P
2. !!P. [|  not P;  not Q |] ==>  not Q
val it = () : unit
> ba 1;
Level 5
!!P.  not P and  not Q ==>  not (P or Q)
1. !!P. [|  not P;  not Q |] ==>  not Q
val it = () : unit
> ba 1;
Level 6
!!P.  not P and  not Q ==>  not (P or Q)
No subgoals!
val it = () : unit
> qed "not_or_intr_dem";
val not_ot_intr_dem = " not ?P and  not ?Q ==>  not (?P or ?Q)" : thm
val it = () : unit
```

Figure 27: deMorgan's Proof via Built in Tactics

4.3.2 Proof Search Tactics

An observation one should be able to make, by looking at the proof in Figure 27, is that for even a simple rule, a large number of steps are performed. The number of steps, in this case, do not pose any problems. However, in more complex proofs this will result in long, tedious, often confusing proofs. To alleviate this, Isabelle supports the creation of user-built proof search tactics.

The proof search tactics is a new package, developed in the form of an ML functor, so that it can fill the void of the generic classical reasoning package missing from VDM-LPF. The new package is needed because LPF is a non-classical logic, thus making the use of the classical reasoning package difficult [AF96].

This package can be found in Appendix B. It follows the same structure as Isabelle's classical reasoner. That is, it defines an ML data structure for holding set of rules. These rules are divided into introduction, elimination, and destruction rules. Rules in these groups are classified as being safe and unsafe rules. Safe rules can be applied without the loss of information and can be backtracked. Unsafe rule may cause loss of information; thus backtracking is not possible.

The new search tactic developed in this package is `lpf_fast_tac`. This tactic is applied in the same manner the built in ones are. However, instead of applying a single rule, a set of rules is provided. Rules from this set are then applied, one by one, until there are no more subgoals left. Safe introduction, elimination, and destruction rules are applied first, followed by the unsafe ones, in that same order. In the case where there are subgoals left, Isabelle displays an appropriate message and leaves it up to the user to decide how to proceed next.

In what follows, we prove the same deMorgan's rule, this time using the `lpf_fast_tac`. A prerequisite is to set up an appropriate set of rules for the proof (depicted in Figure 28). The actual proof is depicted on Figure 29. The advantages of the search proof tactics are obvious; the proof is performed in just one step, while the rule set can be expanded and reused as often as required.

```
> val ruleSet = empty_lpfs addSIs [not_or_intr] addSEs [not_or_elim] addSDs
[not_not_dest];
val ruleSet =
LPFS
{hazDs=[],hazEs=[],hazIs=[],safeDs=[" not  not ?P ==> ?P"],
safeEs=["[|  not (?P or ?Q); [|  not ?P;  not ?Q |] ==> ?R |] ==> ?R"],
safeIs=["[|  not ?P;  not ?Q |] ==>  not (?P or ?Q)"]} : lpf_set
```

Figure 28: Setting up the Rule Set for a deMorgan's Proof

```
> goalw Prop.thy [and_def] "!!P. not P and not Q ==> not (P or Q)";
Level 0
!!P.  not P and  not Q ==>  not (P or Q)
1. !!P.  not ( not  not P or  not  not Q) ==>  not (P or Q)
val it = [] : thm list
> by (lpf_fast_tac ruleSet 1);
Level 1
!!P.  not P and  not Q ==>  not (P or Q)
No subgoals!
val it = () : unit
> qed "not_or_intr_dem1";
val not_or_intr_dem1 = " not ?P and  not ?Q ==>  not (?P or ?Q)" : thm
val it = () : unit
```

Figure 29: deMorgan's Proof via Search Tactics

4.3.3 Rule Sets

As axioms and derived rules are defined, they are added to increasingly larger and more complex rule sets. A rule set is a set of related rules that can be used to derive new rules. These rule sets are used in conjunction with the `lpf_fast_tac` to perform this task.

Rule sets are defined in the same manner as shown in Figure 28. Essentially, they are an ML variable of the `lpf_set` datatype. There are a number of such rule sets defined throughout VDM-LPF. The most important and widely used ones are the following:

- `prop_lpfs` – Rule set for Propositional Logic
- `exists_lpfs` & `forall_lpfs` – Rule sets for Predicate Logic
- `bool_lpfs` – Rule set for the Boolean type.

Note that there exist a number of rules that do not belong in either one of these rule sets. They were not added because they were either too unsafe or not necessary. Nevertheless, they are still available and can be used as desired.

5 Isabelle/Isar 2005 support for VDM-LPF

In this chapter we describe the upgrading process of the VDM-LPF old-style theory development to the new-style or Isar one. Understanding this chapter requires a basic knowledge of VDM-LPF, as presented in Chapter 4, and the basic structures of Isabelle theories old and new styles (also known as classic and Isar respectively) as presented Chapter 3.

The aim of this chapter is to provide a detailed account on the steps undertaken to upgrade VDM-LPF so that it complies with the syntax, proof style, and organization of the latest version of Isabelle, namely Isabelle/Isar 2005. These steps are described in detail in order to enable the reader to retrace our work. When necessary, alternative choices are presented and the rationale behind the final choice is explained. Such information will be of use to others who may wish to covert old-style theories to the new style.

Upgrading the VDM-LPF theory files to Isabelle/Isar was deemed desirable because Isabelle/Isar offered a much improved deductive environment. Isabelle/Isar offers a new theory format that supports interactive development with unlimited undo operations. Moreover, it offers a new language to perform proofs, either by emulating the tactics used in ML proof scripts or by using the new proof language designed to closely resemble proofs as performed by humans. Finally, it offers a document preparation system based on existing PDF-LaTeX technology as part of the Isabelle/Isar theory syntax [Wenzel].

This upgrade was implemented in three stages. The first stage involved an attempt to execute the VDM-LPF theories in their original form, under an older Isabelle build. This served to establish the validity of the theory files before changes to them were applied.

After consultations with the Isabelle community, it was decided that it would be preferable to get VDM-LPF working with the latest version of Isabelle in the classic mode, known as Isabelle/Classic or simply Classic. This mode allows the execution of old style theories without significant modifications.

Finally, the last and most complex step was to transform these old style theories into fully fledged Isar ones. A number of changes were needed to achieve this transformation. First and foremost, all the ML proofs scripts were eliminated and all the proofs rewritten and moved into the theory files. A way to enable access to the VDM-LPF proof search tactics was found. Translation functions were also moved into the theory files. Last but not least a number of relatively small changes, pertinent to syntactic changes of Isabelle's elements, have been performed.

5.1 VDM-LPF in Isabelle94-8

The first step in the upgrading process from VDM-LPF to the Isabelle/Isar level, was to try and use it with an older version of Isabelle, a version that required no changes of the theory files and preferably the one that was used by the original authors. There were two reasons that made us decide to try to use an older version of Isabelle with the original VDM-LPF.

The first reason stemmed from the fact that retrieving VDM-LPF through its advertised repository was not successful. A copy of VDM-LPF was found at the website of Jeremy Dawson (a Post-Doctoral Fellow in the Logic and Computation programme of NICTA). Retrieving a copy in such a manner raised questions on the validity of the theory files. Consequently, using an older version of Isabelle to load the theory files, in their original form, and to perform checks was necessary. It is worth noting that attempts to contact the original authors failed. We were, however, able to contact Jeremy Dawson and inquire on the validity of the theory files. He was confident that the files were unmodified, but he was not certain.

The second reason was that attempts to run VDM-LPF using Isabelle 2005 were failing. That was an expected result given the number of changes made to Isabelle since 1996, the date VDM-LPF was developed. Consequently, starting with a working copy of the original logic was the prudent way forward.

Obtaining an appropriate version of Isabelle for the original VDM-LPF was an easy process. Fortunately, all Isabelle builds starting from 1993 and afterwards are available for download from Isabelle's website. By consulting the available list of builds, it was easy to pick the Isabelle94-8 package, which contains all changes made between 1994 and 1998. The installation was uneventful. The only exception was a small problem pertinent to the underlying ML interpreter, which was quickly resolved.

Invoking Isabelle94-8 was also a simple affair. However, loading of VDM-LPF did not work as expected. The reason for this unexpected result quickly became apparent. Initially Isabelle was invoked in the following manner:

```
${ISABELLE94_8_HOME}/bin/isabelle
```

When Isabelle is invoked in such a way, the pre-built HOL (High Order Logic) which is based on `CPure` is loaded by default. However, VDM-LPF is based on `Pure`. Consequently, invoking Isabelle with the `Pure` logic or another logic that was based on the `Pure` one were the two alternate solutions. The latter was preferred because FOL is one of the pre-built logics and is

present in all installations, while Pure is not. Thus invoking Isabelle in this manner had the desired effect:

```
${ISABELL04_8_HOME}/bin/isabelle FOL
```

Loading the theory files was next. Unfortunately there was no `ROOT.ML` file present, thus a brief search was conducted to identify the top theory. As a convention, in old-style theories, a `ROOT.ML` file is used as the top level file for an Isabelle project. The `VDM_LPF.ML` was identified

```
> ${ISABELLE94_8_HOME}/bin/isabelle FOL
val it = false : bool
> val_commit = fn : unit -> bool
> use_thy "VDM_LPF"
```

Figure 30: Isabelle94-8 - Loading VDM-LPF Theories

as the top level theory and was loaded into Isabelle as shown in Figure 30. Corresponding output is not shown due to its length.

Being successful in loading the VDM-LPF files into Isabelle94-8 boosted our confidence in the validity of the copy of VDM-LPF we obtained. To further convince ourselves we used [Bicarregui+94], upon which the Proof system was based, to compare it with what was in the actual theories. The results were encouraging. All the elements that we examined matched with the ones found in the book. First, we compared the axiomatization of the rules against their Isabelle implementation. We then proceeded to perform some simple proofs. In both these tasks we mainly focused on the propositional and predicate subset of VDM-LPF, simply because it is the most complete. Remaining theories, such as sequences, maps, etc., were only briefly examined, for the purposes of this validation exercise.

Some of the interactive proofs we performed using the Isabelle94-8 client are provided in Figure 31, Figure 32, and Figure 33. In what follows we present a brief explanation of these proofs. Moreover, we only make use of the simple built-in proof tactics, because we were interested in visualizing all the steps of the proof.

The first proof involves the `"q and p ==> p or q"` lemma. The first rule applied is the `or_intr_left` axiom

```
P ==> P or Q
```

which states that if we have a disjunction of two elements and we know that one of these elements is true then we can conclude that the disjunction is also true. This is an introduction rule and as such operates on the conclusion of our lemma. Next the `and_elim` elimination rule

```
[| P and Q; [| P; Q |] ==> R |] ==> R
```

is applied which specifies that if a conjunction is known to be true then we can assume that both elements of such conjunction are true. Since this is an elimination rule it will operate on the assumptions of the lemma. Finally, the proof is completed by assumption.

```
> val [p1] = goalw Prop.thy [] "q and p ==> p or q";
stdIn:23.1-23.50 Warning: binding not exhaustive
          p1 :: nil = ...
Level 0
p or q
 1. p or q
val p1 = "q and p  [q and p]" : thm
> by (cut_facts_tac [p1] 1);
Level 1
p or q
 1. q and p ==> p or q
val it = () : unit
> br or_intr_left 1;
Level 2
p or q
 1. q and p ==> p
val it = () : unit
> be and_elim 1;
Level 3
p or q
 1. [| q; p |] ==> p
val it = () : unit
> ba 1;
Level 4
p or q
No subgoals!
val it = () : unit
```

Figure 31: First Sanity Test of VDM-LPF

The second proof involves a simple predicate logic lemma, `forall x:A & x=x`. The first step in the proof is to expand the `forall_def` definition

```
forall'(A,P) == not (exists x:A & not (P(x)))
```

which states that the universal quantifier is defined in terms of exists and negation. Next the `not_exists_intr` axiom

```
[| !!x.x:A ==> not (P(x)) |] ==> not (exists'(A,P))
```

is applied. This axiom says that it is sufficient to find an arbitrary element where the predicate is not true to satisfy the negation of the existential quantification. Next the double negation axiom `not_not_intr`

```
P ==> not not P
```

is applied with the obvious meaning. The last rule applied is the `eq_self_intr` axiom

```
a:A ==> a = a
```

which says that an element is equal to itself.

```
> goalw Pred.thy [forall_def] "forall x:A & x=x";
Level 0
forall x : A & x = x
 1.  not (exists x : A &  not x = x)
val it = [] : thm list
> br not_exists_intr 1;
Level 1
forall x : A & x = x
 1. !!x. x : A ==>  not  not x = x
val it = () : unit
> br not_not_intr 1;
Level 2
forall x : A & x = x
 1. !!x. x : A ==> x = x
val it = () : unit
> br eq_self_intr 1;
Level 3
forall x : A & x = x
 1. !!x. x : A ==> x : ?A2(x)
val it = () : unit
> ba 1;
Level 4
forall x : A & x = x
No subgoals!
Val it = () : unit
```

Figure 32: Second Sanity Test of VDM-LPF

In the last proof we deviate from propositional and predicate logic and we try to prove something slightly more complicated. This proof involves a VDM-SL conditional statement dealing with variables ranging over the type of natural numbers and Booleans. The lemma `[| n:nat; m:nat; m=n |] (if n=m then true else false) = true` was also successfully proven by first applying the `if_true` axiom:

```
[| b:A; P |] ==> (if P then b else c) = b
```

This axiom is used because the assumption `m=n` points to the fact that the condition of the conditional statement is true. This axiom splits the lemma into two subgoals. The first one was quickly satisfied by the application of the `true_form`

```
true:bool
```

axiom, found in the theory of Boolean types. The second subgoal is satisfied by applying the `eq_symm1`

```
[| a:A; a = b |] ==> b = a
```

axiom which denotes the symmetric property of equality.

```
> val [p1,p2,p3] = goalw Cond.thy [] "[| n:nat; m:nat; m=n |] ==> (if n=m then
true else false) = true";
stdIn:34.1-34.102 Warning: binding not exhaustive
          p1 :: p2 :: p3 :: nil = ...
Level 0
(if n = m then true else false) = true
 1. (if n = m then true else false) = true
val p1 = "n : nat  [n : nat]" : thm
val p2 = "m : nat  [m : nat]" : thm
val p3 = "m = n  [m = n]" : thm
> by (cut_facts_tac [p1,p2,p3] 1);
Level 1
(if n = m then true else false) = true
 1. [| n : nat; m : nat; m = n |] ==> (if n = m then true else false) = true
val it = () : unit
> br if_true 1;
Level 2
(if n = m then true else false) = true
 1. [| n : nat; m : nat; m = n |] ==> true : ?A3
 2. [| n : nat; m : nat; m = n |] ==> n = m
val it = () : unit
> br true_form 1;
Warning: signature of proof state has changed
New theories: Bool, Seq, Map, Set, Nat, BasTy, Sub, Opt, Prod, Union, Let
Level 3
(if n = m then true else false) = true
 1. [| n : nat; m : nat; m = n |] ==> n = m
val it = () : unit
> br eq_symm1 1;
Level 4
(if n = m then true else false) = true
 1. [| n : nat; m : nat; m = n |] ==> m : ?A4
 2. [| n : nat; m : nat; m = n |] ==> m = n
val it = () : unit
> ba 1;
Level 5
(if n = m then true else false) = true
 1. [| n : nat; m : nat; m = n |] ==> m = n
val it = () : unit
> ba 1;
Level 6
(if n = m then true else false) = true
No subgoals!
val it = () : unit
```

Figure 33: Third Sanity Test of VDM-LPF

5.2 Porting to Isabelle 2005 Classic

Executing VDM-LPF in the latest version of Isabelle, but in the classic mode of operation, was chosen as the next step in the upgrading sequence because it was the natural and a relatively small step forward. This entailed the following actions:

(1) renaming three theories that were in conflict with existing FOL theories;
(2) VDM-LPF's proof search tactics had to be modified because of ML syntax errors;
(3) syntactically, Isabelle has evolved since 1996 and changes to the theory files had to be performed to bring them up to date;
(4) finally, VDM-LPF's base logic was changed from Pure to CPure.

Invoking Isabelle 2005 in Classic mode is done as follows:

```
% ${ISABELLE_HOME}/bin/Isabelle -I false
```

Where `-I` stands for Isar mode. Passing the string `false` to the `-I` parameter invokes Isabelle in the classic mode. Removing this parameter or passing any other string value will invoke Isabelle in the Isar mode.

5.2.1 Theory Naming Collisions

A number of VDM-LPF theories shared the same name as their corresponding FOL and HOL ones. Consequently, attempting to run the VDM-LPF top theory file resulted in errors (Figure 34).

```
Error: in '/tmp/isabelle-g_karab17550/Seq_thy.ML', line 22.
Value or constructor (setrange_tr) has not been declared
Found near $( $( $( ...), ......), Abs( "x", ...))

Error: in '/tmp/isabelle-g_karab17550/Seq_thy.ML', line 29.
Value or constructor (eta_exp) has not been declared   Found near eta_exp(f)

Error: in '/tmp/isabelle-g_karab17550/Seq_thy.ML', line 30.
Value or constructor (eta_exp) has not been declared   Found near eta_exp(P)

Exception- Fail "Static errors (pass2)" raised
```

Figure 34: Theory Naming Conflict Error

Two competing solutions were available to the theory naming conflict: make `Pure` a stand alone logic in our local Isabelle installation or rename the offending VDM-LPF theory files. The latter solution was used because it made the redistribution of the upgraded VDM-LPF logic easier.

Six files were renamed and modified, namely from `Nat.thy`, `Nat.ML`, `Map.thy`, `Map.ML`, `Set.thy`, and `Set.ML` to `NatLPF.thy`, `NatLPF.ML`, `MapLPF.thy`, `MapLPF.ML`, `SetLPF.thy`, and `SetLPF.ML`. References to these files and the theories they define were changed accordingly.

5.2.2 ML Syntax Errors

Correcting the naming issues did not result in a successful interpretation of the VDM-LPF theories. A new error, shown in Figure 35, was hindering our progress.

```
Error: in '/home/g/g_karab/thesis/tmp/DIST-VDM-LPF/LPF_Prover.ML', line 32.
end expected but infix was found

Exception- Fail "Static errors (pass 1)" raised
*** Fail "Static errors (pass 1)"
*** At command "use".
Exception- ERROR raised
```

Figure 35: Proof Search Tactics ML Error

The problem is due to the `infix` statement in the `LPF_Prover.ML` file (proof search tactics implementation as described in Chapter 4)—see Figure 36. The cause of the error is that `infix` statements cannot occur with a `sig end` ML block [Paulson96, Milner+97].

```
signature LPF_PROVER =
sig
type lpf_set

val empty_lpfs : lpf_set
val addSIs : lpf_set * thm list -> lpf_set
val addSEs : lpf_set * thm list -> lpf_set
val addSDs : lpf_set * thm list -> lpf_set
val addIs : lpf_set * thm list -> lpf_set
val addEs : lpf_set * thm list -> lpf_set
val addDs : lpf_set * thm list -> lpf_set

infix 4 addSIs addSEs addSDs addIs addEs addDs;

val lpf_atac : int -> tactic
val lpf_step_tac : lpf_set -> int -> tactic
val lpf_fast_tac : lpf_set -> int -> tactic
end;
```

Figure 36: Erroneous ML Code Segment

The solution is involves moving the `infix` statement outside (e.g. after) the `signature` declaration.

5.2.3 Syntactic Changes

Nearly all theory files, `*.ML` and `*.thy` alike, had syntax errors. Thus, correcting them was a fairly long and tedious exercise. Fortunately, all errors were of one particular type. Two instances of this error for `Prop.ML` and `Pred.thy` are given in Figure 37 and Figure 38 The error is easily resolved by inserting a space to the right of the . character, thus: `"P. not true ==> P"` and `"forall'(A,%x. e)"`.

```
Exception- ERROR raised
*** Inner syntax error at: "P.not true ==> P"
*** Expected tokens:  "id" "("
*** The error(s) above occurred for "!! P.not true ==> P"
*** At command "use".
Exception- ERROR raised
```

Figure 37: Syntax Error from Prop.ML as Displayed by Isabelle 2005 Classic

```
*** Inner syntax error at: "x.e )"
*** Expected tokens:  "id" "("
*** The error(s) above occurred in translation pattern "forall'(A,%x.e)"
Exception- ERROR raised
Exception- ERROR raised
```

Figure 38: Syntax Error from Pred.thy as Displayed by Isabelle 2005 Classic

5.2.4 Changing the Base Logic to CPure

The last change performed to complete this stage of the upgrading process was the replacement of VDM-LPF's base logic from Pure to CPure. Identifying the new basic logic only involved a change to the `Basic.thy` file which formed the lowest level theory of VDM-LPF. All other theories are based on this one, either directly or indirectly. Only the theory declaration statement had to be changed from `Basic = Pure +` to `Basic = CPure +`.

`CPure` is identical to the `Pure` logic with one exception; the function application syntax differs. In `Pure`, function arguments are enclosed within parenthesis and separated by commas, whereas in `CPure` spaces separate both the function name from its arguments and the different arguments. To illustrate consider the following translation statement as found in the `Pred.thy` file:

```
"forall_(tbindsn_(tb, tbs), e)" == "forall_(tb, forall_(tbs, e))"
```

When Isabelle is asked to parse this statement it will display the error message as seen in Figure 39. To correct this statement the function will have to be expressed in the following `CPure` complaint syntax:

```
forall_(tbindsn_ tb tbs) e" == "forall_ tb (forall_ tbs e)
```

This kind of modification is not limited to translation statements, it also affected a large number of definitions, axioms, and proofs, making it the longest and most tedious of all the changes performed so far. It should be noted that the change to CPure was not necessary for allowing the theories to work in Isabelle/Classic 2005. However this will allow VDM-LPF to work under Isabelle/HOL, which was also based on the CPure logic. Since our research group makes exclusive use of HOL (vs. FOL), this seemed like the most appropriate choice.

```
*** Inner syntax error at: ", tbs ) , e )"
*** Expected tokens:  "true" "false" "undefined" "_" "op" "\<dots>" "PROP"
***   "OFCLASS" "TYPE" "id" "longid" "var" "..." "\<struct>" "(" "\<Colon>"
"=="
***   "\<equiv>" "\<equiv>\<^sup>?" ")" "<=>" "=>" "or" "and" "::" "==>"
***   "\<Longrightarrow>"
*** The error(s) above occurred in translation pattern
"forall_(tbindsn_(tb,tbs),e)"
Exception- ERROR raised
Exception- ERROR raised
```

Figure 39: Function Syntax Error Because of the Pure to CPure Change.

5.3 Final porting step: Isabelle/Isar 2005

Creating fully-fledged Isar theories out of the old style ones was the last set of modifications we performed to VDM-LPF. This involved a large number of changes which can be categorized into four different kinds:

(1) syntactic changes,
(2) changes pertinent to VDM-LPF proof search tactics,
(3) changes involving the translation functions, and
(4) rewriting all the proofs from their ML proof scripting format into the Isar theory format.

5.3.1 Syntactic Changes

Upgrading the VDM-LPF theories to Isabelle/Isar required a large number of syntactic changes. This was due to the fact that Isabelle/Isar syntax is different from Isabelle/Classic. Most modifications involved changes to the syntax of common constructs to both the Isar and Classic mode. While other modifications involved the replacement of Classic constructs with newly introduced Isar ones. In what follows we will have a closer look at these changes.

5.3.1.1 Theory Declaration

The manner in which theories are declared has changed in Isabelle/Isar as compared to Isabelle/Classic. A new keyword, namely `theory`, was introduced to denote the start of a theory. Additionally, a colon character is used to denote the end of the theory declaration statement and the start of the declarations of new types, definitions, lemmas, etc. As an example consider the following old style theory declaration:

```
VDM_LPF = Bool + Case
```

To perform the same in Isar, one will have to rewrite it as follows:

```
theory VDM_LPF = Bool + Case:
```

Furthemore, Isar theories must denote their ending by the `end` keyword. Classic theory files, only required the `end` keyword if they contained definitions.

5.3.1.2 Truth Judgment

The new `judgment` keyword has been introduced to denote truth judgment statements. A truth judgment statement is the very first step in the creation process of a new object logic and it associates Isabelle's meta-logic with the new object logic [Wenzel]. Consequently the following statement from the classic theory

```
consts
 Trueprop      :: ex => prop   ("(_)" 5)
```

will be written in an Isar theory like

```
judgment
 Trueprop      :: "ex => prop" ("(_)" 5)
```

5.3.1.3 Logical and Syntactic Types

Both logical and syntactic type declarations have changed. Logical types are introduced through the `typedecl` statement hence replacing both `types` and `arities` while syntactic types are introduced by the more appropriate `nonterminals` statement. It is worth noting that the `types` keyword still exists in Isabelle/Isar, however, it is been used for declaring type synonyms, similar to the C-language `typedef` statement.

Consequently, the `ex` type declaration seen in its classic form in Figure 16 would be written in Isar as:

```
typedecl ex
```

The syntactic types

```
types tbinds tbind
```

are rewritten in Isar as

```
nonterminals tbinds tbind
```

This last difference took the author some time to figure out because of its subtlety and somewhat obscure reference in the documentation. In fact, it was only realized when the Isabelle/HOL source code was consulted.

5.3.1.4 Axioms

Axiom declarations have been change in Isar by the introduction of the more appropriate `axioms` keyword as opposed to the formerly used `rules` keyword. The syntax of both these statements is the same for the most part, except for a colon character right after the axiom name. Figure 40, shows sample Isar axiom definitions that are equivalent to the classic ones portrayed in Figure 22.

```
axioms
true-intr: "true"
or-elim:   "[| P or Q; P ==> R; Q ==> R |] ==> R"
contr:     "[|P;  not P|] ==> Q"
```

Figure 40: Isar Axioms Definition

5.3.1.5 Consts and Syntax Declarations

The form of statements of `consts` and `syntax` declarations is slightly different in Isar. The difference is that the signature of a new constant or syntax definition must be enclosed within double quotes. The quotes can be avoided only if a simple type is involved. Thus, the following statement:

```
consts
true'          :: ex                    ("true")
```

remains unchanged in an Isar theories, while the following:

```
consts
not'           :: ex => ex              ("(2 not _/)" [250] 250)
```

has to be changed in the following manner:

```
consts
not'           :: "ex => ex"            ("(2 not _/)" [250] 250)
```

Syntax declarations are modified in the same manner.

5.3.1.6 Constant Definitions

Constant definitions introduced by the `defs` statement are also slightly different in Isar. The differences are similar to the ones for axioms. That is all names must be followed by a colon character. As such, the following classic statement

```
defs
false_def  "false == not true"
```

will have an Isar equivalent form like

```
defs
false_def:  "false == not true"
```

5.3.2 Built-in VDM-LPF Proof Search Tactics

As was mentioned earlier, VDM-LPF's proof search tactics where built to replace Isabelle's classical reasoner package that is inappropriate for a three-valued object logic like LPF. Since our initial goal from the start was to upgrade VDM-LPF to the Isabelle/Isar level, a method had to be established to make these custom built-in proof tactics available to Isar theories.

This method involved two components: first, a mechanism to include the ML module, defined in `LPF_Prover.ML`, into the Isar theories, thus making its definitions (rule set facilities and tactics) available; second, a method capable of using these definitions—to be precise, a method to define the same rule sets, as seen in the ML proof scripts, and a method to reuse the `lpf_fast_tac` in the corresponding Isar styled proofs.

5.3.2.1 Including VDM-LPF's Proof Tactics

Figure 41 shows how VDM-LPF's search proof tactics have been included into Isabelle/Isar theories. The `files "LPF_Prover.ML":` clause, creates a dependency between the `Prop` theory and the `LPF_Prover.ML` file. Moreover, it loads this file in the current theory context. Subsequent theories that import `Prop`, will have all the `LPF_Prover.ML` definitions

```
theory Prop = Basic
files "LPF_Prover.ML":
```

Figure 41: Including LPF_Prover.ML

available. Thus there is no need to add the files clause in any of the remaining theory declarations, for the purpose of making the tactics and rule set facilities available to them.

5.3.2.2 Using VDM-LPF's Proof Tactics

Merely including the VDM-LPF search proof tactics is not enough to be able to use them. Additionally, appropriate ML structures must be defined from the functor in `LPF_Prover.ML`. The rule sets have to be defined and a way to apply the search proof tactics in Isar proofs must be found.

First and foremost, the functor found in `LPF_Prover.ML` must be instantiated into appropriate ML structures. At this point it is worth digressing and mentioning that ML functors are parameterized structures and cannot be used as is. Instead they need to be passed structure(s) of the appropriate signature(s). The result is a new structure which can now be used [Paulson96, Milner+97]. This instantiation is identical to the one performed in the ML proof scripts and is depicted in Figure 42. There is one small difference however: when referring to axioms or theorems one has to use the `thm` ML function to retrieve them as opposed to just using their name. The `ML {* *}` construct allows one to include ML code in the current theory context. This statement is used throughout the theory files when ML code needs to be included for one reason or another.

```
ML
{*
  structure LPF_Prover_Data =
  struct
   val contr = thm "contr"
  end;

  structure LPF_Prover = LPF_Prover_Fun(LPF_Prover_Data);

  open LPF_Prover;
*}
```

Figure 42: Instantiating VDM-LPF's Functors

Defining the different rule sets is also achieved by an `ML {* *}` statement. Thus, the exact ML code as found in the ML proof scripts in used in Isar theories. Again the only difference is that the `thm` function is used to obtain the different axioms and theorems. Figure 43, shows how a VDM-LPF rule set can be defined in Isabelle/Isar theories. The same rule set, in its ML proof scripting form, can be seen on Figure 28. In the same manner all rule sets are defined throughout VDM-LPF's Isar theories.

```
ML
{*
   val ruleSet = empty_lpfs addSIs [thm "not_or_intr"] addSEs [thm
"not_or_elim"] addSDs [thm "not_not_dest"];
*}
```

Figure 43: Rule Set Definition in Isabelle/Isar

After discovering a way of creating the different rule sets, we now need an method to apply them in Isar proofs. Fortunately, Isabelle/Isar provides users with such a method, namely the `tactic` method. This method allows us to use ML type tactics in Isar proofs. We used the tactic method only with VDM-LPF's built in proof search tactics. We could have used this method throughout all proving steps. However, in that case, claiming that we have upgraded the ML style proofs into the Isar tactic emulation style would have been false. To illustrate the tactic method consider the following:

```
apply(tactic {* lpf_fast_tac ruleSet 1 *})
```

An alternative to the tactic method was explored, however, it was not pursued further because it was not the correct approach. This alternative involved using the Isabelle/Isar `method_setup` statement. This statement does what its name implies, it allows for the creation of a new method. Figure 44 shows how a new method, namely `lpf_prop`, was created and was used to prove propositional logic statements by writing:

```
apply(lpf_prop)
```

This method was not pursued because of its inflexibility. Ideally, we should only define one tactic that will be accepting rule sets or some form of rule sets that is compliant with Isabelle/Isar theories. However, lack of proper documentation and time constraints prevented us from moving ahead with such a solution.

```
method_setup lpf_prop =
  {*
  Method.no_args (Method.SIMPLE_METHOD (lpf_fast_tac prop_lpfs 1))
  *}
```

Figure 44: method_setup Illustrated

5.3.3 Syntax Translation Functions

With the elimination of the ML proof scripting, Syntax Translation functions have also undergone changes, albeit smaller when compared to the other components. These functions are developed in ML using the exact same data structures. Consequently, changes to the actual functions were minimal. The majority of changes mostly involved the manner these functions are included into Isar theories.

With the introduction of Isar theories a new set of block statements is introduced specifically for translation functions. These block statements and their general format can be seen in Figure 45. The first two are the ones of interest to us since parse and print translation functions are used

```
parse_translation {* ML code *}
print_translation {* ML code *}
parse_ast_translation {* ML code *}
print_ast_translation {* ML code *}
```

Figure 45: Isar Translation Function Statement

extensively in VDM-LPF. It is worth noting that documentation on translation functions is limited. We were able to find examples by searching through the Isabelle/HOL source code.

Figure 46 displays the Isar complaint print translation function for properly displaying the universal quantifiers with multiple bounded variables. For comparison reasons, the old style translation function can be seen in Figure 20. The differences are very small; actually the only difference is that the ML code is enclosed within a `let in end` ML block statement. Parse translation functions are declared in a similar manner.

```
print_translation
(*
let
  fun eta_exp (e as Abs(_,_,_)) = e
    | eta_exp e =Abs("x",dummyT,e$(Bound 0));

  fun quan_tr' r [A,e] =
      let val Abs(x,_,e') = eta_exp e
          val (x',e'') = variant_abs(x,dummyT,e')
      in
          Const(r,dummyT)$(Const("tbind_",dummyT)$Free(x',dummyT)$A)$e''
      end;
in
  [("forall'",quan_tr' "forall_"),("exists'",quan_tr' "exists_"),
   ("exists1'",quan_tr' "exists1_"), ("iota'",quan_tr' "iota_")]
end;
*)
```

Figure 46: Isar Declaration of Print Translation Functions

5.3.4 Isar-style Proofs

The largest set of changes involved re-writing the ML proofs in the Isar proof language. Two types of Isar proofs are available, the Isar tactic-emulation proofs and the Isar human-readable proof texts. The former was used since it closely resembled the ML proofs that needed to be re-written, it was the natural next task to perform, ample set of examples existed, and an excellent guide was available. Note that rewriting the proofs in Isar human-readable proof texts was considered but was deferred because of the scope and difficulty of this task.

The task of rewriting the ML proofs into Isar tactic-emulation proofs was a matter of finding the appropriate and equivalent Isar constructs. This was facilitated by the existence of an excellent conversion guide [Wenzel] which was followed closely during this set of changes. Rewriting the proofs in Isar emulation tactics was a two-fold process. First, expressing the goal statements in the Isar language. Second, finding the corresponding Isar methods for each of the tactics used in the ML proofs.

5.3.4.1 Goal Statements

There are four types of `goal` and `goalw` statements in the VDM-LPF ML proofs scripts that initiate a proof. Moreover, there are instances where the original authors create their own goal extension statements to facilitate their work. The corresponding Isar statement is `lemma` or its synonym `theorem`. The `lemma` statement is used throughout this rewriting exercise. The left column of Figure 47 shows the general format of the `goal` and `goalw` statements as used in the ML proofs, as well as the proof ending `qed` statements. The refers to the different ML tactics performed to satisfy the proof. On the right column of the same table are the corresponding and equivalent `lemma` statements, as well as the proof ending `done` statement. In this table the Greek letters ϕ and ψ denote formulas, *defl* and *defn* denote definitions, finally *pl* and *pn* denote premises.

goal "ϕ"; Qed "name"	lemma name: "ϕ": done
goalw [*defl*, ,*defn*] "ϕ"; qed "name"	lemma name: "ϕ": apply(unfold *defl* *defn*) done
val [*pl*, ,*pn*]=goal "[|ϕ*l*; ...;ϕ*n* |] ==> ψ"; qed "name"	lemma name: assumes *pl*: "ϕ*l*" and and *pn*: "ϕ*n*" shows "ψ" done
val [*pl*, ,*pn*] = goalw [*defl*, , *defn*] "ϕ"; qed "name"	lemma name: assumes *pl*: "ϕ*l*" and and *pn*: "ϕ*n*" shows "ψ" apply(unfold *defl* *defn*) done

Figure 47: Equivalent goal and lemma Statements

The first form involves rewriting a `goal` statement. This is the simplest form of all proof statements, as such it is very simple to rewrite in Isar. Consider the following ML goal statement with its corresponding proof ending `qed`:

```
goal Prop.thy "!!P. not true ==> P";
...
qed "not_true_elim"
```

this is easily rewritten in the following manner in Isar:

```
lemma not_true_elim : "!!P. not true ==> P"
...
done
```

The differences are minute and there are none in the manner the formula is expressed. Note that the syntax of formulae was left unchanged throughout our upgrading effort.

The second form of proof statements involves a `goalw` statement. Such statements allow for the introduction of more complicated proof statements and for the automatic expansion of definitions. Definitions are passed to the statement in the form of an ML list. All elements must be valid definitions; that is, defined through a pair of `consts` and `defs` statements. Instances of such definitions within the formula are automatically expanded. The new expanded formula is displayed, and all tactics are performed in this expanded formula. As an example consider the following statement:

```
goalw Prop.thy [false_def] "!!P. false ==> P";
...
qed "false_elim"
```

what this does is to replace occurrences, in the formula, of the left hand side of the `false_def` definition with its right side. Considering that `false_def` is defined as `"false == not true"`, the formula will be change to `"!! P. not true ==> P"`. To achieve the same effect in Isar the `goalw` statement must be rewritten in the following manner:

```
lemma false_elim : "!!P. false ==> P"
 apply(unfold false_def)
...
done
```

The formula remains unchanged and the proof is initiated by the `lemma` statement. The expansion of the definition is performed by the `unfold` tactic.

The third form involves a `goal` statement. This time we store the premises or assumptions into ML variables. This is useful when we want to use these premises in the proving process; that is apply them as we apply any other axiom or derived rule. The following example illustrates this technique:

```
val prems = goal Prop.thy "[| not not P; P ==> R |] ==> R";
...
qed "not_not_elim"
```

The two premises will be stored in the `prems` ML list variable, the first at the head of the list while the other as the second element of that list. To rewrite this in Isar a more verbose version of the `lemma` statement is used:

```
lemma not_not_elim :
 assumes p1 : "not not P"
     and p2 : "P ==> R"
 shows "R"
...
done
```

The `assumes` and the `and` keywords store the premises in the `p1` and `p2`, local for this proof, rules while the `shows` keyword contains the formula to be proved, i.e. the conclusion.

The last form is a combination of the previous two `goalw` and `goal` statements. Statements, where both definitions are expanded and premises are stored in ML variables to be used as rules. Consider the following statement:

```
val [p1,p2,p3]= goalw Prop.thy [imp_def] "[| P => Q; not P==>R; Q==>R |] ==> R";
...
qed "imp_elim";
```

where three premises are stored in ML variables and the `imp_def` definition is expanded. The following set of Isar statements achieve the same:

```
lemma imp_elim :
 assumes p1 : "P => Q"
     and p2 : "not P==>R"
     and p3 : "Q==>R"
 shows "R"
 apply(cut_tac p1)
 apply(unfold imp_def)
...
done
```

You will notice that an extra method is applied before the unfolding of the `imp_def` definition. This extra method is adding `p1` as a premise in the target formula we are attempting to prove. Thus, right after `cut_tac` is applied Isabelle will display `"P => Q ==> R"` as the goal formula. This is necessary because unfolding the definition only works on the goal formula and not on any of the local rules formed by the `assumes` and the `and` keywords.

In one of the ML theory files, namely `Pred.ML`, the authors created their own versions of `goal` and `goalw` statements using ML functions. The definitions of these new statements can be seen on Figure 48. These functions combine the `goal` and `goalw` statements with the `cut_facts_tac` tactic. This is done just to save the authors from explicitly using this tactic at every proof. In Isar we treat the use of `goal1` and `goalw1` as if they were expanded into their corresponding definition. Thus a `lemma` statement with an application of the `cut_tac` method is used to replace such goal statements.

```
fun goal1 t s =
    let val prems = goal t s in (by (cut_facts_tac prems 1);prems) end;

fun goalw1 t tl s =
    let val prems = goalw t tl s in (by (cut_facts_tac prems 1);prems) end;
```

Figure 48: goal1 and goalw1 Definitions

5.3.4.2 Tactics

Rewriting the ML styled proof tactics into Isar methods involved two type of changes. First, finding appropriate Isar methods for each ML tactic used. Second, finding appropriate Isar constructs for rewriting ML tacticals. Tacticals are operations on tactics. They create new tactics either on a permanent or on a temporary basis.

Isabelle/Classic provides a large number of tactics and a large number of short form synonyms of these tactics. Fortunately, Isar has consolidated all these different tactics and synonyms into a single set of methods. Figure 49 displays the ML proof tactics used in the original VDM-LPF and their corresponding Isar methods used to replace these tactics. In this table `thm`, `thm1`, and `thmn`, refer to theorems. The first row displays the `cut_facts_tac` and its corresponding Isar `cut_tacs` method that was explained in the previous section.

cut_facts_tac [thm1, ,thmn] 1	cut_tac thm1 thmn
ba 1	Assumption
atac 1	Assumption
br *thm* 1	rule *thm*
rtac *thm* 1	rule *thm*
be *thm* 1	erule *thm*
bd *thm* 1	drule *thm*

Figure 49: ML Proof Tactics and their Equivalent Isar Methods

The next two rows portray the `ba` and the `atac` tactics. Both are shortcuts for the `assume_tac` tactic. To illustrate consider the following equivalent applications of this tactic:

```
ba 1;
atac 1;
by (assume_tac 1);
```

All these statements are equivalent and they will be applied to the first subgoal in the sequence – hence the `1` in the right hand side. The same can be achieved in Isar via the `assumption` method:

```
apply(assumption)
```

There is no need to specify the subgoal in the Isar assumption method because all methods operate by default on the first one in the sequence. Actually, most the of the Isar methods do not allow the specification of the subgoal. Isar, however, provides two helper methods, namely `prefer` *i* and `defer`, that allow the user to change the subgoal sequence.

The next two rows also involve shortcuts of the same tactic, namely `resolve_tac`. This the standard resolution tactic used with introduction rules. Let us consider again an example of a set of equivalent application of the `resolve_tac` and its synonyms:

```
br thm 1;
rtac thm 1;
by (resolve_tac [thm] 1);
```

The same can be done in Isar through the `rule` method:

```
apply(rule thm)
```

Like the `assumption`, the `rule` method operates on the first subgoal in the sequence. Alternatively, Isar provides the `rule_tac` method, which allows for selecting a specific goal. However, its usage is discouraged [Wenzel]. Thus, we refrained from using them in this project.

The ML tactic `be` is also a shortcut of the `eresolve_tac` tactic. This is also a resolution tactic, however, it is used with elimination rules, hence the prefix `e`. The following equivalent ML proof tactic statements:

```
be thm 1;
by (eresolve_tac [thm] 1
```

are rewritten using the `erule` Isar method:

```
apply(erule thm)
```

Similarly, `bd` is a shortcut for the `dresolve_tac` ML tactic. This tactic is used with destruction rules and it is also a resolution tactic. Let us again consider an example:

```
bd thm 1;
by (dresolve_tac [thm] 1);
```

By now the reader is aware of the pattern, thus the Isar method is none other than the `drule` one. Consequently, the above two equivalent tactic applications can be performed in Isar as:

```
apply(drule thm)
```

Both `erule` and `drule` are similar to the `rule` method. As such, they both cannot specifically operate on a different subgoal than the first one and they both have alternatives. These alternatives are the `erule_tac` and `derule_tac`, where the subgoal can be specified. As before, their usage is discouraged [Wenzel].

There are cases in the old style VDM-LPF theories where proof tacticals are used instead of tactics. Tacticals are used to perform operations on tactics. This includes combining tactics, repeating a tactic, and applying a tactic to all the subgoals. These tacticals and their corresponding Isar constructs can be seen on Figure 50.

tac1 THEN … THEN *tacn*	*meth1, …, methn*
EVERY [*tac1, …, tacn*]	*meth1, …, methn*
REPEAT *tac*	*(meth)+*
ALLGOALS *tac*	*(meth)+*
thm1 RSN (i, *thm2*)	*Thm1* [THEN [i] *thm2*]
thm1 RS *thm2*	*Thm1* [THEN *thm2*]

Figure 50: ML Proof Tacticals and their Equivalent Isar Constructs

The `THEN` and `EVERY` tacticals are closely related since the latter is an abbreviation of the former. Consider the following two equivalent application of a tactic formed through these two tacticals:

```
by ((rtac eq_subs_right 1) THEN (atac 1) THEN (atac 1) THEN (atac 1));
By (EVERY [rtac eq_subs_right 1, atac 1, atac 1, atac 1]);
```

They both combine a set of tactics. The `THEN` statement combines two tactics at a time by applying the first to the current proof state then applying the second to the updated proof state.

EVERY operates in the same way. The only difference is that there is no need for nested THEN tacticals.

Both of these tacticals have the same translation in Isar tactic emulation proofs. However, in this case we combine methods and not tactics. This is achieved by separating individual methods by the comma character. Thus the following is the equivalent Isar statement:

```
apply(rule eq_subs_right,assumption,assumption,assumption)
```

The next tactical, namely REPEAT, involves repeating a tactic until it can no longer be applied; that is it can no longer progress the proof any further, either because it was satisfied or it cannot be applied to the current proof state. To illustrate consider the following example:

```
by (REPEAT (atac 1));
```

The same can be performed in Isar by prefixing the plus character, "+", to a method application. Thus the equivalent Isar statement is:

```
apply(assumption)+
```

A closely related tactical to THEN is the ALLGOALS. It is used to apply an ML tactic function with signature `int -> tactic` to all subgoals. The `tactic` type is an Isabelle built in type used to denote tactics. Assuming that the current proof context contains `n` subgoals, ALLGOALS *tacf* abbreviates to *tacf(n)* THEN ... THEN *tacf(1)*[Wenzel]. This difference in ordering is important since there is no Isar construct that can duplicate this behavior. For most instances of ALLGOAL tacticals the + method application prefix can be used to apply the method in all subgoals. However, such an application is not in the right order – often this order is important. During the upgrading process we encountered one such instance where the + prefix couldn't be used. Figure 51 shows the old style proof, while Figure 52 and Figure 53 shows two different Isar solutions. The latter was used for its simplicity, however it is worth noting how the first solution was implemented since it clearly illustrates how the ALLGOALS tactical operates.

```
Val [p] =
goalw1 Pred.thy [forall_def] "(!! y. y:A ==> def P(y)) ==> def forall'(A,P)";
br def_exists 1;
by (ALLGOALS (fn i => lpf_fast_tac (exists_lpfs addDs [p]) i));
qed "def_forall";
```

Figure 51: ALLGOALS Old-Style Proof

```
lemma def_forall:
 assumes p1: "(!! y. y:A ==> def P(y))"
   shows "def (forall' A P)"
 apply(unfold forall_def)
 apply(rule def_exists_inh [THEN def_elim])
   apply(tactic {* lpf_fast_tac (exists_lpfs addDs [thm "p1"]) 3 *})
  apply(tactic {* lpf_fast_tac (exists_lpfs addDs [thm "p1"]) 2 *})
 apply(tactic {* lpf_fast_tac (exists_lpfs addDs [thm "p1"]) 1 *})
done
```

Figure 52: ALLGOALS Isar Translated Proof - Take 1

```
lemma def_forall:
 assumes p1: "(!! y. y:A ==> def P(y))"
   shows "def (forall' A P)"
 apply(unfold forall_def)
 apply(rule def_exists_inh [THEN def_elim])
   apply(tactic {* ALLGOALS (fn i => lpf_fast_tac (exists_lpfs addDs [thm
"p1"]) i) *})
done
```

Figure 53: ALLGOALS Isar Translated Proof - Take 2

Finally two closely related operations on theorems are presented, RSN and RS. These are not tacticals but are presented here because they operate in a similar manner. RSN and RS allow for the modification of existing rules and their subsequent application to a subgoal. The only difference between them is that the latter always uses the first premise of the second rule to modify the first while in the former the premise can be specified. To illustrate consider the following equivalent ML proof RSN and RS statement:

```
br ((hd prems) RS prodn_elim) 1;
br ((hd prems) RSN (1, prodn_elim) 1;
```

In Isar we can apply methods with the same effect using the THEN method attribute. The above statements can be performed in Isar in the following manner:

```
apply(rule p1 [THEN prodn_elim])
apply(rule p1 [THEN [1] prodn_elim)
```

The second statement shows how the THEN method attribute replaces both RS and RSN rule operations.

6 VDM-LPF Case Study

In this chapter we present a number of proofs we performed in order to test, illustrate, and experiment with the newly upgraded embedding of VDM-LPF in Isabelle/Isar 2005. As such, understanding this chapter will require an understanding of the material presented on Chapters 3, 4, and 5.

This case study is comprised of three different sets of proofs: (1) propositional logic proofs, (2) predicate logic proofs, and (3) other proofs. In the first and second set, as their name suggest, we perform proofs pertinent to propositional and predicate logic. These proofs involve elementary theorems as found in an introductory discrete mathematics text book [Rosen91] as well as a number of proofs pertinent to definedness. The remaining set involves a proof that tests the remaining components of the VDM-LPF logic, such as conditionals, the types of Boolean and natural numbers. Some of these proofs are identical to the ones we performed during our sanity tests of the original VDM-LPF logic, as seen in Section 5.1.

6.1 Propositional Logic

A number of propositional logic proofs are performed. These theorems are obtained by consulting [Rosen91]. The aim of this exercise is to become familiar with the upgraded underlying LPF system of VDM-LPF. Most of the theorems are proved successfully by a single application of a VDM-LPF rule, which points to the fact that many of these fundamental laws are already satisfied in this non-classical logic, while other theorems require the introduction of the definedness operator δ. Finally, we perform a small number of proofs involving undefinedness. What follows is a detailed examination of these proofs.

6.1.1 Identity Laws

The first set of proofs involves the Identity laws and can be seen on Figure 54. The first one

```
lemma "p and true ==> p"
```

states that given a conjunction `p and true` we can infer that `p` is true. This is easily proven by applying the `and_elim` derived rule (this rule, as well as all the rules used in this set of proofs, can be seen in Appendix D, under the contents of `Prop.thy`). This is an elimination rule and it will operate on the assumption by changing the goal to:

```
goal (lemma, 1 subgoal):
 1. [| p; true |] ==> p
```

This proof is easily completed by assumption.

The second identity law is similar to the first one, except that this time we are dealing with a disjunction in the hypothesis:

```
lemma "p or false ==> p"
```

This lemma states that if `p or false` is true then we can conclude that `p` is true. The first step to prove this lemma involves another elimination rule, namely `or_elim`. This rule states that we need to prove the conclusion, `R`, twice – once for each term of the disjunction. Thus, we are presented by two subgoals:

```
goal (lemma, 2 subgoals):
 1. p ==> p
 2. false ==> p
```

```
text {* Identity Conjuction *}
lemma "p and true ==> p"
  apply(erule and_elim)
  apply(assumption)
done

text {* Identity Disjunction *}
lemma "p or false ==> p"
 apply(erule or_elim)
  apply(assumption)
 apply(erule false_elim)
done
```

Figure 54: Identity Law Proofs

The first one is satisfied by assumption while for the second is satisfied by another elimination rule, namely `false_elim` – thus, completing this proof.

6.1.2 Domination Laws

The next two proofs involve the domination law equivalencies which are very similar to the identity laws. The proofs are depicted in Figure 55. The first one involves a disjunction in its hypothesis:

```
lemma "p or true ==> true"
```

which specifies that if a disjunction exists where one of its propositions is the `true` value then this is logically equivalent to `true`. We commence by applying the `or_elim` rule, which results into two subgoals:

```
goal (lemma, 2 subgoals):
 1. p ==> true
 2. true ==> true
```

The first subgoal is satisfied by applying the `true_intr` rule. For the second subgoal we have a choice, we can either use the `true_intr` rule for a second time or satisfy it by assumption. We choose the later for no particular reason other than to add variety to our proof. It is also worth noting that this proof can be completed by just a single application of the `true_intr` rule. We choose the longer route for illustration purposes.

```
text {* Domination Disjunction *}
lemma "p or true ==> true"
 apply(erule or_elim)
  apply(rule true_intr)
 apply(assumption)
done

text {* Domination Conjunction *}
lemma "p and false ==> false"
 apply(erule and_elim)
 apply(assumption)
done
```

Figure 55: Domination Law Proofs

The second domination rule, as expected, involves a conjunction in the hypothesis:

```
lemma "p and false ==> false"
```

First, we apply the `and_elim` rule which results in the following subgoal:

```
goal (lemma, 1 subgoal):
 1. [| p; false |] ==> false
```

which is satisfied by assumption.

6.1.3 Idempotent Laws

Idempotent equivalence laws deal with disjunctions and conjunctions involving the same terms. Their proofs are captured in Figure 56. The first one of these laws:

```
lemma "p or p ==> p"
```

deals with a disjunction of the same term which is logically equivalent to that same term. Since this law involves a disjunction in the hypothesis, the `or_elim` rule is the first one we use. This results into two subgoals:

```
goal (lemma, 2 subgoals):
 1. p ==> p
 2. p ==> p
```

Both of these subgoals are easily satisfied by assumption.

```
Text {* Idempotent Disjunction *}
lemma "p or p ==> p"
 apply(erule or_elim)
  apply(assumption)+
done

text {* Idempotent Conjunction *}
lemma "p and p ==> p"
 apply(erule and_elim,assumption)
done
```

Figure 56: Idempotent Law Proofs

Next we prove the idempotent law involving conjunction:

```
lemma "p and p ==> p"
```

This lemma is easily proven by an application of the `and_elim` rule and by assumption. Both these applications take place within the same `apply` statement:

```
apply(erule and_elim,assumption)
```

The decision to combine the application of the `and_elim` and `assumption` methods is done because we have seen the effect these rules have and we knew what the resulting subgoal will look like.

6.1.4 Double Negation

Next we prove the double negation law as seen on Figure 57. This law states that the truth value of a doubly negated term is the same as the value of that term:

```
text {* Double Negation *}
lemma "not not p ==> p"
 apply(erule not_not_dest)
done
```

Figure 57: Double Negation Proof

```
lemma "not not p ==> p"
```

This is easily resolved since there exists an axiom, called `not_not_dest` which is identical to the lemma we want to satisfy. Consequently, by just applying this elimination rule the lemma is satisfied and the proof completed.

6.1.5 Commutative Laws

Commutative laws state that both conjunction and disjunction commute – we can switch their terms without changing their meaning. The proofs can be seen on Figure 58. The first proof deals with the commutative law for disjunction:

```
lemma "p or q ==> q or p"
```

```
text {* Commutative Disjunction *}
lemma "p or q ==> q or p"
 apply(erule or_comm)
done

text {* Commutative Conjunction *}
lemma "p and q ==> q and p"
 apply(erule and_comm)
done
```

Figure 58: Commutative Law Proofs

In the propositional part of VDM-LPF such a property for disjunctions has already been proven in the form of the derived rule `or_comm`. Consequently, the proof is completed by applying this rule, in the following manner:

```
apply(erule or_comm)
```

The conjunction commutative law is expressed in the following manner in Isabelle:

```
lemma "p and q ==> q and p"
```

Like disjunction, there exists a derived rule in the propositional part of VDM-LPF, called `and_comm`. Thus this lemma is satisfied by a single application of the and_comm rule:

```
apply(erule and_comm)
```

6.1.6 Associative Laws

The associative laws are similarly proven, depicted in Figure 59, as the commutative ones. There already exists derived rules in VDM-LPF that have established the associativity of both disjunction and conjunction. Consequently, proving the disjunction associative law

```
text {* Associative Disjunction *}
lemma "(p or q) or r ==> p or (q or r)"
 apply(erule or_assoc_left)
done

text {* Associative Conjunction *}
lemma "(p and q) and r ==> p and (q and r)"
 apply(erule and_assoc_left)
done
```

Figure 59: Associative Law Proofs

```
lemma "(p or q) or r ==> p or (q or r)"
```

under the VDM-LPF logic is done by an application of the `or_assoc_left` rule:

```
apply(erule or_assoc_left)
```

The proof for the conjunction associative law:

```
lemma "(p and q) and r ==> p and (q and r)"
```

is performed similarly, by applying the `and_assoc_left` derived rule:

```
apply(erule and_assoc_left)
```

6.1.7 Distributive Laws

Distributive laws involve both disjunction and conjunction. Similarly to the previous two sections, the distributive properties of the both conjunction and disjunction as defined in VDM-LPF have been established. Consequently, the proofs are easily disposed of by a single application of a rule. The complete proofs are illustrated in Figure 60.

```
text {* Distributive Disjunction *}
lemma "p or (q and r) ==> (p or q) and (p or r)"
 apply(erule or_and_dist_exp)
done

text {* Distributive Conjunction *}
lemma "p and (q or r) ==> (p and q) or (p and r)"
 apply(erule and_or_dist_exp)
done
```

Figure 60: Distributive Law Proofs

6.1.8 De Morgan's Laws

Next we deal with De Morgan's Laws. As in previous sections these theorems are already satisfied in VDM-LPF. Thus, a single rule application is all that is necessary to prove them. The proofs can be seen in Figure 61. The first proof is completed by applying the `not_and_elim_dem` rule while the second by applying the `not_or_elim_dem` rule.

```
text {* De Morgan's *}
lemma "not (p and q) ==> not p or not q"
 apply(erule not_and_elim_dem)
done

text {* De Morgan's *}
lemma "not (p or q) ==> not p and not q"
 apply(erule not_or_elim_dem)
done
```

Figure 61: De Morgan's Laws Proofs

6.1.9 Absorption Laws

In this section we perform proofs for the Absorption Laws. In a welcome change these laws are not satisfied in VDM-LPF. Consequently, a proof that requires more than one rule application is required. The proofs are illustrated in Figure 62. The first law is expressed in Isabelle in the following manner:

```
text {* Absorption Disjunction *}
lemma "p or (p and q) ==> p"
 apply(erule or_elim)
  apply(assumption)
 apply(erule and_elim)
 apply(assumption)
done

text {* Absorption Conjunction *}
lemma "p and (p or q) ==> p"
 apply(erule and_elim,assumption)
done
```

Figure 62: Absorption Laws Proofs

```
lemma "p or (p and q) ==> p"
```

We can only proceed by working from the hypothesis. The first rule to apply is an or elimination rule, namely the `or_elim` rule. Its application results into two subgoals:

```
goal (lemma, 2 subgoals):
 1. p ==> p
 2. p and q ==> p
```

The first one is satisfied by assumption. The second subgoal requires further simplification. The only way to proceed is by working on the hypothesis. We apply a conjunction rule, namely `and_elim` which results in the following subgoal:

```
goal (lemma, 1 subgoal):
              1.  [| p; q |] ==> p
```

which is easily disposed of by assumption.

The second absorption law involves conjunction as the outer most logical connective:

```
lemma "p and (p or q) ==> p"
```

As such, we only need to apply the `and_elim` rule to simplify its hypothesis and then complete the proof by assumption.

6.1.10 Other Logical Equivalences

In this section additional logical equivalences are examined. Their Isabelle representation along with their proofs are captured on Figure 63.

The first one states that the disjunction of a term with its negation is logically equivalent to true. It is expressed, as a lemma, in Isabelle in the following manner:

```
lemma "p or not p ==> true"
```

This is easily satisfied by applying the `true_intr` axiom.

The second logical equivalence states that the conjunction of a term with its negation is logically equivalent to false:

```
lemma "p and not p ==> false"
```

This one is slightly more complicated to satisfy. We start by applying the `and_elim` rule, since the outer most logical connective is a conjunction. This results in the following subgoal:

```
lemma "p or not p ==> true"
 apply(rule true_intr)
done

lemma "p and not p ==> false"
 apply(erule and_elim)
 apply(erule contr)
 apply(assumption)
done

lemma "p => q ==> not p or q"
 apply(unfold imp_def)
 apply(assumption)
done
```

Figure 63: Logical Equivalences Proofs

```
goal (lemma, 1 subgoal):
 1.  [| p;  not p |] ==> false
```

By examining the assumptions we see that this a contradiction, because it is not possible to have the same term and its negation hold at the same time. Consequently, the contr rule is applied which results in the following subgoal:

```
goal (lemma, 1 subgoal):
 1.  p ==> p
```

which is easily disposed by assumption.

Finally, the last logical equivalence in this section deals with the logical connective of implication:

```
lemma "p => q ==> not p or q"
```

It states that implication can be denoted in terms of negation and disjunction. In VDM-LPF, as seen on Chapter 4, implication is defined in exactly such a way. Thus, satisfying this lemma, simply involves unfolding the `imp_def` definition. This unfolding results in the following subgoal:

```
goal (lemma, 1 subgoal):
 1.  not p or q ==>  not p or q
```

which is satisfied by assumption.

6.1.11 Law of the Excluded Middle

While working through the proofs of the previous section, most notably the `p or not p ==> true` logical equivalence, a decision was made to try to prove the reversed statement, namely `true ==> p or not p`. Considering a single true hypothesis is redundant, the formula can be change to `p or not p` without any loss of information. Notice, that this statement is none other than our familiar law of the excluded middle. As we have seen earlier this is not provable in a non-classical logic such as LPF. To be able to perform this proof an extra assumption is added, namely δp. The complete proof can be seen on Figure 64. The addition of the δp assumption changes the lemma in the following way:

```
lemma "def p ==> p or not p"
 apply(erule def_elim)
  apply(rule or_intr_left)
  apply(assumption)
 apply(rule or_intr_right)
 apply(assumption)
done
```

Figure 64: Law of the Excluded Middle Proof

```
lemma "def p ==> p or not p"
```

The first rule we apply is the `def_elim` rule which changes the goal to the following subgoals:

```
goal (lemma, 2 subgoals):
 1. p ==> p or  not p
 2.  not p ==> p or  not p
```

The first subgoal is satisfied by applying the `or_intr_left` rule and then by assumption. The second subgoal is satisfied by applying the `or_intr_right` rule, followed by assumption.

6.1.12 Definedness Derived Rules

A number of proofs were performed involving the definedness operator. These proofs can be seen on Figure 65. All proofs were obtained from [Bicarregui+94]. The first two proofs derive the two introduction rules for definedness, similar to the ones for disjunction, while the third is the definedness elimination rule.

The first one says that if P is true then P is defined and is expressed as:

```
lemma def_intr_left: !!P. P ==> def P
```

The first step is to unfold the definedness definition which results in the following proof state:

```
goal (lemma (def_intr_left), 1 subgoal):
 1. !!P. P ==> P or  not P
```

The conclusion of the subgoal is further simplified by applying the `or_intr_left` rule, which results to the following proof state which is then completed by assumption:

```
goal (lemma (def_intr_left), 1 subgoal):
 1. !!P. P ==> P
```

The second proof follows the same structure as the previous one. The only difference is that the `or_intr_right` is applied in place of the `or_intr_left`.

Lets examine the third proof which is the more interesting of the set. This proof derives the definedness elimination rule. It says that if P is defined, and P infers Q, and the negation of P also infers Q, then Q can be infered. The lemma is introduced using the `assumes and shows` lemma formulae definition because the premises are used as rules to resolve this rule. The initial proof state can be seen below:

```
prems:
  def P
  P ==> Q
  not P ==> Q
goal (lemma def_elim, 1 subgoal):
 1. Q
```

The first statement is to re-introduce the `def P` premise in the subgoal by an application of the `cut_tac` method which results in the following proof state:

```
prems:
  def P
  P ==> Q
  not P ==> Q
goal (lemma def_elim, 1 subgoal):
 1. def P ==> Q
```

Next, the definedness definition is unfolded and the new proof state is:

```
prems:
  def P
  P ==> Q
   not P ==> Q
goal (lemma def_elim, 1 subgoal):
 1. P or  not P ==> Q
```

Having unfolded the definedness definition the assumption is now in terms of disjunction. As such, the obvious next step is to apply the `or_elim` rule, which breaks the subgoal into two simpler ones:

```
prems:
  def P
  P ==> Q
  not P ==> Q
goal (lemma def_elim, 2 subgoals):
 1. P ==> Q
 2.  not P ==> Q
```

These two subgoals are identical to the `p2` and `p3` premises respectively. As such they are resolved by applying each one of these premises to the corresponding subgoal which are then resolved by assumption.

```
lemma def_intr_left : "!!P. P ==> def P"
 apply(unfold def_def)
 apply(rule or_intr_left)
 apply(assumption)
done

lemma def_intr_right : "!!P. not P ==> def P"
 apply(unfold def_def)
 apply(rule or_intr_right)
 apply(assumption)
done

lemma def_elim :
 assumes p1 : "def P"
     and p2 : "P==>Q"
     and p3 : "not P==>Q"
 shows "Q"
 apply(cut_tac p1)
 apply(unfold def_def)
 apply(erule or_elim)
  apply(rule p2)
  apply(assumption)
 apply(rule p3)
 apply(assumption)
done
```

Figure 65: Definedness Derived Rules

6.1.13 Explicit Undefinedness

In this last section on propositional logic proofs, we experiment with the VDM-LPF's `undefined` value. The complete results of these experiments are captured in Figure 66.

We commence by attempting to prove that the disjunction of the `undefined` and `true` values is valid:

```
lemma "undefined or true"
```

This lemma we know is valid and we should be able to prove it because of the monotonic nature of the LPF connectives. The proof is satisfied by applying the `or_intr_right` and `true_intr` axioms.

```
Lemma "undefined or true"
 apply(rule or_intr_right)
 apply(rule true_intr)
done

lemma "undefined and true"
 apply(rule and_intr)
 defer
 apply(rule true_intr)
oops

lemma "undefined and true ==> undefined"
 apply(erule and_elim)
 apply(assumption)
done

lemma "undefined and true ==> true"
 apply(erule and_elim)
 apply(assumption)
done

lemma "undefined or true = true"
oops
```

Figure 66: Explicit Undefinedness Proofs

Next we attempt to prove that the conjunction between `undefined` and `true` values holds.

```
lemma "undefined and true"
```

We are aware that the above lemma is false. Let us see if we can reach the same conclusion in Isabelle using the axioms and rules of VDM-LPF. We first apply the `and_intr` rule and Isabelle displays the following list of subgoals:

```
goal (lemma, 2 subgoals):
 1. undefined
 2. true
```

The second subgoal is satisfied by applying the `true_intr` axiom. However, there exists no rule that will satisfy the first subgoal. As such, we proceed by using the Isabelle `oops` command to acknowledge that this proof does not hold.

The next two lemmas we attempt to prove are based on the assumption that the `undefined and true` conjunction is valid:

```
lemma "undefined and true ==> undefined"
lemma "undefined and true ==> true"
```

Both these lemmas can be proven using the axioms and rules of VDM-LPF, however, only the first statement makes sense. So how is it possible to prove the second statement as well? It is provable because we make use of a wrong assumption.

Finally, we attempt to prove the following lemma:

```
lemma "undefined or true = true"
```

We quickly discover that we are at an impasse. We are not able to find an equality axiom nor a derived rule where it will allow us to proceed from this point onwards. If we want to reason with statements involving equality and undefinedness, further axioms and derived rules have needed to be defined. However, this is beyond the scope of our intended work.

6.2 Predicate Logic

Similarly to the propositional logic proofs, the predicate logic theorems are also obtained from an introductory discrete mathematics textbook [Rosen91]. Furthermore, a final proof (obtained from [Bicarregui+94]) is performed that involves definedness and the universal quantifier.

The first proof is portrayed in Figure 67 and essentially states that the negation of a universal quantifier predicate can be written in terms of an existential one:

```
lemma "not (forall x:A & P(x)) ==> exists x:A & not P(x)"
```

To satisfy this lemma we first unfold the `forall_def` definition. This changes the lemma into the following subgoal:

```
goal (lemma, 1 subgoal):
 1.  not  not (exists x : A &  not P x) ==> exists x : A &  not P x
```

The exists statement in the hypothesis is identical to the one in the conclusion. However, we need to work with the outermost elements, which in this case are the double application of the negation logical connective. Consequently, we apply the `not_not_elim` rule which changes the subgoal to:

```
goal (lemma, 1 subgoal):
 1. exists x : A &  not P x ==> exists x : A &  not P x
```

The proof is completed by assumption.

```
lemma "not (forall x:A & P(x)) ==> exists x:A & not P(x)"
 apply(unfold forall_def)
 apply(erule not_not_elim)
 apply(assumption)
Done
```

Figure 67: First Predicate Logic Proof

Figure 68 shows the second predicate logic proof. This is the mirror image of the previous proof in which it claims that the negation of a predicate involving an existential quantifier can be rewritten in terms of a universal quantifier.

```
lemma "not (exists x:A & P(x)) ==> forall x:A & not P(x)"
```

Again we start by expanding the `forall_def` definition:

```
goal (lemma, 1 subgoal):
 1.  not (exists x : A & P x) ==>  not (exists x : A &  not  not P x)
```

Next we apply the `not_exists_intr` rule which replaces the existential quantifier by its predicate. This rule states that to satisfy an existential predicate, it is sufficient to show that a value exists that satisfies that predicate. The resulting subgoal as displayed by Isabelle can be seen below:

```
goal (lemma, 1 subgoal):
 1. !!x. [|  not (exists x : A & P x); x : A |] ==>  not  not  not P x
```

Next we apply the `not_not_intr` rule in order to simplify the conclusion by eliminating the redundant negations:

```
goal (lemma, 1 subgoal):
 1. !!x. [|  not (exists x : A & P x); x : A |] ==>  not P x
```

At this stage we turn our attention to the hypothesis, since the conclusion is simplified. We apply the `not_exists_dest` rule, which is similar to the `not_exists_intr`, but operating on the assumptions of the subgoal. The resulting subgoal:

```
goal (lemma, 1 subgoal):
 1. !!x. x : A ==> x : A
```

is easily satisfied by assumption.

```
lemma "not (exists x:A & P(x)) ==> forall x:A & not P(x)"
 apply(unfold forall_def)
 apply(rule not_exists_intr)
 apply(rule not_not_intr)
 apply(erule not_exists_dest)
 apply(assumption)
Done
```

Figure 68: Second Predicate Logic Proof

The next proof we perform is identical to the one we performed during our sanity tests of the original VDM-LPF, described in Section 5.1. The corresponding Isabelle/Isar 2005 proof is displayed on Figure 69. This serves not only as a nice example of the usage of VDM-LPF but also as a comparison between the old and new style proofs. The same rules are applied as in Isabelle/Classic. The guidelines for changing old style to new style proofs are followed as they were explained in the previous chapter. For an explanation of the proof steps please consult the fore-mentioned section.

```
Lemma "forall x:A & x=x"
 apply(unfold forall_def)
 apply(rule not_exists_intr)
 apply(rule not_not_intr)
 apply(rule eq_self_intr)
 apply(assumption)
done
```

Figure 69: Third Predicate Logic Proof

The last proof involves a rule concerning the definedness of the universal quantifier. This rule says that if for any arbitrary value, in the domain, predicate P is defined then the universal quantification of predicate P is also defined. The complete proof is displayed on Figure 70. The

only premise is stored in the p1 local rule because it is used in the resolution of this rule. The first step is unfolding the forall_def definition which results in the following proof state:

```
prems:
  ?y : A ==> def P ?y
goal (lemma def_forall, 1 subgoal):
 1. def  not (exists x : A &  not P x)
```

The next rule shows how the THEN method attribute can be used to create new rules. In this particular case the conclusion of the def_exists_inh rule is resolved by the first premise of the def_elim rule and returns the resulting conclusion. This conclusion forms a new rule which is applied to the current subgoal and changes the proof state in the following manner:

```
prems:
  ?y : A ==> def P ?y
goal (lemma def_forall, 3 subgoals):
 1. !!x. x : ?A1 ==> def ?P1 x
 2. exists x : ?A1 & ?P1 x ==> def  not (exists x : A &  not P x)
 3.  not (exists x : ?A1 & ?P1 x) ==> def  not (exists x : A &  not P x)
```

In this subgoal list we can identify a number of schematic variables that can be instantiated by arbitrary values. These schematic variables are replaced automatically by Isabelle via an application of the assumption method. This application creates the following proof state:

```
fixed variables: A, P
prems:
  ?y : A ==> def P ?y
goal (lemma def_forall, 2 subgoals):
 1. exists x : A & P x ==> def  not (exists x : A &  not P x)
 2.  not (exists x : A & P x) ==> def  not (exists x : A &  not P x)
```

Next we decide to simplify the conclusion as much as possible since it involves a more complicated term as compared to the assumption. The def_not-inh is the first rule we apply:

```
prems:
  ?y : A ==> def P ?y
goal (lemma def_forall, 2 subgoals):
 1. exists x : A & P x ==> def (exists x : A &  not P x)
 2.  not (exists x : A & P x) ==> def  not (exists x : A &  not P x)
```

Next we apply the def_exists_inh which replaces the existential quantifier by its predicate:

```
prems:
  ?y : A ==> def P ?y
goal (lemma def_forall, 2 subgoals):
 1. !!x. [| exists x : A & P x; x : A |] ==> def  not P x
 2.  not (exists x : A & P x) ==> def  not (exists x : A &  not P x)
```

Finally, the last simplification to the conclusion is performed by applying the def_not_inh rule. This rule removes the negation operator:

```
prems:
  ?y : A ==> def P ?y
goal (lemma def_forall, 2 subgoals):
 1. !!x. [| exists x : A & P x; x : A |] ==> def P x
 2.  not (exists x : A & P x) ==> def  not (exists x : A &  not P x)
```

Next we turn our attention to the assumption involving the existential quantifier. We apply the exists_elim rule and Isabelle displays the following proof state:

```
prems:
  ?y : A ==> def P ?y
goal (lemma def_forall, 2 subgoals):
 1. !!x y. [| x : A; P y; y : A |] ==> def P x
 2.  not (exists x : A & P x) ==> def  not (exists x : A &  not P x)
```

The conclusion of our premise is now identical to the conclusion of the first subgoal. As such we apply this premise as an introduction rule. This results in an updated proof state (shown below) whereby the first subgoal is resolved by assumption:

```
prems:
  ?y : A ==> def P ?y
goal (lemma def_forall, 2 subgoals):
 1. !!x y. [| x : A; P y; y : A |] ==> x : A
 2.  not (exists x : A & P x) ==> def  not (exists x : A &  not P x)
```

The last subgoal is resolved in a similar manner. We apply successive introduction rules until the conclusion of the rule resembles the conclusion of our premise. In this case however, there is no need to simplify the assumption since the application of the premise, as an introduction rule, is enough to allow the successful completion of the proof by assumption. Because of the similarities in proving this last subgoal to the previous one, detailed illustrations of the proof state are skipped.

```
lemma def_forall:
 assumes p1: "(!!y. y:A ==> def P(y))"
   shows "def (forall x:A & P x)"
 apply(unfold forall_def)
 apply(rule def_exists_inh [THEN def_elim])
   apply(assumption)
  apply(rule def_not_inh)
  apply(rule def_exists_inh)
  apply(rule def_not_inh)
  apply(erule exists_elim)
  apply(rule p1)
  apply(assumption)
 apply(rule def_not_inh)
 apply(rule def_exists_inh)
 apply(rule def_not_inh)
 apply(rule p1)
 apply(assumption)
done
```

Figure 70: Universal Quantifier Definedness Proof

6.3 Other

In this section we perform one proof which was used during our sanity tests. (We essentially did only one such proof because it involves knowledge of VDM-SL, something that was beyond the scope of this thesis. Furthermore, our effort to find such other such VDM-SL based theories as test subjects was hampered due to the need to make use of external tools for the axiomatization of

the VDM-SL specifications or the further enrichment of the existing theories. Unfortunately these tools are no longer accessible.)

The corresponding proof can be seen on Figure 71. It is identical to the proof performed in Section 5.1 as a sanity test and serves as a comparison proof between old and new style proofs. Exactly the same rules were used in the new style proof in the old one. The guidelines as explained in the previous chapter were used to form the Isabelle/Isar proof as seen below. For an explanation of the proof consult the fore-mentioned section.

```
lemma
 assumes p1: "n:nat"
     and p2: "m:nat"
     and p3: "m=n"
   shows "(if n=m then true else false) = true"
 apply(cut_tac p1 p2 p3)
 apply(rule if_true)
  apply(rule true_form)
 apply(rule eq_symm1)
  apply(assumption)
 apply(assumption)
Done
```

Figure 71: Simple VDM-SL Proof

7 Conclusion

One of our stated goals was to investigate how well suited a non-classical logic, such as LPF, is in verifying software specifications. In the course of this investigation we felt the need to understand and experiment with an implementation of a non-classical logic in a theorem proving environment. Being always exposed to classical logics, the need to gain the necessary knowledge and insight in a non-classical logic was essential. We believe the exercise of upgrading VDM-LPF to the latest version of the theorem prover Isabelle enhanced this understanding and increased our confidence on the use of a non-classical logic in verifying specifications. Our confidence was increased because we were able to witness that proof in LPF is not much different from its classical counterpart. We were able to validate fundamental propositional and predicate logic laws as found in a standard discrete mathematics textbooks. The only classical logic property not available to LPF is the law of the excluded middle. This law can be made available in LPF by establishing definedness assumptions. Actually any classical theorem can be re-introduced in LPF by means of definedness assumptions.

We have explored areas of specification languages and their associated tools that further enhanced our understanding. Furthermore, the importance of the underlying logic used in such tools was made evident.

Exploring, understanding, and comparing theorem provers such as Isabelle and PVS was another area of the specification and verification field we examined. Especially enlightening was the generic nature of Isabelle and its facilities for creating new object logics. Understanding unique topics in the field of computer aided theorem proving, such as new logic definition, rule application, and unification, that would have been otherwise not visited, was especially rewarding. We have been able to learn how Isabelle came to be, its history and its design goals. Experimenting with the old-style theories and rewriting them in the new style gave us a unique perspective on Isabelle's evolution.

With our work, we revived VDM-LPF by making it available in the modern incarnation of Isabelle. This logic can now be used by VDM and LPF users alike to either further develop it or learn from it, like we did.

Further, developing VDM-LPF is now possible. On the theoretical level, possible routes would be to further develop existing theories such as types or to add theories for functions. On the practical level, existing theories can further be upgraded to make use of more advanced Isabelle features. The proofs can be further changed to make use of the human-readable Isar proof texts,

as opposed the current Isar tactic emulation method. The built-in proof search tactics can be embedded in the Isar theories, by extending the theory syntax to allow for the definition of rule sets and the use of tactics, without the need to embed ML code in them.

Finally, beyond the availability of a modern realization of VDM's LPF, an added benefit of our work is that it can be used as a guide for upgrading old-style theories into Isar ones. Our detailed account of this upgrading process and the Isabelle tools VDM-LPF made use of provide a wide enough area of changes that can be used for any older type theories to be upgraded. The only Isabelle set of tools VDM-LPF did not make use of was the simplifier and the classical reasoner, since they were inappropriate for a non-classical logic such as LPF.

8 References

[Abrial96] Jean-Raymond Abrial. *The B Book: Assigning Programs to Meanings*. Cambrige University Press, August 1996

[AF96] Sten Agerholm and Jacob Frost. "An Isabelle-based Theorem Prover for VDM-SL". *Proceedings of TPHOLs97*. LNCS, Springer-Verlag, August 1997.

[Appel2] Andrew W. Appel. *Hints on Proving Theorems in Twelf*. Princeton University, February 2000. http://www.cs.princeton.edu/~appel/twelf-tutorial/.

[Aspinal] David Aspinal. *Proof General* [Internet]. Available from http://www.proofgeneral.org [Accessed August 2005]

[Aspinal2] David Aspinal. "Proof General: A generic tool for proof development." In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 1785 of Lecture Notes in Computer Science, pages 38-42. Springer-Verlag, 2000.

[Barnett+04] Mike Barnett, K. Rustan, M. Leino, and Wolfram Schulte. "The Spec# programming system: An overview." In *CASSIS 2004, LNCS* vol. 3362, Springer, 2004.

[BCJ84] H. Barringer, J.H. Cheng, and C.B. Jones, "A Logic Covering Undefinedness in Program Proofs", *Acta Informatica*, vol. 21, no. 3, pp. 251-269, 1984.

[Bicarregui+94] J. C. Bicarregui, J. S. Fitzegerald, P. A. Lindsay, R. Moore, and B. Ritchie. *Proof in VDM: A Practitioner's Guide*. FACIT. Springer-Verlag, 1994.

[Burdy+05] Lilian Burdy, Yoonsik Cheon, David Cok, Michael Ernst, Joe Kiniry, Gary T. Leavens, K. Rustan M. Leino, and Erik Poll. *An overview of JML tools and applications*. International Journal on Software Tools for Technology Transfer, volume 7, number 3, pages 212-232, June 2005

[CJ91] J. H. Cheng and C. B. Jones. *On the usability of logics which handle partial functions*. In Proceedings of The 3rd Refinement Workshop, C. Morgan and J. C. P. Woodcock (eds), pp 51-69, Springer-Verlag, 1991

[Chalin05] P. Chalin, "Logical Foundations of Program Assertions: What do Practitioners Want?" In *Proceedings of the 3rd International Conference on Software Engineering and Formal Methods* (SEFM'05), Koblenz, Germany, September 5-9, 2005 (to appear). Preprint available as ENCS-CSE TR 2005-002.

[Chalin05-1] Patrice Chalin. "Reassessing JML's Logical Foundation". To appear in *Proceedings of the 7th Workshop on Formal Techniques for Java-like Programs* (*FTfJP'05*), Glasgow, Scotland, July, 2005.

[Cheng86] H. J. Cheng. *A logic for partial functions*. Ph.D. Thesis UMCS-86-7-1, Department of Computer Science, University of Manchester, Manchester M13 9PL, England, 1986.

[Church40] A. Church. *A formulation of the simple theory of types*. A Journal of Symbolic Logic, 5: 56-68, 1940.

[ESCJ2] ESC/Java2 site at secure.ucd.ie/products/opensource/ESCJava2.

[GM93] M. J. C. Gordon and T. F. Melham, editors. *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic*. Cambridge University Press, 1993.

[Gordon97] Mike Gordon. *From LCF to HOL: a short history*. Published on WWW, 1997. http://www.cl.cam.ac.uk/users/mjcg/papers/HolHistory.html . Also appeared in [PST2] pages 169-185.

[Hahnle05] Reiner Hahnle."Many-Valued Logic, Partiality, and Abstraction in Formal Specification of Languages." *Logic Journal of the IGPL*, Oxford University Press, 13(4), 415-433, July 2005

[Jone90] C.B. Jones. *Systematic Software Development using VDM*. Computer Science Series. PHI, 2nd Edition, 1990.

[JP03] B. Jacobs and E. Poll, "Java Program Verification at Nijmegen: Developments and Perspective." In *Proceedings of the International Symposium on Software Security - Theories and Systems (ISSS 2003)*, LNCS 3233, pp. 134-153, 2003.

[LBR05] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. *Preliminary Design of JML: A Behavioral Interface Specification Language for Java*. Department of Computer Science, Iowa State University, TR #98-06-rev28, revised July 2005.

[Milner+97] Robin Milner, Mads Tofte, Robert Harper, David MacQueen. *The Definition of Standard ML (Revised)*. The MIT Press, 1997

[NPW98] Tobias Nipkow, L. C. Paulson, and Markus Wenzel. *Isabelle/HOL: A Proof Assistant for Higher Order Logic* Springer-Verlag 1998. ISBN 3-540-43376-7

[OSRS1] S. Owre, N.Shanker, J. M. Rushby, D. W. J. Stringer-Calvert. *PVS System Guide - Version 2.4*. http://pvs.csl.sri.com/doc/pvs-system-guide.pdf. November 2001.

[OSRS2] S. Owre, N.Shanker, J. M. Rushby, D. W. J. Stringer-Calvert. *PVS Language Reference – Version 2.4*. http://pvs.csl.sri.com/doc/pvs-language-reference.pdf. November 2001.

[OSRS3] N.Shanker, S. Owre, J. M. Rushby, D. W. J. Stringer-Calvert. *PVS Prover Guide – Version 2.4*. http://pvs.csl.sri.com/doc/pvs-prover-guide.pdf. November 2001.

[Paulson] L. C. Paulson. *The Isabelle Reference Manual* [Internet]. http://isabelle.in.tum.de/doc/ref.pdf [Accessed August, 2005]

[Paulson90] L. C. Paulson. Isabelle: "The next 700 theorem provers." In Piergiorgio Odifreddi, editor, *Logic and Computer Science*, pages 361-386. Academic Press, 1990.

[Paulson96] L. C. Paulson. *ML for the Working Programmer.* Cambridge University Press, 2nd Edition, 1996.

[PST2] Gordon Plotkin, Colin Stirling, Mats Tofte, editors. *Proof, Language, and Interaction: Essays in Honour of Robin Milner.* The MIT Press, 2000.

[Ranise+03] S. Ranise and D. Deharbe. Light-Weight Theorem Proving for Debugging and Verifying Unit is of Code. *Proc. of the International Conference on Software Engineering and Formal Methods (SEFM03)*, IEEE Computer Society Press, Camberra, Australia, September, 2003.

[Rosen91] Kenneth H. Rosen, Discrete Mathematics and it is Applications. McGraw-Hill, Inc, 2nd Edition, ISBN 0-07-053744-5, 1991

[Spivey92] J. M. Spivey. *The Z Notation: A Reference Manual.* Prentice Hall International Series in Computer Science, 2nd Edition, 1992.

[STP] *Simplify Theorem Prover* [Internet]. http://research.compaq.com/SRC/esc/Simplify.html [Accessed August, 2005]

[Wenzel] Markus Wenzel. *The Isabelle/Isar Reference Manual* [Internet]. http://isabelle.in.tum.de/doc/isar-ref.pdf [Accessed August 2005].

[WP] *WikiPedia* [Internet]. www.wikipedia.org [Accessed August, 2005]

9 Appendix A

Here's is the propositional logic old-style theory file as developed by Sten Agerholm and Jacob Frost:

9.1 Prop.thy

```
(*
Title:          Propositonal VDM-LPF Theory
Author(s):      Jacob Frost IT/DTU  (and Sten Agerholm IFAD)
Copyright:      1996  Jacob Frost IT/DTU (and Sten Agerholm IFAD)

File:           Prop.thy
Version:        1.2
Modified:       10:56:16 11/04/96

This is prototype software. Use at your own risk. Do not distribute or
reuse in any form without prior written consent of the authors.
*)

Prop = Basic +

consts
true'           :: ex                   ("true")
false'          :: ex                   ("false")

not'            :: ex => ex             ("(2 not _/)" [250] 250)

iff'            :: [ex,ex] => ex        ("(_ <=>/ _)" [210,211] 210)
imp'            :: [ex,ex] => ex        ("(_ =>/ _)" [221,220] 220)
or'             :: [ex,ex] => ex        ("(_ or/ _)" [230,231] 230)
and'            :: [ex,ex] => ex        ("(_ and/ _)" [240,241] 240)

def'            :: ex => ex             ("(2def _)" [250] 250)

rules
(* axiom for true *)

true_intr
"true"

(* axioms for negation *)

not_not_intr
"P ==> not not P"
not_not_dest
"not not P ==> P"

contr
"[| not P; P |] ==> Q"

(* axioms for disjunction *)

or_intr_left
"P ==> P or Q"
or_intr_right
"Q ==> P or Q"
or_elim
```

```
"[| P or Q; P==>R; Q==>R |] ==> R"

not_or_intr
"[| not P; not Q |] ==> not (P or Q)"
not_or_elim
"[| not (P or Q); [| not P; not Q |] ==> R |] ==> R"


defs
false_def
"false == not true"
def_def
"def P == P or not P"
imp_def
"P => Q == not P or Q"
and_def
"P and Q == not (not P or not Q)"
iff_def
"P <=> Q == (P => Q) and (Q => P)"

end
```

10 Appendix B

In this Appendix we list the contents of the `LPF_Prover.ML`, that contains VDM-LPF's built-in proof tactics:

10.1 LPF_Prover.ML

```
signature LPF_PROVER_DATA =
sig
val contr : thm          (* [| not P; P |] ==> Q *)
end;


signature LPF_PROVER =
sig
type lpf_set

val empty_lpfs : lpf_set
val addSIs : lpf_set * thm list -> lpf_set
val addSEs : lpf_set * thm list -> lpf_set
val addSDs : lpf_set * thm list -> lpf_set
val addIs : lpf_set * thm list -> lpf_set
val addEs : lpf_set * thm list -> lpf_set
val addDs : lpf_set * thm list -> lpf_set
val lpf_atac : int -> tactic
val lpf_step_tac : lpf_set -> int -> tactic
val lpf_fast_tac : lpf_set -> int -> tactic
end;
infix 4 addSIs addSEs addSDs addIs addEs addDs;



functor LPF_Prover_Fun(LPF_Prover_Data:LPF_PROVER_DATA):LPF_PROVER =
struct
open LPF_Prover_Data;

datatype lpf_set =
LPFS of
{safeIs : thm list, (*safe introduction rules*)
safeEs : thm list, (*safe elimination rules*)
safeDs : thm list, (*safe destruction rules*)
hazIs  : thm list, (*unsafe introduction rules*)
hazEs  : thm list, (*unsafe elimination rules*)
hazDs : thm list}; (*unsafe destruction rules*)


infix 4 addSIs addSEs addSDs addIs addEs addDs;

val empty_lpfs =
LPFS{safeIs = [], safeEs = [], safeDs = [],
hazIs = [], hazEs = [], hazDs = []};

fun (LPFS{safeIs,safeEs,safeDs,hazIs,hazEs,hazDs}) addSIs thms =
LPFS{safeIs=safeIs@thms,safeEs=safeEs,safeDs=safeDs,
hazIs=hazIs,hazEs=hazEs,hazDs=hazDs};

fun (LPFS{safeIs,safeEs,safeDs,hazIs,hazEs,hazDs}) addSEs thms =
LPFS{safeIs=safeIs,safeEs=safeEs@thms,safeDs=safeDs,
hazIs=hazIs,hazEs=hazEs,hazDs=hazDs};

fun (LPFS{safeIs,safeEs,safeDs,hazIs,hazEs,hazDs}) addSDs thms =
```

```
LPFS{safeIs=safeIs,safeEs=safeEs,safeDs=safeDs@thms,
hazIs=hazIs,hazEs=hazEs,hazDs=hazDs};

fun (LPFS{safeIs,safeEs,safeDs,hazIs,hazEs,hazDs}) addIs thms =
LPFS{safeIs=safeIs,safeEs=safeEs,safeDs=safeDs,
hazIs=hazIs@thms,hazEs=hazEs,hazDs=hazDs};

fun (LPFS{safeIs,safeEs,safeDs,hazIs,hazEs,hazDs}) addEs thms =
LPFS{safeIs=safeIs,safeEs=safeEs,safeDs=safeDs,
hazIs=hazIs,hazEs=hazEs@thms,hazDs=hazDs};

fun (LPFS{safeIs,safeEs,safeDs,hazIs,hazEs,hazDs}) addDs thms =
LPFS{safeIs=safeIs,safeEs=safeEs,safeDs=safeDs,
hazIs=hazIs,hazEs=hazEs,hazDs=hazDs@thms};


fun lpf_atac i = (atac i) ORELSE ((etac contr i) THEN (atac i));

fun lpf_step_tac (LPFS{safeIs,safeEs,safeDs,hazIs,hazEs,hazDs}) i =
(lpf_atac i) ORELSE
(resolve_tac safeIs i) ORELSE
(eresolve_tac safeEs i) ORELSE
(dresolve_tac safeDs i) ORELSE
((resolve_tac hazIs i) INTLEAVE
(eresolve_tac hazEs i) INTLEAVE
(dresolve_tac hazDs i));

fun lpf_fast_tac lpfs  =
SELECT_GOAL (DEPTH_SOLVE (lpf_step_tac lpfs 1));
end;
```

11 Appendix C

In here is we present Isabelle's meta-logic syntax. This was generated via the `print_syntax` command:

11.1 Isabelle/CPure Syntax

```
lexicon: "!!" "%" "(" ")" "," "." "..." "::" ";" "==" "==>" "=>" "OFCLASS"
"PROP" "TYPE" "[" "[|" "\<And>" "\<Colon>" "\<Longrightarrow>" "\<Rightarrow>"
  "\<^sub>" "\<dots>" "\<equiv>" "\<equiv>\<^sup>?" "\<index>" "\<lambda>"
"\<lbrakk>" "\<rbrakk>" "\<struct>" "]" "_" "_::" "op" "{" "{}" "|]" "}"
logtypes: dummy it iself fun
prods:
  #prop = "\<And>" idts[0] "." prop[0] => "!!" (0)
  #prop = "\<lbrakk>" asms[0] "\<rbrakk>" "\<Longrightarrow>" prop[1] =>
"_bigimpl" (1)
  #prop = "[|" asms[0] "|]" "==>" prop[1] => "_bigimpl" (1)
  #prop = "(" #prop[0] ")" (1000)
  #prop = "OFCLASS" "(" type[0] "," logic[0] ")" => "_ofclass" (1000)
  #prop = "PROP" aprop[0] => "_aprop" (1000)
  #prop = any[3] "==" any[2] => "==" (2)
  #prop = prop[2] "==>" prop[1] => "==>" (1)
  #prop = any[3] "\<equiv>\<^sup>?" any[2] => "=?=" (2)
  #prop = any[3] "\<equiv>" any[2] => "==" (2)
  #prop = prop[2] "\<Longrightarrow>" prop[1] => "==>" (1)
  #prop = #prop[4] "::" type[0] => "_constrain" (3)
  #prop = "!!" idts[0] "." prop[0] => "!!" (0)
  any = #prop[~1] (~1)
  any = logic[~1] (~1)
  aprop = "\<dots>" => "_DDDOT" (1000)
  aprop = longid (1000)
  aprop = "..." => "_DDDOT" (1000)
  aprop = var (1000)
  aprop = id (1000)
  aprop = "_" => "dummy_pattern" (1000)
  aprop = logic[1000] cargs[1000] => "_applC" (999)
  args = any[0] "," args[0] => "_args" (1000)
  args = any[~1] (~1)
  asms = prop[0] ";" asms[0] => "_asms" (1000)
  asms = prop[~1] (~1)
  cargs = any[1000] cargs[1000] => "_cargs" (1000)
  cargs = any[~1] (~1)
  classes = longid "," classes[0] => "_classes" (1000)
  classes = longid (1000)
  classes = id (1000)
  classes = id "," classes[0] => "_classes" (1000)
  idt = "(" idt[0] ")" (1000)
  idt = id "\<Colon>" type[0] => "_idtyp" (0)
  idt = id (1000)
  idt = id "::" type[0] => "_idtyp" (0)
  idts = idt[1] idts[0] => "_idts" (0)
  idts = idt[~1] (~1)
  index = "\<index>" => "_indexvar" (1000)
  index = => "_noindex" (1000)
  index = "\<^sub>" num_const[0] => "_index" (1000)
  logic = "_" => "dummy_pattern" (1000)
  logic = "\<lambda>" pttrns[0] "." any[3] => "_lambda" (3)
  logic = "\<dots>" => "_DDDOT" (1000)
  logic = "TYPE" "(" type[0] ")" => "_TYPE" (1000)
  logic = longid (1000)
  logic = "..." => "_DDDOT" (1000)
  logic = "(" logic[0] ")" (1000)
```

```
  logic = "\<struct>" index[0] => "_struct" (1000)
  logic = var (1000)
  logic = id (1000)
  logic = "%" pttrns[0] "." any[3] => "_lambda" (3)
  logic = "op" "==" => "==" (1000)
  logic = "op" "==>" => "==>" (1000)
  logic = "op" "\<equiv>" => "==" (1000)
  logic = "op" "\<Longrightarrow>" => "==>" (1000)
  logic = "op" "\<equiv>\<^sup>?" => "=?=" (1000)
  logic = logic[1000] cargs[1000] => "_applC" (999)
  logic = logic[4] "::" type[0] => "_constrain" (3)
  logic = any[4] "\<Colon>" type[0] => "_constrain" (3)
  num_const = num => "_constify" (1000)
  prop = #prop[~1] (~1)
  pttrn = idt[~1] (~1)
  pttrns = pttrn[1] pttrns[0] => "_pttrns" (0)
  pttrns = pttrn[~1] (~1)
  sort = longid (1000)
  sort = "{" classes[0] "}" => "_sort" (1000)
  sort = "{}" => "_topsort" (1000)
  sort = id (1000)
  type = longid (1000)
  type = tvar "::" sort[0] => "_ofsort" (1000)
  type = tvar (1000)
  type = "[" types[0] "]" "=>" type[0] => "_bracket" (0)
  type = "[" types[0] "]" "\<Rightarrow>" type[0] => "_bracket" (0)
  type = "_" => "dummy" (1000)
  type = "(" type[0] "," types[0] ")" id => "_tappl" (1000)
  type = "(" type[0] "," types[0] ")" longid => "_tappl" (1000)
  type = "(" type[0] ")" (1000)
  type = "_::" sort[0] => "_dummy_ofsort" (1000)
  type = tid "\<Colon>" sort[0] => "_ofsort" (1000)
  type = tid (1000)
  type = tid "::" sort[0] => "_ofsort" (1000)
  type = type[1] "\<Rightarrow>" type[0] => "fun" (0)
  type = type[1] "=>" type[0] => "fun" (0)
  type = type[1000] longid => "_tapp" (1000)
  type = type[1000] id => "_tapp" (1000)
  type = id (1000)
  types = type[0] "," types[0] => "_types" (1000)
  types = type[~1] (~1)
print modes: "HTML" "ProofGeneral" "latex" "xsymbols" "xterm" "xterm_color"
consts: "" "" "" "" "!!" "!!" "#prop" "==" "==" "==>" "==>" "==>" "=?=" "Goal"
"Goal" "TYPE" "_DDDOT" "_DDDOT" "_K" "_TYPE" "_abs" "_appl" "_applC" "_aprop"
  "_args" "_asms" "_bigimpl" "_bigimpl" "_bracket" "_bracket" "_cargs"
"_classes" "_constify" "_constrain" "_constrain" "_dummy_ofsort" "_idts"
"_idtyp"
  "_idtyp" "_index" "_indexvar" "_lambda" "_lambda" "_meta_conjunction"
"_mk_ofclass" "_noindex" "_ofclass" "_ofsort" "_ofsort" "_pttrns" "_sort"
"_struct"
  "_tapp" "_tappl" "_topsort" "_types" "all" "any" "aprop" "args" "asms"
"cargs" "classes" "dummy" "dummy" "dummy_pattern" "dummy_pattern" "fun" "fun"
"fun"
  "id" "idt" "idts" "index" "it iself" "logic" "logic" "logic_class" "longid"
"num" "num_const" "prop" "pttrn" "pttrns" "sort" "struct" "tid" "tvar" "type"
  "types" "var" "xnum" "xstr"
parse_ast_translation: "_appl" "_applC" "_bigimpl" "_bracket" "_constify"
"_idtyp" "_indexvar" "_lambda" "_tapp" "_tappl"
parse_rules:
parse_translation: "!!" "_DDDOT" "_K" "_TYPE" "_abs" "_aprop" "_ofclass"
print_translation: "TYPE" "_mk_ofclass" "all"
print_rules:
print_ast_translation: "==>" "_abs" "_idts" "_pttrns" "fun"
```

```
token_translation:
  "": _xstr _xnum _num _var _bound _free _tvar _tfree _class
  "xterm": _var _bound _free _tvar _tfree _class
  "xterm_color": _var _bound _free _tvar _tfree _class
  "HTML": _xstr _var _bound _free _tvar _tfree _class
  "latex": _xstr _xnum _num _var _bound _free _tvar _tfree _class
  "ProofGeneral": _var _bound _free _tvar _tfree _class
```

12 Appendix D

In here we list all the Isar theory files of the upgrade VDM-LPF logic:

12.1 Basic.thy

```
theory Basic = CPure :

typedecl ex

judgment
  (* Natural Deduction *)
  Trueprop  :: "ex => prop" ("(_)" 5)

consts
  undefined'    :: ex       ("undefined")

end
```

12.2 Prop.thy

```
theory Prop = Basic
files "LPF_Prover.ML":

consts
  true'     :: ex           ("true")
  false'    :: ex           ("false")

  not'      :: "ex => ex"       ("(2 not _/)" [250] 250)

  iff'      :: "[ex,ex] => ex"  ("(_ <=>/ _)" [210,211] 210)
  imp'      :: "[ex,ex] => ex"  ("(_ =>/ _)" [221,220] 220)
  or'       :: "[ex,ex] => ex"  ("(_ or/ _)" [230,231] 230)
  and'      :: "[ex,ex] => ex"  ("(_ and/ _)" [240,241] 240)

  def'      :: "ex => ex"       ("(2def _)" [250] 250)

axioms
  (* axiom for true *)

  true_intr :
    "true"

  (* axioms for negation *)

  not_not_intr :
    "P ==> not not P"
  not_not_dest :
    "not not P ==> P"

  contr :
    "[| not P; P |] ==> Q"

  (* axioms for disjunction *)

  or_intr_left :
    "P ==> P or Q"
  or_intr_right :
    "Q ==> P or Q"
  or_elim :
    "[| P or Q; P==>R; Q==>R |] ==> R"
```

```
  not_or_intr :
    "[| not P; not Q |] ==> not (P or Q)"
  not_or_elim :
        "[| not (P or Q); [| not P; not Q |] ==> R |] ==> R"


defs
  false_def:
    "false == not true"
  def_def:
    "def P == P or not P"
  imp_def:
    "P => Q == not P or Q"
  and_def:
    "P and Q == not (not P or not Q)"
  iff_def:
    "P <=> Q == (P => Q) and (Q => P)"

(* Functor setup for LPF_Prover*)

ML
{*
  structure LPF_Prover_Data =
  struct
   val contr = thm "contr"
  end;

  structure LPF_Prover = LPF_Prover_Fun(LPF_Prover_Data);

  open LPF_Prover;
*}

(* derived rules for negation *)

lemma not_not_elim :
 assumes p1 : "not not P"
     and p2 : "P ==> R"
 shows "R"
  apply(rule p1 p2)
  apply(rule not_not_dest)
  apply(rule p1)
done

ML
{*
  val not_lpfs = empty_lpfs addSIs [thm "not_not_intr"]
                            addSEs [thm "not_not_elim"];
*}

lemma not_true_elim : "!!P. not true ==> P"
 apply(erule contr)
 apply(rule true_intr)
done

ML
{*
 val true_lpfs = not_lpfs addSIs [thm "true_intr"]
                          addSEs [thm "not_true_elim"];

(* derived rules for disjunction *)
 val disj_lpfs =
     true_lpfs
```

```
      addSIs [thm "not_or_intr"]
      addSEs [thm "or_elim", thm "not_or_elim"]
      addIs [thm "or_intr_left", thm "or_intr_right"];
*}

(* derived rules for falsity *)

lemma false_elim : "!!P. false ==> P"
 apply(unfold false_def)
 apply(tactic {* lpf_fast_tac disj_lpfs 1 *})
done

lemma not_false_intr : "not false"
 apply(unfold false_def)
 apply(tactic {* lpf_fast_tac disj_lpfs 1 *})
done

ML
{*
  val false_lpfs = disj_lpfs addSIs [thm "not_false_intr"]
                            addSEs [thm "false_elim"];
*}

(* derived rules for implication *)

lemma imp_intr_left : "!!P. not P ==> P => Q"
 apply(unfold imp_def)
 apply(tactic {* lpf_fast_tac false_lpfs 1 *})
done

lemma imp_intr_right : "!!P. Q  ==> P => Q"
 apply(unfold imp_def)
 apply(tactic {* lpf_fast_tac false_lpfs 1  *})
done

lemma imp_elim :
 assumes p1 : "P => Q"
     and p2 : "not P==>R"
     and p3 : "Q==>R"
 shows "R"
 apply(cut_tac p1)
 apply(unfold imp_def)
 apply(tactic {* lpf_fast_tac (false_lpfs addIs [thm "p2",thm "p3"]) 1  *})
done

lemma not_imp_intr : "!!P. [| P; not Q |] ==> not (P => Q)"
 apply(unfold imp_def)
 apply(tactic {*lpf_fast_tac false_lpfs 1*})
done

lemma not_imp_elim :
 assumes p1 : "not (P => Q)"
     and p2 : "[| P; not Q |] ==> R"
 shows "R"
 apply(cut_tac p1)
 apply(unfold imp_def)
 apply(tactic {* lpf_fast_tac (false_lpfs addIs [thm "p2"]) 1 *})
done

ML
{*
  val imp_lpfs =
      false_lpfs
```

```
      addSIs [thm "not_imp_intr"]
      addSEs [thm "imp_elim",thm "not_imp_elim"]
      addIs [thm "imp_intr_left", thm "imp_intr_right"];
*}

(* derived rules for definedness *)

lemma def_intr_left : "!!P. P ==> def P"
 apply(unfold def_def)
 apply(tactic {* lpf_fast_tac imp_lpfs 1 *})
done

lemma def_intr_right : "!!P. not P ==> def P"
 apply(unfold def_def)
 apply(tactic {* lpf_fast_tac imp_lpfs 1 *})
done

lemma def_elim :
 assumes p1 : "def P"
     and p2 : "P==>Q"
     and p3 : "not P==>Q"
 shows "Q"
 apply(cut_tac p1)
 apply(unfold def_def)
 apply(tactic {* lpf_fast_tac (imp_lpfs addIs [thm "p2",thm "p3"]) 1 *})
done

lemma not_def_intr : "!!P.[| P; not P |] ==> not def P"
 apply(unfold def_def)
 apply(tactic {* lpf_fast_tac imp_lpfs 1 *})
done

lemma not_def_elim :
 assumes p1 : "not def P"
     and p2 : "[| P; not P |] ==> Q"
 shows "Q"
 apply(cut_tac p1)
 apply(unfold def_def)
 apply(tactic {* lpf_fast_tac (imp_lpfs addIs [thm "p2"]) 1 *})
done

ML
{*
val def_lpfs =
    imp_lpfs
    addSIs [thm "not_def_intr"]
    addSEs [thm "def_elim",thm "not_def_elim"]
    addIs [thm "def_intr_left", thm "def_intr_right"];
*}

(* derived rules for conjunction *)

lemma and_intr : "!!P.[| P; Q |] ==> P and Q"
 apply(unfold and_def)
 apply(tactic {* lpf_fast_tac def_lpfs 1 *})
done

lemma and_elim :
 assumes p1 : "P and Q"
     and p2 : "[| P; Q |] ==> R"
 shows "R"
 apply(cut_tac p1)
 apply(unfold and_def)
```

```
 apply(tactic {* lpf_fast_tac (def_lpfs addIs [thm "p2"]) 1 *})
done

lemma not_and_intr_left : "!!P. not P ==> not (P and Q)"
 apply(unfold and_def)
 apply(tactic {* lpf_fast_tac def_lpfs 1 *})
done

lemma not_and_intr_right : "!!P. not Q ==> not (P and Q)"
 apply(unfold and_def)
 apply(tactic {* lpf_fast_tac def_lpfs 1 *})
done

lemma not_and_elim :
 assumes p1 : "not (P and Q)"
     and p2 : "not P==>R"
     and p3 : "not Q==>R"
 shows "R"
 apply(cut_tac p1)
 apply(unfold and_def)
 apply(tactic {* lpf_fast_tac (def_lpfs addIs [thm "p2", thm "p3"]) 1 *})
done

ML
{*
 val and_lpfs =
     def_lpfs
     addSIs [thm "and_intr"]
     addSEs [thm "and_elim", thm "not_and_elim"]
     addIs [thm "not_and_intr_left", thm "not_and_intr_right"];
*}

(* rules and tactics for dealing with first-order hypothetical asssumptions *)

lemma mk_hyp_asm_rl_lem1 :
 assumes p1 : "P ==> Q"
     and p2 : "Q ==> R"
     and p3 : "P or R"
 shows "R"
 apply(rule p3 [THEN or_elim])
 apply(rule p2)
 apply(erule p1)
 apply(assumption)
done

lemma mk_hyp_asm_rl_lem2 :
 assumes p1 : "[| P;Q |] ==> PROP T"
     and p2 : "P and Q"
 shows "PROP T"
 apply(cut_tac p2)
 apply(rule p1)
 apply(erule and_elim,assumption)
 apply(erule and_elim,assumption)
done

ML
{*
fun mk_hyp_asm_rl t =
    case compose(t,1,thm "mk_hyp_asm_rl_lem2") of
        [] => t COMP (thm "mk_hyp_asm_rl_lem1") |
        [t'] => mk_hyp_asm_rl t';

fun hyp_asm_tac t = rtac (mk_hyp_asm_rl t);
```

```
*}

(* derived rules for bi-implication *)

lemma iff_intr : "!!P.[| not P or Q; not Q or P |] ==> P <=> Q"
 apply(unfold iff_def)
 apply(tactic {* lpf_fast_tac and_lpfs 1 *})
done

lemma iff_elim :
 assumes p1 : "P <=> Q"
     and p2 : "[| not P; not Q |] ==> R"
     and p3 : "[| Q; P |] ==> R"
 shows "R"
 apply(cut_tac p1)
 apply(unfold iff_def)
 apply(tactic {* lpf_fast_tac (and_lpfs addIs [thm "p2",thm "p3"]) 1 *})
done

lemma not_iff_intr_left : "!!P.[| P; not Q |] ==> not (P <=> Q)"
 apply(unfold iff_def)
 apply(tactic {* lpf_fast_tac and_lpfs 1 *})
done

lemma not_iff_intr_right : "!!P.[| not P; Q |] ==> not (P <=> Q)"
 apply(unfold iff_def)
 apply(tactic {* lpf_fast_tac and_lpfs 1 *})
done

lemma not_iff_elim :
 assumes p1 : "not (P <=> Q)"
     and p2 : "[| P; not Q |] ==> R"
     and p3 : "[| not P; Q |] ==> R"
 shows "R"
 apply(cut_tac p1)
 apply(unfold iff_def)
 apply(tactic {* lpf_fast_tac (and_lpfs addIs [thm "p2",thm "p3"]) 1 *})
done

ML
{*
val iff_lpfs =
    and_lpfs
    addSIs [thm "iff_intr"]
    addSEs [thm "iff_elim", thm "not_iff_elim"]
    addIs [thm "not_iff_intr_left", thm "not_iff_intr_right"];

val prop_lpfs = iff_lpfs;
*}

(* other derived rules from "Proof in VDM *)

(* conjunction *)

lemma and_or_dist_con: "!!P. P and Q or P and R ==> P and (Q or R)"
 apply(tactic {* lpf_fast_tac prop_lpfs 1 *})
done

lemma and_or_dist_exp: "!!P. P and (Q or R) ==> P and Q or P and R"
 apply(tactic {* lpf_fast_tac prop_lpfs 1 *})
done

lemma and_assoc_left: "!!P. P and Q and R ==> P and (Q and R)"
```

```
 apply(tactic {* lpf_fast_tac prop_lpfs 1 *})
done

lemma and_assoc_right: "!!P. P and (Q and R) ==> P and Q and R"
 apply(tactic {* lpf_fast_tac prop_lpfs 1 *})
done

lemma and_comm: "!!P. P and Q ==> Q and P"
 apply(tactic {* lpf_fast_tac prop_lpfs 1 *})
done

lemma and_elim_left: "!!P. P and Q ==> P"
 apply(tactic {* lpf_fast_tac prop_lpfs 1 *})
done

lemma and_elim_right: "!!P. P and Q ==> Q"
 apply(tactic {* lpf_fast_tac prop_lpfs 1 *})
done

lemma and_subs_left:
 assumes p1 : "P and Q"
     and p2 : "P==>R"
 shows "R and Q"
 apply(cut_tac p1)
 apply(tactic {* lpf_fast_tac (prop_lpfs addSIs [thm "p2"]) 1 *})
done

lemma and_subs_right:
 assumes p1 : "P and Q"
     and p2 : "Q==>R"
   shows "P and R"
 apply(cut_tac p1)
 apply(tactic {* lpf_fast_tac (prop_lpfs addIs [thm "p2"]) 1 *})
done

(* definedness *)

lemma def_and_inh: "!!P.[| def P; def Q |] ==> def (P and Q)"
 apply(tactic {* lpf_fast_tac prop_lpfs 1 *})
done

lemma def_and_inh_sqt:
 assumes p1: "def P"
     and p2: "P ==> def Q"
   shows "def (P and Q)"
 apply(cut_tac p1)
 apply(tactic {* hyp_asm_tac (thm "p2") 1 *})
 apply(tactic {* lpf_fast_tac prop_lpfs 1 *})+
done

lemma def_iff_inh: "!!P.[| def P; def Q |] ==> def (P <=> Q)"
 apply(tactic {* lpf_fast_tac prop_lpfs 1 *})
done

lemma def_imp_inh: "!!P.[| def P; def Q |] ==> def (P => Q)"
 apply(tactic {* lpf_fast_tac prop_lpfs 1 *})
done

lemma def_imp_inh_sqt:
 assumes p1: "def P"
     and p2: "P==>def Q"
   shows "def (P => Q)"
 apply(cut_tac p1)
```

```
 apply(tactic {* hyp_asm_tac (thm "p2") 1 *})
 apply(tactic {* lpf_fast_tac prop_lpfs 1 *})+
done

lemma def_not_inh: "!!P. def P ==> def (not P)"
 apply(tactic {* lpf_fast_tac prop_lpfs 1 *})
done

lemma def_or_intr_left: "!!P. Q ==> def (P or Q)"
 apply(tactic {* lpf_fast_tac prop_lpfs 1 *})
done

lemma def_or_intr_right: "!!P. P ==> def (P or Q)"
 apply(tactic {* lpf_fast_tac prop_lpfs 1 *})
done

lemma def_or_inh : "!!P.[| def P; def Q |] ==> def (P or Q)"
 apply(tactic {* lpf_fast_tac prop_lpfs 1 *})
done

lemma def_or_inh_sqt:
 assumes p1: "def P"
     and p2: "not P==>def Q"
   shows "def (P or Q)"
 apply(cut_tac p1)
 apply(tactic {* hyp_asm_tac (thm "p2") 1 *})
 apply(tactic {* lpf_fast_tac prop_lpfs 1 *})+
done

(* bi-implication *)

lemma iff_not_intr: "!!P. P <=> Q ==> not P <=> not Q"
 apply(tactic {*lpf_fast_tac prop_lpfs 1 *})
done

lemma iff_comm: "!!P. P <=> Q ==> Q <=> P"
 apply(tactic {* lpf_fast_tac prop_lpfs 1 *})
done

lemma iff_elim_full: "!!P. P <=> Q ==> P and Q or not P and not Q"
 apply(tactic {* lpf_fast_tac prop_lpfs 1 *})
done

lemma iff_elim_left: "!!P.[| P <=> Q; P |] ==> Q";
 apply(tactic {* lpf_fast_tac prop_lpfs 1 *})
done

lemma iff_elim_left_def: "!!P. P <=> Q ==> def Q"
 apply(tactic {* lpf_fast_tac prop_lpfs 1 *})
done

lemma iff_elim_left_not: "!!P.[| P <=> Q; not P |] ==> not Q"
 apply(tactic {* lpf_fast_tac prop_lpfs 1 *})
done

lemma iff_elim_right: "!!P.[| P <=> Q; Q |] ==> P"
 apply(tactic {* lpf_fast_tac prop_lpfs 1 *})
done

lemma iff_elim_right_def: "!!P. P <=> Q ==> def P"
 apply(tactic {* lpf_fast_tac prop_lpfs 1 *})
done
```

```
lemma iff_elim_right_not: "!!P.[| P <=> Q; not Q |] ==> not P"
 apply(tactic {* lpf_fast_tac prop_lpfs 1 *})
done

lemma iff_intrI: "!!P.[| P; Q |] ==> P <=> Q"
 apply(tactic {* lpf_fast_tac prop_lpfs 1 *})
done

lemma iff_intr_and: "!!P. P and Q ==> P <=> Q"
 apply(tactic {* lpf_fast_tac prop_lpfs 1 *})
done

lemma iff_intr_and_not: "!!P. not P and not Q ==> P <=> Q"
 apply(tactic {* lpf_fast_tac prop_lpfs 1 *})
done

lemma iff_intr_not: "!!P.[| not P; not Q |] ==> P <=> Q"
 apply(tactic {* lpf_fast_tac prop_lpfs 1 *})
done

lemma iff_self_intr: "!!P. def P ==> P <=> P"
 apply(tactic {* lpf_fast_tac prop_lpfs 1 *})
done

(* implication *)

lemma imp_and_left_elim: "!!P.[| P; P and Q => R |] ==> Q => R"
 apply(tactic {* lpf_fast_tac prop_lpfs 1 *})
done

lemma imp_not_conseq:
 assumes p1: "P => R"
     and p2: "not P ==> not Q"
   shows "Q => R"
 apply(cut_tac p1)
 apply(tactic {* lpf_fast_tac (prop_lpfs addIs [thm "p2"]) 1 *})
done

lemma imp_conseq:
 assumes p1: "P => Q"
     and p2: "Q ==> R"
   shows "P => R"
 apply(cut_tac p1)
 apply(tactic {* lpf_fast_tac (prop_lpfs addIs [thm "p2"]) 1 *})
done

lemma imp_contrp: "!!P. P => Q ==> not Q => not P"
 apply(tactic {* lpf_fast_tac prop_lpfs 1 *})
done

lemma imp_elim_left: "!!P.[| P => Q; P |] ==> Q"
 apply(tactic {* lpf_fast_tac prop_lpfs 1 *})
done

lemma imp_elim_right: "!!P.[| P => Q; not Q |] ==> not P"
 apply(tactic {* lpf_fast_tac prop_lpfs 1 *})
done

lemma imp_intr:
 assumes p1: "def P"
     and p2: "P ==> Q"
   shows "P => Q"
 apply(cut_tac p1)
```

```
 apply(tactic {* lpf_fast_tac (prop_lpfs addIs [thm "p2"]) 1 *})
done

lemma imp_intr_left_vac: "!!P. Q ==> P => Q"
 apply(tactic {* lpf_fast_tac prop_lpfs 1 *})
done

lemma imp_intr_right_vac: "!!P. not P ==> P => Q"
 apply(tactic {* lpf_fast_tac prop_lpfs 1 *})
done

lemma imp_self_intr: "!!P. def P ==> P => P"
 apply(tactic {* lpf_fast_tac prop_lpfs 1 *})
done

lemma imp_trans: "!!P. [| P => Q; Q => R |] ==> P => R"
 apply(tactic {* lpf_fast_tac prop_lpfs 1 *})
done

(* negation *)

lemma not_and_elim_dem: "!!P. not (P and Q) ==> not P or not Q"
 apply(tactic {* lpf_fast_tac prop_lpfs 1 *})
done

lemma not_and_elim_left: "!!P.[| P; not (P and Q) |] ==> not Q"
 apply(tactic {* lpf_fast_tac prop_lpfs 1 *})
done

lemma not_and_elim_right: "!!P.[| Q; not (P and Q) |] ==> not P"
 apply(tactic {* lpf_fast_tac prop_lpfs 1 *})
done

lemma not_and_intr_dem: "!!P. not P or not Q ==> not (P and Q)"
 apply(tactic {* lpf_fast_tac prop_lpfs 1 *})
done

lemma not_and_intr_sqt:
 assumes p1: "def P"
     and p2: "P ==> not Q"
   shows "not (P and Q)"
 apply(cut_tac p1)
 apply(tactic {* lpf_fast_tac (prop_lpfs addIs [thm "p2"]) 1 *})
done

lemma not_imp_elim_left: "!!P. not (P => Q) ==> not Q"
 apply(tactic {* lpf_fast_tac prop_lpfs 1 *})
done

lemma not_imp_elim_right: "!!P. not (P => Q) ==> P"
 apply(tactic {* lpf_fast_tac prop_lpfs 1 *})
done

lemma not_or_elim_dem: "!!P. not (P or Q) ==> not P and not Q"
 apply(tactic {* lpf_fast_tac prop_lpfs 1 *})
done

lemma not_or_intr_dem: "!!P. not P and not Q ==> not (P or Q)"
 apply(tactic {* lpf_fast_tac prop_lpfs 1 *})
done

(* disjunction *)
```

```
lemma or_and_dist_con: "!!P.(P or Q) and (P or R) ==> P or Q and R"
 apply(tactic {* lpf_fast_tac prop_lpfs 1 *})
done

lemma or_and_dist_exp: "!!P. P or Q and R ==> (P or Q) and (P or R)"
 apply(tactic {* lpf_fast_tac prop_lpfs 1 *})
done

lemma or_assoc_left: "!!P. P or Q or R ==> P or (Q or R)"
 apply(tactic {* lpf_fast_tac prop_lpfs 1 *})
done

lemma or_assoc_right: "!!P. P or (Q or R) ==> P or Q or R"
 apply(tactic {* lpf_fast_tac prop_lpfs 1 *})
done

lemma or_comm: "!!P. P or Q ==> Q or P"
 apply(tactic {* lpf_fast_tac prop_lpfs 1 *})
done

lemma or_elim_left_not: "!!P.[| P or Q; not P |] ==> Q"
 apply(tactic {* lpf_fast_tac prop_lpfs 1 *})
done

lemma or_elim_right_not: "!!P.[| P or Q; not Q |] ==> P"
 apply(tactic {* lpf_fast_tac prop_lpfs 1 *})
done

lemma or_subs_left:
 assumes p1: "P or Q"
     and p2: "P ==> R"
   shows "R or Q"
 apply(cut_tac p1)
 apply(tactic {* lpf_fast_tac (prop_lpfs addIs [thm "p2"]) 1 *})
done

lemma or_subs_right:
 assumes p1: "P or Q"
     and p2: "Q ==> R"
   shows "P or R"
 apply(cut_tac p1)
 apply(tactic {* lpf_fast_tac (prop_lpfs addIs [thm "p2"]) 1 *})
done

(* falsity *)

lemma false_contr:
 assumes p1: "def P"
     and p2: "P ==> false"
   shows "not P"
 apply(cut_tac p1)
 apply(tactic {* lpf_fast_tac (prop_lpfs addEs [make_elim (thm "p2")]) 1 *})
done

end
```

12.3 Pred.thy

```
theory Pred = Prop:

(* Corresponds to section 14.2 in Proof in VDM *)
ML
{*
```

```
  print_syntax (theory "Prop");
*}


typedecl
  ty

(* type binding lists *)

nonterminals tbinds tbind

print_syntax

syntax
  ""            :: "tbind => tbinds"        ("_")
  tbindsn_   :: "[tbind,tbinds] => tbinds"   ("(_,/ _)")
  tbind_     :: "[idt,ty] => tbind"       ("(_ :/ _)")

print_syntax

(* quantification *)

consts
  forall'       :: "[ty,ex => ex] => ex"
  exists'       :: "[ty,ex => ex] => ex"
  exists1'      :: "[ty,ex => ex] => ex"
  iota'         :: "[ty,ex => ex] => ex"

syntax
  forall_       :: "[tbinds,ex] => ex"     ("(2forall/ _ &/ _)" [100,100] 100)
  exists_       :: "[tbinds,ex] => ex"     ("(2exists/ _ &/ _)" [100,100] 100)
  exists1_      :: "[tbinds,ex] => ex"     ("(2exists1/ _ &/ _)" [100,100] 100)
  iota_         :: "[tbind,ex] => ex"      ("(2iota/ _ & _)" [100,100] 100)

translations
  "forall_ (tbindsn_ tb tbs) e" == "forall_ tb (forall_ tbs e)"
  "forall_ (tbind_ x A) e" => "forall' A (%x. e)"
  "exists_ (tbindsn_ tb tbs) e" == "exists_ tb (exists_ tbs e)"
  "exists_ (tbind_ x A) e" => "exists' A (%x. e)"
  "exists1_ (tbindsn_ tb tbs) e" == "exists1_ tb (exists_ tbs e)"
  "exists1_ (tbind_ x A) e" => "exists1' A (%x. e)"
  "iota_ (tbind_ x A) e" => "iota' A (%x. e)"

print_translation
{*
let
  fun eta_exp (e as Abs(_,_,_)) = e
    | eta_exp e =Abs("x",dummyT,e$(Bound 0));

  fun quan_tr' r [A,e] =
      let val Abs(x,_,e') = eta_exp e
      val (x',e'') = variant_abs(x,dummyT,e')
      in
      Const(r,dummyT)$(Const("tbind_",dummyT)$Free(x',dummyT)$A)$e''
      end;
in
  [("forall'",quan_tr' "forall_"),("exists'",quan_tr' "exists_"),
   ("exists1'",quan_tr' "exists1_"), ("iota'",quan_tr' "iota_")]
end;
*}

consts
  (* type membership *)
```

```
  of'   :: "[ex,ty] => ex"      ("(_ :/ _)" [100,100] 10)
  (* having typing membership as an expression and not a judgement,
     allows substitution for equal values. This also seems to be
     the approach taken in Mural *)

  (* inhabited *)
  inh'  :: "ty => ex"           ("(2inh _)")

  (* equality *)
  eq'   :: "[ex,ex] => ex"      ("(_ =/ _)" [310,311] 310)
  neq'  :: "[ex,ex] => ex"      ("(_ <>/ _)" [310,311] 310)

(* axioms *)

axioms
    exists_elim:
    "[| exists' A P; !!y.[| P(y); y:A |] ==> Q |] ==> Q"
    exists_intr:
    "[| P(a); a:A |] ==> exists' A P"

    not_exists_dest:
    "[| not (exists' A P); a:A |] ==> not P(a)"
    not_exists_intr:
    "[| !!x. x:A ==> not (P(x)) |] ==> not (exists' A P)"

    def_exists_inh:
    "(!!x. x:A ==> def (P(x))) ==> def (exists' A P)"

(* Definitions, page 274 *)

defs
  forall_def:
    "forall' A P == not (exists x:A & not (P(x)))"
  inh_def:
    "inh A == exists x:A & true"

(* Derived rules, pages 274--277 *)

axioms

(*
  def_forall_forall_intr
    "(!!x y.[| x:A; y:B |] ==> def P(x,y)) ==> \
\        def (forall x:A & forall'(B,P(x)))"
  def_forall_inh
    "(!! y.y:A ==> def P(y)) ==> def forall'(A,P)"
*)


(* Equality axioms, page 277 *)

  eq_self_intr:
    "a:A ==> a = a"
  eq_subs_left:
    "[| b:A; a = b; P(b) |] ==> P(a)"
  eq_subs_right:
    "[| a:A; a = b; P(a) |] ==> P(b)"

  def_eq_intr:
    "[| a:A; b:A |] ==> def a = b"

(* Equality definitions, page 277 *)
```

```
defs
  neq_def:
    "e1 <> e2 == not e1 = e2"

(* Equality derived rules, page 277 -- 279 *)

axioms
  def_neq_intr:
    "[| a:A; b:A |] ==> def a <> b"
  eq_or_neq:
    "[| a:A; b:A |] ==> a = b or a <> b"
  eq_cases:
    "[| a:A; b:A; a <> b ==> Q; a = b ==> Q |] ==> Q"
  eq_ext1:
    "[| a:A; a = b; e(a):B |] ==> e(a) = e(b)"
  eq_ext2:
    "[| a:A; a = b; e(b):B |] ==> e(a) = e(b)"
  eq_subs1:
    "[| a:A; a = b; P(b) |] ==> P(a)"
  eq_subs2:
    "[| b:A; a = b; P(a) |] ==> P(b)"
  eq_symm1:
    "[| a:A; a = b |] ==> b = a"
  eq_symm2:
    "[| b:A; a = b |] ==> b = a"
  eq_trans1:
    "[| a:A; a = b; b = c |] ==> a = c"
  eq_trans2:
    "[| b:A; a = b; b = c |] ==> a = c"
  eq_trans3:
    "[| c:A; a = b; b = c |] ==> a = c"
  eq_trans_left1:
    "[| a:A; a = b; a = c |] ==> b = c"
  eq_trans_left2:
    "[| b:A; a = b; a = c |] ==> b = c"
  eq_trans_left3:
    "[| c:A; a = b; a = c |] ==> b = c"
  eq_trans_right1:
    "[| a:A; a = c; b = c |] ==> a = b"
  eq_trans_right2:
    "[| b:A; a = c; b = c |] ==> a = b"
  eq_trans_right3:
    "[| c:A; a = c; b = c |] ==> a = b"
  eq_type_inh_left:
    "[| a:A; b = a |] ==> b:A"
  eq_type_inh_right:
    "[| b:A; b = a |] ==> a:A"


  not_eq_self_intr:
        "a:A ==> not a <> a"
  neq_comm:
        "[| a:A; b:A; a <> b |] ==> b <> a"



(* Other Quantifiers axioms, page 279 *)

axioms
  iota_form:
        "exists1' A P ==> (iota' A P):A"
  iota_intr:
        "exists1' A P ==> P(iota' A P)"
```

```
(* Other Quantifiers definitions, page 279 *)

defs
  exists1_def:
        "exists1' A P == exists x:A & P(x) and (forall y:A & P(y) => x = y)"

(* Other Quantifiers derived rules, page 279--280 *)

axioms
  exists1_eq_intr:
        "a:A ==> exists1 x:A & x = a"
  exists1_or_dist_exp:
        "exists1 x:A & P(x) or Q(x) ==> (exists1' A P) or (exists1' A Q)"
  exists1_to_exists:
        "(exists1' A P) ==> (exists' A P)"
  exists1_elim:
        "[| (exists1' A P) ; \
\           !! x.[| x:A; P(x); forall y:A & P(y) => y = x |] ==> \
\        Q |] ==> \
\        Q"
  exists1_intr:
        "[| a:A; P(a); forall x:A & P(x) => x = a |] ==> (exists1' A P)"
  exists1_same:
        "[| a:A; b:B; P(a); P(b); (exists1' A P) |] ==> a = b"
  exists1_subs:
        "[| (exists1' A P); !!x.[| x:A; P(x) |] ==> Q(x) |] ==> (exists' A P)"
  iota_defn:
        "[| a:A; P(a); (exists1' A P) |] ==> (iota' A P) = a"

  not_exists1_elim:
        "[| a:A; not (exists1' A P) |] ==> \
\        not P(a) or (exists x:A & P(x) and x <> a)"
  not_exists1_intr:
        "[| a:A; b:A; P(a); P(b); a = b |] ==> not (exists1' A P)"
  not_exists1_intr_vac:
        "not (exists' A P) ==> not (exists1' A P)"
  exists1_forall_to_forall_exists:
        "exists1 x:A & (forall' B (P x)) ==> forall y:B & exists x:A & (P x y)"



ML
{*
val exists_lpfs =
    prop_lpfs
    addIs [thm "exists_intr"] addSIs [thm "not_exists_intr"]
    addDs [thm "not_exists_dest"] addSEs [thm "exists_elim"];
*}

(* standard rules for existential quantification *)

lemma not_exists_elim:
 assumes p1: "not (exists' A P)"
     and p2: "a:A"
     and p3: "not P(a) ==> Q"
   shows "Q"
 apply(cut_tac p1 p2)
 apply(tactic {* lpf_fast_tac (exists_lpfs addIs [thm "p3"]) 1 *})
done

(* non-standard rules for existential quantification *)
ML
```

```
{*
val def_exists =  (thm "def_exists_inh") RS (thm "def_elim");
*}

(* standard rules for type inheritance *)

lemma inh_intr: "a:A ==> inh A"
 apply(unfold inh_def)
 apply(tactic {* lpf_fast_tac exists_lpfs 1 *})
done

lemma inh_elim:
 assumes p1: "inh A"
     and p2: "!!a. a:A ==> Q"
   shows "Q"
 apply(cut_tac p1)
 apply(unfold inh_def)
 apply(tactic {* lpf_fast_tac (exists_lpfs addIs [thm "p2"]) 1 *})
done

ML
{*
val inh_lpfs = exists_lpfs addIs [thm "inh_intr"] addSEs [thm "inh_elim"];
*}


(* standard rules for universal quantification *)

lemma forall_intr:
 assumes p1: "!!x. x:A ==> P(x)"
   shows "forall' A P"
 apply(unfold forall_def)
 apply(tactic {* lpf_fast_tac (exists_lpfs addIs [thm "p1"]) 1 *})
done

lemma forall_dest: "!!a.[| (forall' A P); a:A |] ==> P(a)"
 apply(unfold forall_def)
 apply(tactic {* lpf_fast_tac exists_lpfs 1 *})
done

lemma not_forall_intr: "!!a.[| a:A; not P(a) |] ==> not (forall' A P)"
 apply(unfold forall_def)
 apply(tactic {* lpf_fast_tac exists_lpfs 1 *})
done

lemma not_forall_elim:
 assumes p1: "not (forall' A P)"
     and p2: "!!x.[| x:A; not P(x) |] ==> Q"
   shows "Q"
 apply(cut_tac p1)
 apply(unfold forall_def)
 apply(tactic {* lpf_fast_tac (exists_lpfs addIs [thm "p2"]) 1 *})
done

(* non_standard rules for universal quantification *)

lemma def_forall:
 assumes p1: "(!! y. y:A ==> def P(y))"
   shows "def (forall' A P)"
 apply(unfold forall_def)
 apply(rule def_exists_inh [THEN def_elim])
(* This is the order to prove the subgoals if the lpf_fast_tac is to be
   used on it is own. Alternatevely, the ALLGOALS tactical can be used
```

```
   which resolves the subgoals from the Nth to the 1st. *)
(*    apply(tactic {* lpf_fast_tac (exists_lpfs addDs [thm "p1"]) 3 *}) *)
(*   apply(tactic {* lpf_fast_tac (exists_lpfs addDs [thm "p1"]) 2 *}) *)
(*  apply(tactic {* lpf_fast_tac (exists_lpfs addDs [thm "p1"]) 1 *}) *)
 apply(tactic {* ALLGOALS (fn i => lpf_fast_tac (exists_lpfs addDs [thm "p1"])
i) *})
done

ML
{*
val forall_lpfs =
    inh_lpfs
    addIs [thm "not_forall_intr"] addSIs [thm "forall_intr"]
    addDs [thm "forall_dest"] addSEs [thm "not_forall_elim"];
*}

lemma forall_elim:
 assumes p1: "(forall' A P)"
     and p2: "a:A"
     and p3: "P(a) ==> Q"
   shows "Q"
 apply(cut_tac p1 p2)
 apply(tactic {* lpf_fast_tac (forall_lpfs addIs [thm "p3"]) 1 *})
done


(* other rules for existential quantification *)

lemma exists_forall_to_forall_exists: "exists x:A & (forall' B (P x)) ==>
forall y:B & exists x:A & (P x y)"
 apply(tactic {* lpf_fast_tac forall_lpfs 1 *})
done

lemma exists_forall_subs:
 assumes p1: "exists x:A & (forall' B (P x))"
     and p2: " !!x y.[| x:A; y:B; (P x y) |] ==> (Q x y)"
   shows "exists x:A & (forall' B (Q x))"
 apply(cut_tac p1)
 apply(tactic {* lpf_fast_tac (forall_lpfs addIs [thm "p2"]) 1 *})
done

lemma exists_and_dist_exp: "exists x:A & P(x) and Q(x) ==> (exists' A P) and
(exists' A Q)"
 apply(tactic {* lpf_fast_tac exists_lpfs 1 *})
done

lemma exists_and_elim_left: "exists x:A & P(x) and Q(x) ==> (exists' A Q)"
 apply(tactic {* lpf_fast_tac exists_lpfs 1 *})
done

lemma exists_and_elim_right: "exists x:A & P(x) and Q(x) ==> (exists' A P)"
 apply(tactic {* lpf_fast_tac exists_lpfs 1 *})
done

lemma exists_exists_comm: "exists x:A & (exists' B (P x)) ==> exists y:B &
exists x:A & (P x y)"
 apply(tactic {* lpf_fast_tac exists_lpfs 1 *})
done



lemma exists_exists_elim:
 assumes p1: "exists x:A & (exists' B (P x))"
```

```
     and p2: "!!x y.[| x:A; y:B; (P x y) |] ==> Q"
   shows "Q"
 apply(cut_tac p1)
 apply(tactic {* lpf_fast_tac (forall_lpfs addIs [thm "p2"]) 1 *})
done

lemma exists_exists_intr: "[| a:A; b:B; (P a b) |] ==> exists x:A & (exists' B
(P x))"
 apply(tactic {* lpf_fast_tac exists_lpfs 1 *})
done

lemma exists_exists_subs:
 assumes p1: "exists x:A & (exists' B (P x))"
     and p2: "!!x y.[| x:A; y:B; (P x y) |] ==> (Q x y)"
   shows "exists x:A & (exists' B (Q x))"
 apply(cut_tac p1)
 apply(tactic {* lpf_fast_tac (forall_lpfs addIs [thm "p2"]) 1 *})
done

lemma
 assumes p1: "(exists' A P)"
     and p2: "!!x. x:A ==> P(x) <=> Q(x)"
   shows "(exists' A Q)"
 apply(cut_tac p1)
 apply(tactic {* lpf_fast_tac (exists_lpfs addEs [(thm "p2") RS (thm
"iff_elim")]) 1 *})
done

lemma exists_imp_subs:
 assumes p1: "(exists' A P)"
     and p2: "!!x. x:A ==> P(x) => Q(x)"
   shows "(exists' A Q)"
 apply(cut_tac p1)
 apply(tactic {* lpf_fast_tac (exists_lpfs addEs [(thm "p2") RS (thm
"imp_elim")]) 1 *})
done

lemma exists_or_dist_con: "(exists' A P) or (exists' A Q) ==> exists x:A & P(x)
or Q(x)"
 apply(tactic {* lpf_fast_tac exists_lpfs 1 *})
done

lemma exists_or_dist_exp: "exists x:A & P(x) or Q(x) ==> (exists' A P) or
(exists' A Q)"
 apply(tactic {* lpf_fast_tac exists_lpfs 1 *})
done

lemma exists_to_not_forall_dem: "exists x:A & not P(x) ==> not (forall' A P)"
 apply(tactic {* lpf_fast_tac forall_lpfs 1 *})
done

lemma exists_intr_onept: "[| a:A; P(a) |] ==> exists x:A & x = a and P(x)"
 apply(tactic {* lpf_fast_tac (exists_lpfs addIs [thm "eq_self_intr"]) 1 *})
done

lemma exists_split: "exists x:A & (P x x) ==> exists x:A & (exists' A (P x))"
 apply(tactic {* lpf_fast_tac exists_lpfs 1 *})
done

lemma exists_subs:
 assumes p1: "(exists' A P)"
     and p2: "!!x.[| x:A; P(x) |] ==> Q(x)"
   shows "(exists' A Q)"
```

```
 apply(cut_tac p1)
 apply(tactic {* lpf_fast_tac (exists_lpfs addEs [thm "p2"]) 1 *})
done

lemma not_exists_to_forall_dem: "not (exists' A P) ==> forall x:A & not P(x)"
 apply(tactic {* lpf_fast_tac forall_lpfs 1 *})
done

(* other rules for universal quantification *)


lemma forall_forall_comm: "forall x:A & (forall' B (P x)) ==> forall y:B &
forall x:A & (P x y)"
 apply(tactic {* lpf_fast_tac forall_lpfs 1 *})
done

lemma forall_forall_elim: "[| a:A; b:B; forall x:A & (forall' B (P x)) |] ==>
(P a b)"
 apply(tactic {* lpf_fast_tac forall_lpfs 1 *})
done

lemma forall_forall_intr:
 assumes p1: "(!!x y.[| x:A; y:B |] ==> (P x y))"
   shows "forall x:A & (forall' B (P x))"
 apply(tactic {* lpf_fast_tac (forall_lpfs addIs [thm "p1"]) 1 *})
done

lemma forall_forall_subs:
 assumes p1: "forall x:A & (forall' B (P x))"
     and p2: "!!x y.[| x:A; y:B; (P x y) |] ==> (Q x y)"
   shows "forall x:A & (forall' B (Q x))"
 apply(cut_tac p1)
 apply(tactic {* lpf_fast_tac (forall_lpfs addIs [thm "p2"]) 1 *})
done

lemma forall_and_dist_con: "(forall' A P) and (forall' A Q) ==> forall x:A &
P(x) and Q(x)"
 apply(tactic {* lpf_fast_tac forall_lpfs 1 *})
done

lemma forall_and_dist_exp: "forall x:A & P(x) and Q(x) ==> (forall' A P) and
(forall' A Q)"
 apply(tactic {* lpf_fast_tac forall_lpfs 1 *})
done

lemma forall_def_or_inh: "[| forall x:A & def P(x); forall x:A & def Q(x) |]
==> forall x:A & def (P(x) or Q(x))"
 apply(tactic {* lpf_fast_tac forall_lpfs 1 *})
done

lemma forall_def_intr_not: "forall x:A & not P(x) ==> forall x:A & def P(x)"
 apply(tactic {* lpf_fast_tac forall_lpfs 1 *})
done

lemma forall_iff_elim_left_def: "forall x:A & P(x) <=> Q(x) ==> forall x:A &
def Q(x)"
 apply(tactic {* lpf_fast_tac forall_lpfs 1 *})
done

lemma forall_iff_elim_right_def: "forall x:A & P(x) <=> Q(x) ==> forall x:A &
def P(x)"
 apply(tactic {* lpf_fast_tac forall_lpfs 1 *})
done
```

```
lemma forall_iff_subs:
 assumes p1: "(forall' A P)"
     and p2: "!!x. x:A ==> P(x) <=> Q(x)"
   shows "(forall' A Q)"
 apply(cut_tac p1)
 apply(tactic {* lpf_fast_tac (forall_lpfs addEs [(thm "p2") RS (thm
"iff_elim")]) 1 *})
done

lemma forall_iff_subs_def:
 assumes p1: "forall x:A & def P(x)"
     and p2: "!!x. x:A ==> P(x) <=> Q(x)"
   shows "forall x:A & def Q(x)"
 apply(cut_tac p1)
 apply(tactic {* lpf_fast_tac (forall_lpfs addEs [(thm "p2") RS (thm
"iff_elim")]) 1 *})
done

lemma forall_imp_subs:
 assumes p1: "(forall' A P)"
     and p2: "!!x. x:A ==> P(x) => Q(x)"
   shows "(forall' A Q)"
 apply(cut_tac p1)
 apply(tactic {* lpf_fast_tac (forall_lpfs addEs [(thm "p2") RS (thm
"imp_elim")]) 1 *})
done

lemma forall_or_intr_left: "(forall' A P) or (forall' A Q) ==> forall x:A &
P(x) or Q(x)"
 apply(tactic {* lpf_fast_tac forall_lpfs 1 *})
done

lemma forall_or_dist_con: "(forall' A Q) ==> forall x:A & P(x) or Q(x)"
 apply(tactic {* lpf_fast_tac forall_lpfs 1 *})
done

lemma forall_or_intr_right: "(forall' A P) ==> forall x:A & P(x) or Q(x)"
 apply(tactic {* lpf_fast_tac forall_lpfs 1 *})
done

lemma forall_to_exists: "[| (forall' A P); inh A |] ==> (exists' A P)"
 apply(tactic {* lpf_fast_tac forall_lpfs 1 *})
done

lemma forall_to_not_exists_dem: "forall x:A & not P(x) ==> not (exists' A P)"
 apply(tactic {* lpf_fast_tac forall_lpfs 1 *})
done

lemma forall_fix: "forall x:A & (forall' A (P x)) ==> forall x:A & (P x x)"
 apply(tactic {* lpf_fast_tac forall_lpfs 1 *})
done

lemma forall_subs:
 assumes p1: "(forall' A P)"
     and p2: "!!x.[| x:A; P(x) |] ==> Q(x)"
   shows "(forall' A Q)"
 apply(cut_tac p1)
 apply(tactic {* lpf_fast_tac (forall_lpfs addIs [thm "p2"]) 1 *})
done

lemma not_forall_intr_not: "[| a:A; P(a) |] ==> not (forall x:A & not P(x))"
 apply(tactic {* lpf_fast_tac forall_lpfs 1 *})
```

```
done

lemma not_forall_to_exists_dem: "not (forall' A P) ==> exists x:A & not P(x)"
 apply(tactic {* lpf_fast_tac forall_lpfs 1 *})
done

lemma def_forall_forall:
 assumes p1: "(!!x y.[| x:A; y:B |] ==> def (P x y))"
   shows "def (forall x:A & (forall' B (P x)))"
 apply(rule def_forall)
 apply(rule def_forall)
 apply(rule p1)
  apply(assumption)
 apply(assumption)
done

end
```

12.4 Cond.thy

```
theory Cond = Pred:

(* Conditional *)

consts
  if'    :: "[ex,ex,ex] => ex"

nonterminals eifs

syntax
  if            :: "[ex,ex,eifs,ex] => ex"
        ("(if _/ then _/ _/else _)" [100,100,100,100] 100)
  eifs_nil  :: "eifs" ("" 100)
  eifs_cons :: "[ex,ex,eifs] => eifs"
        ("(elseif _/ then _/ _)" [100,100,100] 100)

parse_translation
{*
let
   fun eifs_tr (Const("eifs_nil",_)) e = e
     | eifs_tr (Const("eifs_cons",_) $ e1 $ e2 $ eif) e =
         Syntax.const "if'" $ e1 $ e2 $ (eifs_tr eif e);

  fun if_tr [e1,e2,eifs,e4] =
    Syntax.const "if'" $ e1 $ e2 $ (eifs_tr eifs e4);
(*     Const("if'",dummyT) $ e1 $e2 $ (eifs_tr eifs e4); *)
in [("if",if_tr)]
end;
*}

print_translation
{*
let
  fun eifs_tr' (Const("if'",_)$e1$e2$e3) =
      let val (eifs,e) = eifs_tr' e3 in
        (Const("eifs_cons",dummyT)$e1$e2$eifs,e)
      end
    | eifs_tr' e = (Const("eifs_nil",dummyT),e);

  fun if_tr' [e1,e2,e3] =
    let val (eifs,e) = eifs_tr' e3 in
      (Const("if",dummyT)$e1$e2$eifs$e)
    end;
```

```
in [("if'",if_tr')]
end;
*}


(* Conditional axioms, page 280 *)

axioms
  if_false: "[| c:A; not P |] ==> (if P then b else c) = c"
  if_true:  "[| b:A; P |] ==> (if P then b else c) = b"

(* Conditional derived rules, page 280 *)

axioms
  if_true_ident: "[| a:A; b:A |] ==> (if a = a then b else c) = b"
  if_form:       "[| def P; b:A; c:A |] ==> if P then b else c:A"
  if_form_sqt:   "[| def P; P ==> b:A; not P ==> c:A |] ==> if P then b else
c:A"

(* test *)
lemma "if true then false elseif true then false elseif true then false elseif
false then true else false = false"
oops

end
```

12.5 Let.thy

```
theory Let = Cond:

nonterminals lbinds lbind

consts
  let'  :: "[ex,ex=>ex]=>ex"

syntax
  ""         :: "lbind => lbinds"  ("_")
  lbindsn    :: "[lbind, lbinds] => lbinds" ("(_,/ _)")
  lbind      :: "[idt,ex] => lbind"  ("_ = _")
  "_let"     :: "[lbinds, ex] => ex"  ("(let _/ in _)" [100,100] 100)

translations
  "_let (lbindsn lb lbs) e" == "_let lb (_let lbs e)"
translations
  "_let (lbind x e1) e2" => "let' e1 (%x. e2)"

print_translation
{*
let
fun let_tr' [ea,eb] =
  case eb of
    Abs(x1,T1,e1) =>
      let val (x',e') = variant_abs(x1,dummyT,e1) in
        Const("_let",dummyT)$(Const("lbind",dummyT)$Free(x',T1)$ea)$e'
      end |
    _ => let_tr' [ea,Abs("x",Type("ex",[]),eb$(Bound 0))];
in [("let'",let_tr')]
end;

*}

axioms
  let_defn: "[| e:A; e'(e):B |] ==> (let x = e in e'(x)) = e'(e)"
```

```
lemma let_form: "!!e. [| e:A; e'(e):B |]  ==> let x=e in e'(x) : B"
 apply(rule let_defn [THEN [2] eq_subs_left])
 apply(assumption)+
done

end
```

12.6 Union.thy

```
theory Union = Let:

consts
  unionty'  :: "[ty,ty] => ty"  ("_ | _" [120,121] 120)

axioms
  union_elim:
    "[| u : A | B; !!a. a:A ==> P(a);!!b. b:B ==> P(b) |] ==> P(u)"
  union_intr_left:
    "b:B ==> b:A|B"
  union_intr_right:
    "a:A ==> a:A|B"

  (* derived rules *)

  def_eq_intr_gen:
    "[| a:A; b:B |] ==> def a=b"
  union_assoc_left:
    "a: A | B | C ==> a: A | (B | C)"
  union_assoc_right:
    "a: A | (B | C) ==> a: A | B | C"
  union_comm:
    "a : A | B ==> a : B | A"

End
```

12.7 Prod.thy

```
theory Prod = Union:

(* abstract syntax *)

typedecl prodnum'

consts
  pn_zero'  :: "prodnum'"
  pn_succ'  :: "prodnum' => prodnum'"

typedecl prodtys'
typedecl prodexs'

consts
  prodty2'  :: "[ty,ty] => prodtys'"
  prodtyn'  :: "[ty,prodtys'] => prodtys'"

  prod2'    :: "[ex,ex] => prodexs'"
  prodn'    :: "[ex,prodexs'] => prodexs'"

  prodty'   :: "prodtys' => ty"
  prod'     :: "prodexs' => ex"

  sel'      :: "[ex,prodnum'] => ex"
```

```
(* concrete syntax *)

nonterminals prodtys_ prodexs_

syntax
  prodty2_  :: "[ty,ty] => prodtys_"    ("_ * _" [131,131] 131)
  prodtyn_  :: "[ty,prodtys_] => prodtys_"  ("_ * _" [131,131] 131)
  prodty_id_    :: "id => prodtys_"     ("$_")
  prodty_var_   :: "var => prodtys_"        ("$_")

  prod2_    :: "[ex,ex] => prodexs_"    ("_,/_")
  prodn_    :: "[ex,prodexs_] => prodexs_"  ("_,/_")
  prod_id_  :: "id => prodexs_"     ("$_")
  prod_var_ :: "var => prodexs_"        ("$_")

  prodty_   :: "prodtys_ => ty"     ("_" [131] 130)
  prod_     :: "prodexs_ => ex"     ("mk'_'(_')")
  sel_      :: "[ex,xnum] => ex"        ("_._" [500,501] 500)


translations
  "prodtyn_ t ts" => "(prodtyn' t ts)"
  "prodty2_ t1 t2" => "(prodty2' t1 t2)"
  "prodty_ pts" => "(prodty' pts)"
  "prodty_id_ i" => "i"
  "prodty_var_ v" => "v"

  "prodn_ e es" => "(prodn' e es)"
  "prod2_ e1 e2" => "(prod2' e1 e2)"
  "prod_ pes" => "(prod' pes)"
  "prod_id_ i" => "i"
  "prod_var_ v" => "v"

parse_translation
{*
let
(* parse translations for product selection *)

fun digit_is_to_int [d] = d |
    digit_is_to_int (d::ds) = (digit_is_to_int ds * 10) + d;

fun int_to_prodnum 0 = Const("pn_zero'",dummyT)
  | int_to_prodnum n = Const("pn_succ'",dummyT)$(int_to_prodnum (n-1));

fun string_to_prodnum s =
    int_to_prodnum(digit_is_to_int (tl (map (fn c => ord c - 48) (explode
s))));

fun sel_tr [e,t as (Free(s,_))] =
    (Const("sel'",dummyT)$e$(string_to_prodnum s));
in [("sel_",sel_tr)]
end;
*}

print_translation
{*
let

(* print translations for product selection *)

fun prodnum_to_int (Const("pn_zero'",_)) = 0 |
    prodnum_to_int (Const("pn_succ'",_)$e) = prodnum_to_int e + 1;
```

```
fun int_to_string n =
    let val d = n div 10 in
        if d >= 1
        then int_to_string d^(chr((n mod 10)+48))
        else chr(n+48)
    end;

fun prodnum_to_string pn =
    "#"^(int_to_string (prodnum_to_int pn));

fun sel_tr' [e,pn] =
    Const("sel_",dummyT)$e$(Free(prodnum_to_string pn,dummyT));

(* print translations for products *)

fun prodtys_tr' (Const("prodty2'",_)$t1$t2) =
    Const("prodty2_",dummyT)$t1$t2 |
    prodtys_tr' (Const("prodtyn'",_)$t$pts) =
    Const("prodtyn_",dummyT)$t$prodtys_tr' pts |
    prodtys_tr' i = Const("prodty_id_",dummyT)$i;

fun prodty_tr' [pts] = Const("prodty_",dummyT)$prodtys_tr' pts;


fun prods_tr' (Const("prod2'",_)$t1$t2) =
    Const("prod2_",dummyT)$t1$t2 |
    prods_tr' (Const("prodn'",_)$t$ps) =
    Const("prodn_",dummyT)$t$prods_tr' ps |
    prods_tr' i = Const("prod_id_",dummyT)$i;

fun prod_tr' [ps] = Const("prod_",dummyT)$prods_tr' ps;

in [("sel'",sel_tr'),("prodty'",prodty_tr'),("prod'",prod_tr')]
end;
*}

axioms
  (* rules for product constructors *)

  prod2_elim: "[| p:A*B; !!a b.[| a:A; b:B; mk_(a,b)=p |] ==> P(mk_(a,b)) |]
==> P(p)"
  prodn_elim: "[| p:A*$B; !!a b.[| a:A; mk_($b):$B; mk_(a,$b)=p |] ==>
P(mk_(a,$b)) |] ==> P(p)"

  prod2_intr: "[| a:A; b:B |] ==> mk_(a,b): A*B"
  prodn_intr: "[| a:A; mk_($b): $B |] ==> mk_(a,$b): A*$B"
  prod2_dist: "mk_(a1,b1)=mk_(a2,b2) ==> a1=a2 and b1=b2"
  prodn_dist: "mk_(a1,$b1)=mk_(a2,$b2) ==> a1=a2 and mk_($b1)=mk_($b2)"

  prodexs_subs1: "[| prod'(a) = prod'(b); P(b); prod'(b):prodty'(A)  |]  ==>
P(a)"
  prodexs_subs2: "[| prod'(a) = prod'(b); P(a); prod'(a):prodty'(A)  |]  ==>
P(b)"

  (* rules for product destructors *)

  sel1_prod2_defn: "mk_(a,b):A*B ==> mk_(a,b).#1 = a"
  sel2_prod2_defn: "mk_(a,b):A*B ==> mk_(a,b).#2 = b"
  sel1_prodn_defn: "mk_(a,$b):A*$B  ==> mk_(a,$b).#1 = a"
  seln_prodn_defn: "mk_(a,$b):A*$B  ==> (sel' mk_(a,$b) (pn_succ' pn)) = (sel'
mk_($b) pn)"
```

```
ML
{*
  fun bf t i = by (forward_tac [t] i);
*}

lemma type_subs_right: "!!a. [| a:A; a=b |] ==> b:A"
 apply(rule eq_subs_right)
   back
   apply(assumption)+
done

lemma type_subs_left: "!!a. [| a:A; b=a |] ==> b:A"
 apply(rule eq_subs_left)
   back
   apply(assumption)+
done

lemma eq_subs3: "!!a.[| a:A; a=b; P(b) |] ==> P(a)"
 apply(drule type_subs_right)
  apply(assumption)
 apply(rule eq_subs_left,assumption,assumption,assumption)
done

lemma sel1_prodn_form: "!!p. p:A*$B ==> p.#1:A"
 apply(erule prodn_elim)
 apply(rule sel1_prodn_defn [THEN [2] eq_subs_left])
   apply(assumption)
  apply(rule prodn_intr)
   apply(assumption)+
done

lemma prod2_dest1: "!!a. mk_(a,b):A*B ==> a:A"
 apply(erule prod2_elim)
 apply(drule prod2_dist)
 apply(erule and_elim)
 apply(rule eq_subs_right,assumption,assumption,assumption)
done

lemma prod2_dest2: "!!a. mk_(a,b):A*B ==> b:B"
 apply(erule prod2_elim)
 apply(drule prod2_dist)
 apply(erule and_elim)
 apply(rule eq_subs_right,assumption,assumption,assumption)
done

lemma prodn_dest1: "!!a. mk_(a,$b):A*$B ==> a:A"
 apply(rule sel1_prodn_defn [THEN [2] eq_subs_right])
   back
   apply(erule sel1_prodn_form)
  apply(assumption)
 apply(erule sel1_prodn_form)
done

lemma prodn_dest2: "!!a. mk_(a,$b):A*$B ==> mk_($b):$B"
 apply(erule prodn_elim)
 apply(drule prodn_dist)
 apply(erule and_elim)
 apply(rule eq_subs_right,assumption,assumption,assumption)
done

lemma prod3_elim:
 assumes p1: "p:A*B*C"
     and p2: "!!a b c.[| a:A; b:B; c:C;  mk_(a,b,c)=p |] ==> P(mk_(a,b,c))"
```

```
   shows "P(p)"
 apply(rule p1 [THEN prodn_elim])
 apply(erule prod2_elim)
 apply(rule prodexs_subs2,assumption)
  prefer 2
  apply(rule prod2_intr)
   apply(assumption)+
 apply(rule p2)
    apply(assumption)+
 apply(rule prodexs_subs1,assumption)
  apply(assumption)
 apply(rule type_subs_right)
  prefer 2
  apply(assumption)
 apply(rule prod2_intr)
  apply(assumption)+
done

lemma sel1_prod3_defn: "!!a. mk_(a,b,c):A*B*C ==> mk_(a,b,c).#1 = a"
 apply(erule sel1_prodn_defn)
done

lemma sel2_prod3_defn: "!!a. mk_(a,b,c):A*B*C ==> mk_(a,b,c).#2 = b"
 apply(frule prodn_dest1)
 apply(frule prodn_dest2)
 apply(frule prod2_dest1)
 apply(frule prod2_dest2)
 apply(rule seln_prodn_defn [THEN [2] eq_subs_left])
   apply(rule sel1_prod2_defn [THEN [2] eq_subs_left])
     apply(assumption)+
 apply(erule sel1_prod2_defn)
done

lemma sel3_prod3_defn: "!!a. mk_(a,b,c):A*B*C ==> mk_(a,b,c).#3 = c"
 apply(frule prodn_dest1)
 apply(frule prodn_dest2)
 apply(frule prod2_dest1)
 apply(frule prod2_dest2)
 apply(rule seln_prodn_defn [THEN [2] eq_subs_left])
   apply(rule sel2_prod2_defn [THEN [2] eq_subs_left])
     apply(assumption)+
 apply(erule sel2_prod2_defn)
done

lemma prod3_intr: "!!a.[| a:A; b: B; c:C |] ==> mk_(a,b,c):A*B*C"
 apply(erule prodn_intr)
 apply(erule prod2_intr)
 apply(assumption)
done

lemma prod3_defn: "!!p. p:A*B*C ==> mk_(p.#1,p.#2,p.#3) = p"
 apply(erule prod3_elim)
 apply(rule sel1_prod3_defn [THEN [2] eq_subs_left],assumption)
   apply(erule prod3_intr,assumption,assumption)
 apply(rule sel2_prod3_defn [THEN [2] eq_subs_left],assumption)
  apply(erule prod3_intr,assumption,assumption)
 apply(rule sel3_prod3_defn [THEN [2] eq_subs_left],assumption)
  apply(erule prod3_intr,assumption,assumption)
 apply(rule eq_self_intr)
 apply(erule prod3_intr,assumption,assumption)
done

lemma prod3_form: "!!p. p:A*B*C ==> mk_(p.#1,p.#2,p.#3):A*B*C"
```

```
 apply(rule prod3_defn [THEN [2] eq_subs_left])
   apply(assumption)+
done

end
```

12.8 Opt.thy

```
theory Opt = Prod:

consts
  optty'':: "ty => ty"   ("[_]")
  nil   :: "ex"

axioms
  nil_form: "nil:[A]"
  opt_elim: "[| a:[A]; P(nil); !!x. x:A ==> P(x) |] ==> P(a)"
  opt_intr: "a:A ==> a:[A]"

  (* derived rules *)

 opt_Union_ext_left:   "a:[A] ==> a:[B|A]"
  opt_Union_ext_right: "a:[A] ==> a:[A|B]"
  opt_elim_neq_nil:    "[| a:[A]; a <> nil |] ==> a:A"

end
```

12.9 Sub.thy

```
theory Sub = Opt:

consts
  subty'    :: "[ty,ex => ex] => ty"

syntax
  subty_    :: "[tbind,ex] => ty"   ("<< _ | _ >>")

translations
  "<< x:A | e >>" == "(subty' A (%x. e))"

print_translation
{*
let
  fun eta_exp (e as Abs(_,_,_)) = e
    | eta_exp e =Abs("x",dummyT,e$(Bound 0));

  fun quan_tr' r [A,e] =
      let val Abs(x,_,e') = eta_exp e
      val (x',e'') = variant_abs(x,dummyT,e')
      in
      Const(r,dummyT)$(Const("tbind_",dummyT)$Free(x',dummyT)$A)$e''
      end;
in
  [("subty'",quan_tr' "subty_")]
end;
*}

axioms
  sub_elim:      "a:(subty' A P) ==> P(a)"
  sub_intr:      "[| a:A; P(a) |] ==> a:(subty' A P)"
  sub_supertype: "a:(subty' A P) ==> a:A"

  (* derived rules *)
```

```
  seq_elim_bas:         "[| a:A; !!x. x:A ==> P(x) |] ==> P(a)"
  seq_elim_bas2:        "[| a:A; b:B; !!x y.[| x:A; y:B |] ==> (P x y) |] ==> (P
a b)"
  seq_elim_gen:         "[| a:A; P(a); !!x. [| x:A; P(x) |] ==> Q(x) |] ==>
Q(a)"
  seq_elim_gen2:        "[| a:A; b:B; P(a); Q(b);!!x y.[| x:A; y:B; P(x); Q(y)
|] ==> (R x y) |] ==> (R a b)"
  sub_union_ext_left:  "a:(subty' A P) ==> a:(subty' (B|A) P)"
  sub_union_ext_right: "a:(subty' A P) ==> a:(subty' (A|B) P)"
  sub_subs:             "[| a:(subty' A P); !!y. [| y:A; P(y) |] ==> Q(y) |] ==>
a:(sub A Q)"
end
```

12.10 BasTy.thy

```
theory BasTy = Sub:

consts
  intty'          :: "ty"  ("int")
  ratty' :: "ty"  ("rat")
  tokenty'        :: "ty"  ("token")
  charty'         :: "ty"  ("char")
end
```

12.11 NatLPF.thy

```
theory NatLPF = BasTy:


(* Corresponds to section 14.4 Of "Proof in VDM" *)

consts
  natty' :: "ty"                    ("nat")
  nat1ty'        :: "ty"                    ("nat1")

  zero'          :: "ex"
  succ'          :: "ex => ex"              ("succ'(_')")
  pred'          :: "ex => ex"              ("pred'(_')")
  add'           :: "[ex,ex] => ex"         ("(_ +/ _)" [410,411] 410)
  sub'           :: "[ex,ex] => ex"         ("(_ -/ _)" [410,411] 410)
  mult'          :: "[ex,ex] => ex"         ("(_ */ _)" [420,421] 420)
  div'           :: "[ex,ex] => ex"         ("(_ 'div/ _)" [420,421] 420)
  idiv'          :: "[ex,ex] => ex"         ("(2div/ _)" [420] 420)

  lt'            :: "[ex,ex] => ex"         ("(_ </ _)" [310,311] 310)
  leq'           :: "[ex,ex] => ex"         ("(_ <=/ _)" [310,311] 310)
  gt'            :: "[ex,ex] => ex"         ("(_ >/ _)" [310,311] 310)
  geq'           :: "[ex,ex] => ex"         ("(_ >=/ _)" [310,311] 310)


(* concrete syntax for natural number constants *)

nonterminals digit digit is

syntax
  zero_          :: "digit"       ("0")
  one_           :: "digit"       ("1")
  two_           :: "digit"       ("2")
  three_ :: "digit"       ("3")
  four_          :: "digit"       ("4")
  five_          :: "digit"       ("5")
  six_           :: "digit"       ("6")
  seven_ :: "digit"       ("7")
  eight_ :: "digit"       ("8")
  nine_          :: "digit"       ("9")


  ""             :: "digit => digit is"    ("_")
  digit isn_     :: "[digit is,digit] => digit is" ("__")
  natconst_      :: "digit is => ex"               ("_")


parse_translation
{*
let
  fun digit_to_int (Const(s,_)) =
    case s of
         "zero_" => 0 |
         "one_" => 1 |
         "two_" => 2 |
         "three_" => 3 |
         "four_" => 4 |
         "five_" => 5 |
         "six_" => 6 |
         "seven_" => 7 |
         "eight_" => 8 |
         "nine_" => 9;
```

```
  fun digit_is_to_int (Const("digit_isn_",_)$ds$d) = (digit_is_to_int ds * 10)
+ (digit_to_int d)
    | digit_is_to_int d                          = digit_to_int d;

  fun int_to_succ 0 = Const("zero'",dummyT)
    | int_to_succ n = Const("succ'",dummyT)$(int_to_succ (n-1));

  fun digit_is_to_succ ds = int_to_succ(digit_is_to_int ds);

  fun natconst_tr [ds] = digit_is_to_succ ds;
in
  [("natconst_",natconst_tr)]
end;
*}

print_translation
{*
let
  fun zero_tr' [] = Const("natconst_",dummyT)$Const("zero_",dummyT);

  fun int_to_digit n =
    case n of
        0 => Const("zero_",dummyT) |
        1 => Const("one_",dummyT) |
        2 => Const("two_",dummyT) |
        3 => Const("three_",dummyT) |
        4 => Const("four_",dummyT) |
        5 => Const("five_",dummyT) |
        6 => Const("six_",dummyT) |
        7 => Const("seven_",dummyT) |
        8 => Const("eight_",dummyT) |
        9 => Const("nine_",dummyT);

  fun int_to_digit_is n =
    let val d = n div 10 in
        if d >= 1
        then Const("digit_isn_",dummyT)$(int_to_digit_is d)$(int_to_digit (n
mod 10))
        else int_to_digit n
    end;

  fun succ_to_int (Const("succ'",_)$e) = 1 + succ_to_int e
    | succ_to_int (Const("zero'",_)) = 0
    | succ_to_int _ = raise Match;

  fun succ_tr' [e] =
    Const("natconst_",dummyT)$(int_to_digit_is ((succ_to_int e)+1));
in
  [("zero'",zero_tr'),("succ'",succ_tr')]
end;
*}

(* axioms *)

axioms
  zero_form: "0 : nat"
  succ_form: "n : nat ==> succ(n) : nat"

  nat_indn: "[| n : nat; P(0); !!k. [| k : nat; P(k) |] ==> P(succ(k)) |] ==>
P(n)"

  succ_oneone:    "[| n1 : nat; n2 : nat; succ(n1) = succ(n2) |] ==> n1 = n2"
  succ_neq_zero: "n : nat ==> succ(n) <> 0"
```

```
  add_defn_zero_left: "n : nat ==> 0 + n = n"
  add_defn_succ_left: "[| n1 : nat; n2 : nat |] ==> succ(n1) + n2 = succ(n1 +
n2)"

  mult_defn_zero_left: "n : nat ==> 0 * n = 0"
  mult_defn_succ_left: "[| n1 : nat; n2 : nat |] ==> succ(n1) * n2 = n1 * n2 +
n2"
  (* derived *)

  nat_cases: "[| n : nat; P(0); !!k. k : nat ==> P(succ(k)) |] ==> P(n)"

  add_form:                "[| n1 : nat; n2 : nat |] ==> n1 + n2 : nat"
  add_defn_zero_right:     "n : nat ==> n + 0 = n"
  add_defn_succ_right:       "[| n1 : nat; n2 : nat |] ==> n1 + succ(n2) =
succ(n1 + n2)"
  add_defn_succ_left_comm:  "[| n1 : nat; n2 : nat |] ==> succ(n1 + n2) =
succ(n1) + n2"
  add_comm:                "[| n1 : nat; n2 : nat |] ==> n1 + n2 = n2 + n1"
  add_assoc:                "[| n1 : nat; n2 : nat; n3 : nat |] ==> n1 + n2 + n3
= n1 + (n2 + n3)"

  mult_form:  "[| n1 : nat; n2 : nat |] ==> n1 * n2 : nat"
  mult_comm:  "[| n1 : nat; n2 : nat |] ==> n1 * n2 = n2 * n1"
  mult_assoc: "[| n1 : nat; n2 : nat; n3 : nat |] ==> n1 * n2 * n3 = n1 * (n2 *
n3)"


defs
  nat1_def: "nat1 == << n:nat | n <> 0 >>"
  lt_def:   "n < m == m > n"
  gt_def:   "n > m == exists k:nat1 & m + k = n"
  leq_def:  "n <= m == m >= n"
  geq_def:  "n >= m == exists k:nat & m + k = n"

axioms
  nat1_intr:      "[| n : nat; n <> 0 |] ==> n : nat1"
  nat1_elim:      "n : nat1 ==> n <> 0"
  nat1_indn:        "[| n : nat1; P(succ(0)); !!k. [| k : nat1; P(k) |] ==>
P(succ(k)) |] ==> P(n)"
  nat1_supertype: "n : nat1 ==> n : nat"

  lt_not_refl:  "n : nat ==> not (n < n)"
  lt_trans:      "[| n1 : nat; n2 : nat; n3 : nat; n1 < n2; n2 < n3 |] ==> n1 <
n3"
  lt_neq:       "[| n1 : nat; n2 : nat; n1 < n2 |] ==> n1 <> n2"
  lt_tot_ord:   "[| n1 : nat; n2 : nat |] ==> n1 < n2 or n1 = n2 or n2 < n1"
  lt_zero_nat1: "n : nat1 ==> 0 < n"
  lt_zero_nat:  "[| n1 : nat; n2 : nat; n2 < n1 |] ==> 0 < n1"
  lt_nat_succ:   "n : nat ==> n < succ(n)"
  def_lt:        "[| n1 : nat; n2 : nat |] ==> def n1 < n2"

  gt_not_refl: "n : nat ==> not (n > n)"
  gt_trans:     "[| n1 : nat; n2 : nat; n3 : nat; n1 > n2; n2 > n3 |] ==> n1 >
n3"
  gt_succ_inh: "[| n1 : nat; n2 : nat; n2 > n1 |] ==> succ(n1) > succ(n2)"
  gt_tot_ord:  "[| n1 : nat; n2 : nat |] ==> n1 > n2 or n1 = n2 or n2 > n1"
  def_gt:      "[| n1 : nat; n2 : nat |] ==> def n1 > n2"


  leq_succ_defn: "[| n1 : nat; n2 : nat |] ==> n1 = succ(n2) or n1 <= n2 <=> n1
<= succ(n2)"

  geq_zero_intr: "n : nat ==> n = 0 or n > 0"
  geq_succ_intr: "[| m : nat; n : nat; n > m |] ==> n > succ(m) or n = succ(m)"
```

```
  neq_succ: "[| n1 : nat; n2 : nat; n2 <> n1 |] ==> succ(n1) <> succ(n2)"

  not_lt_zero: "n : nat ==> not (n < 0)"
  eq_zero_iff_leq_zero: "n : nat ==> n = 0 <=> n <= 0"
  add_eq_zero_elim: "[| n1 : nat; n2 : nat; n1 + n2 = 0 |] ==> n1 = 0 and n2 =
0"

end
```

12.12 SetLPF.thy

```
theory SetLPF = NatLPF:

(* expression sequences *)

nonterminals exs

syntax
  ""    :: "ex => exs"            ("_")
  exsn  :: "[ex,exs]=> exs"       ("_,/_")

(* set type constructor *)

consts
  setty'        :: "ty => ty"             ("set of _" [140] 140)


(* set membership *)

consts
  inset'        :: "[ex,ex] => ex"        ("(_ in set/ _)" [310,310] 310)

syntax
  ninset'       :: "[ex,ex] => ex"        ("(_ not in set/ _)" [310,310] 310)

translations
  "ninset' a s" == "not (a in set s)"


(* primitive set constructors *)

consts
  emptyset'     :: "ex"                   ("{}")              (* not VDM-SL! *)
  addtoset'     :: "[ex,ex] => ex"        ("add'(_,_')")      (* not VDM-SL! *)


(* set operations *)

consts
  union'        :: "[ex,ex] => ex"        ("(_ union/ _)" [410,411] 410)
  inter'        :: "[ex,ex] => ex"        ("(_ inter/ _)" [420,421] 420)
  dunion'       :: "ex => ex"             ("(2dunion/ _)" [450] 450)
  dinter'       :: "ex => ex"             ("(2dinter/ _)" [450] 450)
  power'        :: "ex => ex"             ("(2power/ _)" [450] 450)
  diff'         :: "[ex,ex] => ex"        ("(_ \\/ _)" [410,411] 410)
  psubset'      :: "[ex,ex] => ex"        ("(_ psubset/ _)" [310,310] 310)
  subset'       :: "[ex,ex] => ex"        ("(_ subset/ _)" [310,310] 310)

  card'         :: "ex => ex"             ("(2card/ _)" [450] 450)


(* set enumeration *)
```

```
syntax
  enumset       :: "exs => ex"            ("{_}")

translations
  "enumset (exsn e es)" == "(add' e (enumset es))"
  "enumset(e)" == "(add' e emptyset')"


(* set binding lists *)

nonterminals sbinds sbind

syntax
  ""             :: "sbind => sbinds"              ("_")
  sbindsn_       :: "[sbind,sbinds] => sbinds"     ("(_,/ _)")
  sbind_ :: "[idt,ty] => sbind"            ("(_ in set/ _)")


(* set bounded quantifiers *)

consts
  existss'      :: "[ex,ex=>ex] => ex"
  foralls'      :: "[ex,ex=>ex] => ex"
  existsls'     :: "[ex,ex=>ex] => ex"
  iotas'        :: "[ex,ex=>ex] => ex"

syntax
  foralls_      :: "[sbinds,ex] => ex"    ("(2forall/ _ &/ _)" [100,100] 100)
  existss_      :: "[sbinds,ex] => ex"    ("(2exists/ _ &/ _)" [100,100] 100)
  existsls_     :: "[sbinds,ex] => ex"    ("(2existsl/ _ &/ _)" [100,100] 100)
  iotas_        :: "[sbind,ex] => ex"     ("(2iota/ _ &/ _)" [100,100] 100)

translations
  "foralls_ (sbindsn_ sb sbs) e" == "foralls_ (sb foralls_ sbs e)"
  "foralls_ (sbind_ x s) e" => "foralls' s (%x. e)"
  "existss_ (sbindsn_ sb sbs) e" == "existss_ sb (existss_ sbs e)"
  "existss_ (sbind_ x s) e" => "existss' s (%x. e)"
  "existsls_ (sbindsn_ sb sbs) e" == "existsls_ sb (existss_ sbs e)"
  "existsls_ (sbind_ x s) e" => "existsls' s (%x. e)"
  "iotas_ (sbind_ x s) e" => "iotas' s (%x. e)"


(* set comprehension *)

consts
  comp'         :: "[ty, ex => ex, ex => ex] => ex"
  comps'        :: "[ex, ex => ex, ex => ex] => ex"

syntax
  comp          :: "[ex,idt,ty,ex] => ex" ("(1{_ |/ _ : _ &/ _})")
  comps         :: "[ex,idt,ex,ex] => ex" ("(1{_ |/ _ in set _ &/ _)}")

  compid         :: "[idt,ty,ex] => ex"     ("(1{(_ :/ _) &/ _})")         (* not
VDM-SL! *)
  compsid        :: "[idt,ex,ex] => ex"     ("(1{(_ in set/ _) &/ _})")    (* not
VDM-SL! *)
  comptr        :: "[ex,idt,ty] => ex"     ("(1{_ |/ (_ :/ _)})")
  compstr       :: "[ex,idt,ex] => ex"     ("(1{_ |/ (_ in set/ _)})")

translations
  "comp e x A P" => "comp' A (%x. e) (%x. P)"
  "comps e x s P"=> "comps' s (%x. e) (%x. P)"
  "compid x A P" => "comp' A (%x. x) (%x. P)"
  "compsid x s P"=> "comps' s (%x. x) (%x. P)"
```

```
  "comptr e x A" => "comp' A (%x. e) (%x. true)"
  "compstr e x s"=> "comps' s (%x. e) (%x. true)"

(* set range *)

syntax
  setrange_      :: "[ex,ex] => ex"        ("{_,...,_}")

(* axioms *)

axioms
  emptyset_form:     "{} : set of A"
  emptyset_is_empty: "a : A ==> a not in set {}"

  add_form: "[| a : A; s : set of A |] ==> add(a,s) : set of A"

  inset_add_defn:
    "[| a : A; b : A; s : set of A |] ==>
     a in set add(b,s) <=> a = b or a in set s"

  set_indn:
    "[| s : set of A; P({});
        !!a s. [| a : A; s' : set of A;
                  P(s'); a not in set s' |] ==> P(add(a,s')) |] ==>
     P(s)"

  existss_elim:
    "[| s : set of A; (existss' s P);
        !!y. [| y : A; y in set s; P(y) |] ==> Q |] ==> Q"
  existss_intr:
    "[| a : A; s : set of A; a in set s; P(a) |] ==> (existss' s P)"

  not_existss_elim:
    "[| a : A; s : set of A; a in set s; not (existss' s P) |] ==> not P(a)"
  not_existss_intr:
    "[| s : set of A; x : A; !!x. x in set s ==> not P(x) |] ==>
     not (existss' s P)"

  iotas_form: "[| s : set of A; (existsls' s P) |] ==> (iotas' s P) : A"
  iotas_intr: "[| s : set of A; (existsls' s P) |] ==> P(iotas' s P)"

  psubset_defn:
    "[| s1 : set of A; s2 : set of A |] ==>
     s1 psubset s2 <=> (forall a in set s & a in set s)"
  subset_defn:
    "[| s1 : set of A; s2 : set of A |] ==>
     s1 subset s2 <=> s1 psubset s2 and s1 <> s2"

  eq_set_defn:
    "[| s1 : set of A; s2 : set of A |] ==>
     s1 = s2 <=> s1 psubset s2 and s2 psubset s1"
  card_defn_emptyset: "card {} = 0"
  card_defn_add:
    "[| a : A; s : set of A; a not in set s |] ==>
     card add(a,s) = succ(card s)"
  union_form:
    "[| s1 : set of A; s2 : set of A |] ==> (un is1 s2) : set of A"
  inset_union_defn:
    "[| a : A; s1 : set of A; s2 : set of A |] ==>
     a in set s1 union s2 <=> a in set s1 or a in set s2"
  inter_form:
    "[| s1 : set of A; s2 : set of A |] ==> s1 inter s2 : set of A"
  inset_inter_defn:
    "[| a : A; s1 : set of A; s2 : set of A |] ==>
```

```
    a in set s1 inter s2 <=> a in set s1 and a in set s2"
  diff_form:
    "[| s1 : set of A; s2 : set of A |] ==> s1 \\ s2 : set of A"
  inset_diff_defn:
    "[| a : A; s1 : set of A; s2 : set of A |] ==>
     a in set s1 \\ s2 <=> a in set s1 and a not in set s2"
  dunion_form:
    "s : set of set of A ==> dunion s : set of A"
  inset_dunion_defn:
    "[| a : A; s : set of set of A |] ==>
     a in set dunion s <=> (existss' s (inset' a))"
  dinter_form:
    "[| s : set of set of A; s <> {} |] ==> dinter s : set of A"
  inset_dinter_defn:
    "[| a : A; s : set of set of A; s <> {} |] ==>
     a in set dinter s <=> (foralls' s (inset' a))"
  power_form:
    "s : set of A ==> power s : set of set of A"
  inset_power_defn:
    "[| s1 : set of A; s2 : set of A |] ==>
     s1 in set power s2 <=> s1 psubset s2"
  set_comp_form:
    "[| forall x:A & def P(x); !!x. [| x : A; P(x) |] ==> f(x) : B;
        exists s:set of B & forall y:A & P(y) => f(y) in set s |] ==>
     (comp' (set of B) f P) : set of B"
  inset_set_comps_defn:
    "[| b : B; forall x:A & def P(x); !!x. [| x : A; P(x) |] ==> f(x) : B;
        exists s:set of B & forall y:A & P(y) => f(y) in set s |] ==>
     b in set (comp' A f P) <=> (exists a:A & P(a) and b = f(a))"
  comps_defn_set:
    "[| s : set of A; forall x:A & def P(x);
        !!x. [| x : A; P(x) |] ==> f(x) : B;
        exists t:set of B & forall y in set s & P(y) => f(y) in set t |] ==>
     (comps' s f P) = (comp' A f P)"

defs
  foralls_def:  "(foralls' s P) == not (exists x in set s & not P(x))"
  existsls_def: "(existsls' s P) == exists x in set s & P(x) and (forall y in
set s & P(y) => y = x)"


parse_translation
{*
let

fun setrange_tr [e1,e2] =
    Const("comp'",dummyT)$
    Const("natty'",dummyT)$
    Abs("x",dummyT,Bound 0)$
    Abs("x",dummyT,
         Const("and'",dummyT)$
         (Const("leq'",dummyT)$e1$(Bound 0))$
         (Const("leq'",dummyT)$(Bound 0)$e2));
in
  [("setrange_",setrange_tr)]
end;
*}

print_translation
{*
let

fun eta_exp (e as Abs(_,_,_)) = e
  | eta_exp e =Abs("x",dummyT,e$(Bound 0));
```

```
fun comp_tr' [A,e,P] =
    let val Abs(n1,_,e') = eta_exp e
        val Abs(n2,_,P') = eta_exp P
        val new = variant (add_term_names (e',add_term_names (P',[])))
    in
        case (A,e',P') of
            (_,_,Const("true'",_)) =>
                let val n' = Free(new n1,dummyT) in
                    Const("comptr",dummyT)$subst_bounds([n'],e')$n'$A
                end |
            (Const("natty'",_),Bound 0,
             Const("and'",_)$
             (Const("leq'",_)$e1$(Bound 0))$
             (Const("leq'",_)$(Bound 0)$e2)) =>
                Const("setrange_",dummyT)$e1$e2 |
            (_,Bound 0,_) =>
                let val n' = Free(new n2,dummyT) in
                    Const("compid",dummyT)$n'$A$subst_bounds([n'],P')
                end |
             _ =>
                let val n' = Free(new n1,dummyT) in
                    Const("comp",dummyT)$
                     subst_bounds([n'],e')$n'$A$subst_bounds([n'],P')
                end
    end;

fun comps_tr' [s,e,P] =
    let val Abs(n1,_,e') = eta_exp e
        val Abs(n2,_,P') = eta_exp P
        val new = variant (add_term_names (e',add_term_names (P',[])))
    in
        case (e',P') of
            (_,Const("true'",_)) =>
                let val n' = Free(new n1,dummyT) in
                    Const("compstr",dummyT)$subst_bounds([n'],e')$n'$s
                end |
            (Bound 0,_) =>
                let val n' = Free(new n2,dummyT) in
                    Const("compsid",dummyT)$n'$s$subst_bounds([n'],P')
                end |
             _ =>
                let val n' = Free(new n1,dummyT) in
                    Const("comps",dummyT)$
                     subst_bounds([n'],e')$n'$s$subst_bounds([n'],P')
                end
    end;

fun squan_tr' r [s,e] =
    let val Abs(x,_,e') = eta_exp e
        val (x',e'') = variant_abs(x,dummyT,e')
    in
        Const(r,dummyT)$(Const("sbind_",dummyT)$Free(x',dummyT)$s)$e''
    end;


in
  [("foralls'",squan_tr' "foralls_"),("existss'",squan_tr' "existss_"),
   ("existsls'",squan_tr' "existsls_"), ("iotas'",squan_tr' "iotas_"),
   ("comp'",comp_tr'),("comps'",comps_tr')]
end;
*)
```

```
axioms
  emptyset_psubset: "s : set of A ==> {} psubset s"
  singleton_compid: "a : A ==> {a} = {x : A & x = a}"
  singleton_form:   "a : A ==> {a} : set of A"


  forall_to_forall:
     "[| s : set of A;
         forall x:A & x in set s => P(x) |] ==>
      (foralls' s P)"
  foralls_elim:
     "[| a : A; s : set of A; a in set s; (foralls' s P) |] ==> P(a)"
  foralls_intr:
     "[| s : set of A;
         !!y. [| y : A; y in set s |] ==> def P(y) |] ==>
      (foralls' s P)"
  foralls_to_forall:
     "[| s : set of A; (foralls' s P) |] ==>
      forall x:A & x in set s => P(x)"

  def_foralls_inh:
     "[| s : set of A;
         !!y. [| y : A; y in set s |] ==> def P(y) |] ==>
      def (foralls' s P)"
  def_existss_inh:
     "[| s : set of A;
         !!x. [| x : A; x in set s |] ==> def P(x) |] ==>
      def (existss' s P)"
  def_inset:
     "[| a : A; s : set of A |] ==> def a in set s"
  def_inter_emptyset:
     "[| s1 : set of A; s2 : set of A |] ==> def s1 inter s2 = {}"
  def_psubset:
     "[| s1 : set of A; s2 : set of A |] ==> def s1 psubset s2"
  def_emptyset:
     "s : set of A ==> def s = {}"


  eq_emptyset_intr:
     "[| s : set of A; !!a. a : A ==> a not in set s |] ==> s = {}"
  eq_set_intr_psubset:
     "[| s1 : set of A;
         s2 : set of A;
         s1 psubset s2;
         s2 psubset s1 |] ==>
      s1 = s2"
  eq_set_intr_sqt:
     "[| s1 : set of A;
         s2 : set of A;
         !!a. [| a : A; a in set s1 |] ==> a in set s2;
         !!b. [| b : A; b in set s2 |] ==> b in set s1 |] ==>
      s1 = s2"


  exists_to_existss:
     "[| s : set of A;
         exists x:A & x in set s and P(x) |] ==>
      (existss' s P)"
  existss_to_exists:
     "[| s : set of A;
         (existss' s P) |] ==>
      exists x:A & x in set s and P(x)"
  existsls_elim:     "[| s : set of A; (existsls' s P);
                         !!y. [| y : A; y in set s; P(y);
```

```
                        forall x in set s & P(x) => x = y |] ==> Q |] ==>
                    Q"
  exists1s_intr:
     "[| a : A; s : set of A; a in set s; P(a);
         forall y in set s & P(y) => y = a |] ==> (exists1s' s P)"


  inset_singleton_elim:
     "[| a : A; b : A; b in set {a} |] ==> b = a"
  inset_singleton_intr:
     "a : A ==> a in set {a}"
  inset_singleton_intr_eq:
     "[| a : A; b = a |] ==> b in set {a}"
  inset_inter_elim:
     "[| a : A; s1 : set of A; s2 : set of A; a in set s1 inter s2 |] ==>
      a in set s1 and a in set s2"
  inset_inter_elim_left:
     "[| a : A; s1 : set of A; s2 : set of A; a in set s1 inter s2 |] ==>
      a in set s2"
  inset_inter_elim_right:
     "[| a : A; s1 : set of A; s2 : set of A; a in set s1 inter s2 |] ==>
      a in set s1"
  inset_inter_I:
        "[| a : A; s1 : set of A; s2 : set of A; a in set s1; a in set s2 |]
==>
         a in set s1 inter s2"
  inset_notinset_contr:
        "[| a : A; b : A; s : set of A; a in set s; b not in set s |] ==> a <>
b"
  inset_or_notinset:
        "[| a : A; s : set of A |] ==> a in set s or a not in set s"
  inset_union_elim:
        "[| a : A; s1 : set of A; s2 : set of A; a in set s1 union s2 |] ==>
         a in set s1 or a in set s2"
  inset_union_intr:
        "[| a : A; s1 : set of A; s2 : set of A; a in set s1 or a in set s2 |]
==>
         a in set s1 union s2"
  inset_union_intr_left:
        "[| a : A; s1 : set of A; s2 : set of A; a in set s2 |] ==>
         a in set s1 union s2"
  inset_union_intr_right:
        "[| a : A; s1 : set of A; s2 : set of A; a in set s1 |] ==>
         a in set s1 union s2"
  inset_dunion_elim:
        "[| a : A; s : set of set of A; a in set dunion s |] ==> (existss' s
(inset' a))"
  inset_dunion_intr:
        "[| a : A; s : set of set of A; (existss' s (inset' a)) |] ==> a in set
dunion s"
  inset_add_elim:
        "[| a : A; b : A; s : set of A; a in set (add b s) |] ==> a = b or a in
set s"
  inset_add_intr:
        "[| a : A; b : A; s : set of A; a = b or a in set s |] ==> a in set
(add b s)"
  inset_add_intr_elem:
        "[| a : A; s : set of A |] ==> a in set (add a s)"
  inset_add_intr_elem_eq:
        "[| a : A; s : set of A; a = b |] ==> a in set (add b s)"
  inset_add_intr_set:
        "[| a : A; b : A; s : set of A; a in set s |] ==> a in set (add a s)"
  inset_cases:
        "[| a : A; s : set of A; a in set s ==> Q; a not in set s ==> Q |] ==>
```

```
Q"
  inset_diff_elim:
        "[| a : A; s1 : set of A; s2 : set of A; a in set s1 \\ s2 |] ==>
         a in set s1 and a not in set s2"
  inset_diff_elim_left:
        "[| a : A; s1 : set of A; s2 : set of A; a in set s1 \\ s2 |] ==> a not
in set s2"
  inset_diff_elim_right:
        "[| a : A; s1 : set of A; s2 : set of A; a in set s1 \\ s2 |] ==> a in
set s1"
  inset_diff_intr:
        "[| a : A; s1 : set of A; s2 : set of A; a in set s1; a not in set s2
|] ==>
         a in set s1 \\ s2"
  inset_set_comp_elim:
        "[| b : B; forall x:A & def P(x); !!x. [| x : A; P(x) |] ==> f(x) : B;
            exists s:set of B & forall y:A & P(y) => f(y) in set s;
            b in set (comp' A f P) |] ==>
          exists a:A & P(a) and b = f(a)"
  inset_set_comp_intr:
        "[| b : B; forall x:A & def P(x); !!x. [| x : A; P(x) |] ==> f(x) : B;
            exists s:set of B & forall y:A & P(y) => f(y) in set s;
            exists a:A & P(a) and b = f(a) |] ==>
          b in set (comp' A f P)"
  inset_set_comp_intr_fun:
        "[| a : A; P(a); forall x:A & def P(x);
            !!x. [| x : A; P(x) |] ==> f(x) : B;
            exists s:set of B & forall y:A & P(y) => f(y) in set s |] ==>
          f(a) in set (comp' A f P)"
  inset_interval_defn:
        "[| i : nat; j : nat; k : nat |] ==> k in set {i,...,j} <=> i <= k and
k <= j"
  inset_those_defn:
        "exists s:set of A & forall y:A & P(y) => y in set s ==>
         a in set {x:A & P(x)} <=> P(a)"
  inset_those_elim:
        "[| a : A; forall x:A & def P(x);
            exists s:set of A & forall y:A & P(y) => y in set s;
            a in set {x:A & P(x)} |] ==>
          P(a)"
  inset_those_intr:
        "[| a : A; P(a); forall x:A & def P(x);
            exists s:set of A & forall y:A & P(y) => y in set s |] ==>
           a in set {x:A & P(x)}"
  inter_singleton_defn_inset:
        "[| a : A; s : set of A; a in set s |] ==> {a} inter s = {a}"
  inter_singleton_emptyset_elim:
        "[| a : A; s : set of A; {a} inter s = {} |] ==> a not in set s"
  inter_union_dist_left:
        "[| s1 : set of A; s2 : set of A; s3 : set of A |] ==>
         s1 inter (s2 union s3) = s1 inter s2 union s1 inter s3"
  inter_union_dist_right:
        "[| s1 : set of A; s2 : set of A; s3 : set of A |] ==>
         (s1 union s2) inter s3 = s1 inter s3 union s2 inter s3"
  inter_union_emptyset_elim_left:
        "[| s1 : set of A; s2 : set of A; s3 : set of A; (s1 union s2) inter s3
= {} |] ==>
         s2 inter s3 = {}"
  inter_union_emptyset_elim_right:
        "[| s1 : set of A; s2 : set of A; s3 : set of A; (s1 union s2) inter s3
= {} |] ==>
         s1 inter s3 = {}"
  inter_union_left_emptyset_intr:
        "[| s1 : set of A; s2 : set of A; s3 : set of A; s1 inter s3 = {};
```

```
            s2 inter s3 = {} |] ==>
          (s1 union s2) inter s3 = {}"
  inter_union_right_emptyset_intr:
        "[| s1 : set of A; s2 : set of A; s3 : set of A; s1 inter s2 = {};
            s1 inter s3 = {} |] ==>
         s1 inter (s2 union s3) = {}"
  inter_assoc:
        "[| s1 : set of A; s2 : set of A; s3 : set of A |] ==>
         s1 inter s2 inter s3 = s1 inter (s2 inter s3)"
  inter_Comm:
        "[| s1 : set of A; s2 : set of A |] ==> s1 inter s2 = s2 inter s1"
  inter_comp:
        "[| s1 : set of A; s2 : set of A |] ==>
         s1 inter s2 = {x : A & x in set s1 and x in set s2 }"
  inter_defn_emptyset_right:
        "s : set of A ==> s inter {} = {}"
  inter_add_intr_inset:
        "[| a : A; s1 : set of A; s2 : set of A; a in set s2 |] ==>
         add(a,s1) inter s2 = add(a,s1 inter s2)"
  inter_add_intr_notinset:
        "[| a : A; s1 : set of A; s2 : set of A; a not in set s2 |] ==>
         add(a,s1) inter s2 = s1 inter s2"
  inter_intr_right_psubset:
        "[| s1 : set of A; s2 : set of A; s1 psubset s2 |] ==>
         s1 inter s2 = s1"
  inter_self:
        "s : set of A ==> s inter s = s"
  not_add_psubset_intr_elem:
        "[| a : A; s1 : set of A; s2 : set of A; a not in set s2 |] ==>
         not add(a,s1) psubset s2"
  not_add_psubset_intr_set:
        "[| a : A; s1 : set of A; s2 : set of A; not s1 psubset s2 |] ==>
         not add(a,s1) psubset s2"
  ninset_singleton_intr:
        "[| a : A; b : B; a <> b |] ==> a not in set {b}"
  notinset_inter_elim:
        "[| a : A; s1 : set of A; s2 : set of A; a not in set s1 inter s2 |]
==>
         a not in set s1 or a not in set s2"
  notinset_inter_intr_left:
        "[| a : A; s1 : set of A; s2 : set of A; a not in set s2 |] ==>
         a not in set s1 inter s2"
  notinset_inter_intr_right:
        "[| a : A; s1 : set of A; s2 : set of A; a not in set s1 |] ==>
         a not in set s1 inter s2"
  notinset_union_elim:
        "[| a : A; s1 : set of A; s2 : set of A; a not in set s1 union s2 |]
==>
         a not in set s1 and a not in set s2"
  notinset_union_elim_left:
        "[| a : A; s1 : set of A; s2 : set of A; a not in set s1 union s2 |]
==>
         a not in set s2"
  notinset_union_elim_right:
        "[| a : A; s1 : set of A; s2 : set of A; a not in set s1 union s2 |]
==>
         a not in set s1"
  notinset_union_intr:
        "[| a : A; s1 : set of A; s2 : set of A; a not in set s1; a not in set
s2 |] ==>
         a not in set s1 union s2"
  notinset_add_elim:
        "[| a : A; b : A; s : set of A; a not in set add(b,s) |] ==>
         a <> b and a not in set s"
```

```
  notinset_add_elim_left:
        "[| a : A; b : A; s : set of A; a not in set add(b,s) |] ==> a not in
set s"
  notinset_add_elim_right:
        "[| a : A; b : A; s : set of A; a not in set add(b,s) |] ==> a <> b"
  notinset_add_intr:
        "[| a : A; b : A; s : set of A; a <> b; a not in set s |] ==>
         a not in set add(b,s)"
  notinset_diff_elim:
        "[| a : A; s1 : set of A; s2 : set of A; (Nin a (s1 \\ s2)) |] ==>
         a not in set s1 or a in set s2"
  notinset_diff_intr_left:
        "[| a : A; s1 : set of A; s2 : set of A; a in set s2 |] ==> a not in
set s1 \\ s2"
  notinset_diff_intr_right:
        "[| a : A; s1 : set of A; s2 : set of A; a in set s1 |] ==> a not in
set s1 \\ s2"
  notinset_psubset_intr:
        "[| a : A; s1 : set of A; s2 : set of A; s1 psubset s2; a not in set s2
|] ==>
         a in set s1"
  notinset_those_elim:
        "[| a : A; forall x:A & def P(x);
            exists s:set of A & forall y:A & P(y) => y in set s;
            a not in set {x:A & P(x)} |] ==>
         not P(a)"
  notinset_those_intr:
        "[| a : A; not P(a); forall x:A & def P(x);
            exists s:set of A & forall y:A & P(y) => y in set s |] ==>
         a not in set {x:A & P(x)}"
  psubset_add_add_intr:
        "[| a : A; b : A; s : set of A |] ==>
         add(a,add(b,s)) psubset add(b,add(a,s))"
  psubset_elim:
        "[| a : A; s1 : set of A; s2 : set of A; a in set s1; s1 psubset s2 |]
==>
         a in set s2"
  psubset_intr:
        "[| s1 : set of A; s2 : set of A;
           !!a. [| a : A; a in set s1 |] ==> a in set s2 |] ==>
         s1 psubset s2"
  psubset_self:
        "s : set of A ==> s psubset s"
  psubset_trans:
        "[| s1 : set of A; s2 : set of A; s3 : set of A; s1 psubset s2; s2
psubset s3 |] ==>
         s1 psubset s3"
  union_inter_dist_left:
        "[| s1 : set of A; s2 : set of A; s3 : set of A |] ==>
         s1 union s2 inter s3 = (s1 union s2) inter (s1 union s3)"
  union_inter_dist_right:
        "[| s1 : set of A; s2 : set of A; s3 : set of A |] ==>
         s1 inter s2 union s3 = (s1 union s3) inter (s2 union s3)"
  union_add_left_intr:
        "[| a : A; s1 : set of A; s2 : set of A |] ==>
         add(a,s1) union s2 = add(a,s1 union s2)"
  union_add_right_intr:
        "[| a : A; s1 : set of A; s2 : set of A |] ==>
         s1 union add(a,s2) = add(a,s1 union s2)"
  union_assoc:
        "[| s1 : set of A; s2 : set of A; s3 : set of A |] ==>
         s1 union s2 union s3 = s1 union (s2 union s3)"
  union_comm:
        "[| s1 : set of A; s2 : set of A |] ==> s1 union s2 = s2 union s1"
```

```
  union_compid:
        "[| s1 : set of A; s2 : set of A |] ==>
         s1 union s2 = {x : A & x in set s1 or x in set s2 }"
  union_defn_emptyset_left:
        "s : set of A ==> {} union s = s"
  union_defn_emptyset_left_rev:
        "s : set of A ==> s = {} union s"
  union_defn_emptyset_right:
        "s : set of A ==> s union {} = s"
  union_defn_emptyset_right_rev:
        "s : set of A ==> s = s union {}"
  union_defn_those:
        "[| forall x:A & def P(x); forall w:A & def Q(w);
            exists s:set of A & forall y:A & P(y) => y in set s;
            exists t:set of A & forall z:A & P(z) => z in set t |] ==>
         {z : A & P(z) or Q(z) } = {x:A & P(x)} union {x:A & Q(x)}"
  union_intr_left_psubset:
        "[| s1 : set of A; s2 : set of A; s1 psubset s2 |] ==> s1 union s2 =
s2"
  union_psubset:
        "[| s1 : set of A; s2 : set of A; s3 : set of A; s1 psubset s3;
            s2 psubset s3 |] ==>
         s1 union s2 psubset s3"
  union_self:
        "s : set of A ==> s union s = s"
  dunion_union_dist:
        "[| s1 : set of set of A; s2 : set of set of A |] ==>
         dunion (s1 union s2) = dunion s1 union dunion s2"
  dunion_comp:
        "s : set of set of A ==> dunion s = {x : A & (existss' s (inset' x)) }"
  dunion_defn_emp:
        "dunion {} = {}"
  dunion_defn_one:
        "s : set of A ==> dunion {s} = s"
  dunion_defn_add:
        "[| s1 : set of A; s2 : set of set of A |] ==>
         dunion add(s1,s2) = s1 union dunion s2"
  add_psubset_elim_left:
        "[| a : A; s1 : set of A; s2 : set of A; add(a,s1) psubset s2 |] ==>
         s1 psubset s2"
  add_psubset_elim_right:
        "[| a : A; s1 : set of A; s2 : set of A; add(a,s1) psubset s2 |] ==>
         a in set s2"
  add_psubset:
        "[| a : A; s1 : set of A; s2 : set of A; a in set s2; s1 psubset s2 |]
==>
         add(a,s1) psubset s2"
  add_to_uni:
        "[| a : A; s : set of A |] ==> add(a,s) = {a} union s"
  add_abs:
        "[| a : A; s : set of A |] ==> add(a,add(a,s)) = add(a,s)"
  add_add_form:
        "[| a : A; b : A; s : set of A |] ==> add(a,add(b,s)) : set of A"
  add_comm:
        "[| a : A; b : A; s : set of A |] ==> add(a,add(b,s)) =
add(b,add(a,s))"
  add_comp:
        "[| a : A; s : set of A |] ==> add(a,s) = {x : A & x = a or x = s }"
  add_diff_psubset:
        "[| a : A; s1 : set of A; s2 : set of A |] ==>
         add(a,s1) \\ s2 psubset add(a,s1 \\ s2)"
  add_red:
        "[| a : A; s : set of A; a in set s |] ==> add(a,s) = s"
```

```
  card_eq_zero_intr:
        "s = {} ==> card s = 0"
  card_noteq_zero_elim:
        "[| s : set of A; card s <> 0 |] ==> s <> {}"
  card_defn_uni:
        "[| s1 : set of A; s2 : set of A |] ==>
         card (s1 union s2) = add(card s1,card s2) - card (s1 inter s2)"
  card_form:
        "s : set of A ==> card s : nat"
  diff_eq_emptyset_defn:
        "[| s1 : set of A; s2 : set of A |] ==> s1 \\ s2 = {} <=> s1 psubset
s2"
  diff_inter_DeM:
        "[| s1 : set of A; s2 : set of A |] ==>
         s1 \\ s2 inter s3 = s1 \\ s2 union (s1 \\ s3)"
  diff_inter_I:
        "[| s1 : set of A; s2 : set of A; s3 : set of A |] ==>
         (s1 \\ s2) inter s3 = s1 inter s3 \\ s2"
  diff_psubset:
        "[| s1 : set of A; s2 : set of A; s3 : set of A |] ==> s1 \\ s2 psubset
s1"
  diff_union_DeM:
        "[| s1 : set of A; s2 : set of A; s3 : set of A |] ==>
         s1 \\ (s2 union s3) = s1 \\ s2 union (s1 \\ s3)"
  diff_add_intr_inset:
        "[| a : A; s1 : set of A; s2 : set of A; a in set s2 |] ==>
         add(a,s1) \\ s2 = s1 \\ s2"
  diff_add_intr_notinset:
        "[| a : A; s1 : set of A; s2 : set of A; a not in set s2 |] ==>
         add(a,s1) \\ s2 = add(a,s1 \\ s2)"
  diff_comp:
        "[| s1 : set of A; s2 : set of A |] ==>
         s1 \\ s2 = {x : A & x in set s1 and x not in set s2 }"
  diff_defn_emptyset_left:
        "s : set of A ==> {} \\ s = {}"
  diff_defn_emptyset_right:
        "s : set of A ==> s \\ {} = s"
  diff_intr_psubset:
        "[| s1 : set of A; s2 : set of A; s1 psubset s2 |] ==> s1 \\ s2 = {}"
  diff_self:
        "s : set of A ==> s \\ s = {}"
  finset_image:
        "[| s : set of A; !!x. x : A ==> f(x) : B |] ==>
         exists t:set of B & forall x in set s & f(x) in set t"
  inh_to_emptyset:
        "[| a : A; s : set of A; a in set s |] ==> s <> {}"
  ini_setrange_one_form:
        "n : nat ==> {succ(0),...,n} : set of nat1"
  ini_setrange_form:
        "n : nat ==> {0,...,n} : set of nat1"
  setrange_diff_defn:
        "[| i : nat; j : nat |] ==> {succ(i),...,j} = {0,...,j} \\ {0,...,i}"
  setrange_emp:
        "[| i : nat; j : nat; j < i |] ==> {i,...,j} = {}"
  setrange_fin:
        "[| i : nat; j : nat |] ==>
         exists s:set of nat & forall y:nat & i <= y and y <= j => y in set s"
  setrange_form:
        "[| i : nat; j : nat |] ==> {i,...,j} : set of nat"
  noteq_set_inh:
        "[| s : set of A; s <> {} |] ==> exists a:A & a in set s"
  power_comp:
        "s : set of A ==> power s = {t : set of A & t psubset s}"
  set_Un_ext_left:
```

```
      "s : set of A ==> s : set of (B | A)"
  set_Un_ext_right:
      "s : set of A ==> s : set of (A | B)"
  set_comp_form_set_ident:
      "[| s : set of A; !!x. [| x : A; x in set s |] ==> f(x) : B;
          exists t:set of B & forall y in set s & f(y) in set t |] ==>
       {f(x) | x in set s} : set of B"
  set_comp_form_set_ident_global:
      "[| s : set of A; !!x. x : A ==> f(x) : B |] ==> {f(x) | x in set s} :
set of B"
  set_comp_rewrite:
      "[| forall x:A & P(x) <=> Q(x); !!x. [| x : A; P(x) |] ==> f(x) : B;
          exists s:set of B & forall y:A & P(y) => f(y) in set s |] ==>
       {f(x) | x:A & P(x)} = {f(x) | x:A & Q(x)}"
  set_image_form:
      "[| s : set of A; !!x. x : A ==> f(x) : B |] ==> {f(x) | x in set s} :
set of B"
  those_eq_form:
      "a : A ==> {x : A & x = a} : set of A"
  those_inset_form:
      "s : set of A ==> {e : A & e in set s } : set of A"
  those_or_form:
      "[| forall x:A & def P(x); forall x:A & def Q(x);
          exists s:set of A & forall y:A & P(y) => y in set s;
          exists t:set of A & forall z:A & P(z) => z in set t |] ==>
       {x : A & P(x) or Q(x) } : set of A"
  those_to_emptyset:
      "forall x:A & not P(x) ==> {x:A & P(x)} = {}"
  those_form:
      "[| forall x:A & def P(x);
          exists s:set of A & forall y:A & P(y) => y in set s |] ==>
       {x:A & P(x)} : set of A"
  those_form_inset_notinset:
      "[| s1 : set of A; s2 : set of A |] ==>
       {x : A & x in set s1 and x not in set s2 } : set of A"
  those_form_rewrite:
      "[| forall y:A & P(y) <=> Q(y);
          exists s:set of A & forall y:A & P(y) => y in set s |] ==>
       {x:A & Q(x)} : set of A"
  those_intr:
      "s : set of A ==> s = {e : A & e in set s }"
  those_rewrite:
      "[| forall y:A & P(y) <=> Q(y);
         exists s:set of A & forall y:A & P(y) => y in set s |] ==>
       {x:A & P(x)} = {x:A & Q(x)}"
  those_weaken:
      "[| forall x:A & def P(x); forall w:A & def Q(w);
          exists s:set of A & forall y:A & P(y) => y in set s;
          exists t:set of A & forall z:A & Q(z) => z in set t;
          !!x. [| x : A; P(x) |] ==> Q(x) |] ==>
       {x:A & P(x)} psubset {x:A & Q(x)}"
  psubset_inter_left:
      "[| s1 : set of A; s2 : set of A |] ==> s1 inter s2 psubset s1"
  psubset_inter_right:
      "[| s1 : set of A; s2 : set of A |] ==> s1 inter s2 psubset s2"
  eq_inter_diff_diff:
      "[| s1 : set of A; s2 : set of A |] ==> s1 inter (s1 \\ s2) = s1 \\ s2"
  eq_inter_diff_emptyset:
      "[| s1 : set of A; s2 : set of A |] ==> s1 inter (s2 \\ s1) = {}"
  psubset_union_left:
      "[| s1 : set of A; s2 : set of A |] ==> s1 psubset s1 union s2"
  psubset_union_right:
      "[| s1 : set of A; s2 : set of A |] ==> s2 psubset s1 union s2"
  eq_union_diff_union:
```

```
        "[| s1 : set of A; s2 : set of A |] ==> s1 union (s2 \\ s1) = s1 union
s2"
  eq_diff_diff_inter:
        "[| s1 : set of A; s2 : set of A |] ==> s1 \\ (s1 \\ s2) = s1 inter s2"


End
```

12.13 MapLPF.thy

```
theory MapLPF = SetLPF:

(* General apply - will also be used for sequences and functions *)

(* abstract syntax *)

typedecl appexs'

consts
  appl'     :: "ex => appexs'"
  appn'     :: "[ex,appexs'] => appexs'"
  app'          :: "[ex,appexs'] => ex"

(* concrete syntax *)

nonterminals appexs_

(* since appexs is not in class logic the concrete syntax does not allow
   things like f@((a,b)) or f@(a,(b,c))
*)

syntax
  appl_     :: "ex => appexs_"      ("_")
  appn_     :: "[ex,appexs_] => appexs_"    ("_,/_")
  app_          :: "[ex,appexs_] => ex"         ("_@'(_')" [500,501] 500)

translations
  "appn_ e es" == "appn' e es"
  "appl_ e" == "appl' e"
  "app_ e es" == "app' e es"

(* Finite Maps: section 14.6, pages 302-314 in "Proof in VDM" *)

consts
  mapty'        :: "[ty,ty] => ty"        ("map _ to _" [110,110] 110)
  inmap'        :: "[ty,ty] => ty"        ("inmap _ to _" [110,110] 110)

(* map constructors *)

  emptymap'     :: "ex"                   ("{|->}")
  addtomap'     :: "[ex,ex,ex] => ex"     ("madd'(_,_,_')")

(* map operations *)

  inverse'      :: "ex => ex"             ("inverse _" [430] 430)
  merge'        :: "ex => ex"             ("merge _" [450] 450)

  dom'          :: "ex => ex"             ("dom _" [450] 450)
  rng'          :: "ex => ex"             ("rng _" [450] 450)

  domsub'       :: "[ex,ex] => ex"        ("(_ <-:/ _)" [440,441] 440)
  domres'       :: "[ex,ex] => ex"        ("(_ <:/ _)"  [440,441] 440)
```

```
  rngsub'       :: "[ex,ex] => ex"        ("(_ :->/ _)" [440,441] 440)
  rngres'       :: "[ex,ex] => ex"        ("(_ :>/ _)" [440,441] 440)

  munion'       :: "[ex,ex] => ex"        ("(_ munion/ _)" [410,411] 410)
  mcompose'     :: "[ex,ex] => ex"        ("(_ comp/ _)" [611,610] 610)
  modify'       :: "[ex,ex] => ex"        ("(_ ++/ _)" [410,411] 410)

(* maplet lists*)

nonterminals maplet maplets

syntax
  ""            :: "maplet => maplets"            ("_")
  mapletsn_     :: "[maplet,maplets] => maplets"  ("_,/ _")
  maplet_       :: "[ex,ex] => maplet"            ("_ |-> _")


(* map enumeration *)

syntax
  enummap_      :: "maplets => ex"        ("{_}")

translations
  "enummap_ (mapletsn_ (maplet_ e1 e2) mls)" == "madd(e1,e2,enummap_(mls))"
  "enummap_ (maplet_ e1 e2)" == "madd(e1,e2,{|->})"


(* map comprehension *)

consts
  mcomp'        :: "[ty,ex=>ex,ex=>ex,ex=>ex] => ex"
  mcomps'       :: "[ex,ex=>ex,ex=>ex,ex=>ex] => ex"

syntax
  mcomp_        :: "[maplet,tbind,ex] => ex" ("(1{_ |/ _ &/ _})")
  mcomps_       :: "[maplet,sbind,ex] => ex" ("(1{_ |/ _ &/ _})")

  mcomptr_      :: "[maplet,tbind] => ex"    ("(1{_ |/ _})")
  mcompstr_     :: "[maplet,sbind] => ex"    ("(1{_ |/ _})")


translations
  "mcomp_ (maplet_ e1 e2) (tbind_ x A) P" => "mcomp' A (%x. e1) (%x. e2) (%x.
P)"
  "mcomps_ (maplet_ e1 e2) (sbind_ x s) P"=> "mcomps' s (%x. e1) (%x. e2) (%x.
P)"

  "mcomptr_ (maplet_ e1 e2) (tbind_ x A)" => "mcomp' A (%x. e1) (%x. e2) (%x.
true)"
  "mcompstr_ (maplet_ e1 e2) (sbind_ x s)" => "mcomps' s (%x. e1) (%x. e2) (%x.
true)"

consts
  compatible        :: "[ex,ex] => ex"    (* not VDM-SL! *)

  onetoone        :: "ex => ex"  (* one-to-one map *)


(* axioms *)
axioms
  emptymap_form: "{|->} : map A to B"

  madd_form:
```

```
  "[| a : A; b : B; m : map A to B |] ==> madd(a,b,m) : map A to B"
madd_modify:
  "[| a : A; b1 : B; b2 : B; m : map A to B |] ==>
   madd(a,b1,madd(a,b2,m)) = madd(a,b2,m)"
madd_comm:
  "[| a : A; b : B; c : A; d : B; m : map A to B; a <> c |] ==>
       madd(a,b,madd(c,d,m)) = madd(c,d,madd(a,b,m))"

map_indn:
  "[| m0 : map A to B; P({|->});
      !!a b m. [| a : A; b : B;
                  m : map A to B; P(m); a not in set dom m |] ==>
      P(madd(a,b,m)) |] ==>
   P(m0)"

dom_defn_emptymap: "dom {|->} = {}"
dom_defn_madd:
  "[| a : A; b : B; m : map A to B |] ==> dom madd(a,b,m) = add(a,dom m)"

rng_defn_emptymap: "rng {|->} = {}"
rng_defn_madd_ninset:
  "[| a : A; b : B; m : map A to B; a not in set dom m |] ==>
   rng madd(a,b,m) = add(b,rng m)"

domsub_defn_emptymap: "s : set of A ==> s <-: {|->} = {|->}"
domsub_defn_madd_ninset:
  "[| a : A; b : B; m : map A to B; s : set of A; a not in set s |] ==>
   s <-: madd(a,b,m) = madd(a,b,s <-: m)"

domres_defn_emptymap: "s : set of A ==> s <: {|->} = {|->}"
domres_defn_madd_ninset:
  "[| a : A; b : B; m : map A to B; s : set of A; a not in set s |] ==>
   s <: madd(a,b,m) = s <: m"

rngsub_defn_emptymap: "s : set of A ==> {|->} :-> s = {|->}"
rngsub_defn_madd_inset:
  "[| a : A; b : B; m : map A to B; s : set of A; b in set s |] ==>
   madd(a,b,m) :-> s = {a} <-: (m :-> s)"
rngsub_defn_madd_ninset:
  "[| a : A; b : B; m : map A to B; s : set of A; b not in set s |] ==>
   madd(a,b,m) :-> s = madd(a,b,m :-> s)"

rngres_defn_emptymap: "s : set of A ==> {|->} :> s = {|->}"
rngres_defn_madd_inset:
  "[| a : A; b : B; m : map A to B; s : set of A; b in set s |] ==>
   madd(a,b,m) :-> s = madd(a,b,m :> s)"
rngres_defn_madd_ninset:
  "[| a : A; b : B; m : map A to B; s : set of A; b not in set s |] ==>
   madd(a,b,m) :> s = {a} <-: (m :> s)"

at_defn_madd_eq:
  "[| a : A; b : B; m : map A to B |] ==> madd(a,b,m)@(a) = b"
at_defn_madd_neq:
  "[| a : A; b : B; c : A; m : map A to B; c <> a; c in set dom m |] ==>
   madd(a,b,m)@(c) = m@(c)"

eq_map_defn:
  "[| m1 : map A to B; m2 : map A to B; dom m1 = dom m2;
      forall a in set dom m1 & m1@(a) = m2@(a) |] ==>
   m1 = m2"

modify_defn_emptymap_right:
  "m : map A to B ==> m ++ {|->} = m"
```

```
  modify_defn_madd:
    "[| a : B; b : B; m1 : map A to B; m2 : map A to B |] ==>
     m1 ++ madd(a,b,m2) = madd(a,b,m1 ++ m2)"

  compatible_defn:
    "[| m1 : map A to B; m2 : map A to B |] ==>
       (compatible m1 m2) <=>
     (forall a in set dom m1 inter dom m2 & m1@(a) = m2@(a))"

  onetoone_defn:
    "m : map A to B ==>
     onetoone(m) <=>
     (forall x in set dom m & forall y in set dom m & m@(x) =
      m@(y) => x = y)"

  comp_defn_emptymap: "m : map A to B ==> m comp {|->} = {|->}"
  comp_defn_madd:
    "[| a : A; b : B; m1 : map A to B; m2 : map A to B;
        rng m2 psubset dom m1; a not in set dom m2;
        b not in set dom m1 |] ==>
     m1 comp madd(a,b,m2) = madd(a,m1@(b),m1 comp m2)"

  merge_defn_emptymap: "merge {} = {|->}"
  merge_defn_madd:
    "[| m : map A to B; s : set of (map A to B);
        forall m1 in set add(m,s) & (foralls' (add(m,s)) (compatible m1)) |]
==>
     merge add(m,s) = m ++ merge s"

  inverse_defn:
    "m : map A to B ==> inverse m = {m@(a) |-> a | a in set dom m}"

  compm_form:
    "[| forall x : A & def P(x); !!x. [| x : A; P(x) |] ==> f(x) : B;
        !!x. [| x : A; P(x) |] ==> g(x) : C;
        exists s : set of B & forall y : A & P(y) => f(y) in set s;
        forall a1 : A, a2 : A &
               P(a1) and (P(a2) and f(a1) = f(a2)) => g(a1) = g(a2) |] ==>
     {f(x) |-> g(x) | x : A & P(x)} : map B to C"

  dom_defn_compm:
    "[| forall x : A & def P(x); !!x. [| x : A; P(x) |] ==> f(x) : B;
        !!x. [| x : A; P(x) |] ==> g(x) : C;
        exists s : set of B & forall y : A & P(y) => f(y) in set s;
        forall a1 : A, a2 : A &
               P(a1) and (P(a2) and f(a1) = f(a2)) => g(a1) = g(a2) |] ==>
      dom {f(x) |-> g(x) | x : A & P(x)} = (comp' A f P)"

  at_defn_compm:
    "[| b : B; forall x : A & def P(x);
        !!x. [| x : A; P(x) |] ==> f(x) : B;
        !!x. [| x : A; P(x) |] ==> g(x) : C;
        exists s : set of B & forall y : A & P(y) => f(y) in set s;
        forall a1 : A, a2 : A &
               P(a1) and (P(a2) and f(a1) = f(a2)) => g(a1) = g(a2);
        b in set dom {f(x) |-> g(x) | x : A & P(x)} |] ==>
      {f(x) |-> g(x) | x : A & P(x)}@(b) =
      (iota c : C & forall x : A & P(x) and b = f(x) => c = g(x))"

  compm_defn_set:
    "[| s : set of A; forall x : A & def P(x);
        !!x. [| x : A; P(x) |] ==> f(x) : B;
        !!x. [| x : A; P(x) |] ==> g(x) : C;
```

```
        exists s : set of B & forall y : A & P(y) => f(y) in set s;
        forall a1 : A, a2 : A &
               P(a1) and (P(a2) and f(a1) = f(a2)) => g(a1) = g(a2) |] ==>
    dom {f(x) |-> g(x) | x in set s & P(x)} =
    {f(x) |-> g(x) | x : A & x in set s and P(x)}"

(* definitions *)

defs
  inmap_def: "inmap A to B == << x : map A to B | onetoone(x) >>"

(* derived rules *)

axioms
  emptymap_onetoone:   "onetoone({|->})"
  emptymap_dom_disj:   "m : map A to B ==> dom {|->} inter dom m = {}"
  emptymap_form_inmap: "{|->} : inmap A to B"

  msingleton_form: "[| a : A; b : B |] ==> {a |-> b} : map A to B"

  comp_form:
    "[| m1 : map B to C; m2 : map A to B; rng m2 psubset rng m1 |] ==>
     m1 comp m2 : map A to C"

  def_emptymap: "m : map A to B ==> def m = {|->}"
  def_inset_dom_madd_intr:
    "[| a1 : A; a2 : A; b : B; m : map A to B |] ==>
     def a1 in set dom madd(a2,b,m)"
  def_inset_rng_intr: "[| b : B; m : map A to B |] ==> def b in set rng m"
  def_inset_dom_intr: "[| a : A; m : map A to B |] ==> def a in set dom m"
  def_compatible:
    "[| m1 : map A to B; m2 : map A to B |] ==> def (compatible m1 m2)"
  def_onetoone: "m : map A to B ==> def onetoone(m)"
  def_onetoone_pred:
    "[| a : A; b : A; m : map A to B |] ==>
     def (a in set dom m and b in set dom m and m@(a) = m@(b))"

  domres_form: "[| m : map A to B; s : set of A |] ==> s <: m : map A to B"

  domsub_form: "[| m : map A to B; s : set of A |] ==> s <-: m : map A to B"
  domsub_defn: "m : map A to B ==> {} <-: m = m"
  domsub_defn_singleton_ninset:
    "[| a : A; m : map A to B; a not in set dom m |] ==> {a} <-: m = m"
  domsub_defn_madd_singleton_eq:
    "[| a : A; b : B; m : map A to B |] ==>{a} <-: madd(a,b,m) = {a} <-: m"
  domsub_defn_madd_singleton_neq:
    "[| a1 : A; a2 : A; b : B; m : map A to B; a1 <> a2 |] ==>
     {a} <-: madd(a2,b,m) = madd(a2,b,{a} <-: m)"

  inset_dom_madd_elim:
    "[| a1 : A; a2 : A; b : B; m : map A to B;
        a1 in set dom madd(a2,b,m) |] ==>
     a1 = a2 or a1 in set dom m"
  inset_dom_madd_elim_ninset:
    "[| a1 : A; a2 : A; b : B; m : map A to B;
        a1 in set dom madd(a2,b,m); a2 not in set dom m |] ==>
     a1 = a2 or a1 in set dom m and (Neq a2 a1)"
  inset_dom_madd_intr_elem:
    "[| a : A; b : B; m : map A to B |] ==> a in set dom madd(a,b,m)"
  inset_dom_madd_intr_map:
    "[| a1 : A; a2 : A; b : B; m : map A to B; a1 in set dom m |] ==>
     a1 in set dom madd(a2,b,m)"
  inset_dom_compm_intr_fun:
```

```
  "[| a : A; P(a); forall x : A & def P(x);
      !!x. [| x : A; P(x) |] ==> f(x) : B;
      !!x. [| x : A; P(x) |] ==> g(x) : C;
      exists s : set of B & forall y : A & P(y) => f(y) in set s;
      forall a1 : A, a2 : A &
             P(a1) and (P(a2) and f(a1) = f(a2)) => g(a1) = g(a2) |] ==>
   f(a) in set dom {f(x) |-> g(x) | x : A & P(x)}"
inset_rng_madd_intr_elem:
  "[| a : A; b : B; m : map A to B |] ==> b in set rng (m ++ {a |-> b})"
inset_rng_modify_msingleton_intr_elem:
  "[| a : A; b : B; m : map A to B |] ==> b in set rng madd(a,b,m)"
inset_rng_madd_intr_map:
  "[| a : A; b1 : B; b2 : B; m : map A to B; b2 in set rng m;
      a not in set dom m |] ==>
   b2 in set rng madd(a,b1,m)"
inset_rng_elim:
  "[| b : B; m : map A to B; b in set rng m |] ==>
   exists a in set dom m & m@(a) = b"
inset_rng_intr_elimx:
  "[| b : B; m : map A to B; exists a in set dom m & m@(a) = b |] ==>
   b in set rng m"
inset_rng_intr_at_bimap:
  "[| a : A; m : inmap A to B; a in set dom m |] ==> m@(a) in set rng m"

not_inset_dom_emptymap_intr: "a : A ==> a not in set dom {|->}"
not_inset_rng_emptymap_intr: "b : B ==> b not in set rng {|->}"

ninset_dom_domsub_intr_singleton:
  "[| a : A; m : map A to B |] ==> a not in set dom ({a} <-: m)"
ninset_dom_madd_elim:
  "[| a1 : A; a2 : A; b : B; m : map A to B;
      a1 not in set dom madd(a2,b,m) |] ==>
   a1 <> a2 and a1 not in set dom m"
ninset_dom_madd_elim_left:
  "[| a1 : A; a2 : A; b : B; m : map A to B;
      a1 not in set dom madd(a2,b,m) |] ==>
   a1 not in set dom m"
ninset_dom_madd_elim_right:
  "[| a1 : A; a2 : A; b : B; m : map A to B;
      a1 not in set dom madd(a2,b,m) |] ==>
   a1 <> a2"
ninset_dom_inverse_intr:
  "[| b : B; m : inmap A to B; b not in set rng m |] ==>
   b not in set dom (inverse m)"
ninset_rng_madd_elim:
  "[| a : A; b1 : B; b2 : B; m : map A to B;
      b1 not in set rng madd(a,b2,m); a not in set dom m |] ==>
   b1 <> b2 and b1 not in set rng m"
ninset_rng_madd_elim_left:
  "[| a : A; b1 : B; b2 : B; m : map A to B;
      b1 not in set rng madd(a,b2,m); a not in set dom m |] ==>
   b1 not in set rng m"
ninset_rng_madd_elim_right:
  "[| a : A; b1 : B; b2 : B; m : map A to B;
      b1 not in set rng madd(a,b2,m) |] ==>
   b1 <> b2"
ninset_rng_inverse_intr:
  "[| a : A; m : inmap A to B; a not in set dom m |] ==>
   a not in set rng (inverse m)"

modify_form:
  "[| m1 : map A to B; m2 : map A to B |] ==> m1 ++ m2 : map A to B"
modify_assoc:
```

```
  "[| m1 : map A to B; m2 : map A to B; m3 : map A to B |] ==>
   m1 ++ m2 ++ m3 = m1 ++ (m2 ++ m3)"
modify_comm:
  "[| m1 : map A to B; m2 : map A to B; (compatible m1 m2) |] ==>
   m1 ++ m2 = m2 ++ m1"
modify_defn_emptymap_left: "m : map A to B ==> {|->} ++ m = m"
modify_preserves_dom_psubset:
  "[| m1 : map A to B; m2 : map A to B; m3 : map A to B;
      dom m1 psubset dom m2 |] ==>
   dom m1 psubset dom (m1 ++ m2)"
modify_self: "m : map A to B ==> m ++ m = m"

rngres_form: "[| m : map A to B; s : set of B |] ==> m :> s : map A to B"
rngres_singleton_not_empty_elim:
  "[| b : B; m : map A to B; m :> {b} <> {|->} |] ==> b in set rng m"
rngres_defn_singleton_ninset:
  "[| b : B; m : map A to B; b in set rng m |] ==> m :> {b} = {|->}"

rngsub_form: "[| m : map A to B; s : set of B |] ==> m :-> s : map A to B"

psubset_dom_modify_intr:
  "[| m1 : map A to B; m2 : map A to B |] ==>
   dom m1 psubset dom (m1 ++ m2)"

munion_form:
  "[| m1 : map A to B; m2 : map A to B; compatible(m1)(m2) |] ==>
   m1 munion m2 : map A to B"
munion_comm:
  "[| m1 : map A to B; m2 : map A to B; compatible(m1)(m2) |] ==>
   m1 munion m2 = m2 munion m1"
munion_assoc:
  "[| m1 : map A to B; m2 : map A to B; m3 : map A to B;
      compatible(m1)(m2); compatible(m2)(m3);
      compatible(m1)(m3) |] ==>
   m1 munion m2 munion m3 = m1 munion (m2 munion m3)"

madd_modify_defn_inset:
  "[| a : A; b : B; m1 : map A to B; m2 : map A to B;
      a in set dom m2 |] ==>
   madd(a,b,m1) ++ m2 = m1 ++ m2"
madd_modify_defn_ninset:
  "[| a : A; b : B; m1 : map A to B; m2 : map A to B;
      a not in set dom m2 |] ==>
   madd(a,b,m1) ++ m2 = madd(a,b,m1 ++ m2)"
madd_modify_defn_compatible:
  "[| a : A; b : B; m1 : map A to B; m2 : map A to B;
      compatible(madd(a,b,m1))(m2) |] ==>
   madd(a,b,m1) ++ m2 = madd(a,b,m1 ++ m2)"
madd_modify_indent:
  "[| a : A; b : B; m : map A to B; a in set dom m;
      m@(a) = b |] ==>
   madd(a,b,m) = m"
madd_to_modify:
  "[| a : A; b : B; m : map A to B |] ==> m ++ {a |-> b} = madd(a,b,m)"
madd_defn_domsub_msingleton_eq:
  "[| a : A; b : B; m : map A to B |] ==>
   madd(a,b,m) = madd(a,b,{a} <-: m)"
madd_extract:
  "[| a : A; m : map A to B; a in set dom m |] ==>
   m = madd(a,m@(a),{a} <-: m)"
madd_form_bimap:
  "[| a : A; b : B; m : inmap A to B; a not in set dom m;
      b not in set rng m |] ==>
```

```
    madd(a,b,m) : inmap A to B"


at_form:
  "[| a : A; m : map A to B; a in set dom m |] ==> m@(a) : B"
at_inmap_form:
  "[| a : A; m : inmap A to B; a in set dom m |] ==> m@(a) : B"
at_defn_modify_msingleton_eq:
  "[| a : A; b : B; m : map A to B |] ==> (m ++ {a |-> b})@(a) = b"
at_defn_modify_msingleton_neq:
  "[| a : A; b : B; a2 : A; m : map A to B; a2 in set dom m;
      a2 <> a1 |] ==>
   (m ++ {a |-> b})@(a) = m@(a2)"
at_defn_modify_madd_eq:
  "[| a : A; b : B; m1 : map A to B; m2 : map A to B |] ==>
   (m1 ++ madd(a,b,m2))@(a) = b"

at_defn_modify_left:
 "[| a : A; m1 : map A to B; m2 : map A to B; a not in set dom m2;
     a in set dom m1 |] ==>
  (m1 ++ m2)@(a) = m1@(a)"
at_defn_modify_right:
  "[| a : A; m1 : map A to B; m2 : map A to B; a in set dom m2 |] ==>
   (m1 ++ m2)@(a) = m2@(a)"
at_defn_compm_fun:
  "[| a : A; P(a); forall x : A & def P(x);
      !!x. [| x : A; P(x) |] ==> f(x) : B;
      !!x. [| x : A; P(x) |] ==> g(x) : C;
      exists s : set of B & forall y : A & P(y) => f(y) in set s;
      forall a1 : A, a2 : A &
             P(a1) and (P(a2) and f(a1) = f(a2)) => g(a1) = g(a2) |] ==>
   {f(x) |-> g(x) | x : A & P(x)}@(f(a)) = g(a)"
at_defn_map_comp_left_set:
  "[| a : A; s : set of A; a in set s;
      !!x. [| x : A; x in set s |] ==> f(x) : B |] ==>
   {x |-> f(x) | x in set s}@(a) = f(a)"

inmap_onetoone:
  "[| a1 : A; a2 : A; m : inmap A to B; a1 in set dom m;
      a2 in set dom m; m@(a1) = m@(a2) |] ==>
   a1 = a2"
inmap_onetoone_not:
  "[| a1 : A; a2 : A; m : inmap A to B; a1 in set dom m;
      a2 in set dom m; a1 <> a2 |] ==>
   m@(a1) <> m@(a2)"
inmap_elim: "m : inmap A to B ==> onetoone(m)"
inmap_form: "[| m : map A to B; onetoone(m) |] ==> m : inmap A to B"
inmap_indn:
  "[| m0 : inmap A to B; P({|->});
      !!a b m. [| a : A; b : B; m : map A to B; P(m); a not in set dom m;
                  b not in set rng m |] ==>
      P(madd(a,b,m)) |] ==>
   P(m0)"
inmap_supertype: "m : inmap A to B ==> m : map A to B"
inmap_unique_rng_elem:
  "[| a : A; m : inmap A to B; a in set dom m |] ==>
   exists1 b : B & b = m@(a)"

compatible_modify_intr_left:
  "[| m1 : map A to B; m2 : map A to B; m3 : map A to B;
      (compatible m1 m3); (compatible m2 m3) |] ==>
   (compatible (m1 ++ m2) m3)"
compatible_modify_intr_right:
```

```
  "[| m1 : map A to B; m2 : map A to B; m3 : map A to B;
      (compatible m1 m2); (compatible m1 m3) |] ==>
   (compatible m1 (m2 ++ m3))"
compatible_madd_elim_left_ninset:
  "[| a : A; b : B; m1 : map A to B; m2 : map A to B;
      a not in set dom m1; (compatible madd(a,b,m1) m2) |] ==>
   (compatible m1 m2)"
compatible_madd_elim_right:
  "[| a : A; b : B; m1 : map A to B; m2 : map A to B;
      a in set dom m2; (compatible madd(a,b,m1) m2) |] ==>
   m2@(a) = b"
compatible_comm:
  "[| m1 : map A to B; m2 : map A to B; (compatible m1 m2) |] ==>
   (compatible m2 m1)"
compatible_defn_emptymap_left:
  "m : map A to B ==> (compatible ({|->}) m)"
compatible_defn_emptymap_right:
  "m : map A to B ==> (compatible m ({|->}))"
compatible_elim:
  "[| m1 : map A to B; m2 : map A to B; (compatible m1 m2) |] ==>
   forall a in set dom m1 inter dom m2 & m1@(a) = m2@(a)"
compatible_intr:
  "[| m1 : map A to B; m2 : map A to B;
      forall a in set dom m1 inter dom m2 & m1@(a) = m2@(a) |] ==>
   (compatible m1 m2)"

dom_form: "m : map A to B ==> dom m : set of A"
dom_form_inmap: "m : inmap A to B ==> dom m : set of A"
dom_emptymap_intr: "m = {|->} ==> dom m = {}"
dom_domsub_defn:
  "[| m : map A to B; s : set of A |] ==> dom (s <-: m) = dom m \\ s"
dom_neq_emps_elim: "[| m : map A to B; dom m <> {} |] ==> m <> {|->}"
dom_madd_int_elim_left:
 "[| a : A; b : B; m1 : map A to B; m2 : map A to B;
     dom madd(a,b,m1) inter dom m2 = {} |] ==>
  dom m1 inter dom m2 = {}"
dom_madd_int_elim_right:
  "[| a : A; b : B; m1 : map A to B; m2 : map A to B;
      dom madd(a,b,m1) inter dom m2 = {} |] ==>
   a not in set dom m2"
dom_defn_modify:
  "[| m1 : map A to B; m2 : map A to B |] ==>
   dom (m1 ++ m2) = dom m1 union dom m2"
dom_defn_madd_inset:
  "[| a : A; b : B; m : map A to B; a in set dom m |] ==>
   dom madd(a,b,m) = dom m"
dom_defn_inverse: "m : map A to B ==> dom (inverse m) = rng m"
dom_defn_compm_left_set:
  "[| s : set of A; !!x. [| x : a; x in set s |] ==> f(x) : B |] ==>
   dom {x |-> f(x) | x in set s} = s"
dom_finite_intrmp_rng_finite:
  "[| forall x : A & def P(x); !!x. [| x : A; P(x) |] ==> f(x) : B;
      !!x. [| x : A; P(x) |] ==> g(x) : C;
      exists s : set of B & forall y : A & P(y) => f(y) in set s;
      forall a1 : A, a2 : A &
             P(a1) and (P(a2) and f(a1) = f(a2)) => g(a1) = g(a2) |] ==>
   exists t : set of C& forall a : A & P(a) => g(a) in set t"

inverse_form: "m : inmap A to B ==> inverse m : inmap B to A"

onetoone_modify_msingleton_elim_ninset_rng:
  "[| a : A; b : B; m : map A to B; onetoone(m ++ {a |-> b});
      a not in set dom m |] ==>
```

```
   b not in set rng m"
onetoone_madd_elim_ninset_rng:
  "[| a : A; b : B; m : map A to B; onetoone(madd(a,b,m));
      a not in set dom m |] ==>
   b not in set rng m"
onetoone_madd_elim_ninset_map:
  "[| a : A; b : B; m : map A to B; onetoone(madd(a,b,m));
      a not in set dom m |] ==>
   onetoone(m)"
onetoone_elim:
  "[| m : map A to B; onetoone(m) |] ==>
      forall x in set dom m & forall y in set dom m &
   m@(x) = m@(y) => x = y"
onetoone_intr:
  "[| m : map A to B;
      forall x in set dom m & forall y in set dom m &
               m@(x) = m@(y) => x = y |] ==>
   onetoone(m)"

map_Union_ext_dom_left:  "m : map A to B ==> m : map C | A to B"
map_Union_ext_dom_right: "m : map A to B ==> m : map A | C to B"
map_Union_ext_rng_left:  "m : map A to B ==> m : map A to C | B"
map_Union_ext_rng_right: "m : map A to B ==> m : map A to B | C"


compm_form_left:
  "[| forall x : A & def P(x); !!x. [| x : A; P(x) |] ==> f(x) : B;
      exists s : set of B & forall y : A & P(y) => f(y) in set s |] ==>
   {x |-> f(x) | x : A & P(x)} : map A to B"
compm_form_left_set:
  "[| s : set of A; !!x. [| x : A; P(x) |] ==> f(x) : B |] ==>
   {x |-> f(x) | x in set s} : map A to B"
compm_form_set_ident:
  "[| s : set of A; !!x. x : A ==> f(x) : B;
      !!x. [| x : A; P(x) |] ==> g(x) : C;
      forall a1 in set s & forall a2 in set s &
          f(a1) = f(a2) => g(a1) = g(a2) |] ==>
   {f(x) |-> g(x) | x in set s} : map B to C"
compm_left_defn_add:
  "[| a : A; s : set of A; f(a) : B;
      !!x. [| x : A; x in set s |] ==> f(x) : B |] ==>
   {x |-> f(x) | x in set add(a,s)} =
   {x |-> f(x) | x in set s} ++ {a |-> f(a)}"

merge_form:
  "[| s : set of (map A to B);
      forall m1 in set s & (foralls' s (compatible m1)) |] ==>
   merge s : map A to B"

rng_form:       "m : map A to B ==> rng m : set of B"
rng_form_inmap: "m : inmap A to B ==> rng m : set of B"
rng_defn:       "m : map A to B ==> rng m = (compstr (dom m) (at m))"
rng_defn_madd:
  "[| a : A; b : B; m : map A to B |] ==>
   rng madd(a,b,m) = add(b,rng ({a} <-: m))"
rng_defn_madd_inset:
  "[| a : A; b : B; m : map A to B; a in set dom m |] ==>
   rng madd(a,b,m) = add(b,rng ({a} <-: m))"
rng_defn_inverse: "m : inmap A to B ==> rng (inverse m) = dom m"
rng_defn_compm:
  "[| forall x : A & def P(x);
      !!x. [| x : A; P(x) |] ==> f(x) : B;
      !!x. [| x : A; P(x) |] ==> g(x) : C;
```

```
          exists s : set of B & forall y : A & P(y) => f(y) in set s;
          forall a1 : A, a2 : A &
                 P(a1) and (P(a2) and f(a1) = f(a2)) => g(a1) = g(a2) |] ==>
      rng {f(x) |-> g(x) | x : A & P(x)} = (comp' A g P)"


print_translation
{*
let

fun eta_exp (e as Abs(_,_,_)) = e
    | eta_exp e =Abs("x",dummyT,e$(Bound 0));

fun mcomp_tr' [A,f1,f2,P] =
    let val Abs(n1,_,e1) = eta_exp f1
        val Abs(n2,_,e2) = eta_exp f2
        val Abs(n2,_,P') = eta_exp P
        val new =
            variant
                (add_term_names(
                    e1,
                    (add_term_names (e2,add_term_names (P',[])))))
    in
        case (e1,e2,P') of
            (_,_,Const("true'",_)) =>
                let val n' = Free(new n1,dummyT) in
                    Const("mcomptr_",dummyT)$
                    (Const("maplet_",dummyT)$
                        subst_bounds([n'],e1)$
                        subst_bounds([n'],e2))$
                    (Const("tbind_",dummyT)$n'$A)
                end |
             _ =>
                let val n' = Free(new n1,dummyT) in
                    Const("mcomp_",dummyT)$
                    (Const("maplet_",dummyT)$
                        subst_bounds([n'],e1)$
                        subst_bounds([n'],e2))$
                    (Const("tbind_",dummyT)$n'$A)$
                    subst_bounds([n'],P')
                end

    end;

fun mcomps_tr' [s,f1,f2,P] =
    let val Abs(n1,_,e1) = eta_exp f1
        val Abs(n2,_,e2) = eta_exp f2
        val Abs(n2,_,P') = eta_exp P
        val new =
            variant
                (add_term_names(
                    e1,
                    (add_term_names (e2,add_term_names (P',[])))))
    in
        case P' of
            Const("true'",_) =>
                let val n' = Free(new n1,dummyT) in
                    Const("mcompstr_",dummyT)$
                    (Const("maplet_",dummyT)$
                        subst_bounds([n'],e1)$
                        subst_bounds([n'],e2))$
                    (Const("sbind_",dummyT)$n'$s)
                end |
```

```
            _ =>
                let val n' = Free(new n1,dummyT) in
                    Const("mcomps_",dummyT)$
                    (Const("maplet_",dummyT)$
                        subst_bounds([n'],e1)$
                        subst_bounds([n'],e2))$
                    (Const("sbind_",dummyT)$n'$s)$
                    subst_bounds([n'],P')
                end

    end;

in
  [("mcomp'",mcomp_tr'),("mcomps'",mcomps_tr')]
end;
*}


End
```

12.14 Seq.thy

```
theory Seq = MapLPF:

(* Finite Sequences: section 14.7, pages 314-318 in "Proof in VDM" *)

consts
  seqty'   :: "ty => ty"     ("seq of _" [140] 140)
  seq1ty'  :: "ty => ty"     ("seq1 of _" [140] 140)


(* constructors *)

  emptyseq'     :: "ex"                    ("[]")
  cons          :: "[ex,ex] => ex"


(* destructors *)

  hd'           :: "ex => ex"              ("(2hd/ _)" [450] 450)
  tl'           :: "ex => ex"              ("(2tl/ _)" [450] 450)


(* other operations *)
  append'       :: "[ex,ex] => ex"         ("(_^/_)" [410,411] 410)
  len'          :: "ex => ex"              ("(2len/ _)" [450] 450)
  inds'         :: "ex => ex"              ("(2inds/ _)" [450] 450)
  elems'        :: "ex => ex"              ("(2elems/ _)" [450] 450)
  conc'         :: "ex => ex"              ("(2conc/ _)" [450] 450)


(* sequence enumeration *)

syntax
  enumseq_      :: "exs => ex"             ("[_]")

translations
  "enumseq_ (exsn e es)" == "cons e (enumseq_ es)"
  "enumseq_(e)" == "cons e emptyseq'"


(* sequence comprehension *)
```

```
consts
  scomps'       :: "[ex,ex=>ex,ex=>ex] => ex"

syntax
  scomps_       :: "[ex,sbind,ex] => ex" ("(1[_ |/ _ &/ _])")
  scompstr_     :: "[ex,sbind] => ex"    ("(1[_ |/ _])")
  scompsid_     :: "[sbind,ex] => ex"    ("(1[_ &/ _])")  (* not VDM-SL! *)

translations
  "scomps_ e (sbind_ x s) P" => "scomps' s (%x. e) (%x. P)"
  "scompstr_ e (sbind_ x s)" => "scomps' s (%x. e) (%x. true)"
  "scompsid_ (sbind_ x s) P" => "scomps' s (%x. x) (%x. P)"


(* sequence range *)

syntax
  seqrange_       :: "[ex,ex] => ex"        ("(1[_,...,/_])")


(* primitive rules (axioms) *)

axioms
  emptyseq_form:  "[] : seq of A"
  cons_form_seq1: "[| a : A; s : seq of A |] ==> (cons a s) : seq1 of A"
  hd_defn_cons:   "[| a : A; l : seq of A |] ==> hd (cons a s) = a"
  tl_defn_cons:   "[| a : A; l : seq of A |] ==> tl (cons a s) = l"
  append_defn_emptyseq_left:
    "l : seq of A ==> []^l = l"
  append_defn_cons_left:
    "[| a:A1; s1:seq of A; s2:seq of A |] ==>
     (cons a s1)^s2=(cons a (s1^s2))"
  sapply_defn_hd: "s:seq1 of A ==> s@(succ(0)) = hd s"
  sapply_defn_tl:
    "[| s : seq1 of A; i : nat1; i <> succ(0); i <= len s |] ==>
     s@(i) = (tl s)@(i - succ(0))"

(* definitions *)

defs
  seq1_def: "seq1 of A == << s : seq of A | s <> [] >>"
  inds_def: "inds s == {succ(0),...,len s}"

(* derived rules *)

axioms
  len_def:   "len s == if s=[] then 0 else succ(len (tl s))"
  elems_def: "elems s == if s=[] then {} else add(hd s,elems (tl s))"
  conc_def:  "conc s == if s=[] then [] else (hd s)^conc (tl s)"

(* derived rules *)

axioms
  def_emptyseq_intr: "s:seq of A ==> def (s=[])"
  eq_seq_defn_cons:
    "[| a1:A; a2:A; s1:seq of A; s2:seq of A |] ==>
     (cons a1 s2)=(cons a2 s2) <=> a1=a2 and s1=s2"
  eq_seq1_defn:
    "[| s1:seq1 of A; s2:seq1 of A |] ==>
     s1=s2 <=> hd s1=hd s2 and tl s1=tl s2"
  inset_elems_cons_elim:
    "[| a:A; b:A; s:seq of A; a in set elems (cons b s) |] ==>
     a=b or a in set elems s"
```

```
inset_inds_elim:
  "[| s:seq1 of A; n:nat1; n in set inds s |] ==> n <= len s"
notinset_elems_emptyseq_intr:
  "a:A ==> a not in set elems []"
notinset_inds_emptyseq_intr:
  "a:A ==> a not in set inds []"
append_assoc:
  "[| s1:seq of A; s2:seq of A; s3:seq of A |] ==> s1^s2^s3 = s1^(s2^s3)"
append_defn_emptyset_right:
  "s:seq of A ==> s^[] = s"
append_form:
  "[| s1:seq of A; s2:seq of A |] ==> s1^s2:seq of A"

singletons_noteq_emptyseq:
  "a:A ==> [a] <> []"
singletons_form:
  "a:A ==> [a]:seq of A"

sapp_form: "[| s:seq1 of A; i:nat1; i<=len s |] ==> s@(i):A"

conc_defn_emptyset:
  "conc [] = []"
conc_defn_cons:
  "[| s1:seq of A; s2:seq of seq of A |] ==>
   conc (cons s1 s2) = s1^conc s2"
conc_form:
  "s:seq of seq of A ==> conc s:seq of A"

cons_noteq_emptyseq:
  "[| a:A; s:seq of A |] ==> (cons a s) <> []"
cons_to_append:
  "[| a:A; s:seq of A |] ==> [a]^s=(cons a s)"
cons_form:
  "[| a:A; s:seq of A |] ==> (cons a s):seq of A"
cons_intr:
  "s:seq1 of A ==> (cons (hd s) (tl s)) = s"

elems_to_those:
  "s:seq of A ==> elems s = {s@(i) | i in set inds s}"
elems_defn_emptyseq:
  "elems [] = {}"
elems_defn_append:
  "[| s1:seq of A; s2:seq of A |] ==>
   elems (s1^s2) = elems s1 union elems s2"
elems_defn_append_cons:
  "[| a:A; s1:seq of A; s2:seq of A |] ==>
   elems((cons a s1)^s2) = add(a,elems(s1^s2))"
elems_defn_cons:
  "[| a:A; s:seq of A |] ==> elems (cons a s) = add(a,elems s)"
elems_defn_cons_singleton:
  "[| a:A; s:seq of A |] ==> elems (cons a s) = {a} union elems s"
elems_form:
  "s:seq of A ==> elems s:set of A"
elems_form_seq1:
  "s:seq1 of A ==> elems s:set of A"

hd_defn_singletons:
  "a:A ==> hd [a] = a"
hd_form:
  "s:seq1 of A ==> hd s:A"
inds_defn_emptyseq:
  "inds [] = {}"
inds_form:
```

```
    "s:seq of A ==> inds s:set of nat1"
  len_defn_emptyseq:
    "len [] = 0"
  len_defn_append:
    "[| s1:seq of A; s2:seq of A |] ==> len(s1^s2) = len s1 + len s2"
  len_defn_singletons:
    "a:A ==> len [a] = succ(0)"
  len_defn_cons:
    "[| a:A; s:seq of A |] ==> len (cons a s) = succ(len s)"
  len_defn_seq1:
    "s:seq1 of A ==> len s = succ(len tl s)"
  len_form:
    "s:seq of A ==> len s:nat"
  len_form_seq1:
    "s:seq1 of A ==> len s:nat1"

  seq1_elim:
    "s:seq1 of A ==> s <> []"
  seq1_intr:
    "[| s:seq of A; s <> [] |] ==> s:seq1 of A"
  seq1_hnf:
    "[| s:seq1 of A;
        !! h t.[| h:A; t:seq of A; P(t) |] ==> P(cons h t) |] ==>
     P(s)"
  seq1_indn:
    "[| s:seq1 of A; !!a. a:A ==> P([a]);
        !! h t.[| h:A; t:seq1 of A; P(t) |] ==> P(cons h t) |] ==>
     P(s)"
  seq1_supertype:
    "s:seq1 of A ==> s:seq of A"

  seq_append_indn:
    "[| s:seq of A; P([]); !!a. a:A ==> P([a]);
        !!s1 s2.[|s1:seq of A; s2:seq of A; P(s1); P(s2) |] ==>
          P(s1^s2) |] ==>
     P(s)"
  seq_or_ext_right:
    "s:seq of A ==> s:A | B"
  seq_or_ext_left:
    "s:seq of A ==> s:B | A"
  seq_sep:
    "s:seq of A ==> s=[] or (exists h:A, t:seq of A & s=(cons h t))"

  tl_defn_singletons:
    "a:A ==> tl [a] = []"
  tl_form:
    "s:seq1 of A ==> tl s:seq of A"

parse_translation
{*
let

fun setrange_tr [e1,e2] =
    Const("comp'",dummyT)$
    Const("natty'",dummyT)$
    Abs("x",dummyT,Bound 0)$
    Abs("x",dummyT,
    Const("and'",dummyT)$
    (Const("leq'",dummyT)$e1$(Bound 0))$
    (Const("leq'",dummyT)$(Bound 0)$e2));

fun seqrange_tr [e1,e2] =
    Const("scomps'",dummyT)$
```

```
    (setrange_tr [e1,e2])$
    (Abs("x",dummyT,Bound 0))$
    (Abs("x",dummyT,Const("true'",dummyT)));

in
  [("seqrange_",seqrange_tr)]
end;

*}

print_translation
{*
let

fun eta_exp (e as Abs(_,_,_)) = e
  | eta_exp e =Abs("x",dummyT,e$(Bound 0));

fun scomps_tr' [s,f,P] =
    let val Abs(n1,_,e) = eta_exp f
        val Abs(n2,_,P') = eta_exp P
        val new =
            variant (add_term_names (e,add_term_names (P',[])))
    in
        case (s,e,P') of
        (Const("comp'",_)$
         Const("nat",_)$
             Abs(_,_,Bound 0)$
             Abs(_,_,
         Const("and'",_)$
             (Const("leq'",_)$e1$(Bound 0))$
             (Const("leq'",_)$(Bound 0)$e2)),
         Bound 0,
         Const("true'",_)) =>
        Const("seqrange_",dummyT)$e1$e2 |
            (_,_,Const("true'",_)) =>
                let val n' = Free(new n1,dummyT) in
                    Const("scompstr_",dummyT)$
                    subst_bounds([n'],e)$
                    (Const("sbind_",dummyT)$n'$s)
                end |
            (_,Bound 0,_) =>
                let val n' = Free(new n1,dummyT) in
                    Const("scompsid_",dummyT)$
                    (Const("sbind_",dummyT)$n'$s)$
                    subst_bounds([n'],P')
                end |
            _ =>
                let val n' = Free(new n1,dummyT) in
                    Const("scomps_",dummyT)$
                    subst_bounds([n'],e)$
                    (Const("sbind_",dummyT)$n'$s)$
                    subst_bounds([n'],P')
                end
    end;
in
  [("scomps'",scomps_tr')]
end;
*}

lemma "[false,true,false] : seq1 of Nat"
oops

lemma "[0,...,succ(0)]"
```

```
oops

lemma "[ x | x in set {0,...,succ(0)} ]"
oops

lemma "[ x | x in set { x | x:nat & 0<=x and x<=succ(0)} ]"
oops

end
```

12.15 Bool.thy

```
theory Bool = Seq:

(* Booleans: section 14.7, pages 319-321 in "Proof in VDM" *)

consts
  boolty' :: "ty"   ("bool")

(* axioms *)

axioms
  def_to_bool: "def P ==> P:bool"
  bool_to_def: "P:bool ==> def P"
  iff_to_eq:   "P <=> Q ==> P = Q"

(* derived *)

axioms
  forall_form:
    "(!!y. y:A ==> P(y):bool) ==> (forall x:A & P(x)):bool"
  foralls_form:
    "[| s:set of A; !!y.[| y:A; y in set s |] ==> P(y):bool |] ==>
     (forall x in set s & P(x)):bool"

  and_form:     "[| e1:bool; e2:bool |] ==> e1 and e2:bool"
  and_form_sqt: "[| e1:bool; e1 ==> e2:bool |] ==> e1 and e2:bool"

  bool_eval: "P:bool ==> P=true or P=false"

  eq_to_iff: "[| P:bool; P=Q |] ==> P <=> Q"
  eq_form:   "[| a:A; b:A |] ==> a=b:bool"

  exists_form:
    "(!!y. y:A ==> P(y):bool) ==> (exists x:A & P(x)):bool"
  existss_form:
    "[| s:set of A; !!y.[| y:A; y in set s |] ==> P(y):bool |] ==>
     (exists x in set s & P(x)):bool"

  exists1_form:
    "(!!y. y:A ==> P(y):bool) ==> (exists1 x:A & P(x)):bool"
  exists1s_form:
    "[| s:set of A; !!y. [| y:A; y in set s |] ==> P(y):bool |] ==>
     (exists1 x in set s & P(x)):bool"

  iff_form:       "[| e1:bool; e2:bool |] ==> e1 <=> e2:bool"
  iff_subs_left:  "[| Q <=> R; P(R) |] ==> P(Q)"
  iff_subs_right: "[| Q <=> R; P(Q) |] ==> P(R)"

  imp_form:     "[| e1:bool; e2:bool |] ==> e1 => e2:bool"
  imp_form_sqt: "[| e1:bool; e1 ==> e2:bool |] ==> e1=>e2 : bool"

  inset_form: "[| a:A; s:set of A |] ==> a in set s:bool"
```

```
  lt_form: "[| n1:nat; n2:nat |] ==> n1<n2 : bool"

  leq_form: "[| n1:nat; n2:nat |] ==> n1<=n2 : bool"

  not_form: "P:bool ==> not P : bool"

  noteq_form: "[| P:A; Q:A |] ==> P<>Q : bool"

  notinset_form: "[| a:A; s:set of A |] ==> a not in set s : bool"

  or_form:     "[| P:bool; Q:bool |] ==> P or Q : bool"
  or_form_sqt: "[| P:bool; not P ==> Q:bool |] ==> P or Q : bool"

  psubset_form: "[| s1:set of A; s2:set of A |] ==> s1 psubset s2 : bool"

  false_form: "false:bool"

  true_form:  "true:bool"

ML
{*
val bool_lpfs =
  prop_lpfs addIs
   [thm "forall_form", thm "foralls_form", thm "and_form",
    thm "and_form_sqt", thm "exists_form",
    thm "existss_form", thm "exists1_form", thm "exists1s_form",
    thm "iff_form", thm "imp_form", thm "imp_form_sqt", thm "inset_form",
    thm "lt_form", thm "leq_form", thm "not_form", thm "noteq_form",
    thm "notinset_form", thm "or_form", thm "or_form_sqt",
    thm "psubset_form", thm "false_form", thm "true_form"];
*}

end
```

12.16 Case.thy

```
theory Case = Let:

(* case_match *)

lemma case_match: "!! e.[| P(e); e:A; e1(e):B |] ==>
                   (let x = e in if P(x) then e1(x) else e2(x)) = e1(e)";
 apply(rule eq_trans3)
   prefer 2
   apply(rule let_defn)
    prefer 4
    apply(rule if_true)
     apply(tactic {* lpf_fast_tac (prop_lpfs addIs [thm "if_form_sqt"]) 1 *})+
done

(* case_not_match *)

lemma case_not_match: "!! e.[| not P(e); e:A; e2(e):B |] ==>
                       (let x = e in if P(x) then e1(x) else e2(x)) =
                       (let x = e in e2(x))"
 apply(rule eq_trans3)
   prefer 2
   apply(rule let_defn )
    prefer 4
    apply(rule eq_trans2)
      prefer 2
      apply(rule if_false)
```

```
       prefer 4
       apply(rule eq_symm1)
        prefer 2
        apply(rule let_defn)
        apply(tactic {* lpf_fast_tac (prop_lpfs addIs [thm "if_form_sqt",thm
"let_form"]) 1 *})+
done

ML {* val case_others = thm "let_defn"; *}

end
```

12.17 VDM_LPF.thy

```
theory VDM_LPF = Bool + Case:

end
```