LFTOP: An LF based approach to domain specific reasoning

Jianmin Pang

LFTOP: AN LF BASED APPROACH TO DOMAIN SPECIFIC REASONING

by

Jianmin Pang

Submitted in conformity with the requirements
for the degree of Ph.D
Department of Computer Science
University of Durham

Abstract

Specialized vocabulary, notations and inference rules tailored for the description, analysis and reasoning of a domain is very important for the domain. For domain-specific issues researchers focus mainly on the design and implementation of domain-specific languages (DSL) and pay little attention to the reasoning aspects. We believe that domain-specific reasoning is very important to help the proofs of some properties of the domains and should be more concise, more reusable and more believable. It deserves to be investigated in an engineering way. Type theory provides good support for generic reasoning and verification. Many type theorists want to extend uses of type theory to more domains, and believe that the methods, ideas, and technology of type theory can have a beneficial effect for computer assisted reasoning in many domains. Proof assistants based on type theory are well known as effective tools to support reasoning. But these proof assistants have focused primarily on generic notations for representation of problems and are oriented towards helping expert type theorists build proofs efficiently. They are successful in this goal, but they are less suitable for use by non-specialists. In other words, one of the big barriers to limit the use of type theory and proof assistant in domain-specific areas is that it requires significant expertise to use it effectively.

We present LFTOP — a new approach to domain-specific reasoning that is based on a type-theoretic logical framework (LF) but does not require the user to be an expert in type theory. In this approach, users work on a domain-specific interface that is familiar to them. The interface presents a reasoning system of the domain through a user-oriented syntax. A middle layer provides translation between the user syntax and LF, and allows additional support for reasoning (e.g. model checking). Thus, the complexity of the logical framework is hidden but we also retain the benefits of using type theory and its related tools, such as precision and machine-checkable proofs. The approach is being investigated through a number of case studies. In each case study, the relevant domain-specific specification languages and logic are formalized in Plastic. The relevant reasoning system is designed and customized for the users of the corresponding specific domain. The corresponding lemmas are proved in Plastic. We analyze the advantages and shortcomings of this approach, define some new concepts related to the approach, especially discuss issues arising from the translation between the different levels. A prototype implementation is developed. We illustrate the approach through many concrete examples in the prototype implementation.

The study of this thesis shows that the approach is feasible and promising, the relevant methods and technologies are useful and effective.

Acknowledgements

I would like to thank everyone who has helped me with my research. In particular, I am very grateful to my supervisors Paul Callaghan and Zhaohui Luo. They have provided a great many suggestions for research topics, and without the foundation of their works on computer assisted reasoning my thesis would have been very different. They also gave me many support when I met problems in research and life. I missed the discussions while we had barbecues in your backyard.

I would like to thank my Ph.D proposal examiners, Professor Malcolm Munro and Dr. Alex Coddington, for their valuable advice in the beginning of my research.

Thanks also go to James McKinna, Steven Bradley, Conor McBride, Yong Luo, Xingyuan Zhang, Edwin Brady, Robert Pollack and all other members of the CARG Group, Durham. You are all great!

I would like to thank my elder sister, Ying Pang, and the rest of my family who have provided encouragement, love, and support throughout.

Finally, my greatest thanks go to my wife, Bin Wang. She takes care of my daily life and my lovely daughter so that I can focus on my research. Without her love, encouragement, and confidence in me, the ups and downs of the research would have been a harder ride. This thesis is dedicated to her with all my love.

Thanks also to Durham University and the Department of Computer Science for the financial support through the Durham University Studentship.

Declaration

I declare that this thesis was composed by myself, and the work reported herein is my own unless explicitly declared otherwise. Some parts of the work have already been published in [Pang et al., 2005a; Callaghan et al., 2001; Pang et al., 2005b; Pang and Zhao, 2005; Pang et al., 2006a; Pang et al., 2006b].

Copyright Notice

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The last thing one knows in constructing a work is what to put first.

— Blaise Pascal, French thinker, mathematician, and scientist

1.1 Domain-specific reasoning

Specialised vocabulary, notations and inference rules tailored for the description, analysis and reasoning of a domain is very important for the domain. First, the role of notations and rules is cognitive in nature as they provide support for basic describing and reasoning tasks. Second, notations also have an important social role as communication interfaces between different, and possibly diverse, technical specialities involved in the domain.

In addition to using specialised vocabulary and notations for description and analysis, reasoning tasks are becoming prominent tasks in some domains. The domain users hope that computer can be used not only for computing and editing, but also for reasoning. There are many common characteristics in the aspect of reasoning even for different domains. Based on the common characteristics a good engineering approach is required for supporting domain-specific reasoning.

For users who do reasoning and verification work in their domain, a good computer-supported reasoning system should be powerful and convenient, i.e. the system should be domain oriented, easy to learn and use, can do all things which can be done by domain users with pen and paper, and a lot more that they can't. It should also be 'believable' in the sense that developments are rigorously checked by machine. This is especially important in domains where safety or security are key properties.

Till now, for domain-specific issues researchers focus mainly on the design and implementation of domain-specific languages (DSL) and pay little attention to the reasoning aspects. We believe that domain-specific reasoning is very important to help the proofs of some properties of the domains and should be more concise, more reusable and more believable. It

deserves to be investigated in an engineering way.

1.2 Type theory, LF and generic reasoning

Type theory provides good support for generic reasoning and verification. Many type theorists want to extend uses of type theory to more domains, and believe that the methods, ideas, and technologies of type theory can have a beneficial effect for computer assisted reasoning in many domains. Proof assistants based on type theory are well known as effective tools to support reasoning. But these proof assistants have focused primarily on generic notations for representation of problems and are oriented towards helping expert type theorists build proofs efficiently. They are successful in this goal, but they are less suitable for use by non-specialists. There are many model checkers which do not support reasoning strongly (such as Edinburgh Concurrency Workbench) applied in domain-specific areas, while the development of domain-specific reasoning techniques in the area of proof assistant has been remarkably slow. But we can't require that all domain users (e.g. potential users in domains which could benefit from type-theory based reasoning) are expert type theorists or that they should learn type theory first. i.e. one of the big barriers to limit the use of type theory and proof assistant in domain-specific areas is that it requires significant expertise to use it effectively.

LF is a simple type theory which allows particular type theories to be specified clearly in it. The version of LF we use is presented in [Luo, 1994], which introduces the type theory UTT by specifying it in LF. (Note: this kind of LF is different from Edinburgh LF [Harper et al., 1987].) LF provides a definition mechanism and the four basic concepts required in a type theory: types, objects in types, families of types, and objects of families of types. It makes a distinction between the *types*, which are required for a specific application, and *kinds*, which are part of the framework. LF is extended by declaring new constants and computation rules involving those constants. For example, inductive types can be added by declaring a type name, constructors, an elimination operator, and rules for computation. Luo [Luo, 1994] gives a schema which allows the addition of a large class of inductive types. These features allow LF as a suitable framework to be based on.

1.3 Motivation of this thesis

We believe that reasoning systems can be built using type theory technology as a framework, but which can be used via a domain-specific interface. Such an interface would provide functionality recognizable to an expert in the domain, and would support reasoning work in that domain. But the correctness of the functionality would be supported by the underlying type theory technology. In particular, we propose that this kind of system can be developed with a general purpose 'framework' type theory, in which the domain is first formalized (by

an expert) and an interface built on top of it. So in reasoning of domain-specific properties a hidden underlying support from a proof assistant (which is an implementation of the 'framework' type theory) is available and does not put extra burden on domain users, i.e. we envisage the following scenario: an implementation of a logical framework (LF) provides the core reasoning functionality and works as a server. A type theory expert encodes the type theory needed for an application domain in LF, formalizes the domain in the type theory, and gives a mapping between LF syntax and a concrete syntax for the application domain. A domain user who is an non-expert in type theory can then use the resulting system by working in the new interface which operates as a client, whilst being exposed to the minimum amount of details about the underlying type theory system. We advocate the use of this approach to provide a composition mechanism that retains the benefits of both domain-specific and generic proof assistants approaches. This is one of the motivations of our research.

LF has been implemented in Plastic [Callaghan and Luo, 2000b], a form of a proof assistant. But there is no big application based on the LF based proof assistant till now. Is it suitable to big applications? This is another one of the motivations of the research of this thesis.

1.4 The major contribution

The major contribution of this thesis is that we present and investigate a sound and practical approach to domain-specific reasoning based on LF, achieving this through theoretical work, actual implementation and evaluation. The following are some major steps which lead to the substantiation of this claim.

1. The presentation of the approach
   - The description of the approach
   - The methodology about the approach
   - The structure of the approach
2. The prototyping implementation of the approach
   - The interface design
   - The design of the protocol for the communications between user-level and Plastic-level.
3. The case studies
   - The case study on concurrency.
     - The formalisation of the relevant concepts of concurrency in LF.
     - The formal proving of related axioms, inference rules, lemmas and theorems.

  - The prototyping implementation of the domain-specific interface.
- The case study on LAZY-PCF+SHAR [1].
  - The formalisation of the relevant concepts of LAZY-PCF+SHAR in *LF*.
  - The formal proving of related axioms, inference rules, lemmas and theorems.
  - The description of the reason why we use Plastic directly other than to design a new interface.
- The whole work can be seen as an experiment with use of a restricted type theory (here we mean *LF*).

1.5 Related work

For the computer assisted verification of domain-specific properties, there are two main approaches, distinguished by their use (or not) of a theorem prover.

1.5.1 Theorem prover based approach

As its name, theorem prover based approach is an approach which is based on a theorem prover. The following is a list of the major systems which belong to this kind of approach.

- Isabelle [Paulson, 1994] [Paulson, 2005] provides a theorem prover based approach for users to specify concrete theory by providing a few declarations of abstract and concrete syntax, primitive proof rules and the support to user defined macros and translation functions in ML. Isabelle/Isar [Wenzel, 2002] aims to a versatile environment for human-readable formal proof documents, but doesn't explicitly consider domain-specific issues. Our approach is different from this approach by using a concise but powerful type theoretic logical framework *LF* and by constructing an interface for the domain users. Our approach presents both support to user convenience and strict type theoretic style correctness.

- In fact, users can use type theoretic proof assistants such as Coq [Project, 2004] and Lego [Pollack, 2005] to construct their proof directly by embedding the domain concepts in them. The syntax extensions mechanism of Coq can help users to set up their favorite syntax, but in my opinion, this mechanism is mainly a syntax sugar or abbreviation mechanism, the user level command is limited to tactics of Coq, so it cannot use the theorems and lemmas of the domain in a natural way. Meanwhile Coq or Lego theory may include extra features that the domain does not have, e.g. universes.

[1] LAZY-PCF+SHAR is a lazy version of the functional language PCF(Programming language for Computable Functions) extended by adding explicit substitution in order to formalize the semantics of lazy evaluation.

- Yu [Yu, 1999; Yu and Luo, 1997] takes a hybrid approach, investigating the integration of model checker and proof assistant (Lego) in a number of domains. However, the relation between the model checker and proof assistant is loose, because domain users operate on Lego directly, in a way which does not reflect domain-specific proof procedure.

- PVS[Owre et al., 1997] allows to specify and to verify systems using higher-order logic and presents an integration of tables, types and model checking. A relevant tool TAME [Archer and Heitmeyer, 1997] is designed to support "human-style" reasoning in a particular mathematical model (Lynch-Vaandrager timed automata) through an appropriate mechanism from top layer to PVS. It concentrated on how to design the underlying theorem proving support for it, rather than a high-level interface. In other words, TAME focused on developing PVS strategies for proof steps that closely resemble the steps in hand proofs and not concern about the forms of the expressions in user layer.

- Z/EVES [Saaltink, 1997] is an interactive system which can be used to develop or analyze a Z specification. It is based on the EVES system, and uses the EVES prover to carry out its proof steps, but without the knowledge of EVES or its language (Verdi). Z/EVES has focus on the Z-interface implementation above EVES, not focus on the generic way of building domain-specific interfaces as our LFTOP approach. Z/EVES cannot guarantees that any result on EVES can be translated back to Z. But some restrictions are imposed to operations to ensure that the result can be translated back to Z. [Saaltink, 1997] page 29.

1.5.2 Non theorem prover based approach

Most systems of this approach are based on model checkers, and used in a single domain. For example, the system Truth (Truth/SLC) [Leucker and Noll, 2000] is a design and verification platform for concurrent systems. Its aim is to offer a modular verification system which can be easily adjusted to different settings. It is implemented in Haskell directly and does not depend on any existing theorem prover. It presents some level of user convenience, but *loses the support from theorem prover.* In our opinion, the proof which is done under this system is less convincing than a proof which is done under a theorem prover or proof assistant with the power of proof term generation. For example, soundness depends on a large body of code, which is infeasible to check for errors.

1.6 The structure of the thesis

This thesis is divided into nine chapters. The present chapter introduces the material of this thesis, provides some background concepts and presents the motivations of the thesis.

The rest of the thesis makes extensive use of the logical framework LF and its implementation Plastic.

Chapter 2 concentrates on the statement of problem, the aims of the thesis and the preestablished criteria for the aims.

Chapter 3 introduces the preliminary concepts, relevant theories, their implementations and some basic concepts formalized in LF.

In chapter 4 we give the outline of the system supporting domain specific reasoning. Architectures, methodologies, techniques and the common things for reasoning in different domains are studied in this chapter.

In chapter 5 we present a case study in concurrency. We first analyze the domain, choose CCS as the specification language, LTS as the semantic model, μ-calculus as the specification logic and formalize them in LF. Then we design the user level syntax for the reasoning system for the domain, i.e. design and implement the rules and the user level interface.

In chapter 6 we present another case study in LAZY-PCF+SHAR. A similar but concise discussion is provided in this chapter. The main difference of this case study from the case study in chapter 5 is that we use the interface of Plastic directly. This is due to the observation on the suitability of the direct application of Plastic to this domain.

In chapter 7 we study the issues about interface. We focus on the aspects of the design principle, the protocol and implementations. Especially we design a protocol called ULPIP for the communications between user-level and Plastic-level.

In chapter 8 we discuss translation issues in this thesis. We study the translation between different levels, prove some properties of the translations.

Finally in chapter 9 we conclude our work and review the work left open by this thesis.

Chapter 2

Statement of problem

The important thing in life is to have a great aim, and the determination to attain it.

— JOHANN WOLFGANG VON GOETHE , GERMAN POET AND DRAMATIST

This chapter presents the problems and aims of this thesis. Some criteria for their achievement are outlined also.

2.1 Aim 1: The analysis of requirements of domain-specific reasoning

2.1.1 Basic assumptions of our research

As indicated in chapter 1, reasoning tasks are becoming prominent tasks in some domains. The domain users hope that computer can be used not only for computation, but also for reasoning. There are many different ways to support the reasoning work in many domains by computer system. For example, a lot of specific reasoning tools which are direct implementations of formal systems of specific domains and generic reasoning tools such as proof assistants are already used for reasoning. We present a new approach and want to investigate the approach to support the domain-specific reasoning under some assumptions. The basic assumptions for our research are:

- We want to benefit from the research results of type theory, so we use a type theory based proof assistant for the underlying reasoning instead of design a domain-specific reasoning system from scratch for each domain.
- We want domain users to work in their domain-oriented way with familiar syntax and semantics instead of being a type theory expert first.

- We do not exclude using type theory directly by some domain users, because type theory is exactly suitable for some domains. Of course the use is through an interface which we recommend Proof General.
- The requirements of domain users who want to do the domain-specific reasoning are different from the requirements of type theory experts with more general interests; so it doesn't follow that we need the same techniques for both of them.

From above we can see that this isn't a matter of providing forms of sugaring for expert users, but a serious attempt to study and understand the issues behind producing computer assisted reasoning tools in a variety of domains, which will in time lead to well-engineered systems and a methodology for producing them.

2.1.2 Aims of the LF based domain-specific reasoning approach

Under the above assumptions, we present an LF based approach. We are interested in the following problems:

- Is it a feasible approach?
- What should we do in this approach in order to realize our intention?

In fact, this is one of the motivations of the research of this thesis. We summarize the aims as follows:

1. To analyze the characteristics of domain-specific reasoning.
2. To get better understanding on the issues behind producing computer assisted reasoning tools in a variety of domains.
3. To provide an architecture of the presented approach and investigate the feasibility of this approach.
4. To provide a relevant methodology (i.e. methods, process etc.) of the approach.
5. Based on the above architecture and methodology, to do case studies in some domains and implement a prototype system based on the case studies.
6. Through the case studies, to investigate the suitability of the approach and analyze the advantages and disadvantages of this approach.

The criteria for above aims are as follows :

1. An analysis of the characteristics of domain-specific reasoning which includes domain-specific notation, higher-level abstraction, design reuse etc.
2. To have learnt from the case studies some of the issues behind producing domain-specific computer assisted reasoning tools.

3. An architecture of the approach and the analysis of feasibility of it.
4. The existence and suitability of a relevant methodology and process.
5. Some case studies for concrete domains including the work of formalization, parser, communication protocols and translations between different levels; implementations of the prototype systems based on the case studies.
6. An analysis of suitability of the approach and the validation of it through case studies.

2.2 Aim 2: The analysis of LF and Plastic as a basis to support domain-specific reasoning

LF has been implemented in Plastic [Callaghan and Luo, 2000b], a form of a proof assistant. But there are no big applications based on the LF based proof assistant till now. Is it suitable for big applications? This is another one of the motivations of the research of this thesis. So we have these aims to complete:

1. To analyze the issues of LF as an underlying basis for domain-specific reasoning.
2. Try to implement some big applications to make sure that this kind of system is suitable for big applications.
3. Try to get feedback from direct application of this system to answer the questions such as which are the theoretical and practical benefits and defects of using it instead of other proof assistants.

The criteria for the above aims may be as follows:

1. An analysis of LF's suitability related to be an underlying reasoning basis.
2. Some big applications implemented in the system as a proof of the capability of the system.
3. A summary about the benefits and defects of using LF.

But these works can be seen as the formalization works in Aim1 for some domains. Especially the case study in Chapter 6 can be seen as the proof of the capability of LF and Plastic. So there are no strict separation between Aim1 and Aim2.

2.3 Aim 3: The theoretical aspects of the approach

There will be some theoretical aspects related to the approach and these will be studied during the work.

The criteria for this aim is the relevant presentations of the corresponding analysis and proofs.

Chapter 3

Preliminaries

Histories make men wise; poems witty; the mathematics subtle; natural philosophy deep ; moral grave ; logic and rhetoric able to contend .

— FRANCIS BACON , BRITISH PHILOSOPHER

The main purpose of this chapter is to introduce the notations used throughout the thesis, and make the thesis more self-contained.

3.1 Typed Lambda-calculus

As presented in [Barendregt, 1990], the lambda calculus was originally conceived by Church [Church, 1932] as part of a general theory of functions and logic, intended as a foundation for mathematics. Although the entire system turned out to be inconsistent, as pointed out in Kleene and Rosser [Kleene and Rosser, 1935], the subsystem dealing with functions only became a successful model for computable functions. This system is called now the *lambda calculus*. Representing computable functions as λ-terms, i.e. as expressions in the lambda calculus, gives rise to so-called *functional programming*. There are also typed versions of the lambda calculus. Curry [Curry, 1934] introduced a typed variant of the lambda calculus, called combinatory logic. Church [Church, 1940] gave his formulation of the simple theory of types. The two original papers of Curry and Church introducing typed versions of the lambda calculus give rise to two different families of systems. In the typed lambda calculi *à la Curry* terms are those of the type-free theory. Each term has a set of possible types. In the system *à la Church* the terms are annotated versions of the type-free terms. Each that is derivable from the way the term is annotated. The Curry and Church approaches to typed lambda calculus correspond to two paradigms in programming. In many important systems, especially those *à la Church*, it is the case that terms that do have a type always possess a normal form. By the unsolvability of the halting problem this implies that not all computable functions can be represented by a typed term [Barendregt, 1990].

The λ-calculus with its β-reduction rule is very useful for formalizing mathematics and computing expression. It is also a useful tool for expressing semantics of programming language. And β-reduction satisfies the Church-Rosser (diamond) property. However, for untyped (or type-free) λ-terms, some of them cannot be reduced to normal forms, this means they have infinite reduction sequence [Sørensen and Urzyczyn, 1998]. This is one reason why we need simply typed λ-calculus. In simply typed λ-calculus system (i.e. $\lambda \rightarrow$), the properties such as Church-Rosser, Subject Reduction, and Strong Normalization hold [Sørensen and Urzyczyn, 1998]. This means every well-typed λ-term can be reduced to a normal form, keep its type, and reduce to the same normal form no matter which way it is reduced. Therefore we can easily figure out whether two typable terms are β-equal by just reducing the terms to their respective normal forms and comparing them.

3.2 Type theory

Type theory is designed originally as a basis for formalising constructive mathematics. But scientists have found a lot of applications of it in computer science. Type theory offers a coherent treatment of two related but different fundamental notions in computer science: Computation and logical inference. This makes it possible for one to program, to understand and to reason about programs in a single formalism. Meanwhile type theory can provide nice abstraction mechanisms which support conceptually clear development of specifications, programs, and proofs. The gap in other specification languages between the programming language and the specifications vanished. So we can say that type theory is a very useful theory to support the technology for computer assisted reasoning, such as formalized mathematics or program verification.

Our description here is based on the work of Martin-Löf, in particular Martin-Löf's book [Martin-Löf, 1984] and Nordström, Petersson and Smith's book [Nordström et al., 1990]. But we call the entities "types" where Martin-Löf calls the entities in his theory "sets".

3.2.1 Objects, types, and rules

Type theory can be viewed as a formal language based on a conceptual organization of objects. A type is a collection of objects with some common property or structure. In type theory we are interested in the validity of some property or whether some object has a property. We are also interested in whether different expressions denote the same object in a type. Type theory contains rules for making judgements of the following four forms:

- A is a type.
- A and B are equal types.
- a is an object of the type A.

- a_1 and a_2 are equal objects of the type A.

Among the objects of a type, some are called *canonical objects*, which are the values of objects of the type under computation. A canonical object is in a form that the outermost constructor is an introduction constant. This form is called *canonical form*. The notion of *computation* is a basic concept in type theory, which generates an equivalence relation, the *computational equality* between the basic expressions in the language of type theory. In order to guarantee the harmony between the different uses of the entities in the type theory, the computation should have the property that *every object has a unique value under computation and the objects which are computationally equal have the same value.*

There is a common pattern in the rules for introducing types in type theory. Each type will be defined by giving rules in each of four general categories:

- The **formation rules** for A describe under which conditions we may infer that A is a type and when two types A and B are equal.
- The **introduction rules** define the type A in that they prescribe how the canonical objects are formed and when two canonical objects are equal. The constructors for the type are introduced in these rules.
- The **elimination rules** are a kind of structural induction rules. They allow us to define functions or programs on the type. The function, which is a primitive non-canonical constant associated with the type, is introduced in this kind of rule. It is the function which makes it possible to do pattern-matching and primitive recursion over the objects in the type.
- The **equality rules** describe the equalities which are introduced by the computation rules for the function associated with the type. They relate the introduction and elimination rules. They show how the function defined by the elimination rule behaves on the canonical objects of the type.

3.2.2 The principle of propositions-as-types

The principle of *propositions as types* is based on the observation by Curry and Howard [Curry and Feys, 1958] [Howard, 1980] of the close correspondence between systems of natural deduction for intuitionistic logical inference and type systems. It can be viewed as a fundamental idea in the justification of type theory as a foundation for constructive mathematics or as a basis for specification and verification of programs.

The basic idea of this principle is that any proposition P corresponds to a type $\mathbf{Prf}(P)$, the type of its proofs, and a proof of P corresponds to an object of type $\mathbf{Prf}(P)$. Furthermore, one can assert a proposition to be true if and only if one has a proof of the proposition, that is, an object of the type of its proofs. So the truth of a proposition is understood by the inhabitance of the type of proofs of the proposition. The notion of canonical objects for type

Prf(P) gives a notion of *canonical or direct proofs* of proposition P, while the non-canonical objects of type **Prf**(P) may be called *indirect proofs* of proposition P.

There is a fundamental distinction between propositions which are formulas describing properties and facts, and judgements, which are assertions of whether formulas are true. On the basis of this distinction, a type theory with sufficient logical type structures has an internal logic and presents a logical language rather different from that of set theory or that of logic programming.

The system studied by Curry and Howard were systems for which there was an equivalence between propositions and types. This equivalence holds for various logics and type theories: for example, an extension of the simply typed lambda calculus corresponds to full intuitionistic first-order propositional logic, as developed by Howard [Howard, 1980]; and System F corresponds to second-order propositional logic, as the former type theory and the equivalence were studied by Girard [Girard, 1972]. For this reason the propositions-as-types principle is also referred to as an isomorphism. According to the propositions-as-types principle, we have mapped proofs of a proposition to objects of the type of proofs of the proposition. The judgements of type theory must therefore be decidable, so that we can tell from the form of a judgement $M : A$ that M is indeed an object of the type A.

An alternative view is that although type theory provides a framework in which to understand both logical inference and computation, we need not identify these two things. We can treat propositions as types, but not vice versa. Zhaohui Luo [Luo, 1994] lists several reasons for viewing the identification of propositions and types as unnatural: Firstly, the logic of our system should be independent of the objects studied in it; secondly, certain types such as the natural numbers do not intuitively correspond to propositions; thirdly, type theory is often considered open to the addition of new types representing new computational or mathematical objects, but the addition of these objects should not change the way we reason in the logic. Furthermore, results about the conservativity of type theories which identify propositions and types over their related logic [Berardi, 1990; Luo, 1990b] show that this identification does not correspond to the traditional way of formulating logics.

3.3 The Logical Framework and its application

3.3.1 Logical Framework (LF)

A logical framework may be used in various ways. The Edinburgh Logical Framework [Harper et al., 1987] has been studied for formalisation of logical systems based on the idea of judgements-as-types. Martin-Löf's logical framework(see Part III of [Nordström et al., 1990]) has been proposed by Martin-Löf to present his intensional type theory. The logical framework (LF) which we are interested here is Martin-Löf's LF with type labels on all binders (i.e.,$[x : K]k$ rather than $[x]k$). The extra type labels ensure that type checking is decidable for this LF, where as for Martin-Löf's LF it is only decidable for a subset of terms

[Barthe and Sørensen, 2000][Callaghan and Luo, 2000b]. We can use LF as a meta-language to specify type theories. LF is a simple type system with terms of the following forms:

- **Type**
- $El(A)$
- $(x : K)K'$
- $[x : K]k'$
- $f(k)$

where the free occurrences of variable x in K' and k' are bound by the binding operators $(x : K)$ and $[x : K]$, respectively. There are five forms of judgements in LF:

- Γ **valid**, which asserts that Γ is a valid context;
- $\Gamma \vdash K$ **kind**, which asserts that K is a kind;
- $\Gamma \vdash k : K$, which asserts that k is an object of kind K;
- $\Gamma \vdash k = k' : K$, which asserts that k and k' are equal objects of kind K; and
- $\Gamma \vdash K = K'$, which asserts that K and K' are equal kinds

The rules in LF are given in figure 3.1. We can use it to customize a specific type theory, or use it as a small type theory directly.

There are several reasons to be interested in LF.

- Theoretically, it allows a clearer and more satisfactory presentation. Specifically, there is a clear distinction between an object language (to be used for reasoning and programming) and the meta-level mechanisms which are used to define the object language.
- As LF itself is a concise type theory, so its implementation system (Plastic) does not have much inherent properties which are not easy to be waived.

Luo introduces LF as a meta-language for specifying a type theory (e.g. UTT(Unified Type Theory))

3.3.2 Specifying type theories in LF

To specify a type theory we just need to do two kinds of things. Firstly, we should declare new constants. Secondly, we should give computation rules(usually about the new constants). Formally, declaring a new constant k of kind K by writing k:K, is to extend the type theory(specified by means of LF) to which the constant is introduced by the following inference rule:

Contexts and assumptions

$$\frac{}{\langle\rangle\ \textbf{valid}} \qquad \frac{\Gamma \vdash K\ \textbf{kind}\quad x \notin FV(\Gamma)}{\Gamma, x{:}K\ \textbf{valid}} \qquad \frac{\Gamma, x{:}K, \Gamma'\ \textbf{valid}}{\Gamma, x{:}K, \Gamma' \vdash x : K}$$

General equality rules

$$\frac{\Gamma \vdash K\ \textbf{kind}}{\Gamma \vdash K = K} \qquad \frac{\Gamma \vdash K = K'}{\Gamma \vdash K' = K} \qquad \frac{\Gamma \vdash K = K' \quad \Gamma \vdash K' = K''}{\Gamma \vdash K = K''}$$

$$\frac{\Gamma \vdash k : K}{\Gamma \vdash k = k : K} \qquad \frac{\Gamma \vdash k = k' : K}{\Gamma \vdash k' = k : K} \qquad \frac{\Gamma \vdash k = k' : K \quad \Gamma \vdash k' = k'' : K}{\Gamma \vdash k = k'' : K}$$

Equality typing rules

$$\frac{\Gamma \vdash k : K \quad \Gamma \vdash K = K'}{\Gamma \vdash k : K'} \qquad \frac{\Gamma \vdash k = k' : K \quad \Gamma \vdash K = K'}{\Gamma \vdash k = k' : K'}$$

Substitution rules

$$\frac{\Gamma, x{:}K, \Gamma'\ \textbf{valid} \quad \Gamma \vdash k : K}{\Gamma, [k/x]\Gamma'\ \textbf{valid}}$$

$$\frac{\Gamma, x{:}K, \Gamma' \vdash K'\ \textbf{kind} \quad \Gamma \vdash k : K}{\Gamma, [k/x]\Gamma' \vdash [k/x]K'\ \textbf{kind}} \qquad \frac{\Gamma, x{:}K, \Gamma' \vdash K'\ \textbf{kind} \quad \Gamma \vdash k = k' : K}{\Gamma, [k/x]\Gamma' \vdash [k/x]K' = [k'/x]K'}$$

$$\frac{\Gamma, x{:}K, \Gamma' \vdash k' : K' \quad \Gamma \vdash k : K}{\Gamma, [k/x]\Gamma' \vdash [k/x]k' : [k/x]K'} \qquad \frac{\Gamma, x{:}K, \Gamma' \vdash k' : K' \quad \Gamma \vdash k_1 = k_2 : K}{\Gamma, [k_1/x]\Gamma' \vdash [k_1/x]k' = [k_2/x]k' : [k_1/x]K'}$$

$$\frac{\Gamma, x{:}K, \Gamma' \vdash K' = K'' \quad \Gamma \vdash k : K}{\Gamma, [k/x]\Gamma' \vdash [k/x]K' = [k/x]K''} \qquad \frac{\Gamma, x{:}K, \Gamma' \vdash k' = k'' : K' \quad \Gamma \vdash k : K}{\Gamma, [k/x]\Gamma' \vdash [k/x]k' = [k/x]k'' : [k/x]K'}$$

The kind Type

$$\frac{\Gamma\ \textbf{valid}}{\Gamma \vdash \textbf{Type}\ \textbf{kind}} \qquad \frac{\Gamma \vdash A : \textbf{Type}}{\Gamma \vdash El(A)\ \textbf{kind}} \qquad \frac{\Gamma \vdash A = B : \textbf{Type}}{\Gamma \vdash El(A) = El(B)}$$

Dependent product kinds

$$\frac{\Gamma \vdash K\ \textbf{kind} \quad \Gamma, x{:}K \vdash K'\ \textbf{kind}}{\Gamma \vdash (x{:}K)K'\ \textbf{kind}} \qquad \frac{\Gamma \vdash K_1 = K_2 \quad \Gamma, x{:}K_1 \vdash K_1' = K_2'}{\Gamma \vdash (x{:}K_1)K_1' = (x{:}K_2)K_2'}$$

$$\frac{\Gamma, x{:}K \vdash k : K'}{\Gamma \vdash [x{:}K]k : (x{:}K)K'} \qquad (\xi) \qquad \frac{\Gamma \vdash K_1 = K_2 \quad \Gamma, x{:}K_1 \vdash k_1 = k_2 : K}{\Gamma \vdash [x{:}K_1]k_1 = [x{:}K_2]k_2 : (x{:}K_1)K}$$

$$\frac{\Gamma \vdash f : (x{:}K)K' \quad \Gamma \vdash k : K}{\Gamma \vdash f(k) : [k/x]K'} \qquad \frac{\Gamma \vdash f = f' : (x{:}K)K' \quad \Gamma \vdash k_1 = k_2 : K}{\Gamma \vdash f(k_1) = f'(k_2) : [k_1/x]K'}$$

$$(\beta)\quad \frac{\Gamma, x{:}K \vdash k' : K' \quad \Gamma \vdash k : K}{\Gamma \vdash ([x{:}K]k')(k) = [k/x]k' : [k/x]K'} \qquad (\eta)\quad \frac{\Gamma \vdash f : (x{:}K)K' \quad x \notin FV(f)}{\Gamma \vdash [x{:}K]f(x) = f : (x{:}K)K'}$$

Figure 3.1: The inference rules of LF from [Luo, 1994].

$$\frac{\Gamma \textbf{ valid}}{\Gamma \vdash k{:}K}$$

and, for a kind K which is either **Type** or of the form $El(A)$, asserting a computation rule by writing

$$k = k'{:}K \quad \text{for } k_i{:}K_i (i = 1, ..., n),$$

is to extend the type theory by the following equality inference rule,

$$\frac{\Gamma \vdash k_i{:}K_i (i = 1, ..., n) \;\; \Gamma \vdash k{:}K \;\; \Gamma \vdash k'{:}K}{\Gamma \vdash k = k'{:}K}$$

The special kind **Type** in LF corresponds to the conceptual universe of types in the type theory to be specified. Let T be any type theory specified in LF. Then, a $T - context$ is a context of the form $x_1{:}El(A_1), ..., x_n{:}El(A_n)$, and T has the following five forms of judgements (where Γ is any $T - context$):

- Γ **valid**, which asserts that Γ is a valid $T - context$;
- $\Gamma \vdash A{:}$**Type**, which asserts that A is a type;
- $\Gamma \vdash a{:}El(A)$, which asserts that a is an object of type A;
- $\Gamma \vdash a = b{:}El(A)$, which asserts that a and b are computationally equal objects of type A in the sense that they compute to the same value; and
- $\Gamma \vdash A = B{:}$**Type**, which asserts that A and B are equal types in the sense that they have the same objects.

A judgement in a type theory specified in LF is *derivable* if it is derivable in the system of LF extended by the constants and computation rules specifying the type theory. Once a type theory is specified, the user uses the type theory rather than the LF language, except that he may use LF as a definitional mechanism which may be implemented in a proof development system, e.g. Plastic. In such a use of logical framework as a meta-language, one does not use the meta-logic embedded in LF to reason about objects in the type theory, but should use the internal logic in the specified type theory for reasoning. An inductive schema is introduced to LF. The essential idea is that each finite sequence of inductive schemata specifies a collection of introduction rules (each schema in the sequence determines one of them) and hence generates an inductive data type whose meaning is given by the introduction rules, the associated elimination and computation rules. We use LF_Θ to express the LF with inductive schema.

3.3.3 The type theory UTT

The type theory UTT is specified in LF_Θ. It consists of an internal logic, a large class of inductive data types, and universes.

3.3.3.1 Internal logic SOL and definition of Π

The internal logic(called SOL) consists of a universe *Prop* of logical propositions and their proof types. The logical universe *Prop* is impredicative since universal quantification can be formed for any type A and (meta-level)predicate P over A. Similar to ECC [Luo, 1990a], many of the usual logical operators can be defined by means of the impredicative universal quantification. However, the internal logic SOL of UTT by itself is only a second-order logic (hence the name). There are no types of internal predicates or internal relations in SOL over which universal quantification may be possible. The internal logic is introduced by declaring the following constants:

$$\begin{aligned}
Prop &: \mathbf{Type}\\
\mathbf{Prf} &: (Prop)\mathbf{Type}\\
\forall &: (A{:}\mathbf{Type})((A)Prop)Prop\\
\Lambda &: (A{:}\mathbf{Type})(P{:}(A)Prop)((x{:}A)\mathbf{Prf}(P(x)))\mathbf{Prf}(\forall(A,P))\\
E_{\forall} &: (A{:}\mathbf{Type})(P{:}(A)Prop)(R{:}(\mathbf{Prf}(\forall(A,P)))Prop)\\
&\quad ((g{:}(x{:}A)\mathbf{Prf}(P(x)))\mathbf{Prf}(R(\Lambda(A,P,g))))\\
&\quad (z{:}\mathbf{Prf}(\forall(A,P)))\mathbf{Prf}(R(z))
\end{aligned}$$

and asserting the following computation rule:

$$E_{\forall}(A,P,R,f,\Lambda(A,P,g)) = f(g){:}\mathbf{Prf}(R(\Lambda(A,P,g))).$$

Then, the usual application operator can be defined as

$$\begin{aligned}
\mathbf{App} \;=_{df}\; & [A{:}\mathbf{Type}][P{:}(A)Prop][F{:}\mathbf{Prf}(\forall(A,P))][a{:}A]\\
& E_{\forall}(A,P,[G{:}\mathbf{Prf}(\forall(A,P))]P(a),[g{:}(x{:}A)\mathbf{Prf}(P(x))]g(a),F),
\end{aligned}$$

which satisfies the equality (the β-rule for Λ and **App**):

$$\mathbf{App}(A,P,\Lambda(A,P,g),a) = g(a){:}\mathbf{Prf}(P(a)).$$

In LF, we can introduce dependent product types by declaring the following constants:

$\Pi : (A{:}\mathbf{Type})((A)\mathbf{Type})\mathbf{Type}$

$\lambda : (A{:}\mathbf{Type})(B{:}(A)\mathbf{Type})((x{:}A)B(x))\Pi(A,B)$

$$E_\Pi : (A{:}\mathbf{Type})(B{:}(A)\mathbf{Type})(C{:}(\Pi(A,B))\mathbf{Type})$$
$$((g{:}(x{:}A)B(x))C(\lambda(A,B,g)))(z{:}\Pi(A,B))C(z)$$

and asserting the following computation rule:

$E_\Pi(A,B,C,f,\lambda(A,B,g)) = f(g) : C(\lambda(A,B,g))$.

For Π-types, the application operator can be defined as follows:

$\mathbf{app} =_{df} [A{:}\mathbf{Type}][B{:}(A)\mathbf{Type}][F{:}\Pi(A,B)][a{:}A]$
$E_\Pi(A,B,[G{:}\Pi(A,B)]B(a),[g{:}(x{:}A)B(x)]g(a),F)$

SOL together with the Π-types is essentially a formulation of the Calculus of Constructions [Coquand and Huet, 1988] in LF.

3.3.3.2 Inductive types

Inductive types in UTT are based on the notion of inductive schemata. Any finite sequence of inductive schemata specifies a collection of introduction rules(each schema in the sequence determines one of them) and hence generates an inductive type whose meaning is given by the introduction rules (and the associated elimination and computation rules). The similar idea has been considered by Gentzen [Gentzen, 1935], Prawitz [Prawitz, 1973; Prawitz, 1974], etc. for traditional logical systems, and by Martin-Löf[Martin-Löf, 1984], Backhouse [Backhouse, 1988], Dybjer [Dybjer, 1991], and Coquand and Mohring [Coquand and Paulin-Mohring, 1990] for type theories.

For example, a type Nat of natural numbers can be defined as $Nat =_{df} \mathcal{M}[\bar{\Theta}]$, where $\bar{\Theta}$ represents the kinds of the constructors— in this case, kinds X and $X \to X$ where X is a placeholder for the name of the inductive type. The associated introduction operator are $zero =_{df} \iota_1[\bar{\Theta}]{:}Nat$ and $succ =_{df} \iota_2[\bar{\Theta}]{:}Nat \to Nat$. The elimination operator and computation rules are as the following:

$E_{Nat} =_{df} E[\bar{\Theta}]$
$: (C{:}Nat \to \mathbf{Type})(c{:}C(zero))\ (f{:}(x{:}Nat)C(x) \to C(succ(x)))(n{:}Nat)C(n)$,

$E_{Nat}(C,c,f,zero) = c$
$E_{Nat}(C,c,f,succ(x)) = f(x, E_{Nat}(C,c,f,x))$.

3.3.3.3 Universes

The universes in UTT are the impredicative universe $Prop$ and the predicative universes $Type_i (i \in \omega)$ in Tarski style. In this style types in universes are represented by codes (i.e., names) and a decoding function which maps such names to appropriate types. This is contrasted against Russell style, where codes and the types they represent are identified. The theory ECC contains a hierarchy of universes in Russell style.

A typical example of using universe is to prove the distinctness of constructors of inductive types [Smith, 1988]. Callaghan gave a proof in Plastic for the boolean type, i.e.

$true \neq_{Bool} false$; see section 4.5.2 of [Callaghan and Luo, 2000b] for details. Such distinctness cannot be proved without universes [Smith, 1988].

UTT also has the nice meta-theoretic properties such as subject reduction and strong normalisation. Goguen had proved these properties in his Ph.D. thesis [Goguen, 1994].

3.3.4 Distinction between dependent product kind and Π-types

Dependent product kind and Π-types are two different notions which often cause confusion. We can list the following differences between dependent product kind and Π-type:

- The notion of dependent product kind is one in the meta-framework, while Π-type is a notion of some object type theories.
- The dependent product kind provides parameterisation mechanisms which can be used to define a type theory, while a Π-type is an inductively defined construct representing the *type* of dependent function in an object language.
- An important difference is that there is a notion of elimination for Π-types but not for dependent product kinds.

For example, an object of dependent product kind[1]

$$(A{:}\mathbf{Type})(B{:}(x{:}A)\mathbf{Type})\mathbf{Type}$$

is a family of types parameterised by a type A and a family of types B indexed by objects of type A. Representing such a family by means of an "internal" Π-type is inappropriate and leads to possible misunderstandings in use of type theory.

3.3.5 A new version of Logical Framework PAL+

In fact, a clearer explanation for distinguishing meta and object concepts is from a new version of Logical Framework PAL+[Luo, 2003] . PAL+ is a lambda-free logical framework which takes parameterisation and definitions as the basic notions to provide schematic mechanisms for specification of type theories and their use in practice. It is also a logical framework for specification and implementation of type theories, such as Martin-Löf's type theory or UTT. As in Martin-Löf's logical framework [Nordström et al., 1990] and the above LF, computational rules can be introduced and are used to give meanings to the declared constants. However, PAL+ only allows one to talk about the concepts that are intuitively in the object type theories: types and their objects, and families of types and families of objects of types. In particular, in PAL+, one cannot directly represent families of families of entities, which could be done in other logical frameworks by means of lambda abstraction. Just as implied in its name, PAL+ can be seen as a successor of de Bruijn's PAL for Automath [de Bruijn, 1980]. Compared with PAL, PAL+ allows one to represent

[1] In fact Π is declared as a constant of this kind in UTT

parametric concepts such as families of types and families of non-parametric objects, which can be used by themselves as totalities as well as when they are fully instantiated. Such parametric objects are represented by local definitions (let-expressions). PAL+ is a correct meta-language for specifying type theories (e.g., dependent type theories), as it has the advantage of exactly capturing the intuitive concepts in object type theories, and that its implementation realises the actual use of type theories in practice. Luo [Luo, 2003] studies the meta-theory of PAL+ by developing its typed operational semantics and shows that it has nice meta-theoretic properties.

As a complete implementation of PAL+ has not been done yet, so we still use *LF* and its implementation system Plastic as the basis of our research.

3.4 Implementations of type theory

Several systems which are based on type theories have been implemented. An early implementation of type theory with many important contributions is de Bruijn's Automath project [de Bruijn, 1980].In this project de Bruijn introduced the idea of using type theory as a system which can serve as a framework for implementing logics, by giving a system which formalizes the underlying principles which mathematicians agree upon.

Lego [Luo and Pollack, 1992] implements several different type theories: The Edinburgh Logical Framework [Harper et al., 1987]; ECC [Luo, 1990a] and the Pure Calculus of Constructions [Coquand and Huet, 1988].

Coq [Project, 2004] is a Proof Assistant based on the Calculus of Inductive Constructions. ALF [Magnusson and Nordström, 1994] is a structure editor for Martin-Löf's type theory in the Logical Framework, including a window-based user interface. Nuprl [Constable et al., 1986] implements a variant of Martin-Löf's polymorphic and extensional type theory, and unlike some other type theories, type checking in Nuprl is not decidable, so the elements of propositional types should not be interpreted as proofs but merely represents the computational contents of the associated proposition.

Isabelle [Paulson, 1999] is an interactive theorem prover that supports a variety of logics, such as higher order logic (HOL), Zermelo Fraenkel set theory (ZF), and constructive type theory (CTT).

Plastic [Callaghan and Luo, 2000b] is an implementation of typed *LF* with coercive subtyping and universes. It is different from Lego and Coq because it is not intended to be used directly by expert users but as the underlying layer for other systems. Our further study will be based upon Plastic.

3.5 Model checking

A *model checking problem* is a problem of checking whether a given model satisfies a given property:

$$M \models \psi$$

where, the model M represents a design, and the property ψ represents its correctness criteria. In general, as a popular automatic verification technique, model checking has focused on automatic decision procedures for solving its verification problem. The basic idea is to determine whether a model satisfies a property expressed as a temporal logic formula by searching the state space of the model thoroughly. Therefore, to guarantee the termination, the model is often restricted to a finite state system, and properties are expressed in a propositional temporal logic like CTL or LTL, for which finite-state model checking is known to be *decidable*. The main obstacle encountered by model checking is the so-called *state explosion problem* that the size of the state transition graph grows exponentially while the size of the system grows linearly. But it is important to understand that model checking problem is not limited to finite state systems or propositional logics, symbolic model checking [McMillan, 1992; McMillan, 2005] can be used to deal with the state explosion problem.

ACM awarded the 1998 ACM Kanellakis Award for Theory and Practice to Randal E. Bryant, Edmund M. Clarke, Jr., E. Allen Emerson, and Kenneth L. McMillan for their invention of "symbolic model checking", a method of formally checking system designs widely used in the computer hardware industry. The technique has shown significant promise when used for software verification and in other areas.

Symbolic model checking is one of the most important formal techniques used in the computer and semiconductor industries today, and the SMV program, originally developed by Kenneth McMillan as part of his Ph.D. program, is one of the most widely used verification tools. These industries face a complexity explosion of near-crisis proportions, with six-month design cycles in which products of unprecedented complexity have to be "right" the first time for companies to survive. Symbolic model checking offers design teams shorter time to market and increased product integrity, which explains the rapid adoption of this technology by all leading semiconductor companies.

Model checking is a technique for verifying finite state concurrent systems such as sequential circuit designs and communication protocols. It has a number of advantages over traditional approaches that are based on simulation, testing, and deductive reasoning. In particular, model checking is automatic and usually quite fast. Also, if the design contains an error, model checking will produce a counterexample that can be used to pinpoint the source of the error. The method has been used successfully in practice to verify real industrial designs, and many companies are beginning to market commercial model checkers.

We want to include model checking technology in our approach to deal with some suitable domain-specific problems.

3.6 Theorem proving

Unlike model checking, theorem proving utilizes the proof inference technique in some proof system for solving the general validity of a problem.

DeBruijn's Automath project was an early and influential investigation into techniques for mechanically proof-checking mathematics [de Bruijn, 1980]. Van Jutting [Jutting, 1977] formalized all of a foundational text on elementary analysis — Landau's "Grundlagen" — in Automath. Recently, more mathematics has been formalized in the MIZAR system [Trybulec, 2005].MIZAR is based on classical first-order predicate logic, extended with second order schema, and Tarski-Grothendieck set theory. Roughly speaking, this set theory is like Zermelo-Fraenkel set theory, extended with uncountably many inaccessible cardinals. Till July 2005, over two thousands of definitions of mathematical concepts and thirty thousands of theorems are included in the Mizar database. All work done in Mizar is grouped into articles. Articles are published in a Journal of Formalized Mathematics which is largely automatically type-set from information in the MIZAR database. The subjects of the articles have been mostly in the fields of analysis, topology and algebra (including some universal algebra and category theory). I think that the keys to MIZAR's success are as follows:

- It started with a set theoretic framework which is known to be theoretically adequate for all of mathematics, including category theory.
- A rich type theory was layered on top of the set theory. The type theory allows for the definition of subtypes and parameterized types, and has a structure facility for the definition of algebraic classes. The system copes automatically with set subtyping relationships between elements of classes that have different underlying signatures.
- Much effort has been put into the organization of articles in the MIZAR database to ease and speed cross-referencing between articles.

In terms of applying theorem provers to hardware and software verification, the NQTHM system of Boyer and Moore [Boyer and J S. Moore, 1979; Boyer and J S. Moore, 1997] is fruitful. Accomplishments include the checking the RSA public key encryption algorithm [Boyer and Moore, 1984] and the verification of microprocessor designs [Hunt et al., 1992]. NQTHM has also been used to formalize Gödel incompleteness theorem [Shankar, 1986]. The generation of proofs in NQTHM is highly automated. The user commonly only guides proofs by perhaps giving a few high level hints and suggesting useful lemmas. NQTHM automatically guesses how to do inductions and how to prove the subgoals of inductions. NQTHM also has a linear arithmetic decision procedure tightly integrated in with the the prover program. But NQTHM's logic is weak: it is quantifier free and includes a theory of recursive functions over Lisp like S-expressions. Its strength is roughly that of Primitive Recursive Arithmetic (PRA). This logic is too weak for abstract algebra: there is no way to

define algebraic classes of objects and reason with them in ways common in algebra, though 'functional instantiation' extensions do allow some basic algebraic reasoning.

The HOL system [Gordon and Melham, 1993a]is a tactic based interactive theorem prover with a classical logic similar to Church's simple theory of types [Church, 1940] but with the addition of a type polymorphism scheme similar to that found in the ML functional programming language. This theory is slightly weaker than ZF set theory. HOL has mostly been used in domains related to hardware and software verification, though its foundational theories are quite general purpose and some success has been had with more abstract mathematics. Recently, many works about formalization of network protocols in HOL have been done [Steve Bishop and et al., 2005].

We pay more attention to the theorem provers which are based on intuitionistic type theory and using "proposition as type" principle. They include Alf [Magnusson and Nordström, 1994], LEGO [Luo and Pollack, 1992] [Pollack, 2005], Coq [Project, 2004] and Plastic [Callaghan and Luo, 2000b] etc. Alf is a proof editor based on Martin-Löf's type theory and explicit substitution. Coq uses the Calculus of Constructions [Coquand and Huet, 1988]. Lego uses ECC [Luo, 1990a] and Plastic uses LF. They are *LCF-style* theorem provers [Gordon et al., 1979]. Usually the problem itself is represented as a *sequent*. The sequent used in *natural deduction* is in the form of $\Gamma \vdash F$. We say a sequent *holds* when it satisfies its intended semantics. In general, theorem provers cannot prove theorem without guidance of users, i.e. they are interactive system and only experienced experts can use them effectively.

From now on we'll focus on the applications of Plastic.

3.7 Some basic concepts formalized in LF

3.7.1 A brief introduction to Plastic

Plastic is an implementation of LF with inductive types, universes and coercive subtyping. It is a proof assistant with a similar style as Lego and Coq. Plastic is implemented in functional language Haskell. It is best used with Aspinall's Proof General interface for xemacs. Currently Plastic uses a script-based model of interaction.

3.7.1.1 The syntax of Plastic

The following Figure 3.2 shows the correspondence of LF syntax and Plastic syntax.

Currently Plastic maintains a simple linear context. Its context may contain hypotheses (or assumptions), declarations of inductive types, and global definitions. Plastic provides a form of meta-variable to fill in information which is inferrable with simple unification techniques. New meta-variables may be added to the context at any time by claiming a name of a given type. Fresh meta-variables can appear in a term as either named (e.g. ?lemma1, where

LF syntax	`Plastic syntax`	Explanation
$(x{:}K)K'$	`(x:K)K'`	dependent products
$K \to K'$	`K -> K'`	non-dependent products
$[x{:}K]k$	`[x:K]k`	λ-abstractions
$f(a,b)$	`f a b`	function application
$\forall$	`FA`	universal quantification
Λ	`LL`	a constructor, which builds proofs of quantifications over propositions.
$E_\forall$	`E_FA`	the elimination operator of $\forall$

Figure 3.2: The correspondence of LF syntax and Plastic syntax from [Callaghan and Luo, 2001]

lemma1 is a name chosen by the user) or unnamed (e.g. symbol ?). If the meta-variables are not solved by constraints within the term, then they are added to the relevant context.

3.7.1.2 Syntactic Sugar in Plastic

Binders in specified type theories (eg SOL) are not easy to use, so a few things are implemented in Plastic to make it more palatable.

There are three forms:

By Arrow regard an arrow as an infix operator between two terms, which produces a non-dependent binding. Converted to an application of a non-dependent binder to the two terms. Eg `a -> b`

By Symbol follow the general pattern `left_bracket id : term right_bracket term`. A selection of bracketing symbols is available. Eg `{x:A}B`

By Identifier look like a functional operation binding except the opening square bracket is preceded by the name of the binder. Eg `FA[x:A]B`

For example, `tautology` can be written as below. Braces denote "for all" binding, and `=>` denotes propositional implication.

```
tautology = {p:Prop}p => p
tautology = FA [p:Prop] Imp p p
tautology = FA Prop([p:Prop] Imp p p) --expanded version.
```

3.7.1.3 Coercive subtyping implementation in Plastic

The notion of Coercive Subtyping is first inducted into LF in [Luo, 1999]. Since then many studies on it are done in [Luo and Soloviev, 1999; Luo, 2004]. Coercive Subtyping is viewed as a mechanism of abbreviation of the meta-language (LF), not a part of a particular object

type theory. The mechanism of subtyping is expressed as a fundamental part of LF. So object type theories can make use of it by virtue of their definition in LF.

A coercion is a function $c{:}K \rightarrow K'$, which lifts an object of kind K to kind K'. The coercive definition rule is as follows:

$$\frac{f{:}(x{:}K)K' \quad k_0{:}K_0 \quad K_0 <_c K}{f(k_0) = f(c(k_0)){:}[c(k_0)/x]K'}$$

The rule says that, if f is a functional operation with domain K, k_0 is an object of K_0, and c is a coercion from K_0 to K, then $f(k_0)$ abbreviates $f(c(k_0))$ and is equal to $f(c(k_0))$ by definition.

Z.Luo has extended the framework LF with subtyping relations represented by the following kinds of coercions [Luo, 2004]:

- *Simple coercion:* it represents subtyping between two types. For example $Even$ is a subtype of Nat.
- *Parameterised coercions:* it represents subtyping (or subfamily relation) between two families of types indexed by objects of the same type. For example each vector type $Vec(A, n)$ can be taken as a subtype of that of lists $List(A)$, parameterised by the index n, where the coercion would map a vector $< a_1, ..., a_n >$ to the list $[a_1, ..., a_n]$.
- *Coercion between parameterised inductive types:* General schematic rules are provided to represent natural propagation of the basic coercions to other structured (or parameterised) inductive types. For example, $\Sigma(A, B)$ is a subtype of $\Sigma(A', B')$ if A is a subtype of A' and B is a subfamily of B'.

Plastic can be used to test ideas on coercive subtyping. Plastic implements parameterised and dependant coercions, non-dependent subkinding, and the lifting of coercions over inductive types [Callaghan and Luo, 2001]. The coercions are implemented by coercion insertion during type checking. It is justified by the coercion completion results [Soloviev and Luo, 2000]. The relevant parameters are calculated by using the meta-variable mechanism.

3.7.1.4 Modules

A Plastic program consists of a collection of modules. Its form is as follows:

```
> module Modulename where;
> import importedmodule1;
...
> import importedmodulen;
%%declarations, definitions and proof scripts
...
```

Technically speaking, a module is a sequence of declarations, definitions and proof scripts which begins with the keyword `module`. Concrete examples can be seen in the following subsections.

3.7.1.5 Inductive type and family in Plastic

The syntax for definition of simple inductive type is as follows:

```
> Inductive [D:Type]
>   Constructors
>   [C1:M1]
...
>   [Cn:Mn]
```

Where `D` is the name of the new defined type and `Mi (i=1,...n)` is a term in the form of `(x1:N1)...(xm:Nm)D` or `D. N1,...Nm` are existed types or `D` itself. We can see that the syntax is similar to Lego. For example, natural number type `Nat` is introduced like this:

```
> Inductive [Nat:Type]
>   Constructors
>   [zero:Nat]
>   [succ:(n:Nat)Nat]
```

The constants `Nat`, `zero`, and `succ` are declared in the current context, and the elimination rule `E_Nat` is defined as per Luo's scheme.

Inductive families are introduced similarly, with dependent product kinds instead of just `Type`:

```
> Inductive
>       <parameters>
>       [Vec:(n:Nat)Type]
>       <flags>
>   Constructors
>       [vnil:Vec(zero)]
>       [vcons:(m:Nat)(x:A)(l:Vec(m))Vec(succ(m))]
```

`<flags>` are options which affect what is generated for the inductive family. `<parameters>` are declarations which are in force for the inductive family, and (by default) discharged after it is created.

3.7.1.6 Inductive Relations

Plastic supplies Lego-style inductive relations. To define an inductive relation the `Relation` flag in the declaration is needed. The only difference in handling from conventional inductive

types is that Prf must be included everywhere it is required and a Prop is yielded for the relation.

For example: for natural numbers, the "less than or equal" relation:

```
> Inductive [le:(x,y:El Nat)El Prop] Relation
> Constructors
>   [le0 : (m:El Nat)Prf(le zero m)]
>   [leS : (m,n:El Nat)(ih:Prf(le m n))Prf(le (succ m) (succ n))];
```

Because there is no syntactic sugar for handling Props, so Prf must be added explicitly. The resulting elimination operator is this (El is omitted in this output):

```
Hyp E_le :
    (C_le:Nat -> Nat -> Prop)
    ((m:Nat)(Prf (C_le zero m))) ->
    ((m:Nat)
     (n:Nat)
     (Prf (le m n))
     -> (Prf (C_le m n)) -> (Prf (C_le (succ m) (succ n)))) ->
    (x:Nat)
    (y:Nat)
    (Prf (le x y)) -> (Prf (C_le x y))
```

3.7.1.7 Inductive Relations with Large Elimination

As an extension to the above, relations may be given large elimination, that is: instead of producing Props, it may produce (larger) Types. It is triggered with the flag Relation_LE (replacing the plain Relation flag). This follows what Lego does, and allows equality to be defined as an inductive relation with more useful elimination behavior, as shown below.

```
> Inductive [A:Type]
>           [Eq : (x,y:El A)El Prop]
>   Relation_LE
>   Constructors
>           [eqr:(a:El A)Prf(Eq a a)];
```

Hence the elimination operator is this:

```
E_Eq :
    (A:Type)
    (C_Eq:El A -> El A -> Type)
    ((a:El A)El (C_Eq a a))
    -> (x:El A)(y:El A)El (Prf (Eq A x y)) -> El (C_Eq x y)
```

3.7.1.8 Development of proofs in Plastic

In the style of Lego and Coq a goal-directed proof state controls which subgoals the user must prove next. Expert users like this style. Plastic used a more flexible model, where user can work on any unsolved meta-variable in the current linear context. Further, user need not completely solve a meta-variable before attempting another meta-variable. This flexibility is identified as being useful to applications like Mathematical Vernacular [Callaghan and Luo, 1998].

Meta-variables are the central notion: proof is the process of developing instantiations for them. A `Claim` for the kind which represents the goal is the first line for a proof. The proof commands may act on that claimed meta-variable, which could introduce further meta-variables (i.e. sub-goals). The following are some of the main commands:

Refine t: this is a command similar to Lego's corresponding command. It computes a term t to instantiate a meta-variable (e.g. M) of known type. It is implemented in terms of the meta-variable preprocessor. Cut is applied to M using the computed term t, and any new meta-variables arising (i.e. sub-goals) are inserted in the context immediately before M. In other words, the evaluation of a refinement command of the form `Refine t` proceeds as follows:

- First, the system tries to check whether the term `t` is well-typed in the current context. If it is not well-typed, the system should report error message, Otherwise do the next step.
- Second, the system tries to unify the current goal with the type of the refinement term `t`. There are several possibilities:
  1. the unification succeeds: that means the current goal is proved.
  2. the unification fails: then, the system tries to specialize the refinement term `t` by applying it to a new meta-variable of the right type. There are two possibilities: if the specialization succeeds, then several new goals are generated in order to prove the current goal. If the specialization fails, then the refinement step fails.

Intros: when used on a meta-variable $M{:}(x{:}K)K'$, the context appearing after and including M is replaced with a hypothesis $x{:}K$ and a new meta-variable $M'{:}K'$. This creates a branch in the context.

Return: it marks closure of an Intros, i.e. all meta-variables introduced by (and since) the Intros have been solved. The action is to abstract the solution for M' by $x{:}K$ and cut the result into the context existing prior to the corresponding Intros command.

ReturnAll: It closed all opened Intros, namely, all meta-variables introduced by all Intros have been solved. It releases all hypotheses.

We shall give some concrete proofs and examples of applications in following sections and chapters.

3.7.2 Sets and relevant constants and operators

Set theory and the theorem of fixed points are very useful to give a clear semantic interpretation of formal systems. There are different representations of set. We use logical predicates to represent set in this thesis. Although there are many papers which pay more attention to setoids [Barthe et al., 2003], which are more suitable to support extensional concepts such as quotients and subsets, but we don't want to use setoids here and we think that a simple treatment to set is enough.

At first, we define some special sets and operators in λ-notation as follows:

$$
\begin{aligned}
\text{Pred} &= \lambda A{:}Type.A \rightarrow Prop \\
\text{Fullset} &= \lambda A{:}Type.\lambda x{:}A.tautology \\
\text{Emptyset} &= \lambda A{:}Type.\lambda x{:}A.absurd \\
\text{Meet} &= \lambda A{:}Type.\lambda B,C{:}Pred(A).\lambda x{:}A.(B(x)\ and\ C(x)) \\
\text{Union} &= \lambda A{:}Type.\lambda B,C{:}Pred(A).\lambda x{:}A.(B(x)\ or\ C(x)) \\
\text{Not} &= \lambda A{:}Type.\lambda B{:}Pred(A).\lambda x{:}A.(not\ B(x)) \\
\text{Minus} &= \lambda A{:}Type.\lambda B,C{:}Pred(A).Meet\ A\ B\ (Not\ A\ C) \\
\text{Subset} &= \lambda A{:}Type.\lambda B,C{:}Pred(A).\forall x{:}A(B(x) \rightarrow C(x)) \\
\text{Eqset} &= \lambda A{:}Type.\lambda B,C{:}Pred(A).and\ (Subset\ A\ B\ C)(Subset\ A\ C\ B) \\
\text{Single} &= \lambda A{:}Type.\lambda x{:}A.Eq\ x
\end{aligned}
$$

Then, we give some relevant predefined elements (Pi_, La_, ap_) which are defined in a system module called Function in Plastic. The following is an episode for the definitions:

```
%----------------------------------------------------------------
> module Function where;

Function spaces (non-dependent)

> Inductive
>       [ A,B: Type ]
>       [Pi_ : Type ]
>       Constructors
>       [ La_ : (f:(x:El A) El B) Pi_];


Now, the means to use a Pi type.

> Claim ap_ : (A,B:Type)(f:Pi_ A B) -> (_:A)B;
```

```
> Intros A B pi a;
> Refine E_Pi_ ? ? ([_:Pi_ A B]B) ? pi;
> Intros f;
> Refine f a;
> ReturnAll;
%------------------------------------------------------------------
```

In addition, *tautology* and *absurd* are defined in Higher-Order logic as follows:

$$\begin{aligned} P \Rightarrow Q &=_{df} \forall\, x{:}\ \mathrm{Prf}(P).Q \\ \text{tautology} &=_{df} \forall\, P{:}\mathrm{Prop}.\ P \Rightarrow P \\ \text{absurd} &=_{df} \forall\, P{:}\mathrm{Prop}.\ P \end{aligned}$$

In Plastic they are defined in a system module (called Sol_Basics) for Second-Order logic, the following is an episode for the definitions :

```
%------------------------------------------------------------------
> module Sol_Basics where;

Definitions of Common Logical Constants

> import Sol;

%------------------------------------------------------------------
Tautology.

> [tautology = {P:Prop}P=>P : Prop];
> Claim prf_tautology : Prf tautology;
> Refine LL;
> Intros P;
> Refine LL;
> Intros p;
> Refine p;
> ReturnAll;

%------------------------------------------------------------------
Absurd.

> [absurd = {A:Prop}A : Prop ];
>
> [not = [A:Prop]A => absurd : Prop -> Prop];
```

```
> Claim E_absurd : Prf (absurd => {N:Prop}N);
> Refine LL;
> Intros x;
> Refine x;
> ReturnAll;
%-------------------------------------------------------------
```

Finally, We can define set and related constants and operators in Plastic as follows:

```
%-------------------------------------------------------------
> [Pred = [A:Type](Pi_ A Prop)];

> [Fullset = [A:Type](La_ A Prop [x:A]tautology)];

> [Emptyset = [A:Type](La_ A Prop [x:A]absurd)];

> [Meet = [A:Type][B:(Pred  A)][C:(Pred  A)](La_ A Prop
>         [x:A](and (ap_ A Prop B x) (ap_ A Prop C x)))];

> [Union = [A:Type][B:(Pred  A)][C:(Pred  A)](La_ A Prop
>        [x:A](or (ap_ A Prop B x) (ap_ A Prop C x)))];

> [Pnot = [A:Type][B:(Pred  A)](La_ A Prop
>                                 [x:A](not (ap_ A Prop B x)))];
> [Minus = [A:Type][B:(Pred  A)][C:(Pred  A)]
>                 Meet A B (Pnot A C)];

> [Subset = [A:Type][B:(Pred  A)][C:(Pred  A)](FA A
>           [x:A]((ap_ A Prop B x) => (ap_ A Prop C x)))];

> [Eqset =  [A:Type][B:(Pred  A)][C:(Pred  A)]( and (Subset A B C)
>           (Subset A C B))];

> [Single = [A : Type][x: A]( La_ A Prop ([y: A](Eq A x y)))];

%-------------------------------------------------------------
```

From the above definitions we can see that the definitions are longer than the corresponding definitions in Lego. The reason is that Plastic has no lego-like implicit syntax and thus requires most things to be made explicit.

3.7.3 Fix points and their properties

The theory of fixed points is very useful for giving denotational semantics of programming languages. It can also be used in program analysis and verification of program properties, etc. In this subsection, based on the above definitions, we give some formalization of the theory and theorems which are formally proved in Plastic. In the following definitions we assume that ϕ is a monotonic function from power set of E to power set of E.

Definition 1 *(Prefixed point:) A subset $S \subseteq E$ is a* ***prefixed point*** *of ϕ if $\phi(S) \subseteq S$*

Definition 2 *(Postfixed point:) A subset $S \subseteq E$ is a* ***postfixed point*** *of ϕ if $S \subseteq \phi(S)$*

Definition 3 *(Fixed point:) A subset $S \subseteq E$ is a* ***fixed point*** *of ϕ if S is both a prefixed point and a postfixed point of ϕ.*

Definition 4 *(Greatest Fixed point:) A subset $S \subseteq E$ is a* ***greatest fixed point*** *of ϕ if S is a fixed point and for any fixed point T of ϕ, $T \subseteq S$.*

Definition 5 *(Least Fixed point:) A subset $S \subseteq E$ is a* ***least fixed point*** *of ϕ if S is a fixed point and for any fixed point T of ϕ, $S \subseteq T$.*

The following are main formal definitions related to the above definitions:

```
> [Mono = [A:Type][F: Pi_ (Pred  A) (Pred  A)][C,D: (Pred  A)]
>     (( Subset A C D) => (Subset A (ap_ (Pred  A) (Pred  A) F C)
>                                    (ap_ (Pred  A) (Pred  A) F D)))];
> [F_Mono:(A:Type)(F: Pi_ (Pred  A) (Pred  A))
>                                    (C,D: (Pred  A))( Prf(Mono A F C D))];

> [prefixp = [A:Type][F:(Pi_ (Pred  A)) (Pred  A))] [P: Pred   A]
>           ( Subset A (ap_ (Pred  A)) (Pred  A) F P) P)];

> [postfixp = [A:Type][F:(Pi_ (Pred  A) (Pred  A))] [P: Pred  A]
>           ( Subset A  P (ap_ (Pred  A) (Pred  A) F P))];

> [lfixp = [A:Type][F:(Pi_ (Pred  A) (Pred  A))] (La_ A Prop [x:A]
>          ({P: (Pred  A)}((prefixp A F P) => (ap_ A Prop P x))))];

> [gfixp = [A:Type][F:(Pi_ (Pred  A) (Pred  A))](La_ A Prop [x:A](Ex (Pred  A)
>         ([P: (Pred  A)] (and (postfixp A F P) (ap_ A Prop P x))))) ];
```

We have proved the relevant properties of set and fixed points using Plastic. These form our bases for defining $\mu - calculus$. The following are main theorems related to fixed points we have proved, but we just show a proof of one theorem:

- **Theorem 3.7.1** *(Tarski[Tarski, 1955])Let E be a set, P(E) be the power set of E and* Φ*:*$P(E) \rightarrow P(E)$ *be a monotonic function (i.e.* $\forall S, S' \in P(E)(S \subseteq S' \rightarrow \Phi(S) \subseteq \Phi(S'))$*), Then* Φ *has a least fixed point* $\mu S.\Phi(S)$ *and a greatest fixed point* $\nu S.\Phi(S)$ *given by*
  $\mu S.\Phi(S) = \cap\{S' \subseteq E | \Phi(S') \subseteq S'\}$
  $\nu S.\Phi(S) = \cup\{S' \subseteq E | S' \subseteq \Phi(S')\}$
  $\mu S.\Phi(S)$ *is the least prefixed point since it is the meet of all the prefixed points.* $\nu S.\Phi(S)$ *is the greatest postfixed point since it is the union of all the postfixed points.*

- **Theorem 3.7.2** *For every prefixed point P, least fixed point is a subset of P,i.e.:* $\forall P.prefp(F, P) \rightarrow lfp(F) \subseteq P$

- **Theorem 3.7.3** *Least fixed point is a prefixed point, i.e.:*
  $prefp(F, lfp(F))$
  Proof *The following is our proof of this theorem in Plastic:*

```
%----------------------------------------------------------------

Least fixpoint is a prefixed point

> Claim lfixp_isprefixp:  (F1:(Pi_(Pred(A)) (Pred(A))))
>                         (Prf(prefixp F1 (lfixp F1)));
> Intros F1;
> Refine LL;
> Intros x;
> Refine LL;
> Intros H;
> Refine LL;
> Intros x1;
> Refine LL;
> Intros H1;
> Refine App ? ? (App ? ? H1);
> Refine App ? ? ( App ? ? (App ? ? (F_Mono A F1 (lfixp F1)  x1 )) x);
> 2 Refine App ? ?  lfixp_lessp;
> 2 Refine H1;
> Refine H;
> ReturnAll;
```

```
> lfixp_isprefixp;
%-------------------------------------------------------------------
```

Q.E.D.

The reason why we give this example here is to show the reasoning style of Plastic in this area, other theorems can be proved similarly.

- **Theorem 3.7.4** *Least fixed point is a postfixed point, i.e.:*
  $postfp(F, lfp(F))$
- **Theorem 3.7.5** *Every postfixed point P is a subset of greatest fixed point, i.e.:*
  $\forall P.postfp(F, P) \rightarrow P \subseteq gfp(F)$
- **Theorem 3.7.6** *Greatest fixed point is a prefixed point, i.e.:*
  $prefp(F, gfp(F))$
- **Theorem 3.7.7** *Greatest fixed point is a postfixed point, i.e.:*
  $postfp(F, gfp(F))$
- **Theorem 3.7.8** *(Reduction lemma [Kozen, 1983; Winskel, 1989])*
  $\forall P.P \subseteq gfp(F) \leftrightarrow P \subseteq F(gfp(\lambda Q.(P \cup F(Q))))$
- **Theorem 3.7.9** *(Least fix point fold and unfold)*
  $\forall P.P \subseteq lfp(F) \leftrightarrow P \subseteq F(lfp(F) \cup P)$
- **Theorem 3.7.10** *(Greatest fix point base)*
  $\forall P.P \subseteq P' \rightarrow P \subseteq gfp(\lambda Q.(P' \cup F(Q)))$
- **Theorem 3.7.11** *(Greatest fix point fold and unfold)*
  $\forall P.P \subseteq gfp(F) \leftrightarrow P \subseteq F(gfp(F) \cup P)$

3.8 Summary

We present preliminaries of the thesis in this chapter. Based on these basic concepts we can expand our study on the goals of the thesis.

Higher order logic and inductive data type are two important features in LF (compared with some non-theorem prover approach) which help us to formalise the required concepts very easily. From the above we can see that the power of expressive higher order logic simplifies the encoding of several concepts such as set, predicate and fixed points. We can also see that the inductive data type is very useful to formalise data types. Their concrete application will be discussed in following chapters of the thesis.

Chapter 4

The outline of the approach

Something attempted, something done.

— HENRY WADSWORTH LONGFELLOW, AMERICAN POET

This chapter presents the outline of our approach. We emphasize the difference of our approach and other approaches from architectures, the underlying theories and methodologies.

4.1 Our approach

For the computer assisted domain-specific reasoning, there are two main approaches, distinguished by their use (or not) of a theorem prover. We know that theorem prover based approaches lack strong support for domain-specific syntax and proof style; systems of the non theorem prover based approaches lack the certainty of proof and genericity. Our approach aims to reduce the weakness of the above approaches and help the domain users who are not type theoretic experts to do proof in a familiar syntax, and with the support of an *exactly customized* type theory. It aims at a balance between user convenience and certainty of proof.

There are new advantages too, since type theory provides clear methods, useful tools and good ideas about how to do computer assisted formal proof. In particular, induction is an important and powerful technique in type theory. We expect that providing reasoning tools which offer good support for induction, via the underlying type theory basis, will lead to a better appreciation of proofs and to wider use of such tools in many domains.

The key feature of our approach is its use of LF and its associated reasoning techniques to formalize a problem domain and to present a domain-oriented interface suitable for use by people who aren't experts in type theory. We just need to do some work for formalizations, translations and communications to implement a reasoning system for each domain. That

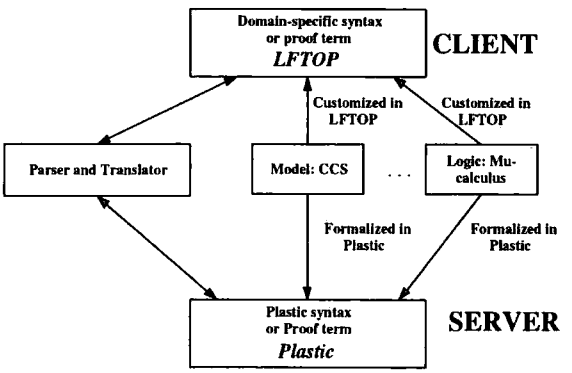

Figure 4.1: The architecture implied by the approach

saves much work. i.e. the formalization and translation work are the main work in this approach other than the concrete reasoning system.

4.1.1 An architecture of the approach

An architecture implied by the approach is shown in figure 4.1. It is a client-server structure, where the implementation of LF is a 'server' which handles the important reasoning steps. Clients use this reasoning functionality, and present a simplified version of it to users. There are three layers:

- The upper layer is a domain oriented interface which is operated by the domain user. For a specific domain, it gives access to the domain language, specification logic, semantic model and the reasoning system by using notations which are familiar to the domain users. These notations are customized in our new interface (Called LFTOP), so domain users need not use the underlying LF notations directly.

- In the bottom layer, every component in the upper layer is represented by its corresponding formalization in LF. Because we use Plastic [Callaghan and Luo, 2000b] as the implementation of LF. So the format of the formalization follows Plastic (See Chapter 7 for more details).

- The middle layer provides a bridge between the upper layer and the bottom layer. It includes tools such as parsers and translators. These tools implement the automatic transformation from domain-specific syntax and proof terms to Plastic syntax and proof terms.

Following Aspinall [Aspinall, 2000], we have designed an XML-based protocol ULPIP for communications between the various layers. This encodes the dialogues that can occur between the various layers. This may be extended with domain-specific features.

4.1.2 A methodology

Using this approach for each specific domain we should do all or some of the following steps:

- Formalize the domain-specific specification language, logic and semantic model in LF.
- Compare the original syntax and the formalized syntax. Implement parsers and translators to do the translation automatically.
- Design the reasoning system and commands which can be used by domain users in the interface.
- Find the corresponding command, group of commands or prove new lemmas in Plastic to simulate the effect of each user command in the interface.
- Implement the translations of user level commands to Plastic commands.
- Design and implement a concise protocol for communications between user level interface and the underlying Plastic system.
- Design and implement GUI related issues.

We shall follow the above steps in our case studies of Chapter 5 and 6. We can see them as the applications of this methodology.

4.2 The techniques we use in this approach

Many technologies should be used in this approach. We list them as follows:

- Functional programming technology: We use functional programming language Haskell to implement our system. We get a lot of benefit from the features of high order, list comprehension and lazy evaluation [Pang and Zhao, 2005; Pang et al., 2005b; Pang et al., 2006b] in it. As Plastic is implemented in Haskell, so the combination of our implementation and Plastic implementation is convenient.
- Parser technology: We use Happy [Gill and Marlow, 2005] as a generator of the Parser. We just need to present the BNF format of the domain specific language, Happy can generate the corresponding Haskell modules as the Parser.
- Translation technology: Using Grammar-directed translation technology the work of translation and parsing is done at the same time. The translations are clear for understanding.
- Model checking technology: We intend to use model checking technology to solve some subproblems which have finite states. But this was not attempted in this thesis.

- Communication technology: How to deal with the problem of communications between the layers? We can design some protocols for communication. We can also use the framework of Proof General [Aspinall, 2005a].

- *LF* based computer assisted reasoning technology: We use *LF* based computer assisted reasoning technology in the bottom layer. A lot of benefits are gotten in a way which domain users need not have the knowledge of LF in detail.

We use the above technologies in our approach. The details will be given in the following chapters.

4.3 The common things for different domains in this approach

There are many common things even for different domains in this approach. The following are the main common things.

- The communication protocols: Different domain can use same communication protocols. We design communication protocols called ULPIP for this purpose. We describe these protocols in HaXML. Meanwhile the framework of Proof General can be used directly.

- The underlying type system: In our approach we use Plastic (the implementation of *LF*) as the underlying type system for all domains.

- The similar translation modules: We use Grammar-directed translation technology, the translation modules are similar both in skeleton and strategy.

The reuse of the above items provides a good way for us to implement the relevant things.

4.4 The role of type theory and its framework

Type theory and the relevant logical framework (LF) are the basis of this approach. The formal reasoning is carried out in a system (here we mean Plastic) which is an implementation of the logical framework. The benefits from type theory and the logical framework can be obtained naturally. Research results from type theory (such as proof assistants) can be used without any difficulty. Especially, the correctness guaranteed by type theory increases the credibility of the work done in the approach. In fact, we can view the approach as programming the formal system in a better programming language (ie, writing the key parts in type theory rather than in C/Java/Haskell).

4.5 Discussion

In this chapter we give an outline of our approach. The approach tries to inherit all the advantages over non-type-theory based approach from type theory based approach in a way where the users need not have a lot of knowledge of type theory. In our design we use three layers of the interface to attain this effect. For example, it can inherit *the proof terms* which is one of the major differences between type theory based theorem provers with other non-type-theory based theorem provers and automatic verifiers. Proof terms are λ-terms of which the correctness can be checked by type checking algorithms implemented in a type theory based proof assistant. Therefore proof terms give us more confidence on the proof. The proof checking of Plastic helps to ensure the correctness of the reasoning in our approach in a way that is not noticed by domain users. All these features are benefit from the structure of the approach.

However the structure of this approach is more complicated than most other approaches. But this should be balanced against the positive features, and we believe the balance is in our favor. For this kind of system, the overheads in a multi-layer approach are relatively small, so 'efficiency' is not a big problem here.

Chapter 5

Case study: concurrency

If you want to understand today, you have to search yesterday.

— PEARL BUCK, AMERICAN FEMALE WRITER

This chapter presents a specific domain —concurrency— as a domain for the case study.

Building on Yu's work in Lego [Yu, 1999; Yu and Luo, 1997], we choose concurrency as the domain of interest for this case study. This domain is relatively complex, requiring the interaction of three formal systems, hence a demanding case study. Some issues of this work has been published in [Pang et al., 2005a].

Firstly, this chapter introduces the basic relevant concepts of concurrency and then we give a deep study of it.

5.1 Domain analysis

Concurrent systems are quite different from ordinary sequential systems. Instead of focusing on input-output behavior and termination of the sequential systems, they focus on the *interactions and communications* between components. Usually the interactions and communications are described by competing for access to shared resources which is corresponding to *shared variable model* or exchanging messages which is corresponding to *message passing model.* Process algebra represents a mathematically rigorous framework for modelling concurrent systems of interacting processes.

5.1.1 Process algebra

The term process algebra includes a collection of theories that support mathematically rigorous (in)equational reasoning about systems consisting of concurrent, interacting processes. The field grew out of a seminal book due to Milner [Milner, 1980] and has been an active area of research since then. In particular, researchers have developed a number of different

process algebraic theories in order to capture different aspects of system behavior; however, each such formalism generally includes the following characteristics:

1. A language, or algebra, is defined for describing systems.
2. A behavioral equivalence is introduced that is intended to relate systems whose behavior is indistinguishable to an external observer.
3. Equational rules, or axioms, are developed that permit proofs of equivalences between systems to be conducted in a syntax driven manner.

Some formalisms include a refinement ordering, in this case, the theories allow one to determine if a system is "greater than or equal to" (i.e. refines) another. The relevant literature typically refers to each theory as a process algebra; so the field of process algebra contains many process algebras. Process algebras derive their motivation from the fact that a system design often consists of several different descriptions of the system involving different levels of detail. The behavioral equivalence or refinement relation provided by a process algebra may be used to determine whether these different descriptions conform to one another. More specifically, higher-level descriptions of system behavior may be related to lower-level ones using the equivalence or refinement ordering supplied by the algebra. Related systems may be used interchangeably inside larger system descriptions; this facilitates compositional system verification, since low-level designs of system components may be checked in isolation against their high-level designs. This section surveys some of the main features of process algebra. The next subsection introduces CCS, the process algebra that we use throughout the chapter to illustrate the principles we cover in the case study.

Calculus for Communicating Systems (CCS) is a good example of the message passing model. The application of the approach LFTOP in this domain involves three formal systems and their associated technology. These are:

- *Specification language:* The state system under consideration is described in a specification language which usually is a kind of process algebra, such as Calculus for Communicating Systems (CCS), or Communicating Sequential Processes (CSP). We choose CCS as our specification language and focus on Pure CCS [Milner, 1989] in this chapter.
- *Semantics:* The specification is transformed into a representation which is a semantic model, e.g. LTS (Labelled Transition System),or Timed Automata. We use LTS as the semantic model to give the relevant operational semantics.
- *Logic:* Properties to be checked are given as formulas of a specification logic, such as μ-calculus, Propositional Linear Temporal Logic (PLTL), or Computation Tree Logic (CTL). We choose μ-calculus as the logic here: it is sufficiently powerful for our purposes, and other temporal logics can be defined as abbreviations of μ-calculus.

5.1.2 CCS: Calculus for Communicating System

The Calculus for Communicating System is an algebraic theory intended to describe communications between, and computations of, abstract processes. It is an algebra for specifying and reasoning about concurrent systems. As an algebra, CCS provides a set of terms, operators and axioms that can be used to write and manipulate algebraic expressions. The expressions define the elements of a concurrent system and the manipulations of these expressions reveal how the system behaves.

The operators in the set may be used to construct system descriptions from definitions of subsystems. The basic building blocks of these descriptions and system definitions in all existing process algebras are actions. Intuitively, actions represent atomic, uninterrupted execution steps, with some actions denoting internal execution and others representing potential interactions with its environment that the system may engage in. In CCS, both communication and computation are abstractly represented by actions. In other words, actions represent either inputs/outputs on ports or internal computation steps. The former are sometimes called external, as they require interaction from the environment. To formalize these intuitions, let $A = L \cup \{\tau\}$ be a set of actions, τ be a distinguished action called 'silent' action which models *internal* or *invisible* or *idling* actions. L is a set of labels having two disjoint subset: L^+ is a set of *names*, and L^- is a set of *co-names*. We let a, b, c range over names, $\bar{a}, \bar{b}, \bar{c}$ range over co-names, and α, β range over A. If $l \in L$, then its complement action $\bar{l} \in L$, and we have $\bar{\bar{l}} = l$.

Then an action in CCS has one of the following three forms.

- a, where $a \in L^+$, represents the act of receiving a signal on port a.
- $\bar{a}$, where $a \in L^+$, in other words, $\bar{a} \in L^-$, represents the act of emitting a signal on port a.
- τ is a distinguished action called 'silent' action which models *internal* or *invisible* or *idling* actions.

The syntax of the pure CCS version can be expressed as follows:
$E ::= Nil \mid X \mid \alpha.E \mid E_1 + E_2 \mid E_1|E_2 \mid E\backslash L \mid E[f] \mid rec\ X.E$
where

- Nil (called empty process) represents stopped or deadlocked computation, so it cannot perform any actions.
- X is a process variable.
- $\alpha.E$ (called prefix) can perform action α and then behave as E.
- $E_1 + E_2$ (called summation) represents choice — the process can evolve either as E_1 or as E_2.

- $E_1|E_2$ (called parallel composition) represents the parallel independent performing of E_1 and E_2 or communicating through complement actions of them.
- $E\backslash L$ (called restriction) represents a process which behaves like E but cannot perform actions in L or their complement actions.
- $E[f]$ (called relabelling) behaves like E, but the actions are renamed by a bijection $f{:}L \rightarrow L$, where f has the property that $f(\bar{l}) = f(\bar{l})$; we can extend the domain of f to A and let $f(\tau) = \tau$.
- $rec\ X.E$ (called recursion) represents a recursive process which behaves like the process E applied to $rec\ X.E$.

Value-passing CCS is a process calculus in which actions consist of sending and receiving values through communication ports, and the transmitted data can be tested using a conditional construct.

The syntax form of the value-passing version is as follows:

$E{::} = Nil \mid X \mid p(x).E \mid p'(e).E \mid \tau.E \mid E_1 + E_2 \mid E_1|E_2 \mid E\backslash L \mid E[f] \mid if\ b\ then\ E \mid Rec\ X.E$

We just need to describe the different expressions from pure CCS as follows:

- $p(x).E$ (called input prefix) behaves as a process which can receive a value, say v, over channel p, and bind the result to a variable x, binding results in substitution [v/x] of the formal parameter x by the actual parameter v;
- $p'(e)$ (called output prefix) behaves as a process which send value e over channel p;
- $if\ b\ then\ E$ behaves as E when b is true, otherwise no action is done.

5.1.3 LTS: Labelled Transition System

Labelled Transition System (LTS) is very useful in representing operational semantics of formal systems such as CCS. Its definition is as follows.

Definition 6 *A labelled transition system T is a triple $(S, L, \rightarrow)$, where S is a set of states, L is a set of transition labels and $\rightarrow \subseteq S \times L \times S$ is a transition relation.*

Normally we write $s \xrightarrow{a} s'$ for $(s, a, s') \in \rightarrow$, and $\xrightarrow{a}$ for the relation $\{(s, s') | s \xrightarrow{a} s'\}$.

When LTS is used to describe the operational semantics of concurrent systems the labels are interpreted as actions which can take place in the system. The system is considered as being in one particular state at any given time, changing states by performing actions in accordance with the transition relation.

In an LTS, if we set a state as a start state, then the LTS is called rooted LTS. It is a quadruple $(S, L, \rightarrow, p)$, where p is the start state (also be called root), the others are the same as in the above triple.

5.1.4 μ-calculus

5.1.4.1 Previous logics

For previous logics J. Bradfield and C. Stirling gave a good introduction in [Bradfield and Stirling, 2003]. Many explanations in the following paragraphs are extracted from theirs for self-containment of this thesis.

Hennessy-Milner Logic(HML) [Hennessy and Milner, 1980] is a primitive modal logic of actions. In addition to the boolean operators the syntax of HML has a modality $< a > \phi$, where a is a process action. A structure for the logic is a labelled transition system. The constants tt and ff are two atomic formulas of the logic. The meaning of $< a > \phi$ is "it is possible to do an a-action to a state where ϕ holds". By inductively defining when a state (a process) of a transition system has a property, the formal semantics are given; for example, $E \models < a > \phi$ iff $\exists F.E \xrightarrow{a} F \wedge F \models \phi$. Some notions of variable or atomic proposition to the logic are added, in this case we provide a valuation which maps a variable to the set of states at which it holds. The expressive power of HML in this form is quite limited: a given HML formula can only make statements about a given finite number of steps into the future. In fact, we can say that HML was introduced not so much as a language to express properties, but rather as an aid to understanding process equivalence: two (image-finite) processes are equivalent iff they satisfy the same HML formulae exactly. In order to obtain the expressivity desired in practice, stronger logics are needed.

The logic Propositional Dynamic Logic (PDL) [Fischer and Ladner, 1979] [Pratt, 1976] is both a development of modal logics and a development of Floyd-Hoare style logics. PDL enriches the labels in the modalities of HML. Although it is not widely used these days, but it is worth presenting it briefly for historical reasons. PDL is an extension of HML in the circumstance that the action set has some structure.

Computation Tree Logic (CTL) [Clarke and Emerson, 1981] is an alternative extension of HML, with some extra "temporal" operators which permit expression of liveness and safety properties, i.e. it includes further modalities. For the semantics, we need to consider "runs" of a process. A run of E_0 is a sequence $E_0 \xrightarrow{a_0} E_1 \xrightarrow{a_1} E_2 \xrightarrow{a_2} \ldots$ which may have finite or infinite length; if it has finite length then its final process is a "sink" process which has no transitions. A run $E_0 \xrightarrow{a_0} E_1 \xrightarrow{a_1} E_2 \xrightarrow{a_2} \ldots$ has the property $\phi U \psi$, "ϕ until ψ", if there is an $i \geq 0$ such that $E_i \models \psi$ and for all j:$0 \leq j < i$, $E_j \models \phi$.

E_0	$\xrightarrow{a_0}$	E_1	$\xrightarrow{a_1}$	...	E_{i-1}	$\xrightarrow{a_{i-1}}$ E_i	$\xrightarrow{a_i}$ ...
$\models$		$\models$		...	$\models$	$\models$	
ϕ		ϕ		...	ϕ	ψ	

The formula $F\phi = (ttU\phi)$ means "ϕ eventually holds" and $G\phi = \neg(ttU\neg\phi)$ means "ϕ always holds". For each "temporal" operator such as U there are two modal variants, a strong variant ranging over all runs of a process and a weak variant ranging over some run of a process. We present a strong version with $\forall$ and a weak version with $\exists$. If HML is extended with the two kinds of until operator, the resulting logic is a slight but inessential

variant of CTL. By inductively defining when a state (process) has a property the formal semantics are given. For instance $E \vDash \forall[\phi U \psi]$ iff every run of E has the property $\phi U \psi$.

Some variants and enrichments of CTL appear to allow free mixing of path operators and quantifiers. CTL* [Emerson and Lei, 1986] and ECTL* [Vardi and Wolper, 1983] are two representatives. The CTL* formula $\forall[P\ U\ \exists FQ]$ is also a CTL formula, but $\forall[P\ U\ FQ]$ is not, because the F is not immediately governed by a quantifier. Hence traditional temporal logics as advocated by Manna and Pnueli [Manna and Pnueli, 1981] and others are also covered by the extensions. In these logics, time is a linear sequence of instants, corresponding to the states of just one execution path through the program. One can define a logic on paths which has operators $\bigcirc\phi$ "in the next instant (on this path) ϕ is true", and $\phi U \psi$ "ϕ holds until ψ holds (on this path)"; and then a system satisfies a formula if all execution paths satisfy the formula.

We are interested in the following aspects:

1. *The small model property:* Many modal logics have a very good property — the small model property; that is if a formula is satisfiable, then it is satisfiable by a small model of some bounded size. Provided that model-checking is decidable, which is the case for almost all temporal logics, this gives decidability of satisfiability for the logic immediately, as one need simply check every transition system up to the size bound. The technique used to establish the small model property for PDL (and hence HML) is a classical technique in modal logic, that of filtration. Let E be a possibly infinite-state process with the property ϕ, and let T be its transition system. Let Γ be the set of all subformulas of ϕ and their negations: in the case of PDL one also counts $< \alpha > \psi$ and $< \beta > \psi$ as subformulas of $< \alpha \cup \beta > \psi$, $< \beta > \psi$ as a subformula of $< \alpha; \beta > \psi$; and $< \alpha > \psi$ as subformula of $< \alpha^* > \psi$. The size of Γ is proportional to $|\phi|$ (the length of ϕ). One then filters T through Γ by defining an equivalence relation on the states of T, $E \equiv F$ iff $\forall \psi \in \Gamma . E \models \psi \Leftrightarrow F \models \psi$. We define the filtered model to have states $T/\equiv$ and with atomic action relations given by $[E] \xrightarrow{a} [F]$ iff $\exists E' \in [E], F' \in [F] . E' \xrightarrow{a} F'$. The number of equivalence classes is at most $2^{|\Gamma|}$ and so is $O(2^{|\phi|})$. It shows that the filtered model is indeed a model, in that $[E] \models \psi$ iff $E \models \psi$ for $\psi \in \Gamma$. Consequently if ϕ is a satisfiable PDL formula, then it has a model with size $O(2^{|\phi|})$, and in fact $2^{|\phi|}$ suffices — see [Fischer and Ladner, 1979] for more details.

   Although CTL, CTL* and modal μ-calculus all have the finite model property, the filtration technique does not apply. If one filters T through a finite set Γ containing $\forall FQ$ unintended loops may be added. For example if T is $E_i \xrightarrow{a} E_{i+1}$ for $1 \leq i < n$ and Q is only true at state E_n then $E_i \models \forall FQ$ for each i. But when n is large enough the filtered model will have at least one transition $E_j \xrightarrow{a} E_i$ when $i \leq j < n$, with the consequence that $E_i \not\models \forall FQ$. The initial approach to showing the finite model property utilises semantic tableaux where one explicitly builds a model for a satisfiable formula with small size. But this technique is very particular, and subsequent more

sophisticated methods based on automata are used for optimal results.

2. *CTL model-checking:* Apart from the common things, CTL has some obvious differences from PDL. At first, although it is a state-based logic, but it uses path operators internally — evaluating the formula $\forall[\phi U \psi]$ at a state involves considering all paths from that state. Thus, at first glance, one might expect to lose the obvious exponential upper bound on model-checking. However, this turns out not to happen, and in fact CTL is not difficult to model-check. This was shown in [Clarke et al., 1986] by a direct construction; it also follows from the fact that CTL is a simple fragment of the modal μ-calculus. The model-checking procedure of [Clarke et al., 1986] is an example of a global technique. This procedure proceeds by model-checking subformulas from the bottom up, doing a full pass over the state space for subformulas before considering the superformula. Here is an English outline of the algorithm in the original paper:

   - to check $tt, \neg\phi, \phi_1 \wedge \phi_2$, check the subformulas and perform the boolean operation;
   - to check $< a > \phi, [a]\phi$, check the subformula ϕ, and then apply the semantic definitions;
   - to check $\exists[\phi U \psi]$, check the subformulas, then find the states at which ψ holds, and trace backwards along paths on which ϕ holds;
   - to check $\forall[\phi U \psi]$, check the subformulas, then make a depth-first traversal of the system, doing the following: if a state satisfies ψ, mark it as satisfying $\forall[\phi U \psi]$; otherwise, if it fails ϕ, mark it as failing $\forall[\phi U \psi]$; otherwise, after processing the successors, mark it as satisfying $\forall[\phi U \psi]$ iff all its successors do.

In fact, CTL can be translated into modal μ-calculus. The relevant algorithms include global, backward-looking model-checking algorithms, and local forward-looking algorithms. From today's perspective, it is interesting to see that this previous CTL algorithm has elements of both: the code for $\exists[\phi U \psi]$ is doing exactly the computation by approximation of the μ-calculus translation; but the code for $\forall[\phi U \psi]$ is doing tableau model-checking.

5.1.4.2 A brief introduction to μ-calculus

The use of fixedpoint operators is an important defining feature of μ-calculus. Using them in program logics goes back at least to D. Park [Park, 1969]. However, using them in modal logics of programs dates from work of Pratt, Emerson, Clarke and Kozen. Pratt's version [Pratt, 1982] used a fixedpoint operator like the minimization operator of recursion theory, and this has not been studied further. Fixedpoint operators were added by Emerson and Clarke to a temporal logic to capture fairness and other correctness properties [Emerson and Clarke, 1980]. Kozen [Kozen, 1983] introduced the modal μ-calculus which we still use today, and established a lot of basic results. The expressive power of modal μ-calculus is intimately connected to finite–state automata on infinite trees [Vardi and Wolper, 1986]. Classically,

a μ-formula denotes a predicate on states. Typical properties to be expressed and analyzed are safety and liveness assertions. The formula $\nu x. < a > x$, for example, denotes the set of all states allowing for an infinite sequence of a-actions. Especially, the modal operator $< a >$ (for action a) constructs a property of the actual state from a property of a next state. Thus, it relates present to a (possibly infinite) future. For detailed explanation of various modal (and temporal) logics consult, e.g., Colin Stirling in [Stirling, 1992].

5.1.4.3 A positive version of μ-calculus with tagging fixed points

Considering that we are based on an intuitionistic type theory, we choose a positive version of μ-calculus with tagging fixed points [Winskel, 1989]. It is enough to express all the temporal properties we need. Because formulas with negation operators can be transformed to some normal forms with negation operators occurring only before atomic formulas [Walukiewicz, 1995]. The data irrelevant version of the tagged μ-calculus is as follows:

$$\Phi ::= Z \mid \Phi \vee \Phi \mid \Phi \wedge \Phi \mid < K > \Phi \mid [K]\Phi \mid \mu Z.U\Phi \mid \nu Z.U\Phi$$

where U is a tag which is a subset of states, K ranges over subset of labels and Z ranges over a set of assertion variables.

The tag-free fixed points $\mu Z.\Phi$ and $\nu Z.\Phi$ are special cases with empty tag. The formula *true* is defined as $tt =_{def} \nu Z.Z$, and the formula *false* is defined as $ff =_{def} \mu Z.Z$.

5.1.4.4 Semantics of μ-calculus :

We use labelled transition system $(S, L, \{\xrightarrow{l} : l \in L\})$ to give the operational semantics of μ-calculus. Here S is a set of states, L is a set of transition labels, and for each $l \in L$ a transition relation $\xrightarrow{l}$ is a subset of $S \times S$, i.e. $\xrightarrow{l} \subseteq S \times S$. The semantics of formula ϕ is represented by $[\![\phi]\!]_\rho$ (where $[\![\phi]\!]_\rho \subseteq S$) and it is given by induction on the structure of ϕ as follows.

$$\begin{aligned}
[\![Z]\!]_\rho &= \rho(Z) \\
[\![\Phi \vee \Psi]\!]_\rho &= [\![\Phi]\!]_\rho \cup [\![\Psi]\!]_\rho \\
[\![\Phi \wedge \Psi]\!]_\rho &= [\![\Phi]\!]_\rho \cap [\![\Psi]\!]_\rho \\
[\![< K > \Phi]\!]_\rho &= \{s \in S | \exists \alpha \in K. \exists s' \in S. s \xrightarrow{\alpha} s' \textit{ and } s' \in [\![\Phi]\!]_\rho\} \\
[\![[K]\Phi]\!]_\rho &= \{s \in S | \forall \alpha \in K. \forall s' \in S. s \xrightarrow{\alpha} s' \textit{ implies } s' \in [\![\Phi]\!]_\rho\} \\
[\![\mu Z.U\Phi]\!]_\rho &= \{s \in S | \forall P \subseteq S. [\![\Phi[P/Z]]\!]_\rho / U \subseteq P \textit{ implies } s \in P\} \\
[\![\nu Z.U\Phi]\!]_\rho &= \{s \in S | \exists P \subseteq S. P \subseteq [\![\Phi[P/Z]]\!]_\rho \cup U \textit{ and } s \in P\}
\end{aligned}$$

where the environment ρ assigning a subset of S to each assertion variable Z.

Properties of concurrent system are usually represented by assertions which are formulas of μ-calculus. The judgement that a state s satisfies a property Φ is now defined by : $s \vdash \Phi$ iff $s \in [\![\Phi]\!]_\rho$ for all ρ.

5.2 Congruences and Reasoning in CCS

5.2.1 Congruences for CCS

A congruence for an algebra is an equivalence relation with the following extra substitution property: equivalent systems can be used interchangeably inside any larger system. We can explain this property formally. Define a context $C[\,]$ to be a system description with a "hole", $[\,]$; given a system description p_1, then, $C[p_1]$ represents the system obtained by "filling" the hole with p_1. Then an equivalence $\approx$ is a congruence for a language, if whenever $p_1 \approx p_2$, then $C[p_1] \approx C[p_2]$ for any context $C[\,]$ built using operators in the language. From the above description we can see that relations that are congruences for some languages maybe not for others. Congruence is an important concept in analysis of the relevant systems.

In process algebras a notion of behavioral congruence [Cleaveland and Smolka, 1999] often be used as a basis for system analysis. In this subsection we define a relation that relates systems with respect to their "observable" behavior and study congruences for CCS. We first define an equivalence relation on states in an arbitrary LTS in each case; since CCS may be viewed as an LTS, these relations may then be used to link CCS system descriptions. The suitability of the equivalence from the standpoint of the observable behavior is considered. Furthermore whether or not the relation is a congruence for CCS is studied.

5.2.1.1 Trace Equivalence

Language equivalence is a well-studied equivalence in state machine theory, where two machines are equivalent if they accept the same sequences of symbols. Individual CCS system descriptions may be converted into rooted LTS's. But rooted LTS does not contain accepting states, and consequently we cannot use the notion of language equivalence from finite-state machine theory directly. However, if every state in a rooted LTS is accepting state, then the language of the machine contains the execution sequences, or traces, that a machine may engage in. So, to relate two descriptions of a system exactly when the machines for them have the same traces might be a reasonable attempt at defining a behavioral equivalence for CCS.

Definition 7 *Let $(P, A, \rightarrow)$ be a LTS.*

1. *Let $s = a_0 ... a_{n-1} \in A^*$ be a sequence of actions. Then $p \xrightarrow{s} p'$ if there are states $p_0, ..., p_n$ such that $p = p_0, p_i \xrightarrow{a_i} p_{i+1}$ for $0 \leq i < n$, and $p' = p_n$.*

2. *s is a strong trace of p if there exists p' such that $p \xrightarrow{s} p'$. We use $S(p)$ to represent the set of all strong traces of p.*

3. *$p \approx_S q$ exactly when $S(p) = S(q)$.*

The reason to use the term *strong traces* is that the definition given above does not distinguish between internal and external actions (i.e. all may appear in a strong trace). In contrast, τ action is treated in a special manner in the traditional definition of traces.

Since CCS is an LTS whose states are system descriptions, so the definition of $\approx_S$ might be used to CCS systems. Unfortunately, since CCS permits the definition of nondeterministic systems, $\approx_S$ suffers from severe deficiencies which is illustrates as in the following examples.

1. Let $p_1 =_{df} a.p_1$ and $p_2 =_{df} a.p_2 + a.Nil$. Then $p_1 \approx_S p_2$, however p_2 can reach a "deadlocked" state (i.e. Nil) after an a-transition while p_1 cannot.

2. Let p be $a.b.Nil+a.c.Nil$ and q be $a.(b.Nil+c.Nil)$. Then $S(p) = S(q) = \{\varepsilon, a, ab, ac\}$, So $p \approx_S q$. However, after an a-transition q can perform both a b and a c, whereas p must reject one or the other of these possibilities.

From the above examples we can see that even though two nondeterministic systems have the same traces, they may go through inequivalent states in performing them [1]. In particular, trace equivalent systems can have different deadlocking behavior. So this kind of trace equivalence is not adequacy for nondeterministic systems such as CCS.

5.2.1.2 Bisimulation Equivalence

The observation in the previous paragraph suggests that a nondeterministic system such as CCS needs an equivalence which has a recursive flavor: execution sequences for equivalent systems ought to *pass through* equivalent states. This intuition leads the definition of bisimulation, or strong equivalence.

Definition 8 *Let $(S, A, \rightarrow)$ be an LTS. A relation $R \subseteq S \times S$ is a bisimulation if, whenever $< p, q > \in R$, then the following conditions hold for any $a \in A$ and $p', q' \in S$.*

1. *if $p \xrightarrow{a} p'$, then $q \xrightarrow{a} q'$ for some q' such that $< p', q' > \in R$,*

2. *if $q \xrightarrow{a} q'$, then $p \xrightarrow{a} p'$ for some p' such that $< p', q' > \in R$.*

From the above definition we can see that if two systems are related by a bisimulation, then it is possible for each to simulate the other's behavior. For a relation to be a bisimulation, related states must be able to *match* transitions of each other by moving to related states. Next we shall give bisimulation equivalent for two states. Two states are bisimulation equivalent exactly when a bisimulation relating them is found.

Definition 9 *Systems p and q are bisimulation equivalent, or bisimilar, if there exists a bisimulation R containing $< p, q >$. We write $p \sim q$ whenever p and q are bisimilar.*

As CCS may be viewed as an LTS, so we can use $\sim$ to relate CCS processes. The following examples show some differences of bisimulation equivalent with trace equivalent.

[1] This situation cannot occur in deterministic systems.

1. $p_1 \nsim p_2$, where $p_1 =_{df} a.p_1$, $p_2 =_{df} a.p_2 + a.Nil$.
2. $a.b.Nil + a.c.Nil \nsim a.(b.Nil + c.Nil)$
3. $a.b.Nil + a.b.Nil \sim a.b.Nil$

Bisimulation equivalence has a lot of pleasing properties.

1. For any LTS it is indeed an equivalence; i.e. the relation $\sim$ is reflexive, symmetric and transitive.
2. $\sim$ implies $\approx_S$ and if the LTS is deterministic in the sense that every state has at most one outgoing transition per action, then $\sim$ coincides with $\approx_S$.
3. It can be shown in a precise sense that two equivalent systems must have the same *deadlock potential.*
4. $\sim$ is a congruence for CCS, in other words, if $p \sim q$, then p and q may be used interchangeably inside any larger system.

However, for the process algebras which allow asynchronous execution such as CCS, $\sim$ does suffer from a major flaw: it is too sensitive to internal computation. In particular, the definition does not take account of the speciality of the action τ. For example, the systems $a.\tau.b.Nil$ and $a.b.Nil$ are not bisimulation equivalent, even though an external observer cannot detect the difference between them. Nevertheless, $\sim$ has been studied extensively in many literatures, and for process algebras in which internal computation in one component can affect the behavior of other components indeed, it is a reasonable basis for verification. But this flaw is an inducement to consider other equivalence to suit CCS.

5.2.1.3 Observational Equivalence for CCS

From above subsubsection we can see that bisimulation is too sensitive to internal computation. This subsubsection presents a coarsening of bisimulation equivalence that is intended to relax the sensitivity of the former to internal computation. The introduction of *weak* transitions starts a further progress in new equivalent relations.

Definition 10 *Let $(P, A, \rightarrow)$ be an LTS with $\tau \in A$, and let $p \in P$.*

1. *if $s \in A^*$, then $\hat{s} \in (A - \{\tau\})^*$ represents the action sequence obtained by deleting all occurrences of τ from s.*
2. *Let $s \in (A - \{\tau\})^*$, then $p \stackrel{s}{\Rightarrow} p'$ if there exists s' such that $p \stackrel{s'}{\Rightarrow} p'$ and $s = \hat{s}'$*

From the above definition we can see that ŝ returns the *visible content* (i.e. non-τ elements) of sequence s; in particular, if $a = \tau$ then $\hat{a} = \varepsilon$; if $a \in A - \{\tau\}$ then $\hat{a} = a$. In addition, if a sequence of τ-transitions leads from p to p' then $p \stackrel{\varepsilon}{\Rightarrow} p'$.

We now define weak bisimulations as follows.

Definition 11 *Let $(P, A, \rightarrow)$ be an LTS with $\tau \in A$, then a relation $R \subseteq P \times P$ is a weak bisimulation if, whenever $< p, q > \in R$, then the following hold for all $a \in A$ and $p', q' \in P$:*

1. *if $p \xrightarrow{a} p'$, then $q \stackrel{\hat{a}}{\Rightarrow} q'$ for some q' such that $< p', q' > \in R$*
2. *if $q \xrightarrow{a} q'$, then $p \stackrel{\hat{a}}{\Rightarrow} p'$ for some p' such that $< p', q' > \in R$*

States p and q are observationally equivalent, or weakly bisimilar, or weakly equivalent, if there exists a weak bisimulation R containing $< p, q >$. In this case we write $p \approx q$

As CCS is an LTS whose action set contains τ, the definition of $\approx$ may be used to relate CCS system descriptions. We have the following observations.

1. for any process p, $\tau.p \approx p$
2. $a.\tau.b.\tau.Nil \approx a.b.Nil$

Consequently weakly bisimilar would appear to be a viable candidate for relating CCS system descriptions. But unfortunately it is not a congruence for CCS. To see the reason, consider the context C[ ] given by [ ] + b.Nil. Let p be $\tau.a.Nil$, q be $a.Nil$, then we know that $p \approx q$. However, $C[p] \not\approx C[q]$. To see this, note that $C[p] \xrightarrow{\tau} a.Nil$. This transition must be matched by a weak ε-labelled transition from C[q]. But the only such transition C[q] is $C[q] \stackrel{\varepsilon}{\Rightarrow} C[q]$. However, $a.Nil \not\approx C[q]$, since the latter can engage in a b-labelled transition that cannot be matched by the former.

This shortage of $\approx$ arises from the interplay between + and the initial internal computation that a system might engage in, in particular, the only CCS operator that *breaks* the congruence-hood of $\approx$ is +. Milner [Milner, 1980; Milner, 1989] solved this problem by adopting a way which is to focus on finding the largest CCS congruence $\approx^L$ that implies $\approx$. Such a largest congruence is guaranteed to exist [Hennessy and Milner, 1985].

Definition 12 *Let $(P, A, \rightarrow)$ be an LTS with $\tau \in A$, and let $p, q \in P$, then $p \approx^L q$ if the following hold for all $a \in A$ and $p', q' \in P$.*

1. *if $p \xrightarrow{a} p'$, then $q \stackrel{a}{\Rightarrow} q'$ for some q' such that $p' \approx q'$.*
2. *if $q \xrightarrow{a} q'$, then $p \stackrel{a}{\Rightarrow} p'$ for some p' such that $p' \approx q'$.*

We have the following remarks about the above definition.

1. It should be noted that for $p \approx^L q$ to hold, any τ-transition of p must be matched by a $\stackrel{\tau}{\Rightarrow}$-transition of q. In particular, this weak transition must consist of a non-empty sequence of τ-transitions.
2. The definition is not recursive; the targets of initial matching transitions need only be related by $\approx$.

3. $\approx^L$ is a congruence for CCS indeed and is the largest CCS congruence entailing $\approx$, i.e. $p \approx^L q$ implies $p \approx q$, and for any other congruence R such that pRq implies $p \approx q$, pRq also implies $p \approx^L q$.

Consider the following examples.

1. $\tau.a.Nil \not\approx^L a.Nil$, since the $\xrightarrow{\tau}$ transition of the former cannot be matched by a $\stackrel{\tau}{\Rightarrow}$ transition of the latter.

2. $a.\tau.b.Nil \approx^L a.b.Nil$

3. For any p,q, if $p \approx q$, then $\tau.p \approx^L \tau.q$.

5.3 Formalization of the domain

5.3.1 Formalization of CCS

The formalization is based on the system Plastic which is an implementation of LF. The Plastic system is a concise system with very few predefined types. We use the predefined type Nat as the basic type to help us in the formalization of actions. The formalization of actions is as follows:

```
> [Base = Nat];

> Inductive
>        [Actb:Type]
>    Constructors
>    [base:(b:El Base)Actb]
>    [comp:(b:El Base)Actb];

> Inductive
>        [Act:Type]
>    Constructors
>    [tau:Act]
>    [act:(a:El Actb)Act];

> [Comp = [x:El Actb] E_Actb ([b:El Actb] Actb) ([a:El Base](comp a))
>  ([a1:El Base](base a1)) x:El Actb -> El Actb];
```

We formalise processes in an inductive type like the following:

```
> [Var = Nat];
```

```
> Inductive
>       [Process:Type]
>    Constructors
>    [Nil:Process]
>    [var:(v:El Var)Process]
>    [dot:(a:El Act)(E: Process)Process]
>    [choice:(E1: Process)(E2: Process)Process]
>    [par:(E1: Process)(E2: Process)Process]
>    [hide:(E: Process)(L:El (List Actb))Process]
>    [ren:(E: Process)(f: El (List (Pair  Base Base)))Process]
>    [rec: (E: Process)Process];
```

The operational semantics of CCS processes can be given by a labelled transition system with S to be the set of process, L to be the set of actions and the transition relations to be defined by the following transition rules:

$$\textbf{Dot:}\ \frac{}{a.E \xrightarrow{a} E} \qquad \textbf{ChoiceL:}\ \frac{E_1 \xrightarrow{a} E'}{E_1 + E_2 \xrightarrow{a} E'}$$

$$\textbf{ChoiceR:}\ \frac{E_2 \xrightarrow{a} E'}{E_1 + E_2 \xrightarrow{a} E'} \qquad \textbf{ParL:}\ \frac{E_1 \xrightarrow{a} E'}{E_1|E_2 \xrightarrow{a} E'|E_2}$$

$$\textbf{ParR:}\ \frac{E_2 \xrightarrow{a} E'}{E_1|E_2 \xrightarrow{a} E_1|E'} \qquad \textbf{Tau1:}\ \frac{E_1 \xrightarrow{a} E_1' \quad E_2 \xrightarrow{a'} E_2'}{E_1|E_2 \xrightarrow{\tau} E_1'|E_2'}$$

$$\textbf{Tau2:}\ \frac{E_1 \xrightarrow{a'} E_1' \quad E_2 \xrightarrow{a} E_2'}{E_1|E_2 \xrightarrow{\tau} E_1'|E_2'} \qquad \textbf{Hide:}\ \frac{E \xrightarrow{a} E'}{E\backslash L \xrightarrow{a} E'\backslash L}(a, a' \notin L)$$

$$\textbf{HideT:}\ \frac{E \xrightarrow{\tau} E'}{E\backslash L \xrightarrow{\tau} E'\backslash L} \qquad \textbf{Rename:}\ \frac{E \xrightarrow{a} E'}{E[f] \xrightarrow{f(a)} E'[f]}$$

$$\textbf{Rec:}\ \frac{E[(rec\ x.E)/x] \xrightarrow{a} E'}{rec\ x.E \xrightarrow{a} E'}$$

We define the transition relation as an inductive relation with large elimination. The above rules are corresponding to the constructors of the inductive relation.

```
> Inductive
>       [TRANS:(a:El Act)(E1:El Process)(E2:El Process)El Prop] Relation_LE
>    Constructors
>    [Dot:(a:El Act)(p:El Process) Prf( TRANS a (dot a p) p)]
```

```
>    [Chol:(a:El Act)(p:El Process)(p1:El Process)(p2:El Process)
>    (t:Prf(TRANS a p1 p))Prf(TRANS a (choice p1 p2) p)]

>    [Chor:(a:El Act)(p:El Process)(p1:El Process)(p2:El Process)
>    (t:Prf(TRANS a p2 p))Prf(TRANS a (choice p1 p2) p)]

>    [Parl:(a:El Act)(p,p1,p2:El Process)(t:Prf(TRANS a p1 p))
>    Prf(TRANS a (par  p1 p2) (par p p2))]

>    [Parr:(a:El Act)(p,p1,p2:El Process)(t:Prf(TRANS a p2 p))
>    Prf(TRANS a (par  p1 p2) (par p1 p))]

>    [Tau1:(n:El Base)(p1,p2,q1,q2:El Process)
>    (t1:Prf(TRANS (act (base n)) p1 q1))
>    (t2:Prf(TRANS (act (comp n)) p2 q2))
>     Prf(TRANS tau (par p1 p2) (par q1 q2))]

>    [Tau2:(n:El Base)(p1,p2,q1,q2:El Process)
>    (t1:Prf(TRANS (act (comp n)) p1 q1))
>    (t2:Prf(TRANS (act (base n)) p2 q2))
>    Prf(TRANS tau (par p1 p2) (par q1 q2))]

>    [Hide:(a:El Actb)(p,q:El Process)(L:El (List Actb))
>    (t:Prf(TRANS (act a) p q))
>    (p1:El (Prf(is_false (Or (member (Actb) Eq_Actb a L)
>    (member (Actb) Eq_Actb  (Comp a) L)))))
>    Prf(TRANS (act a) (hide p L) (hide q L))]

>    [Hidet:(p,q:El Process)(L:El (List Actb))
>    (t:Prf(TRANS tau p q))
>    Prf(TRANS tau (hide p L)(hide q L))]

>    [Ren:(a:El Act)(p,q:El Process)(f: El (List (Pair Base Base)))
>    (t:Prf(TRANS a p q))
>     Prf(TRANS (rename f a )(ren p f)(ren q f))]
>    [Rec: (a: El Act)(p1,p2: El Process)
>    (t: Prf(TRANS a  (subst p1 (succ zero) (rec p1)) p2))
>    Prf(TRANS a (rec p1) p2)]  ;
```

Where (*subst* t n s) means to replace the variables in term t which are equal to n with s, i.e. we use de Bruijn's index for expressing substitutions. So (*subst* p1 (succ zero) (rec p1)) is p1[(rec p1)/x]. This method to express substitutions bring some difficulties in translation issues. We shall do more explanation in section 5.7.4.

It is dangerous if we use inductive relation with large elimination without careful consideration about avoiding paradox. Adams [Adams, 2004] pointed out that using large elimination without limitation may lead to paradox [2]. But our formalization is safe, because we did not use the dangerous features. Meanwhile, in order to make sure its safety we have used the corresponding formalization of this transition (i.e. the above TRANS) in Lego with ordinary inductive relation. There is no such kind of paradox in Lego, because there is no universe Type which is its own type.

5.3.2 Formalization of μ-calculus

We use Plastic's second order logic to formalize μ-calculus with the help of our previous formalization and proof of properties of set and fixed point. The details are as follows:

```
> [Label = Act :Type];

> Inductive
>       [Modality :Type]
>    Constructors
>    [Modal: (L:El (List Label))Modality]
>    [Nmodal: (L:El (List Label))Modality];


> [State = Process];

> [MTRANS = [K: Modality][s1: State][s2 : State](Ex Label ([a: Label]
>           (and (Eq Bool (Modal_check a K) true) (TRANS a s1 s2))))];

> [ Form  = Pred  State];
> [ Tag = Pred State ];

> [MuvarF : Pi_ Var Form];


> [MuOr = [A,B: Form] (Union State A B)];
```

[2] Dr. Callaghan will issue his modified version of Plastic to avoid this kind of paradoxes.

```
> [MuAnd = [A,B: Form] (Meet State A B)];


> [MuDia = [K: Modality][F: Form]( La_ State Prop ([s: State](Ex State
>          ([s': State] (and (MTRANS K s s') (ap_ State Prop F s'))))))];


> [MuBox = [K: Modality][F: Form]( La_ State Prop ([s: State](FA State
>          ([s': State] ((MTRANS K s s') => (ap_ State Prop F s'))))))];



> [MuTagnu = [T: Tag ][F: (Pi_ Form Form)]( La_ State Prop ([s: State]
>        (Ex Form ([P: Form] (and (Subset State P (Union State
>        (ap_ Form Form F P) T ))(ap_ State Prop P s))))))];


> [MuTagmu = [T: Tag ][F: (Pi_ Form Form)]( La_ State Prop ([s: State]
>        (FA Form ([P: Form] ((Subset State  (Minus  State
>        (ap_ Form Form F P) T ) P) => (ap_ State Prop P s))))))];
```

The above formalization of syntax and semantics give us a basis for representing the domain concepts and properties. On LF level we can prove some useful lemmas which are corresponding to the rules on the user level. Meanwhile these give an another way to validate that the rules are correct.

5.4 User level reasoning system

The basic reasoning steps which a user can make are given by a user-level reasoning system. This can be described via several groups of rules, derived from the standard rules of μ-calculus and CCS, and augmented with several useful lemmas.

We divide the rules into the following groups.

5.4.1 Rules that do not involve the process operators.

These rules do not depend on CCS components.

- *Rules related to basic logic:*

  This is a simple logic including the basic operators for $and(\wedge)$, $or(\vee)$ etc. They are mainly used to express and prove the side conditions which appear in some of the

μ-calculus rules and combined assumptions. Here we just list the *and-relevant* rules as follows:

$$\textbf{Pair: } \frac{\Phi \quad \Psi}{\Phi \wedge \Psi} \qquad \textbf{Fst: } \frac{\Phi \wedge \Psi}{\Phi} \qquad \textbf{Snd: } \frac{\Phi \wedge \Psi}{\Psi}$$

- *Rules related to* set*:*

  These are basic rules which deal with set membership.

$$\textbf{Singlein:} \frac{}{a \in \{a\}} \qquad \textbf{Inr :} \frac{s \in V}{s \in U \cup V} \qquad \textbf{Inl: } \frac{s \in U}{s \in U \cup V}$$

- *Rules related to μ-calculus:*

  The following show a subset of the rules related to tagged μ-calculus.

$$\textbf{True: } \frac{}{s \vdash tt} \qquad \textbf{Dia with s': } \frac{s' \vdash \Phi}{s \vdash < K > \Phi}(s \xrightarrow{K} s')$$

$$\textbf{Exintro with } a\textbf{: } \frac{s \xrightarrow{a} s'}{s \xrightarrow{K} s'}(a \in K) \qquad \nu\textbf{\_unfold: } \frac{s \vdash \Phi[\nu Z.(U \cup \{s\})\Phi/Z]}{s \vdash \nu Z.U\ \Phi}(s \notin U)$$

where notation $s \xrightarrow{a} s'$ means (s, s') in the transition relation $\xrightarrow{a}$ and $s \xrightarrow{K} s'$ means $\exists a \in K.s \xrightarrow{a} s'$.

5.4.2 Rules for the process operators.

The following are a subset of the rules for pure CCS. The rules with prefix **Inv_** are not standard rules in CCS, but are lemmas added to the reasoning system for user convenience.

$$\textbf{Dot: } \frac{}{a.E \xrightarrow{a} E} \qquad \textbf{Rec: } \frac{E[(rec\ X.E)/X] \xrightarrow{a} E'}{rec\ X.E \xrightarrow{a} E'}$$

$$\textbf{Inv\_dot: } \frac{a.E_1 \xrightarrow{b} E_2}{E_1 = E_2 \wedge a = b} \qquad \textbf{Inv\_rec: } \frac{rec\ X.E_1 \xrightarrow{a} E_2}{E_1[rec\ X.E_1/X] \xrightarrow{a} E_2}$$

Users can apply commands which are corresponding to the above rules in the system to prove relevant properties.

5.5 User level syntax

We design the user level syntax of the concepts of this domain by trying to keep their original form. The following is the description of the syntax of the user level in BNF:

ids	::=	letter quasiletter*
letter	::=	a | ... | z | A | ... | Z
digit	::=	0 | ... | 9
quasiletter	::=	letter | digit | _ | '
nontauact	::=	ids | ids~
act	::=	nontauact | tau
topcmd	::=	ids : process '|−' muform | ids : process − act − > process
		| ids : Allnat act proc
proc	::=	Nil | ids | act . proc | proc + proc | proc '|' proc |
		proc \ {nontauact,...,nontauact}
		| proc [nontauact/nontauact,...,nontauact/nontauact] | Rec ids proc
muform	::=	ids | muform '||' muform | muform & muform | < K > muform
		| [K] muform | Mu ids.Tag muform | Nu ids.Tag muform
K	::=	− | {act,...,act} | −{act,...,act}
Tag	::=	{-} | {ids,...,ids}
cmd	::=	Fst ids | Snd ids | RDia ids | RRec | RDot | RPair | RSinglein |
		RTrue | RExintro ids | Rend | RNuunfold | Rvpair | Rbox ids ids |
		RNubase | Rinr | Rinl | Rhypchange ids ids ids | Rinverdotl ids ids |
		Rinverrec ids ids ids ids | Rinverchoi ids ids ids ids |
		Rinverpar ids ids ids ids| Req | RModule ids | RImport ids |
		RUndo | RChol | RChor | RParl | RParr | RTau1 ids |
		RTau2 ids | RHide | RHidet | RRen | RIndn act proc ids | RHyp ids

In the above BNF description, "act" represents act of CCS, "topcmd" represents target or goal we want to prove, "proc" represents process of CCS, "muform" represents the formula of μ-calculus, "K" represents a set of acts, "Tag" represents Tags of the μ-calculus and "cmd" represents commands which user can use in the interface. In addition, the quoted parts by ' ' (such as '|−' and '||') are used for distinguishing them from the meta-symbols.

Users will use the above syntax to define their concurrent system and prove the properties of this system. Obviously the syntax is very similar to the domain users.

5.6 Translation between different levels

The translation between different levels is very important in this approach. It realizes an important step of the implicit support of the proof assistant Plastic. Using grammar-directed technology, the translation can be implemented automatically.

5.6.1 The translation from user level to LF level

To implement the translation from user level to LF level is to implement the transformation from the user level grammar to LF level grammar. We just present the outline of the

translation in this chapter, concrete discussions will be given in chapter 8.

5.6.1.1 The translation of CCS concepts

The translation of CCS concepts includes the translation of the CCS grammar of the concepts to the corresponding formalised parts of them in LF. This work is fulfilled by the parser and translator of the CCS automatically. We just need to customise the parser and the translator. The concrete parts of them, especially the translation of the relevant parts, will be given in chapter 8.

5.6.1.2 The translation of LTS concepts

The translation of LTS concepts includes the translation of the LTS grammar of the concepts to the corresponding formalised parts of them in LF. This work is fulfilled by the parser and translator of the LTS automatically also. We just need to customise the parser and the translator. The concrete parts of them, especially the translation of the transition relation, will be given in chapter 8.

5.6.1.3 The translation of μ-calculus concepts

The translation of μ-calculus concepts includes the translation of the μ-calculus grammar of the concepts to the corresponding formalised parts of them in LF. This work is fulfilled by the parser and translator of the μ-calculus automatically also. We just need to customize the parser and the translator. The concrete parts of them, especially the translation of the relevant parts, will be given in chapter 8.

5.6.2 The translation from LF level to user level

The translation from LF level to user level is the reverse process of the the translation from user level to LF level. More details please to refer to chapter 8. In fact, we may not do this translation thoroughly, as when we use commands on user level, from the view of domain users, it just needs to do the transformation on user level. We can keep this transformation and then translate the result of the transformation to LF level, furthermore check the result to see whether this is consistent with the result of the LF level.

5.7 Some examples

The following examples have been chosen to illustrate the approach and issues that arise from it, in particular for translation, rather than to show new functionality enabled by the approach.

5.7.1 Ticking clock

Firstly, we use an example of ticking clock to show the basic aspects of this approach. This example also shows a translation problem which we'll discuss in detail in chapter 8. This example was taken from [Stirling, 1992; Yu, 1999] and discussed in [Pang et al., 2002].

$$Cl =_{def} tick.Cl$$

This process can just perform only one action *tick* and it will *tick* forever. We can use the CCS syntax $rec\ x.\ tick.x$ to express the process [3].

The ticking clock process has a simple property: the clock is able to *tick*. We can express the goal of proving this property in the following form:

$$Abletick:\ Cl\ \vdash\ <\{tick\}>tt$$

where the judgement p: $s \vdash\ \Phi$ means that we want to prove that process s satisfies the property Φ and use the name p to memorize the property.

The table 5.1 and table 5.2 show our proof on the two different levels. The rows with `Goal` in the first column show the relevant parts of the proof state, and rows with `CMD` show the command issued by the user. The User-level column shows what the user should expect to see, whilst the Plastic column shows the corresponding LF form or the equivalent command sequence for Plastic. In our prototype, we can complete the proof by issuing just the user-level commands, and the prototype is able to translate the more complex LF terms back to their simple user-level forms.

From table 5.1 and table 5.2 we can see that the goals and commands on the user-level are more concise and user oriented, one step on the user-level proof usually corresponds to several steps in Plastic. On the user-level steps some information about the real parameters should be given for translating the interface command to Plastic commands. In fact, the Plastic level proof can be hidden from the user and the group of commands in Plastic which corresponding to one command on user-level is linked by tactical.

5.7.2 Simple communication protocol

Now we use an example of a simple communication protocol, taken from [Cleaveland et al., 1993] and discussed in [Pang et al., 2005a].

The protocol specification can be formalized as the parallel combination of three basic processes: a sender, a receiver and a medium that connects sender and receiver. The sender initially waits for a message to send, after which it passes the message to the medium using the channel *from* and then awaits an acknowledgement on the channel *ack_to*. When the medium receives a message along its channel *from* it makes it available on its channel *to*, and when it receives an acknowledgement on its channel *ack_from* it makes it available on its channel *ack_to*. When the receiver gets a message on channel *to*, it announces that

[3] in [Milner, 1989], this process should be expressed by $fix(X = tick.X)$, so the CCS syntax here is a little bit different from it.

	User Level	Plastic Level
Goal	Abletick: Cl |- <{tick}> tt	Claim Abletick: Prf (ap_ Process Prop (MuDia (Modal (cons Label tick (nil Label))) Mutt) Cl)
CMD	Rule Dia Cl	Refine App ? ? (lemma_dia_ccs ? ? ? Cl ?);
Goal	$?_2$ Cl $\stackrel{\{tick\}}{\rightarrow}$ Cl $?_1$ Cl |- tt	? m2 : Prf(MTRANS (Modal (cons Label tick (nil Label))) Cl Cl) ? m1 : Prf(ap_ State Prop Mutt Cl)
CMD	Rule True	Refine lemma_true;
Goal	$?_2$ Cl $\stackrel{\{tick\}}{\rightarrow}$ Cl	? m2 : Prf(MTRANS (Modal (cons Label tick (nil Label))) Cl Cl)
CMD	Rule Exintro tick	Refine LL; Intros _; Refine LL; Intros H; Refine App ? ? (App ? ? H tick);
Goal	$?_3$ (tick in {tick}) $\wedge$ (Cl $\stackrel{tick}{\rightarrow}$ Cl)	? m3 : Prf(and (Eq Bool (Modal_check tick (Modal (cons Label tick (nil Label)))) true) (TRANS tick Cl Cl))
CMD	Rule Pair	Refine App ? ? (App ? ? p_pair ?);
Goal	$?_5$ tick in {tick} $?_4$ Cl $\stackrel{tick}{\rightarrow}$ Cl	? m5 : Prf(Eq Bool (Modal_check tick (Modal (cons Label tick (nil Label)))) true) ? m4 : Prf(TRANS tick Cl Cl)
CMD	Rule Rec	Refine Rec;

Table 5.1: Proof procedure for ticking clock (part I)

	User Level	Plastic Level
Goal	$?_5$ tick in {tick}	? m5 : Prf(Eq Bool (Modal_check tick (Modal (cons Label tick (nil Label)))) true)
	$?_6$ tick .Cl $\stackrel{tick}{\rightarrow}$ Cl	? m6 : Prf (TRANS tick (subst (dot tick (var one)) (succ zero) (rec (dot tick (var one)))) Cl)
CMD	Rule Dot	Refine Dot;
Goal	$?_5$ tick in {tick}	? m5 : Prf (Eq Bool (Modal_check tick (Modal (cons Label tick (nil Label)))) true)
CMD	Rule Singlein	Refine App ? ? Eq_refl;
Goal	(no new goal)	(no new goal)
CMD	Rule end	ReturnAll;

Table 5.2: Proof procedure for ticking clock (part II)

the message is available for receipt and then sends an acknowledgement along the channel *ack_from*. The corresponding and assistant processes in the interface are defined as follows:

```
SENDER = rec X (send.from~.ack_to.X);
MEDIUM = rec X (from.to~.X + ack_from.ack_to~.X);
RECEIVER = rec X (to.receive~.ack_from~.X);
PROTOCOL = (SENDER | MEDIUM | RECEIVER) \{from,to,ack_from,ack_to};
SENDER1 = from~.ack_to.(rec X send.from~.ack_to.X);
PROTOCOL1 = (SENDER1 | MEDIUM | RECEIVER)\{from,to,ack_from,ack_to};
```

The following are the translations of the above definitions in *LF*:

```
%------------------------------------------------------------------------
Def SERVICE = rec (dot send (dot receive (var one))) : El Process
Def SENDER = rec (dot send (dot from' (dot ack_to (var one)))) : El Process
Def SENDER1 = dot from' (dot ack_to SENDER) : El Process
Def MEDIUM
    = rec
       (choice
     (dot from (dot to' (var one)))
     (dot ack_from (dot ack_to' (var one))))
    : El Process
Def RECEIVER
```

```
    = rec (dot to (dot receive' (dot ack_from' (var one)))) : El Process
Def PROTOCOL
    = hide
       (par (par SENDER MEDIUM) RECEIVER)
       (cons
     Actb
     fromb
     (cons Actb ack_tob (cons Actb tob (cons Actb ack_fromb (nil Actb)))))
    : El Process
Def PROTOCOL1
    = hide
       (par (par SENDER1 MEDIUM) RECEIVER)
       (cons
     Actb
     fromb
     (cons Actb ack_tob (cons Actb tob (cons Actb ack_fromb (nil Actb)))))
    : El Process
%-------------------------------------------------------------------------
```

The PROTOCOL process has a simple property: it is able to *send*. We can express the goal of proving this property in the following form:

$$Abletosend\text{: } PROTOCOL \vdash < \{send\} > tt$$

Table 5.3 (continued in table 5.4) shows a proof of this property on two different levels. It is easy to see that the forms on user level are much simpler than those on LF level.

5.7.3 Example with infinite state space

Now we consider some examples with infinite state space. We choose an example from [Dam, 1995]. This example presents a counter and its property "a counter can count forever". In this thesis we want to prove that the counter is always able to perform up. Use CCS notation, the counter can be expressed in the following form:

```
Cnt = rec x.up.(x|down.Nil)
```

The corresponding processes in our interface are defined as the following:

```
Cnt =  rec X  up.(X | (down. Nil));
```

	User Level	Plastic Level
Goal	Abletosend: PROTOCOL |- <{send}> tt	Claim Abletosend: Prf (ap_ Process Prop (MuDia (Modal (cons Label send (nil Label))) Mutt) PROTOCOL)
CMD	Rule Dia PROTOCOL1	Refine App ? ? (lemma_dia_ccs ? ? ? PROTOCOL1 ?);
Goal	$?_2$ PROTOCOL $\overset{\{send\}}{\rightarrow}$ PROTOCOL1 $?_1$ PROTOCOL1 |- tt	? m2 : Prf(MTRANS (Modal (cons Label send (nil Label))) PROTOCOL PROTOCOL1) ? m1 : Prf(ap_ State Prop Mutt PROTOCOL1)
CMD	Rule True	Refine lemma_true;
Goal	$?_2$ PROTOCOL $\overset{\{send\}}{\rightarrow}$ PROTOCOL1	? m2 : Prf(MTRANS (Modal (cons Label send (nil Label))) PROTOCOL PROTOCOL1)
CMD	Rule Exintro send	Refine lemma_Exintro ? ? ? send
Goal	$?_4$ send in {send} $?_3$ PROTOCOL $\overset{send}{\rightarrow}$ PROTOCOL1	? m4 : Prf(Eq Bool (Modal_check send (Modal (cons Label send (nil Label)))) true) ? m3 : Prf(TRANS send PROTOCOL PROTOCOL1)
CMD	Rule Hide	Refine Hide;
Goal	$?_4$ send in {send} $?_5$ (SENDER | MEDIUM | RECEIVER) $\overset{send}{\rightarrow}$ (SENDER1 | MEDIUM | RECEIVER) $?_6$ not (send in {from,ack_to, to,ack_from} $\vee$ send~in {from,ack_to,to,ack_from})	? m4 : Prf(Eq Bool (Modal_check send (Modal (cons Label send (nil Label)))) true) ? m5 : Prf (TRANS send (par (par SENDER MEDIUM) RECEIVER) (par (par SENDER1 MEDIUM) RECEIVER))) ? m6 : Prf (is_false (Or (member Actb Eq_Actb sendb (cons Actb fromb (cons Actb ack_tob (cons Actb tob (cons Actb ack_fromb (nil Actb)))))) (member Actb Eq_Actb (Comp sendb) (cons Actb fromb (cons Actb ack_tob (cons Actb tob (cons Actb ack_fromb (nil Actb))))))
CMD	Rule Hyp Distinct	Refine Distinct;
Goal	$?_4$ send in {send} $?_5$ (SENDER | MEDIUM | RECEIVER) $\overset{send}{\rightarrow}$ (SENDER1 | MEDIUM | RECEIVER)	? m4 : Prf(Eq Bool (Modal_check send (Modal (cons Label send (nil Label)))) true) ? m5 : Prf (TRANS send (par (par SENDER MEDIUM) RECEIVER) (par (par SENDER1 MEDIUM) RECEIVER)))

Table 5.3: Proof procedure for simple protocol (part I)

	User Level	Plastic Level
CMD	Rule Parl	Refine Parl;
Goal	$?_4$ send in {send}	? m4 : Prf(Eq Bool (Modal_check send (Modal (cons Label send (nil Label)))) true)
	$?_7$ (SENDER | MEDIUM) $\stackrel{send}{\rightarrow}$ (SENDER1 | MEDIUM)	? m7 : Prf (TRANS send (par SENDER MEDIUM) (par SENDER1 MEDIUM)))
CMD	Rule Parl	Refine Parl;
Goal	$?_4$ send in {send}	? m4 : Prf(Eq Bool (Modal_check send (Modal (cons Label send (nil Label)))) true)
	$?_8$ SENDER $\stackrel{send}{\rightarrow}$ SENDER1	? m8 : Prf (TRANS send SENDER SENDER1))
CMD	Rule Rec	Refine Rec;
Goal	$?_4$ send in {send}	? m4 : Prf(Eq Bool (Modal_check send (Modal (cons Label send (nil Label)))) true)
	$?_9$ send .SENDER1 $\stackrel{send}{\rightarrow}$ SENDER1	? m9 : Prf (TRANS send (subst (dot send (dot from~(dot ack_to (var one)))) (succ zero) (rec (dot send (dot from~(dot ack_to (var one)))))) SENDER1))
CMD	Rule Dot	Refine Dot;
Goal	$?_4$ send in {send}	? m4 : Prf(Eq Bool (Modal_check send (Modal (cons Label send (nil Label)))) true)
CMD	Rule Singlein	Refine App ? ? Eq_refl;
Goal	(no new goal)	(no new goal)
CMD	Rule end	ReturnAll;

Table 5.4: Proof procedure for simple protocol (part II)

The following is the translation of the above definition in LF:

```
%------------------------------------------------------------
Def Cnt = (rec  (dot up
                (par (var (succ (zero))) (dot down Nil)))) ;
%------------------------------------------------------------
```

This is an example with infinite state space. Systems which based on model checker technology are hard to prove properties of the counter. But in our approach to prove this property is not difficult.

The property "`Always able to perform up`" can be expressed in CCS as the following form:

$$\nu X. < up > tt \wedge \texttt{[-]X}.$$

So "counter has this property" can be expressed as follows:

$$\texttt{Cnt} \vdash \nu X. < up > tt \wedge \texttt{[-]X}.$$

According to the semantics of ν operator, this goal can be split to the following sub-goals:

$$\exists S. S \subseteq (< up > tt \wedge [-]S) \quad \ldots\ldots \quad (5.1)$$

and $\texttt{Cnt} \in S \quad \ldots\ldots \quad (5.2)$

We take the infinite set { `cnt(i)|i ∈ Nat` } as this S, where `cnt(0)=Cnt`, `cnt(1)=Cnt|(down.Nil)`,...,`cnt(i+1)=cnt(i)|down.Nil`). In other word, S can be defined as:

$$S =_{df} \lambda\ \texttt{s: Process } \exists\ \texttt{n:Nat. Eq s cnt(n)}\ .$$

We can split (5.1) to two separate sub-goals:

$$S \subseteq < up > tt \quad \ldots\ldots \quad (5.3)$$

and $S \subseteq [-]S \quad \ldots\ldots \quad (5.4)$

Sub-goal (5.2) can be proved by the membership of S. Sub-goal (5.3) can be split to

$$\forall s \in S\ \exists s'. s \xrightarrow{up} s'$$

and $s' \in tt$

by the semantics of <> operator. Take `s'` as `s|down.Nil`, the two sub-goals are changed to:

$$\forall s \in S. s \xrightarrow{up} s|down.Nil \quad \ldots\ldots \quad (5.5)$$

and $s|down.Nil \in tt \quad \ldots\ldots \quad (5.6)$

Sub-goal (5.6) is proved easily. Sub-goal (5.5) can be proved by induction on natural number.

Table 5.5 shows a proof of (5.5) on two different levels. It is easy to see that the forms on user level are similar to standard CCS notation and much simpler than those on LF level.

By the semantics of [ ] operator, sub-goal (5.4) is:

$$\forall s \in S\ \forall s'\ \exists \alpha. s \xrightarrow{\alpha} s'\ \texttt{implies}\ s' \in S$$

It can be proved by induction on natural number also. So we get the proofs by means of the semantics of μ-calculus formulas and induction.

The above example shows that infinite state space is no problem for this approach. We can list the reason as follows:

- First, Plastic is not limited to finite state system. It relies on the techniques such as structural induction to prove properties in infinite domains.
- Second, CCS is not limited to finite state system.

	User Level	Plastic Level
Goal	Alwaysup: Allnat zero Cntn	Claim Alwaysup: (n: Nat) Prf (TRANS up (Cntn n) (Cntn (succ n)))
CMD	Rule RIndn zero Cntn H	Refine E_Nat ([n1:Nat]Prf(TRANS up (Cntn n1) (Cntn (succ n1)))); Intros n H ;
Goal	$?_1$ (Cntn zero) $\stackrel{up}{\rightarrow}$ (Cntn (succ zero)) $?_2$ (Cntn (succ n)) $\stackrel{up}{\rightarrow}$ (Cntn (succ (succ n)))	? m1 : Prf (TRANS up (Cntn zero) (Cntn (succ zero))) ? m2 : El (Prf (TRANS up (Cntn (succ n))(Cntn (succ (succ n)))))
CMD	Rule Parl	Refine Parl;
Goal	$?_1$ (Cntn zero) $\stackrel{up}{\rightarrow}$ (Cntn (succ zero)) $?_3$ (Cntn n) $\stackrel{up}{\rightarrow}$ (Cntn (succ n))	? m1 : Prf (TRANS up (Cntn zero) (Cntn (succ zero))) ? m3 : El (Prf (TRANS up (E_Nat ([n1:El Nat]State) Cnt ([n1:El Nat][S:El State] par S (dot down Nil)) n) (E_Nat ([n1:El Nat]State) Cnt ([n1:El Nat][S:El State] par S (dot down Nil)) (succ n))))
CMD	Rule Hyp H	Refine H;
Goal	$?_1$ (Cntn zero) $\stackrel{up}{\rightarrow}$ (Cntn (succ zero))	? m1 : Prf (TRANS up (Cntn zero) (Cntn (succ zero)))
CMD	Rule Rec	Refine Rec ;
Goal	$?_4$ up.((Cntn (succ zero)) | down.Nil) $\stackrel{up}{\rightarrow}$ ((Cntn (succ zero)) | down.Nil)	? m4 : El (Prf (TRANS up (subst (dot up (par (var (succ zero)) (dot down Nil))) (succ zero) (rec (dot up (par (var (succ zero)) (dot down Nil))))) (Cntn (succ zero))))
CMD	Rule Dot	Refine Dot;
Goal	(no new goal)	(no new goal)
CMD	Rule end	ReturnAll;

Table 5.5: Proof procedure for Counter's property

- Third, μ-calculus is not limited to finite state system. In the process of finding a proof, however, users often gain invaluable insight into the system or the property being proved.

5.7.4 Some observations from the examples

From the above examples, we make some observations:

- Automatic translation between user level and LF level is feasible. This translation is a superset of the implicit syntax mechanisms provided by most proof assistants. Although this example is simple, it already demonstrates several non-trivial features.

- Consider the translation of subgoal ?m6 in the example of ticking clock: the user level form is significantly simpler than the LF version. In particular, the LF form uses the term operator `subst`, which has been programmed as part of the formalization. There is currently no user-level indication or representation of `subst`: we treat it as an inherent *mechanism* of the formalization, and hence one that does not need to be shown. The user is interested in concrete processes, not hypothetical ones which are subject to substitution. Hence, uses of `subst` must be normalised away, to show the term after substitution. Plastic now implements a normalization operation which removes the obvious use of a set of operations by computation. We shall return to this issue in chapter 8.

- Almost all rules on the user level correspond to lemmas on LF level. This keeps the correspondence between the two levels explicit and implies that we need not translate every concepts on LF level to the user level, but just those which have some clear user level correspondence. This also makes the translation from LF level to user level feasible.

5.8 Discussion

In this chapter, to the domain of concurrency, we have analyzed the characteristics of domain-specific reasoning, formalized the notations of CCS, LTS and μ-calculus in LF. While we do the formalization we get a better understanding not just on the issues behind producing domain-specific computer assisted reasoning tools, but also on the knowledge of the domain.

The architecture, methodology and process of the approach presented in Chapter 4 led us to carry out the case study. Meanwhile this case study enriched them in many aspects.

We get the following feedback:

- The formalization of a domain may include many aspects such as:

  - The notations of the domain which is corresponding to the notations of CCS for the domain of concurrency.
  - The set of rules on user level which is a very important part in the case study and affects the usefulness and convenience of the ultimate system.
  - The notations for the description of semantics which is corresponding to the notations of LTS of the case study.
  - The notations for the description of logic properties which is corresponding to the notations of μ-calculus of the case study.

- The formalization or its skeleton is not just limited to this case study. For example, the skeleton of the formalization of LTS and μ-calculus can be used in other domains. This gives a kind of reusability of the formalizations.

We have demonstrated how our approach is used to prove properties of concurrency through some simple examples. Although some of them have infinite state space, their structures are very simple and therefore can be handled by CCS. The case study shows that our approach combines induction, semantic reasoning, domain-specific interface, abstraction and composition methods with LFTOP to verify the properties of domain specific systems. All of the lemmas and inference rules based by the domain-specific reasoning system are formally proved in Plastic and therefore a coherent system which firmly ensures the correctness of proofs on that level is constructed.

This case study also shows that LF is suitable to be an underlying reasoning basis. Although the structure is complicated, but domain user can get benefit without noticing the complexity of the structure.

Although the formalization work is intended to be done by some experts of type theory. We find that to allow users to develop their own lemmas (i.e. to extend the formalization rather than just to work inside it) is useful. Other things such as how to improve the understandability of proofs(e.g. representing traces of computation and using Natural Language to explain proof steps) are also important. As the size of examples increases, we may also need to study techniques to help users organize their proofs and developments, such as allowing multiple contexts for reasoning. These issues are also listed in our future work in chapter 9.

Chapter 6

Case study: Verification of semantic properties of LAZY-PCF+SHAR

Human felicity is produced not so much by great pieces of good fortune that seldom happen, as by little advantages that occur every day.

— BENJAMIN FRANKLIN, AMERICAN PRESIDENT

As another case study, in this chapter we use the approach to verify some semantic properties of a functional programming language LAZY-PCF+SHAR [Seaman and Iyer, 1996].

In [Seaman and Iyer, 1996] the authors present a very good explanation of the need of sharing. We extract some explanations from [Seaman and Iyer, 1996] here to show the basic notations and problems in this case study.

From a theoretical view, functional languages are easy to reason about, especially within the framework of call-by-name or call-by-value evaluation. But implementing a functional language strictly according to call-by-name causes a lot of problems, especially the problem of efficiency, this is due to the fact that arguments that are referred to more than once are copied and possibly re-evaluated each time they are needed. However, functional languages should have the referential transparency, so this value will always be the same. In practice the unnecessary re-evaluation is usually avoided by sharing the argument among each of its references so that there is only one copy of the argument at any time. When the value of the argument is first needed, the argument is evaluated and the original copy of the argument is replaced by its value. This value is the one used for later references to the argument. So sharing can be characterised by a lack of duplication of the argument and by updating the original copy of the argument when it is evaluated. This method of evaluation is referred

to as call-by-need usually. It provides the same resulting values as call-by-name, but has different behavior due to the reduction of unnecessary re-evaluation. For the call-by-need implementation additional improvements may be made in the usual way by analyzing the behavior of given programs in the implementation and performing some program transformations which improve the behavior of the program without affecting its results. Since the behavior of a program is often depended on by many optimisations, the sharing involved in implementing lazy evaluation must be taken into consideration. Thus an operational model of the call-by-need implementation which is easy to reason with is essential for us to carry out analysis of the programs. In [Seaman and Iyer, 1996] such a model is presented as an operational semantics of lazy evaluation with sharing, this semantics is also proven to yield the same results as the call-by-name semantics.

We know that a substitution is an operation defined externally to the semantics rules. And the fact that function application is defined in terms of substitution is one of the main factor to lead the simplicity of call-by-name and call-by-value semantics. But substitution is the operation that allows an argument to be duplicated. This point makes it unsuitable for formalizing sharing. Thus in order to avoid duplication an operational model of lazy evaluation with sharing must be able to explicitly determine when and how an argument is substituted. By incorporating the actions which carry out substitution explicitly in the semantics rules this can be done. Fortunately, in papers such as [Abadi et al., 1991] [Field, 1990] much work has already been done on explicit substitutions. In these papers the reduction system $\lambda\sigma$ is considered. $\lambda\sigma$ includes some syntax and rules which can carry out substitution explicitly. However, the sharing occurring in lazy evaluation implementations is not captured by this system, because it duplicates arguments and does not update them upon evaluation. In other words, these papers present studies of the reduction system with emphasis on optimality of reduction strategies, while the goal in paper [Seaman and Iyer, 1996] is to more closely model the sharing found in implementations in order to have a more accurate model for analysis.

The paper [Seaman and Iyer, 1996] fixes the reduction strategy by the operational semantics and emphasizes the suitability of the system for reasoning about sharing. That's why we followed Seaman's work in Coq [Seaman and Felty, 1993], and choose verification of properties of the operational semantics of a lazy functional language (called LAZY-PCF+SHAR) as an another domain of interest for case study. Since this domain uses the concept of explicit substitution to deal with the problem of substitution, so it is very different from our previous case study for concurrency. Because the proof technique of this domain is very similar to the technique of LF in Plastic, we can use Plastic directly and need not design a new interface for this domain. This tells us that for some domains we need not define new interfaces, the interface for Plastic is useful for them. Another reason of studying this kind of domain is to check the power of Plastic which is an implementation of LF, to see whether or not the style of explicit substitution in LAZY-PCF+SHAR affects the reasoning in Plastic greatly.

call-by-name: $\dfrac{e[e'/x] \downarrow v}{((\lambda x{:}t.e)e') \downarrow v}$ **call-by-value:** $\dfrac{(e' \downarrow v') \wedge (e[v'/x] \downarrow v)}{((\lambda x{:}t.e)e') \downarrow v}$

Figure 6.1: call-by-name and call-by-value

6.1 The need for explicit substitutions

Substitution can be used in explaining the call-by-name and call-by-value methods. The following shows the process. In order to evaluate the term $((\lambda x{:}t.e)e')$ by call-by-name order, just substitute the term e' for x in e and evaluate. For evaluating the term $((\lambda x{:}t.e)e')$ by call-by-value order, first evaluate e' to v', then substitute v' for x in e and evaluate. An inference rule may be used to describe the formal semantics of application. In order to conclude what is below the line, the premise above the line must be true. Also, if evaluating a term e results in a term v, this is denoted as $e \downarrow v$. Then formally the rules for the evaluation of application can be described as figure 6.1:

From the Figure 6.1 we can see that, this definition of substitution simplifies the formalization of these evaluation orders. But it is still not clear how this definition of substitution could be used to describe call-by-need evaluation. The original argument, e', should be substituted for the occurrence of x which will be accessed first, and the result of evaluating e' should be substituted for any remaining occurrences. However, before the program is run it is not known which occurrence of x will be evaluated first. And the argument e' should be evaluated only if it is needed. The reason is that the details of the actual process of substituting a term for a variable in another term are abstracted away. So, in order to implement lazy-evaluation the suitable semantics needs to be able to control the substitution process so that the substitution and evaluation of arguments can take place while the function body is being evaluated.

Fortunately, in [Abadi et al., 1991] [Field, 1990] explicit substitutions are introduced to implement the idea of incorporating rules into the semantics which directly carry out substitution. They are used to define systems of rewrite rules for the lambda calculus with no prescribed evaluation strategy. Unfortunately, these rules do not capture sharing, though they incorporate explicit rules to carry out substitution. In spite of this, explicit substitution provides a mechanism for a relatively simple formalization of lazy evaluation. Paper [Abadi et al., 1991] gave an example of a system called $\lambda\sigma$ by using explicit substitutions. This calculus evaluates λ-terms which may include unevaluated substitutions.

6.2 Capture of sharing

Though the operational semantics for call-by-need have some similarity to the $\lambda\sigma$-calculus, they differ from explicit substitutions in the following ways in order to capture sharing.

- Substitutions are not allowed to occur within an expression by the syntax of the terms of the language. Instead, a term is evaluated with respect to a single substitution at the outermost level, called the operational semantics environment. This environment is a list of variable bound to expressions which corresponds to an explicit substitution.

- Not as in *app rule* of the $\lambda\sigma$-calculus, which destroys sharing, in function application environments are not duplicated and distributed to subexpressions. This is the second difference which captures the first characterisation of sharing.

- The expression that a variable points to in the environment may be replaced by the value that it evaluates to. This is the third difference which captures the second characterisation of sharing. So the original copy of an argument is allowed to be replaced by its evaluated value. Not as in the $\lambda\sigma$-calculus, environments are not eliminated upon reaching a value, but are maintained throughout the evaluation. This ensures that the value can be used later.

Thus a relation between expression-environment pairs can be used to express an evaluation. This pair of an expression with an environment is called a configuration. A configuration for an expression e and an environment A is denoted as $< e, A >$. A list of binding of typed variable to expressions gives the structure of an environment. It can be formally described as the following form:

$A ::= [\,] | [x{:}t \mapsto e]A$

For convenience, if an environment containing more than one binding, the bindings will be separated by commas instead of square brackets.

The evaluation relation between a program and its final value in terms of inferences and axioms is defined as a natural semantics [Plotkin, 1981] [Kahn, 1987]. This natural semantics can be used to define the operational semantics of LAZY-PCF+SHAR. In this framework, since an expression is evaluated directly to its final value, so this style of semantics is often referred to as "big-step" or "one step" semantics. Properties or theorems about the evaluation relation defined with these semantics can be proved by induction on the height of the proof justifying the evaluation relation.

6.3 Domain analysis

LAZY-PCF+SHAR is a lazy version of the functional language PCF(Programming language for Computable Functions) extended by adding explicit substitution in order to formalize the semantics of lazy evaluation. Its semantics are defined as inference rules in the style of natural operational semantics [Kahn, 1987] or "big step" semantics which is called deductively defined systems. Usually the verification of this kind of properties is done by hand or on paper, but recent years the work can be done by interactive theorem provers. We try this for the case study in this chapter to show that Plastic is powerful enough to be this

kind of theorem provers. Meanwhile we shall get more information about the merits and shortcomings in doing reasoning by using Plastic. We also want to compare the style and characterization of Plastic with those of Coq and Lego.

6.3.1 Syntax of the language

Types:		
t	=	nat | bool $\|t_1 \rightarrow t_2$
Expressions:		
e	=	0 | true | false | x |
		succ(e) | pred(e) |
		iszero(e) | $if(e_1, e_2, e_3)$ |
		$\lambda x{:}t.e$ | $e_1 e_2$ |
		$\mu x{:}t.e$ | $< e, [x{:}t \mapsto e_1] >$

Figure 6.2: The Syntax of LAZY-PCF+SHAR

The syntax of LAZY-PCF+SHAR is shown in Figure 6.2. It includes constants, variables, the conditional, lambda abstraction, primitive functions, function application, μ operator and closure which acts as the syntactic vehicle for implementation of lazy evaluation.

6.3.2 Operational semantics of the language

The type judgement rules of this language are listed in figure 6.3 where Γ is a type environment which is a mapping of variables to types. $\Gamma[s/x]$ denotes a perturbed environment which respects Γ on all variables other than x, and binds x to type s. We say that an expression e has type t in type environment Γ if $\Gamma \vdash e{:}t$ can be justified by inferences based on the type judgement rules.

$$\textbf{C0:}\ \frac{}{\vdash 0{:}nat} \qquad \textbf{CT:}\ \frac{}{\vdash true{:}bool}$$

$$\textbf{CF:}\ \frac{}{\vdash false{:}bool} \qquad \textbf{VAR:}\ \frac{}{\Gamma[t/x] \vdash x{:}t}$$

$$\textbf{CS:}\ \frac{\Gamma \vdash e{:}nat}{\Gamma \vdash succ(e){:}nat} \qquad \textbf{CP:}\ \frac{\Gamma \vdash e{:}nat}{\Gamma \vdash pred(e){:}nat}$$

$$\textbf{CZ:}\ \frac{\Gamma \vdash e{:}nat}{\Gamma \vdash iszero(e){:}bool} \qquad \textbf{COND:}\ \frac{\Gamma \vdash e_1{:}bool \quad \Gamma \vdash e_2{:}t \quad \Gamma \vdash e_3{:}t}{\Gamma \vdash if(e_1, e_2, e_3){:}t}$$

$$\textbf{ABS:}\ \frac{\Gamma[s/x] \vdash e{:}t}{\Gamma \vdash \lambda x{:}s.e{:}s \rightarrow t} \qquad \textbf{APP:}\ \frac{\Gamma \vdash e_1{:}s \rightarrow t \quad \Gamma \vdash e_2{:}s}{\Gamma \vdash e_1 e_2{:}t}$$

$$\textbf{REC:}\ \frac{\Gamma[t/x] \vdash e{:}t}{\Gamma \vdash \mu x{:}t.e{:}t} \qquad \textbf{CLO:}\ \frac{\Gamma \vdash e_1{:}s \quad \Gamma[s/x] \vdash e{:}t}{\Gamma \vdash < e, [x{:}s \mapsto e_1] > {:}t}$$

Figure 6.3: Type rules

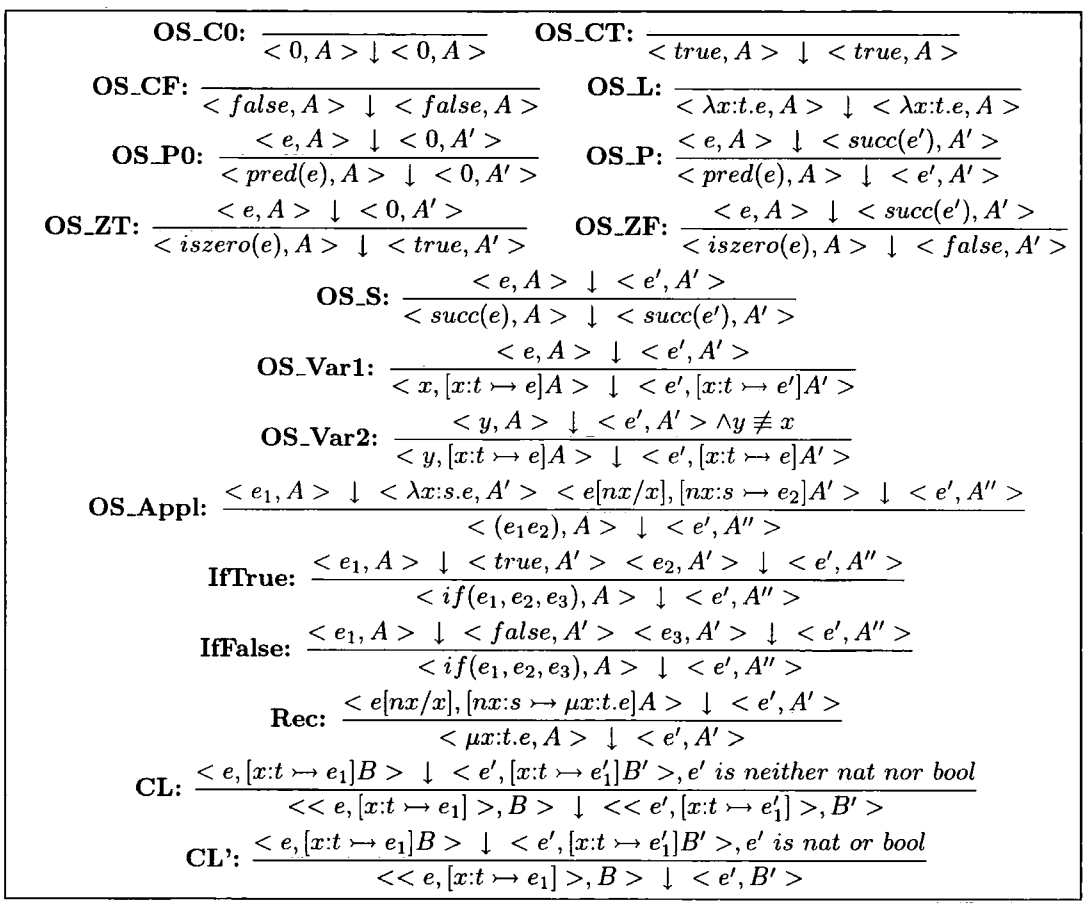

Figure 6.4: The operational semantics of LAZY-PCF+SHAR

The rules in Figure 6.4 are rules to reflect the operational semantics of the language. There is a set NF of expressions which represents the normal form of expressions in the language. It is as follows:

$$NF = 0 \mid true \mid false \mid succ^n(0) \mid F$$

$$F = \lambda x{:}t.e \mid \ < F, [x{:}t \mapsto e_1] >$$

6.4 Special features of this domain

One special feature of this domain is the logic for proving properties. In this domain the rules for operational semantics are not enough to reflect the requirement of the proving of properties. How to design the tool for this kind of domain specific reasoning? We know that many domains have not clear logic for proving properties. Domain users usually use an informal logic for their reasoning. In fact, LF and UTT themselves are good candidate for being a logic. For this kind of domains, we discuss the representation mainly, and let the logic to be the logic of that of LF or UTT. So we use Plastic directly in this domain by using Plastic's interface (i.e. the interface customized in Proof General). Therefore the main work for this kind of domains is the formalization work.

6.5 An implementation of LAZY-PCF+SHAR in LF

We use LF as the meta-language to represent expressions of the object language LAZY-PCF+SHAR. In our encoding here, we focus on the reflection of the explicit substitution.

6.5.1 Translation from LAZY-PCF+SHAR expressions and types to LF expressions

6.5.1.1 Inductive definition of the syntax of LAZY-PCF+SHAR

The following module *Syntax* is our inductive definition of the syntax of LAZY-PCF+SHAR in LF.

```
> module Syntax where;
(*************************************************************************)
(*  syntax.lf:  Inductive definition of the syntax                       *)
(*           of LAZY-PCF+SHAR                                            *)
(*  Includes types, variables and terms                                  *)
(*************************************************************************)
> import Pi;
> import Nat;
```

```
> Inductive
>          [Ty : Type]
>   Constructors
>             [ nat_Ty : Ty]                -- natural numbers
>             [ bool_Ty : Ty]               -- boolean values
>             [ arr : (e1: Ty)(e2: Ty)Ty];  -- function types


> Inductive
>        [Vari : Type]
>    Constructors
>          [X : (i: El Nat)Vari];



> Inductive
>          [Tm : Type]
>   Constructors
>          [o : Tm]                                        -- zero
>          [ttt : Tm]                                      -- true
>          [fff : Tm]                                      -- false
>          [abs : (v: El Vari)(t : El Ty)(tm1 : Tm)Tm]     -- lambda abstractions
>          [appl : (tm1 : Tm)(tm2 : Tm)Tm]                 -- function applications
>          [cond : (tm1 : Tm)(tm2 : Tm)(tm3 : Tm)Tm]       -- if e1 then e2 else e3
>          [var : (v : El Vari)Tm]                         -- variables
>          [suc : (tm1 : Tm)Tm]                            -- successor
>          [prd : (tm1 : Tm)Tm]                            -- predecessor
>          [is_o : (tm1 : Tm)Tm]                           -- zero test
>          [fix : (v: El Vari)(t1: El Ty)(tm1: Tm)Tm]      -- fixed point operator
>          [clos: (tm1 : Tm)(v: El Vari)(t: El Ty)(tm2 : Tm)Tm];
>                                                          -- closure, <x,[x:t->e]>
```

6.5.1.2 Translation of operational semantics rules

The following module OSrules expresses the operational semantics rules in *LF*.

```
> module OSrules where;
(*************************************************************************)
(*  OSrules.lf                                                           *)
(*                                                                       *)
```

```
(*  This file contains the definition of the operational                *)
(*  semantics rules for LAZY-PCF+SHAR, as well as a definition          *)
(*  of the Ap function and some related properties.                     *)
(***********************************************************************)

> import Typecheck;
> import Rename;

(*********************************************)
(*  OScons  (abbrev.)                        *)
(*    (OScons v t e A) ==                    *)
(*     (cons VTT <VT,tm>(<vari,ty>(v,t),e) A)   *)
(*********************************************)
> [OScons = [v: El Vari][t : El Ty][e : El Tm][A: El OS_env]
>           cons VTT (pair VT Tm (pair Vari Ty v t) e) A];

(*****************************************************)
(* Ap:                                               *)
(*    Inductively defines the relation characterised by *)
(*    the Ap function.                               *)
(*                                                   *)
(*    (Ap a F A F' n t) <--> Ap(F,a)=<F',[n:t->a]>   *)
(*        New variables may not come from            *)
(*        the Domain of OS env A.                    *)
(*                                                   *)
(*****************************************************)
> Inductive
>     [Ap : (ptm1,ptm2: El Tm)(pose: El OS_env)(ptm3: El Tm)
>           (pv: El Vari)(pt: El Ty) El Prop]
>           Relation_LE
>    Constructors
>    [Ap_abs: (nv,v:El Vari)(t: El Ty)(a,e,ne: El Tm)(A: El OS_env)
>               (p1: El (Prf(not (member Vari  nv (OS_Dom A)))))
>               (p2: El (Prf(Rename nv v e ne)))
>               Prf( Ap a (abs v t e) A ne nv t)]
>    [Ap_clos: (n,v:El Vari)(s,t: El Ty)(a,e,ne,e1: El Tm)(A: El OS_env)
>                (p1: Prf(Ap a e (OScons v s e1 A) ne n t))
>                Prf (Ap a (clos e v s e1) A (clos ne v s e1) n t)];
```

```
(*************************************)
(*  OSrules                          *)
(*                                   *)
(*  Definition of Operational Semantics *)
(*                                   *)
(*  <e,A> -> <e',A'>                 *)
(*                                   *)
(*************************************)

> Inductive
>    [OSred : (conf1:El Config)(conf2: El Config)El Prop]
>        Relation_LE
>   Constructors
>    [OS_C0: (A: El OS_env)Prf(OSred (cfg o A) (cfg o A))]
>    [OS_CT: (A: El OS_env)Prf(OSred (cfg ttt A) (cfg ttt A))]
>    [OS_CF: (A: El OS_env)Prf(OSred (cfg fff A) (cfg fff A))]
>    [OS_L: (A: El OS_env)(e : El Tm)(t: El Ty)(x: El Vari)
>           Prf(OSred (cfg (abs x t e) A) (cfg (abs x t e)  A))]
>    [OS_P0: (A,A1: El OS_env)(e: El Tm)
>           (p1: Prf(OSred (cfg e A) (cfg o A1)))
>            Prf(OSred (cfg (prd e) A) (cfg o A1))]
>    [OS_P: (A,A1: El OS_env)(e,e1: El Tm)
>            (p1: Prf(OSred (cfg e A) (cfg (suc e1) A1)))
>            Prf(OSred (cfg (prd e) A) (cfg e1 A1))]
>    [OS_ZT: (A,A1: El OS_env)(e: El Tm)
>            (p1: Prf(OSred (cfg e A) (cfg o A1)))
>            Prf(OSred (cfg (is_o e) A) (cfg ttt A1))]
>    [OS_ZF: (A,A1: El OS_env)(e,e1: El Tm)
>            (p1: Prf(OSred (cfg e A) (cfg (suc e1) A1)))
>            Prf(OSred (cfg (is_o e) A) (cfg fff A1))]
>    [OS_S: (A,A1: El OS_env)(e,e1: El Tm)
>            (p1: Prf(OSred (cfg e A) (cfg e1 A1)))
>            Prf(OSred (cfg (suc e) A) (cfg (suc e1) A1))]
>    [OS_Var1: (A,A1: El OS_env)(e,e1: El Tm)(t:El Ty)(x: El Vari)
>               (p1: El (Prf(not (member Vari x (OS_Dom A)))))
>               (p2: Prf(OSred (cfg e A) (cfg e1 A1)))
>               Prf(OSred (cfg (var x) (OScons x t e A))
>                  (cfg e1 (OScons x t e1 A1)))]
>    [OS_Var2: (A,A1: El OS_env)(e,e1: El Tm)(t:El Ty)(x,y: El Vari)
>               (p1 : El (Prf (not (Eq Vari x y))))
```

```
>              (p2: El (Prf(not (member Vari x (OS_Dom A)))))
>              (p3: Prf(OSred (cfg (var y) A) (cfg e1 A1)))
>              Prf(OSred (cfg (var y) (OScons x t e A))
>                 (cfg e1 (OScons x t e A1)))]
>   [OS_Appl: (A,A1,A2: El OS_env)(e1,e2,en1,en2,enf: El Tm)
>              (t:El Ty)(n: El Vari)
>              (p1 : Prf (OSred (cfg e1 A) (cfg en1 A1)))
>              (p2: El (Prf(Ap e2 en1 A en2 n t)))
>              (p3: Prf(OSred (cfg (clos en2 n t e2) A1) (cfg enf A2)))
>              Prf(OSred (cfg (appl e1 e2) A) (cfg enf A2))]
>   [OS_IfTrue: (A,A1,A2: El OS_env)(e1,e2,e3,en: El Tm)
>              (p1 : Prf (OSred (cfg e1 A) (cfg ttt A1)))
>              (p2 : Prf (OSred (cfg e2 A1) (cfg en A2)))
>              Prf(OSred (cfg (cond e1 e2 e3) A) (cfg en A2))]
>   [OS_IfFalse: (A,A1,A2: El OS_env)(e1,e2,e3,en: El Tm)
>              (p1 : Prf (OSred (cfg e1 A) (cfg fff A1)))
>              (p2 : Prf (OSred (cfg e3 A1) (cfg en A2)))
>              Prf(OSred (cfg (cond e1 e2 e3) A) (cfg en A2))]
>   [OS_Fix: (A,A1: El OS_env)(e,e1,en: El Tm)(t:El Ty)(x,nx: El Vari)
>              (p1: El (Prf(not (member Vari nx (OS_Dom A)))))
>              (p2: El (Prf (Rename nx x e e1)))
>              (p3: Prf(OSred (cfg (clos e1 nx t (fix x t e)) A)
>                  (cfg en A1)))
>              Prf(OSred (cfg (fix x t e ) A) (cfg en A1))]
>   [OS_CL: (A,A1: El OS_env)(e,e1,en,e2: El Tm)(s,t:El Ty)(x: El Vari)
>              (p1: Prf(OSred (cfg e (OScons x t e1 A))
>                      (cfg en (OScons x t e2 A1))))
>              (p2: El (Prf (TC (OS_Dom_ty (OScons x t e1 A)) en s)))
>              (p3: El (Prf (not (or (Eq Ty s nat_Ty) (Eq Ty s bool_Ty)))))
>           Prf(OSred (cfg (clos e x t e1 ) A) (cfg (clos en x t e2) A1))]
>   [OS_CL': (A,A1: El OS_env)(e,e1,en,e2: El Tm)(s,t:El Ty)(x: El Vari)
>              (p1: Prf(OSred (cfg e (OScons x t e1 A))
>                      (cfg en (OScons x t e2 A1))))
>              (p2: El (Prf (TC (OS_Dom_ty (OScons x t e1 A)) en s)))
>              (p3: El (Prf (or (Eq Ty s nat_Ty) (Eq Ty s bool_Ty))))
>           Prf(OSred (cfg (clos e x t e1 ) A) (cfg en A1))];
```

6.5.2 An example

Using the above definition we can prove semantic properties of LAZY-PCF+SHAR. We have proved many properties related to LAZY-PCF+SHAR. The successful proofs of these properties reflect the power and suitability of Plastic as the reasoning tool for this domain. To show these, we introduce some definitions firstly.

Definition 13 *(Type Context of an Environment)*
$Context([\,]) = \perp$
$Context([x{:}t \mapsto e]A) = Context(A)[t/x]$
where $\perp$ *is the mapping that is undefined for each variable.*

Definition 14 *(Dom)*
Dom(H) is used to denote the domain of a context H and Dom(A) is used to denote the set of variables which have bindings in the operational semantics environment A.

Definition 15 *(Context extension)*

1. *H is an extension of H;*
2. *if H' is an extension of H and* $x \notin Dom(H')$*, then* $H'[t/x]$ *is an extension of H.*

We can also define this more formally as:

1. $H \vdash H$
2. if $H' \vdash H$ and $x \notin Dom(H')$, then $H'[t/x] \vdash H$.

The following is a semantic property described in LF:
$Ap(a, fun, A) =< b, [n{:}t \mapsto a] > \rightarrow n \notin Dom(A)$

This property shows the following fact: while applying a function fun to an expression a if the environment variable n is used to represent the expression a, then n should not be in the environment before the applying. We give this property a name "ApNewVar", and present a proof in Plastic as the following. The motivation of showing this code here is just to let reader know the profile of the proof. For further understanding please refer to the manual of Plastic [Callaghan, 2000a].

```
> Claim ApNewVar : (a,fun,b: El Tm)(A: El OS_env)(n : El Vari)(t: El Ty)
>                  (p1: El (Prf (Ap a fun A b n t)))
>                  Prf (not (member Vari n (OS_Dom A)));
> Intros a fun b A n t p1;
> Refine E_Ap ([a,fun:El Tm][A: El OS_env][b:El Tm][n: El Vari][t:El Ty]
>            Prf(not (member Vari n (OS_Dom A)))) ? ?  a fun A b n t p1;
> 2 Intros nv;
```

```
> Intros v  t1 tm1 e ne A1 pr1 pr2;
> Refine pr1;
> ReturnAll;
> Intros n1 v s t1 a1 e ne e1 A1 pr1 pr2;
> Refine LL;
> Intros H;
> Refine App ? ? pr2 ?;
> Refine App ? ? (p_inr ? ? ) ?;
> Refine H;
> ReturnAll;
> ApNewVar;
```

Now we discuss the proof of a main theorem— Subject Reduction theorem. First, we need to give some necessary definitions and lemmas.

Definition 16 *(Valid environments)*

1. $[\,]$ *is a valid environment.*
2. *If A is a valid environment and $Context(A) \vdash e{:}t$, then $[x{:}t \mapsto e]A$ is a valid environment.*

This definition implies that the free variables of an expression bound in a valid environment must be bound in the remainder of the environment. This is due to the fact that if an expression has a type in some type context then the free variables of that expression occur in the domain of the type context. The definition of valid environments can be extended to configurations by requiring that the environment of a configuration is valid and that the expression of the configuration has some type in the type context of the environment.

Definition 17 *(Valid configurations)*
If A is a valid environment and for some t, $Context(A) \vdash e{:}t$, then $< e, A >$ is a valid configuration.

This concept is very important. Because the operational semantics are designed to yield meaningful results only when they are applied to valid configurations.

Lemma 6.5.1 *If $< e, A > \downarrow < e', A' >$, then $Context(A') \vdash Context(A)$.*

Proof The proof is by induction on the height of the inference justifying
$< e, A > \downarrow < e', A' >$.

Q.E.D.

This lemma shows that if one configuration evaluates to another configuration, then the type context of the second environment extends the type context of the first environment.

Theorem 6.5.1 *(Subject Reduction Theorem)*
If $Context(A) \vdash e{:}t$, *A is valid, and* $< e, A > \downarrow < e', A' >$, *then* $Context(A') \vdash e'{:}t$, *and A' is valid.*

Proof The proof is carried out by case analysis on the inference justifying $< e, A > \downarrow < e', A' >$ and relevant inductions. The proof is very big, please refer to the appendix A for further details.

Q.E.D.

The theorem shows that the evaluation preserves the type. In other words the type of an expression is the same as the type of its normal form.

6.6 Discussion

Generally speaking, we have done the following tasks in the case study for this domain.

- An analysis on the concepts of the domain, especially on the features of functional programming and explicit substitution;
- A formalization of the concepts of the domain, this includes the definitions of the relevant concepts in LF, the proofs of the relevant lemmas etc.
- An explanation of how to do reasoning in this domain, this includes the proofs of many relevant domain properties.

This case study also gives us a deep understanding of the application of proof assistant Plastic. We learn a lot of features of Plastic in doing domain-specific reasoning.

- The case study shows that proof assistants are suitable to some domains directly. Our approach does not exclude this direct use. For this kind of domains, the suitability of our approach is depended on the power and suitability of the underlying proof assistant (here is Plastic). Our conclusion is: the Plastic system is qualified to be an underlying system.
- The case study can also be seen as a big application of Plastic. Because it includes a lot of formalization work and proof work.
- The metavariable mechanism provided by Plastic is very useful in doing proofs. It gives us a very flexible means when we want to conquer difficult problems. For example, using this mechanism we can prove some temporary lemmas in the environment of the proof procedure of the main property. This concords with the reasoning habit of human being.

- Compare with proofs by hand, doing proofs in Plastic is stricter and more convincing. But the proofs in Plastic are not as understandable as the proofs by hand. But for this domain this shortcoming can be relieved by using a well defined formalization of the domain. For the domain such as we discussed in chapter 5 this shortcoming can be relieved by using a well defined domain specific interface.

- Compare with Coq and Lego, doing proof in Plastic is overloaded with details. The reason is that Plastic is a concise implementation of LF. There are no many modules accompanied Plastic inherently. But this is also a merit of Plastic, because it forces its user to customize a fitted one for his application. This avoids users from using extra properties inherited from the proof assistant.

Chapter 7

The interface

Only those who have the patience to do simple things perfectly ever acquire the skill to do difficult things easily.

— FRIEDRICH SCHILLER, GERMAN DRAMATIST AND POET.

In this chapter we investigate the aspects of interface related to this approach. We focus on the principle of the design and implementation.

7.1 Design principle

7.1.1 General principle for designing domain user interface

Jakob Nielsen [Nielsen, 2005] presented ten general principles for design of user interface. They are called "heuristics" because they are more in the nature of rules of thumb than specific usability guidelines. The following are the ten general principles.

- Visibility of system status: The system should always keep users informed about what is going on, through appropriate feedback within reasonable time.
- Match between system and the real world: The system should speak the users' language, with words, phrases and concepts familiar to the user, rather than system-oriented terms. Follow real-world conventions, making information appear in a natural and logical order.
- User control and freedom: Users often choose system functions by mistake and will need a clearly marked "emergency exit" to leave the unwanted state without having to go through an extended dialogue. Support undo and redo.
- Consistency and standards: Users should not have to wonder whether different words, situations, or actions mean the same thing. Follow platform conventions.

- Error prevention: Even better than good error messages is a careful design which prevents a problem from occurring in the first place.
- Recognition rather than recall: Make objects, actions, and options visible. The user should not have to remember information from one part of the dialogue to another. Instructions for use of the system should be visible or easily retrievable whenever appropriate.
- Flexibility and efficiency of use: Accelerators – unseen by the novice user – may often speed up the interaction for the expert user such that the system can cater to both inexperienced and experienced users. Allow users to tailor frequent actions.
- Aesthetic and minimalist design: Dialogues should not contain information which is irrelevant or rarely needed. Every extra unit of information in a dialogue competes with the relevant units of information and diminishes their relative visibility.
- Help users recognize, diagnose, and recover from errors: Error messages should be expressed in plain language (no codes), precisely indicate the problem, and constructively suggest a solution.
- Help and documentation: Even though it is better if the system can be used without documentation, it may be necessary to provide help and documentation. Any such information should be easy to search, focused on the user's task, list concrete steps to be carried out, and not be too large.

The above ten principles give us a good guide to design general interface. But for different domains and different kind of users the relevant interfaces should have their own principles.

7.1.2 Principle for designing a reasoning interface based on LF

We follow the above principles and consider the speciality of our system. The design of an interface to help LF based reasoning depends on the intended users. Novices need some way to define the goal, to view the result of the reasoning and to provide reasoning guidance. They want a simple interface with limited functionality, so that they do not become confused and/or issue instructions at variance with their intentions. More experienced users may also require ways to define new theories, to browse through libraries of theorems, definitions, etc and to switch between one part of a proof attempt and another. System developers want access to the underlying system. They want multiple views onto the underlying system and reasoning process and want a rich functionality. So different focus leads to different style of interface. But in general a good interface must: assist users to understand the current reasoning attempt; provide mechanisms for them to interact with the reasoning process; avoid bewildering them with too much information, while providing what is required; and help them explore their options without imposing too high a cognitive load.

For our design of the domain-specific interface, we focus on the following issues:

- Use LF as the underlying meta-theory.
- Provide domain-specific user-friendly notations and reasoning with no acquaintance with the underlying meta-theory.
- Can be customized to many domains.

So, the major design principles of LFTOP are as follows:

- Suitable to several domain-specific reasoning system without many changes of their code.
- Support communications between different levels.
- Reuse of customizing steps.
- High level user level language with automated translation into the underlying meta-language.
- Support a variety of user interfaces (GUI, command line).
- Multiple views: Different views make different information and manipulation of information explicit and easy to understand. A declarative representation of a proof, such as a proof term, can have advantages over a procedural representation of a proof, such as a list of tactics. Representing a proof at a high level of abstraction may make its structure clearer. Domain-specific proof support may extend as far as providing views which visualize objects at the level of domain users. More concretely, we are interested in the following principles:
  - *Principle 1:* There should be a number of complementary views of the proof construction and the user should be able to choose to see any number of the views simultaneously.
  - *Principle 2:* Within any view the user should be able to invoke operations that are meaningful in that view.
  - *Principle 3:* For multiple-part commands the interface should provide defaults for any variables that the user does not specify.
  - *Principle 4:* If there is a choice for the default values then the option which results in the simplest proof step and the easiest to undo should be chosen.
  - *Principle 5:* There should be a high level of flexibility in which the user can articulate commands to the prover.
  - *Principle 6:* The user interface should support the user by displaying only information that is relevant in the current state.
  - *Principle 7:* The user interface should support several concurrent proof constructions.

We can divide these to the following three levels:

- At first, most immediate level, an interface should be designed to make the customary interactions with the theorem prover as convenient as possible.
- Secondly, somewhat deeper level, an interface should provide supplementary services in the theorem prover itself.
- Thirdly, even deeper level, an interface should provide derived proof rules that allow the user to reason in familiar ways, e.g., using its favorite logic and syntax.

In order to implement a useful interface, we have done the following:

- The translation from the user level domain-specific language to LF.
- The translation from the user level commands to Plastic commands, tacticals or proved lemmas.
- The maintenance of the correspondences between the two levels. These include mappings between the rules, definitions, and induction schemas used by both levels, as well as the correspondences between explicit terms used in the proofs performed by both levels.
- The communication protocols between the two levels,

Our design of the interface is implemented in Haskell and through two different ways. One is using Proof General as the based tool, the other is implemented directly in Java.

7.2 ULPIP: a protocol for communications between user-level and Plastic-level

Communications between user-level and Plastic-level need a protocol. We have designed a protocol called ULPIP for conducting the User-Level to Plastic-level Interactive Proof. A proof using this protocol may have the following parts:

- Proof begins by issuing a target claim [1]
- Proof proceeds by successive proof steps
- Proofs in different levels keep correspondence.

This protocol should have a good extensibility to different domains. Our case studies try to confirm this point.

[1] on different level the claim has different form

7.2.1 Usage of eXtensible Markup Language (XML)

XML is a descriptive markup language. It is used popularly in the area of data exchange and memory. Now it is one of the main tools for publishing and assembling information in the internet. XML provides both programmers and authors of document with a friendly environment. XML's rigid set of rules helps make documents more readable to both humans and machines. XML is extensible. It allows developers to create their own DTDs[Laurent, 1999] which create extensible tag sets and can be used for many applications. This presents us a clear flexible means to represent and understand a protocol. Meanwhile Aspinall's work [Aspinall, 2000] [Aspinall and Lüth, 2003] inspired us to use XML to do this. In fact, the main reason why we use XML to describe the protocol is that it can offer the following properties we want:

- *Platform independent:* XML is platform independent. If there is an XML parser available on the system, the XML description can be used. If there is no suitable XML parser available for your target platform, XML is so simple that writing your own parser is fairly easy;
- *Consistent:* through the usage of DTD, XML can be consistent. A DTD specifies a set of rules for the XML file to follow;
- *Rapid prototyping:* using a stylesheet the results can be seen immediately in a browser;
- *Constraint definitions:* XML can contain constraint definitions; as well for the form of the XML itself, as for external resources we can add constraint definitions;
- *Easily extensible:* because XML is a metalanguage, it is by nature an extensible language;
- *Reusability:* it is relatively easy to fit an existing piece of XML into another.

7.2.2 DTD for XML documents

The basic unit of XML documents is element. An element consists of a start tag, contents and an end tag. The start tag is at the beginning of the element with the form $< tag - name >$. The end tag is at the end of the element with the form $< /tag - name >$. An element can be nested defined.

Document type definition (DTD) is usually used to present a set of constraint rules for an XML document. A constraint rule is called a declaration of an element. A DTD of an XML document will indicate the legal elements, the correct order of the elements and some constraint rules which the elements should satisfy. A declaration of an element consists of the parts which are quoted by "<>", "!ELEMENT" is followed by the name of the element.

The structure of an XML document can be represented by a tree. The root note is the root element of the DTD, the other notes are the other elements of the DTD, the leaves of

the tree are of the basic element types (#PCDATA or #CDATA [2]). We created our DTDs for the definition of the protocol. DTDs present a clear picture of the protocol syntax, and, especially with a validating parser, can enforce very precise syntactic requirements. Flexibility in adding or changing a protocol is particularly important when designing new functions. This is an area where the XML approach really stands out. It is very simple matter to modify the DTDs, change a dispatch table in the code, and test a new feature or command; it is certainly much easier than modifying ad hoc parsing code. In fact, this flexibility invites software re-use as well.

7.2.3 DTD for the protocol

We benefited from XML and HaXml (a Haskell package for XML) in our protocol design and implementation. Using a DTD file the messages can be divided into several classes. But they can be distinguished in two parts: those sent to Plastic and those originating from Plastic. Messages sent to Plastic include configuration commands, actual proof commands and commands for inspecting various aspects of Plastic state(e.g. the Entity %reqcmd in the following DTD file). Messages sent from Plastic consist of error dialogues or status display and messages to configure user-level components(e.g. the Entity %respcmd in the following DTD file). The following is part of the DTD file for our simple protocol:

```
<?xml version="1.0" encoding="UTF-8"?>

<!-- DTD for ULPIP, the protocol for User Level system and Plastic  -->

<!ENTITY %reqcmd  " setmodule | setimport | setdef | setg | getng |
 getst | getccxt | runtact | undo | end " >

<!ENTITY %respcmd  " ng | ok | err | cctxt " >

<!ENTITY %plasticmsg " askulpmlver | askprefset |  resetpref | setpref | getpref " >

<!ENTITY %ulmsg " tellulpmlver | tellprefset | tellprefval  " >



<!ELEMENT ulpip ( %reqcmd;  | %respcmd; | %plasticmsg; | %ulmsg; )* >
<!ATTLIST ulpip
          version    CDATA #IMPLIED
```

[2]PCDATA stands for "parsed character data" and CDATA stands for "character data" that contains no mark-up. PCDATA is used for entity element content and CDATA is used for attributes

```
          receiver   CDATA #REQUIRED
          origin     CDATA #IMPLIED
          systemid   CDATA #IMPLIED>

<!-- request command message -->

<!ELEMENT setmodule   EMPTY >
<!ATTLIST setmodule
          name   CDATA #REQUIRED >


<!ELEMENT setimport   EMPTY >
<!ATTLIST setimport
          name   CDATA #REQUIRED >

<!ELEMENT setdef   (#PCDATA) >
<!ATTLIST setdef
          name   CDATA #REQUIRED >

<!ELEMENT setg   (#PCDATA) >
<!ATTLIST setg
          name   CDATA #REQUIRED >

<!ELEMENT getng  EMPTY>
<!ELEMENT getst  EMPTY>
<!ELEMENT getccxt  EMPTY>

<!ELEMENT runtact  (#PCDATA) >
<!ELEMENT undo  EMPTY>
<!ELEMENT end  EMPTY>
<!-- response command message -->


<!ELEMENT ng  (goal)*>
<!ELEMENT goal  (#PCDATA)>
<!ATTLIST goal
          name  CDATA #IMPLIED >
```

```
<!ELEMENT ok  EMPTY>
<!ELEMENT err  (#PCDATA)>
<!ATTLIST err
          code  CDATA #REQUIRED>
<!ELEMENT cctxt  (newctxt, lostctxt)>

<!ELEMENT newctxt (defctxt | hypctxt | goalctxt )*>
<!ELEMENT defctxt (defname, defterm, deftype)>
<!ELEMENT hypctxt (hypname, hyptype)>
<!ELEMENT goalctxt (goalname, goaltype)>

<!ELEMENT defname (#PCDATA)>
<!ELEMENT defterm (#PCDATA)>
<!ELEMENT deftype (#PCDATA)>

<!ELEMENT hypname (#PCDATA)>
<!ELEMENT hyptype (#PCDATA)>

<!ELEMENT goalname (#PCDATA)>
<!ELEMENT goaltype (#PCDATA)>

<!ELEMENT lostctxt (defctxt | hypctxt | goalctxt )*>



<!-- configuration message  -->



<!ELEMENT askulpmlver EMPTY>

<!ELEMENT askprefset EMPTY>
<!ATTLIST askprefset
           class  CDATA #REQUIRED>


<!ELEMENT  resetpref  EMPTY>
<!ATTLIST  resetpref
```

```
          class  CDATA #REQUIRED>


<!ELEMENT  setpref  (#PCDATA)>
<!ATTLIST  setpref
           name   CDATA #REQUIRED>

<!ELEMENT  getpref  EMPTY>
<!ATTLIST  getpref
           name   CDATA #REQUIRED>


<!ELEMENT tellulpmlver  EMPTY>
<!ATTLIST tellulpmlver
          version  CDATA #REQUIRED>

<!ELEMENT tellprefset  EMPTY>
<!ATTLIST tellprefset
          type   CDATA #REQUIRED>

<!ELEMENT tellprefval  EMPTY>
<!ATTLIST tellprefval
          name CDATA #REQUIRED>

<!ELEMENT br EMPTY>

.....
```

The reason why we include DTD's in XML documents, is that the DTD is a vocabulary of the XML elements that we use. It lets us do validation in our XML document and it also describes what kind of element for example the 'goalname' element is (e.g. is it PCDATA?).

From the contents of the DTD file we can see that this design has some advantages in readability of code both to human being and machine. This provides a good foundation to the concrete implementation.

To model our protocol faithfully in Haskell, the tool HaXML is used. From the above DTD file, HaXML generates a series of Haskell datetypes, one for each element, along with functions to read and write XML. So the type security given by the DTD extends into our program, detecting the reception of invalid XML immediately and making it nearly impossible to send messages containing invalid XML.

7.3 Implementation issues in our design

7.3.1 Some considerations on the implementation issues

We have the following considerations on the implementation issues:

- We should remember the user proof in the interface other than forget it.
- The construction of a proof is often a trial-and-error process, and keeping track of the current partial proof helps the step back and redo certain parts of it as the proof is being constructed.
- Proof editing and maintenance is an important feature of an interface.

Could we use Proof General in our interface? The answer is yes. We have implemented the interface in two different ways, one is using JAVA and Haskell, the other is using Haskell and Proof General. The first way is more complex than the second one, but it's more specific. The implementation of the second way is simple, it provides a unified Proof General style interface.

7.3.2 An interface in Proof General style

Figure 7.1 is a screenshot of our prototype of the interface in Proof General style:

From the figure 7.1 we can see that LFTOP can be used directly as a domain-specific reasoning tool by domain users. In this figure we show two different windows, one is a window of definitions and commands for domain users, the other is an information window for showing the corresponding information on Plastic level. So, if domain users really care about the process of the underlying reasoning, it is very easy from this window. This provides a convenient means to learn how to do reasoning directly in Plastic .

7.4 Discussion

We have so far presented the principles which we follow in the design of our interface and some issues of implementation. The main idea is to reflect our view of the approach. Our prototype of implementation is based on the above principles. XML presents us a platform independent extendible language for representing protocols. It is easier both to human being and machine to understand the protocol. This can get a more accurate implementation. We are inspired by Aspinall's work for Proof General Kit [Aspinall, 2005b]. But our focus is different from his work. Our purpose is to facilitate different specific domains to one proof assistant, not like Proof General Kit which tries to provide facilities to many different proof assistant other than different domains.

Through the study of this chapter, we got the following results:

1. An analysis of the issues related to design of interface;

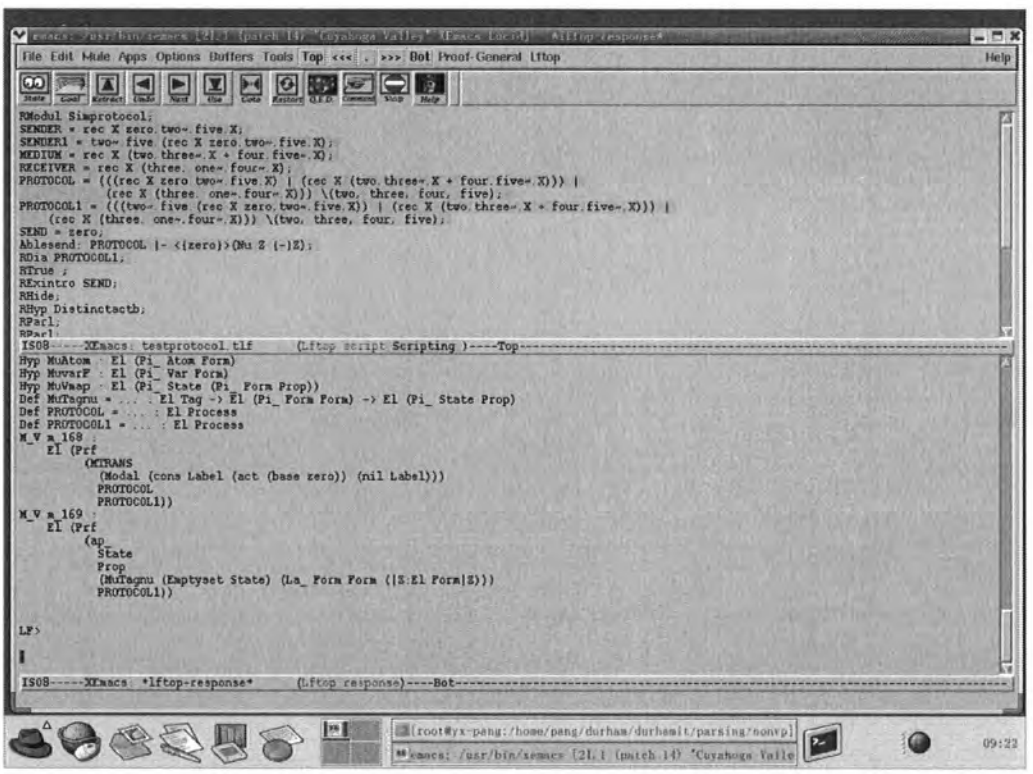

Figure 7.1: A screenshot of the interface in Proof General style

2. A better understanding on the issues of the interface;

3. A protocol for communications between user level and Plastic level.

Our implementation is just a simple prototype and is in a preliminary stage. Many improvements are needed. For example, from the domain user's point of view:

1. There is no easy way for interoperability with other tools.

2. Not enough domain-specific options are provided.

3. There is no mechanism for domain users to customize their favorite style and commands of reasoning.

4. The output from Plastic has not been well organised and decorated yet.

We hope that the protocols and components will be improved in stages by successive generalization to more specific domains.

Chapter 8

Translation issues

Eternal truths will be neither true nor eternal unless they have fresh meaning for every new social situation.

— FRANKLIN ROOSEVELT, AMERICAN PRESIDENT

Translation between different levels in this approach is very important. How to guarantee the successful forward and backward translation is one of the main problems which should be solved in the approach. A good correspondence between these levels is a critical issue.

8.1 Some problems in translations

One interesting question is the effect of computation on the translation. Originally, we intended that the translation is a bi-directional map between the object language and a subset LF_T of the framework language LF. It was suggested that manipulation of terms in LF_T results in further terms in LF_T, e.g. that LF_T is closed under computation. In fact it is incorrect. We found a counter-example as mentioned in [Pang et al., 2002], that is the `subst` problem.

Our revised model is shown in figure 8.1, where a superset LF_M of LF_T contains terms which can result from manipulations of terms in LF_T, either by computation or by appearing as a sub-goal after a rule application, but that LF_M can be mapped back to LF_T by computation involving a distinguished set of operators. Such operators will be elements of the formalization whose purpose is to explain manipulations on terms, but don't correspond to observable phenomena in the domain, hence should not necessarily be shown to users. The aforementioned `subst` fits this pattern. Dr. Callaghan had studied this problem and found a way to solve it. This idea was embodied in the new version of Plastic. Plastic now implements a normalization operation which removes the obvious use of a set of operations by computation. So the above `subst` is removed in the relevant terms after this kind of

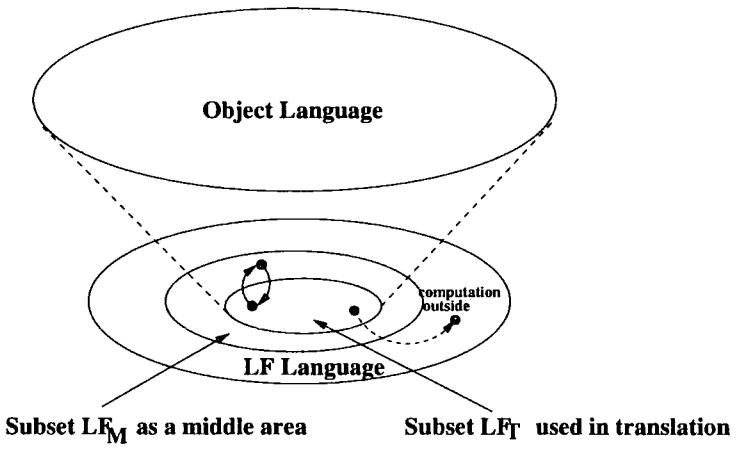

Figure 8.1: Mapping between domain-specific object language and a subset of LF

normalization operation. This feature of Plastic presents a good method to keep the correspondence between the user level and LF level. Using this feature and forcing the use of lemmas or tactics relevant to domains, the results are guaranteed to be in terms of LF_T.

8.2 The translation from user level to LF level

We use $[\![(\Gamma, -)]\!]$ to express the translation map in the context Γ. For translations which are not dependent on the context, we omit the context item. The translation is implemented by parsing and translating in our Haskell and Happy program. Next we'll discuss the case of concurrency mainly. Of course, for the case of Chapter 6, as there is no distinction of the LF layer and user layer, the translations are not needed.

8.2.1 The case of concurrency

In this subsection we discuss the translations of the relevant elements in concurrency.

8.2.1.1 The translation of the predefined actions

The actions in CCS on the user level have three different forms:

- *tau* which express the idling or internal action;
- string which is not ended with ˜ to express base action;
- string which is ended with ˜ to express complement action.

We have the predefined actions named by one, two, ..., ten; and one˜,two˜,...,ten˜on the user level for convenience. These actions can be used without declaration. In fact we can expand the predefined actions by giving the definition of the same string on the LF level.

We use the following types to express actions on LF level.

```
> [Base = Nat];

> Inductive
>        [Actb:Type]
>    Constructors
>    [base:(b:El Base)Actb]
>    [comp:(b:El Base)Actb];

> Inductive
>        [Act:Type]
>    Constructors
>    [tau:Act]
>    [act:(a:El Actb)Act];
```

The translation is:

$$[\![a]\!] = \begin{cases} \text{tau} & \text{if a is tau;} \\ \text{act (base a)} & \text{if a is not ended with } \tilde{}; \\ \text{act (comp b)} & \text{if a is b}\tilde{}; \end{cases}$$

8.2.1.2 The translation of the list of hidden actions

For the list of hidden actions (They should be base actions), we have a quite direct translation as follows:
$[\![\ ]\!]$ = (nil Actb)
$[\![a, A]\!]$ = (cons Actb (base a) $[\![A]\!]$)
where (nil Actb) is the empty list of type Actb on the level of LF.

8.2.1.3 The translation of the list of relabelling

For the list of relabelling (it is needed to consider just base actions), we have a quite direct translation as follows:
$[\![\ ]\!]$ = (nil (Pair Base Base))
$[\![a/b, A]\!]$ = (cons (Pair Base Base) (pair Base Base a b) $[\![A]\!]$)

8.2.1.4 The translation of processes

We use inductive type on LF level to formalize the basic concepts for CCS. For the following CCS syntax:
$E ::= Nil \mid X \mid a.E \mid E1 + E2 \mid E1|E2 \mid E\backslash L \mid E[f] \mid rec\ X.E$
Its formalization is like this:

```
> Inductive
>       [Process:Type]
>    Constructors
>    [Nil:Process]
>    [var:(v:El Var)Process]
>    [dot:(a:El Act)(E: Process)Process]
>    [choice:(E1: Process)(E2: Process)Process]
>    [par:(E1: Process)(E2: Process)Process]
>    [hide:(E: Process)(L:El (List Actb))Process]
>    [ren:(E: Process)(f: El (List (Pair  Base Base)))Process]
>    [rec: (E: Process)Process];
```

The translation is:

- $[\![(\Gamma, Nil)]\!] = Nil$
- $[\![(\Gamma, X)]\!] = \begin{cases} \text{var n} & \text{if} \quad \text{X is the n}^{th} \textit{ item in } \Gamma\text{, here n is a natural number on } LF \text{ level} \\ & \quad\ \text{to reflect deBruijn index;} \\ \text{var X} & \text{if} \quad \text{X is not in } \Gamma. \end{cases}$
- $[\![(\Gamma, a.E)]\!] = dot\ [\![a]\!]\ [\![(\Gamma, E)]\!]$
- $[\![(\Gamma, E1 + E2)]\!] = choice\ [\![(\Gamma, E1)]\!]\ [\![(\Gamma, E2)]\!]$
- $[\![(\Gamma, E1|E2)]\!] = par\ [\![(\Gamma, E1)]\!]\ [\![(\Gamma, E2)]\!]$
- $[\![(\Gamma, E\backslash L)]\!] = hide\ [\![(\Gamma, E)]\!]\ [\![L]\!]$
- $[\![(\Gamma, E[f])]\!] = ren\ [\![(\Gamma, E)]\!]\ [\![f]\!]$
- $[\![(\Gamma, Rec\ X.E)]\!] = rec\ [\![(X{:}\Gamma, E)]\!]$

  where $X{:}\Gamma$ express putting X as the first element of the context.

8.2.1.5 The translation of μ-calculus

The form of μ-calculus formula on the user level is like the following:
$F ::= A \mid F||F \mid F\&F \mid < K > F \mid [K]F \mid Mu\ Z.UF \mid Nu\ Z.UF$

where K is range over subsets of labels, U is a tag which is a subset of states. A is an assertion variable.

For the form of μ-calculus formula on the user level, we have the corresponding definition on LF level as follows:

```
> [Pred = [A:Type](Pi_ A Prop)];
> [Form  = Pred  State];
> [Tag = Pred State ];
> [VarF : Form];
> [MuOr = [A,B: Form] (Union State A B)];
> [MuAnd = [A,B: Form] (Meet State A B)];
> [MuDia = [K: Modality][F: Form]( La_ State Prop ([s: State](Ex State
>          ([s': State] (and (MTRANS K s s') (ap_ State Prop F s'))))))];
> [MuBox = [K: Modality][F: Form]( La_ State Prop ([s: State](FA State
>          ([s': State] ((MTRANS K s s') => (ap_ State Prop F s'))))))];
> [MuTagnu = [T: Tag ][F: (Pi_ Form Form)]( La_ State Prop ([s: State](Ex Form
>        ([P: Form] (and (Subset State P (Union State (ap_ Form Form F P) T ))
>        (ap_ State Prop P s))))))];
> [MuTagmu = [T: Tag ][F: (Pi_ Form Form)]( La_ State Prop ([s: State](FA Form
>        ([P: Form] ((Subset State  (Minus  State (ap_ Form Form F P) T ) P)
>          =>  (ap_ State Prop P s))))))];
```

The translation is:

- $[\![VarF]\!] = VarF$
- $[\![f1\&f2]\!]$ = MuAnd $[\![f1]\!]$ $[\![f2]\!]$
- $[\![f1||f2]\!]$ = MuOr $[\![f1]\!]$ $[\![f2]\!]$
- $[\![[k]f]\!]$ = MuBox $[\![k]\!]$ $[\![f]\!]$
- $[\![< k > f]\!]$ = MuDia $[\![k]\!]$ $[\![f]\!]$
- $[\![Mu\ s\ st\ f]\!]$ = MuTagnu $[\![st]\!]$ (La_ Form Form ([s:Form] $[\![f]\!]$))
- $[\![Nu\ s\ st\ f]\!]$ = NuTagnu $[\![st]\!]$ (La_ Form Form ([s:Form] $[\![f]\!]$))

8.2.1.6 The translation of propositions

The claim to prove that a process has some property on the user level can be translated to a claim to prove a proposition on LF level. For the claim of the following form:
pname: $P \vdash F$
where P is a process, F is a μ-calculus formula and pname is a string as the name for this proof. We have:
$[\![$*pname*: $P \vdash F]\!]$ = Claim pname: Prf (ap_ State Prop [F] [([ ],P)])
where [ ] expresses the empty context.

8.2.1.7 The translation of CCS and μ-calculus rules

The CCS and μ-calculus rules which are corresponding to the commands on the user level are translated to lemmas on LF level. There is a "one to one correspondence" between these commands on user level and lemmas on LF level.

For example, if we want to use the rule

Dia with s': $\frac{s' \vdash \Phi}{s \vdash \langle K \rangle \Phi}(s \xrightarrow{K} s')$

on the user level, we use the command "Rule Dia s' ", this is corresponding to the LF level lemma called *lemma_dia_ccs*, so the translation is:
[[Rule Dia s']] = Refine App ? ? (lemma_dia_ccs ? ? ? s' ?) ?

where s' is a name of a definition for a process which we keep the same name on the two levels, ? is a place holder(i.e. an unnamed metavariable in Plastic).

8.2.2 The translation of definitions

The definitions on user level are translated to the definitions on LF level. For example, for the process definition $SENDER = P$, we do the following translation:
$[\![SENDER = P]\!]$ = [SENDER = $[\![([\,], P)]\!]$]

8.2.3 The translation of declaration

The declarations on user level are translated to the hypothesis on LF level. For example, for the declaration Act a,b ; we do the following translation: [[Act a,b]] = [a,b: Act]

8.3 The translation from LF level to user level

We use $\lfloor - \rfloor$ to express this converse translation. Next we'll discuss the case of concurrency mainly.

8.3.1 The case of concurrency

8.3.1.1 The translation of actions

For actions the translation is as follows:

- $\lfloor$ tau $\rfloor$ = tau
- $\lfloor$ act (base i) $\rfloor$ = i
- $\lfloor$ act (comp i) $\rfloor$ = i~

8.3.1.2 The translation of processes

For process, the translation is as follows:

- $\lfloor$ (Γ, Nil) $\rfloor$ = Nil
- $\lfloor$ (Γ, var v) $\rfloor$ = $\begin{cases} X & \text{if} \quad X \text{ is the } v^{th} \textit{ element in } \Gamma; \\ v & \text{otherwise.} \end{cases}$
- $\lfloor$ (Γ, dot a p)$\rfloor$ = $\lfloor a \rfloor$. $\lfloor$ (Γ, p) $\rfloor$
- $\lfloor$ (Γ, choice p1 p2)$\rfloor$ = $\lfloor$ (Γ, p1) $\rfloor$ + $\lfloor$ (Γ, p2)$\rfloor$
- $\lfloor$ (Γ, par p1 p2) $\rfloor$ = $\lfloor$ (Γ, p1) $\rfloor$ | $\lfloor$ (Γ, p2) $\rfloor$
- $\lfloor$ (Γ, hide p L)$\rfloor$ = $\lfloor$ (Γ, p) $\rfloor$ \ $\lfloor L \rfloor$
- $\lfloor$ (Γ, ren p f) $\rfloor$ = $\lfloor$ (Γ, p)$\rfloor$ [$\lfloor$ f $\rfloor$]
- $\lfloor$ (Γ, rec p) $\rfloor$ = rec X. $\lfloor$ (X:Γ, p) $\rfloor$
  where X is a fresh symbol which is different from the elements of Γ.

8.3.2 The translation of some forms of propositions

For some forms of propositions, we recognize these forms and translate them back to the user level. Some of the forms and the corresponding translations are as follows:

- $\lfloor$ Prf (ap_ State Prop F P) $\rfloor$ = $\lfloor$ ([-],P) $\rfloor$ $\vdash$ $\lfloor$ F $\rfloor$
- $\lfloor$ Prf (and p1 p2) $\rfloor$ = $\lfloor$ p1 $\rfloor$ & $\lfloor$ p2 $\rfloor$
- $\lfloor$ Prf (MTRANS m p1 p2)$\rfloor$ = $\lfloor$ p1 $\rfloor$ - $\lfloor$ m $\rfloor$ $\rightarrow$ $\lfloor$ p2 $\rfloor$
- $\lfloor$ Prf (TRANS a p1 p2)$\rfloor$ = $\lfloor$ p1 $\rfloor$ - $\lfloor$ a $\rfloor$ $\rightarrow$ $\lfloor$ p2 $\rfloor$

In my mind, the rules, definitions and declarations need not be translated back. Because users already know them and type them by themselves.

8.4 The properties of the translations

Lemma 8.4.1 *The translations* $[\![-]\!]$ *and* $\lfloor$ - $\rfloor$ *satisfy the following property: for a user level action* a, $\lfloor [\![a]\!] \rfloor \equiv a$.

Proof Using the definitions of the translations, the proof proceeds by case analysis on the structure of action.

Q.E.D.

Lemma 8.4.2 *The translations* $[\![-]\!]$ *and* $\lfloor$ - $\rfloor$ *satisfy the following property: for a user level process* p *and context* Γ, $\lfloor\ [\![(\Gamma, p)]\!]\ \rfloor \equiv (\Gamma, p)$.

Proof

Using the definitions of the translations, the proof proceeds by induction on the structure of process.

Q.E.D.

Lemma 8.4.3 *The translations* $[\![-]\!]$ *and* $\lfloor$ - $\rfloor$ *satisfy the following property: for a user level* μ*-calculus formula* F, $\lfloor\ [\![F]\!]\ \rfloor \equiv F$.

Proof

Using the definition of the translations, the proof proceeds by induction on the structure of μ-calculus formula.

Q.E.D.

Lemma 8.4.4 *The translations* $[\![-]\!]$ *and* $\lfloor$ - $\rfloor$ *satisfy the following property: for a user level proposition* P, $\lfloor\ [\![P]\!]\ \rfloor \equiv P$.

Proof

Using the definition of the translations, the proof proceeds by induction on the structure of proposition.

Q.E.D.

8.5 The proof of the adequacy property

Theorem 8.5.1 *Our translations* $[\![-]\!]$ *and* $\lfloor$ - $\rfloor$ *are adequate, i.e. they satisfy the following* ***adequacy condition for inference****: If* G *is a goal on the user level under assumptions* $A_1, A_2, ...A_m$*; we use a command* δ *on this goal and get* $G_1, G_2, ..., G_n$ *as the subgoals on the user level. Then* $[\![G]\!]$ *is a goal on the LF level under assumptions* $[\![A_1]\!], [\![A_2]\!], ...[\![A_m]\!]$ *, and after apply* $[\![\delta]\!]$*, we get* G'_1, G'_2,..., G'_n *as the subgoals, and* $\lfloor\ G'_i\ \rfloor \equiv G_i$, $[\![G_i]\!] \equiv G'_i$ *for* i = *1,2, ...n. Where* $\equiv$ *express the syntactical equal under* α *conversion.*

Proof We prove this theorem by case analysis on the commands:

- For the command "Fst h", we have $[\![Fst\ h]\!]$ = Refine App ? ? p_fst h .
  The h must be a name for a hypothesis of the form "p & q" and the goal on the user level is p. After applying this command, the goal is solved (i.e. there is no subgoal). On the *LF* level, the goal is $[\![p]\!]$ and h is a name for a hypothesis of the form $[\![p\ \&\ q]\!]$, After applying "Refine App ? ? p_fst h", the goal is solved; so the theorem holds for this case.

  A similar proof can be used to the command "Snd h".

- For the command "RDia s' " (i.e. Rule Dia s'), we have
  $[\![RDia\ s']\!]$ = Refine App ? ? (lemma_dia_ccs ? ? ? s' ?) ?.
  The goal on the user level must be in the form of $s \vdash < K > F$. After applying this command, the subgoals are: $s' \vdash F$ and $s \xrightarrow{K} s'$.
  The goal on the LF level is: Prf (ap_ Process Prop ($[\![< K > F]\!]$) $[\![s]\!]$);
  after applying the corresponding command:
  "Refine App ? ? (lemma_dia_ccs ? ? ? s' ?) ?",
  the subgoals are :
  (Prf (ap_ State Prop $[\![F]\!]$ $[\![s']\!]$)) and (Prf (MTRANS $[\![K]\!]$ $[\![s]\!]$ $[\![s']\!]$)),
  according to the definition of the translation $\lfloor$ - $\rfloor$, we can see that:
  $\lfloor$ (Prf (ap_ State Prop $[\![F]\!]$ $[\![s']\!]$)) $\rfloor \equiv s' \vdash F$,
  and $\lfloor$ (Prf (MTRANS $[\![K]\!]$ $[\![s]\!]$ $[\![s']\!]$)) $\rfloor \equiv s \xrightarrow{K} s'$.
  By the definition of the translation $[\![-]\!]$, we know that:
  $[\![s' \vdash F]\!] \equiv$ Prf (ap_ State Prop $[\![F]\!]$ $[\![s']\!]$)
  and $[\![s \xrightarrow{K} s']\!] \equiv$ Prf (MTRANS $[\![K]\!]$ $[\![s]\!]$ $[\![s']\!]$)
  So the theorem holds for this case.

- Consider the command "RRec" (i.e. Rule Rec), we have $[\![RRec]\!]$ = Refine Rec, the goal on the user level must be in the form of $(rec\ X.p1) \xrightarrow{a} p2$.
  After applying this command, the subgoal is: $p1[(rec\ X.p1)/X] \xrightarrow{a} p2$.
  The goal on the LF level is : $(Prf(TRANS\ [\![a]\!]\ [\![rec\ X.p1]\!]\ [\![p2]\!]))$;
  after applying the corresponding command "Refine Rec", the subgoal is:
  $(Prf(TRANS\ [\![a]\!](subst\ [\![p1]\!]\ one\ (rec\ [\![p1]\!]))\ [\![p2]\!]))$.
  By our definition of $\lfloor$ - $\rfloor$, we can see that:
  $\lfloor$ (Prf (TRANS $[\![$a$]\!]$ (subst $[\![$p1$]\!]$ one $(rec\ [\![p1]\!]))$ $[\![p2]\!]$) $\rfloor$
  $\equiv \lfloor$ ($\lfloor$ subst $\rfloor$ $[\![p1]\!]$ one $(rec\ [\![p1]\!]))$ $\rfloor \xrightarrow{a} p2$
  $\equiv p1[(rec\ X.p1)/X] \xrightarrow{a} p2$.
  By our definition of $[\![-]\!]$, we can see that:
  $[\![p1[(rec\ X.p1)/X] \xrightarrow{a} p2]\!] = (Prf(TRANS\ [\![a]\!](subst\ [\![p1]\!]\ one\ (rec\ [\![p1]\!]))\ [\![p2]\!])$.
  So the theorem holds for this case.

  Remark: If the above $(rec\ X.p1)$ are predefined using a name, then after using this command, the predefined name for this will be changed to $(rec\ X.p1)$.

- Consider the command "RDot" (i.e. Rule Dot), we have $[\![RDot]\!]$ = Refine Dot, the goal on the user level must be in the form of $a.p \xrightarrow{a} p$. After applying this command, the goal is solved. The goal on the LF level is : $(Prf(TRANS\ [\![a]\!]\ [\![a.p]\!]\ [\![p]\!]))$;
  after applying the corresponding command "Refine Dot", the goal is solved. So the theorem holds for this case.

- For the command "RTrue" (i.e. Rule True), we have $[\![RTrue]\!]$ = Refine lemma_true, the goal on the user level must be in the form of $s \vdash tt$. After applying this command,

the goal is solved. The goal on the LF level is: $(Prf(ap\_\ State\ Prop\ Mutt\ s))$;
after applying the corresponding command "Refine lemma_true", the goal is solved.
So the theorem holds for this case.

- Consider the command "RPair" (i.e. Rule Pair), we have
$[\![RPair]\!]$ = Refine App ? ? (App ? ? p_pair ?) ?, the goal on the user level must be in the form of $p1$ & $p2$. After applying this command, the subgoals are $p1$ and $p2$. The goal on the LF level is:
(Prf (and $[\![p1]\!]'$ $[\![p2]\!]'$)). where $[\![p1]\!] \equiv$ (Prf ($[\![p1]\!]'$)) and $[\![p2]\!] \equiv$ (Prf ($[\![p2]\!]'$)) ;
after applying the corresponding command "Refine App ? ? (App ? ? p_pair ?) ?", the subgoals are (Prf ($[\![p1]\!]'$)) and (Prf ($[\![p2]\!]'$)). By the definition of $[\![-]\!]$, $\lfloor$ - $\rfloor$ and lemma 8.4.4, the theorem holds for this case.

- Consider the command "RChol" (i.e. Rule Chol), we have $[\![RChol]\!]$ = Refine Chol, the goal on the user level must be in the form of: $p1 + p2 \xrightarrow{a} q$.
After applying this command, the subgoal is: $p1 \xrightarrow{a} q$.
The goal on the LF level is: (Prf (TRANS $[\![a]\!]$ $[\![p1 + p2]\!]$ $[\![q]\!]$));
after applying the corresponding command "Refine Chol", the subgoal is:
(Prf (TRANS $[\![a]\!]$ $[\![p1]\!]$ $[\![q]\!]$)). By our definition of $\lfloor$ - $\rfloor$, we can see that:
$\lfloor$ (Prf (TRANS $[\![a]\!]$ $[\![p1]\!]$ $[\![q]\!]$)) $\rfloor \equiv p1 \xrightarrow{a} q$.
By our definition of $[\![-]\!]$, we can see that:
$[\![p1 \xrightarrow{a} q]\!] \equiv$ (Prf (TRANS $[\![a]\!]$ $[\![p1]\!]$ $[\![q]\!]$)). So the theorem holds for this case.

  We can use the similar proof to the commands RChor, RParl, RParr, RHide, RHidet and RRen.

- Consider the command RTau1 n (i.e. Rule Tau1 n), we have
$[\![RTau1\ n]\!]$ = Refine Tau1 n, the goal on the user level must be in the form of $(p1|p2) \xrightarrow{tau} (q1|q2)$. After applying this command, the subgoals are:
$p1 \xrightarrow{n} q1$ and $p2 \xrightarrow{\tilde{n}} q2$.
The goal on the LF level is :
(Prf (TRANS $[\![tau]\!]$ $[\![p1|p2]\!]$ $[\![q1|q2]\!]$));
after applying the corresponding command "Refine Tau1 n", the subgoals are:
(Prf (TRANS $[\![n]\!]$ $[\![p1]\!]$ $[\![q1]\!]$)) and (Prf (TRANS $[\![\tilde{n}]\!]$ $[\![p2]\!]$ $[\![q2]\!]$)).
By our definition of $\lfloor$ - $\rfloor$, we can see that :
$\lfloor$ (Prf (TRANS $[\![n]\!]$ $[\![p1]\!]$ $[\![q1]\!]$)) $\rfloor \equiv p1 \xrightarrow{n} q1$
and $\lfloor$ (Prf (TRANS $[\![\tilde{n}]\!]$ $[\![p2]\!]$ $[\![q2]\!]$)) $\rfloor \equiv p2 \xrightarrow{\tilde{n}} q2$. By our definition of $[\![-]\!]$, we can see that: $[\![p1 \xrightarrow{n} q1]\!] \equiv$ (Prf (TRANS $[\![n]\!]$ $[\![p1]\!]$ $[\![q1]\!]$))
and $[\![p2 \xrightarrow{\tilde{n}} q2]\!] \equiv$ (Prf (TRANS $[\![\tilde{n}]\!]$ $[\![p2]\!]$ $[\![q2]\!]$)).

  So the theorem holds for this case.

  We can use the similar proof to the commands RTau2 n.

- Consider the command "RSinglein" (i.e. Rule Singlein), we have:
  $[\![RSinglein]\!]$ = Refine App ? ? Eq_refl ?,
  the goal on the user level must be in the form of "$s \in \{s\}$". After applying this command, the goal is solved. The goal on the LF level is :
  (Prf (Eq Bool (Modal_check s (Modal (cons Label s (nil Label)))) true)),
  after applying the corresponding command "Refine App ? ? Eq_refl ?", the goal is solved. So the theorem holds for this case.

- For the command "RExintro a", we have:
  $[\![RExintro\ a]\!]$ = Refine lemma_Exintro ? ? ? a,
  the goal on the user level must be in the form of $p1 \xrightarrow{m} p2$. After applying this command, the subgoals are: "$a \in m$" and "$p1 \xrightarrow{a} p2$".
  The goal on the LF level is : (Prf (MTRANS $[\![m]\!]$ $[\![p1]\!]$ $[\![p2]\!]$)),
  after applying the corresponding command : "Refine lemma_Exintro ? ? ? a",
  the subgoals are: (Prf (Eq Bool (Modal_check $[\![a]\!]$ $[\![m]\!]$) true))
  and (Prf (TRANS $[\![a]\!]$ $[\![p1]\!]$ $[\![p2]\!]$)).
  By the definitions of $\lfloor$ - $\rfloor$ and $[\![-]\!]$, we can see the theorem holds for this case.

- Consider the command "Rend" (i.e. Rule end), because this rule does not be used to any goals, so the theorem holds for this case.

- Consider the command "RHyp s" (i.e. Rule Hyp s), we have $[\![RHyp\ s]\!]$ = Refine s. The form of the goal on the user level depends on the hypothesis named by s. After using this command, the goal is solved. The form of the goal on LF level depends on the corresponding hypothesis of the same name s. After applying the corresponding command "Refine s", the goal is solved. So the theorem holds for this case.

- Consider the command "RNuunfold" (i.e. Rule Nuunfold), we have:
  $[\![RNuunfold]\!]$ = Refine App ? ? Nu_unfold ?.
  The goal on the user level must be in a form of "s $\vdash$ Nu x t f". After using this command, the subgoal is in a form of "s $\vdash$ f[(Nu x (t$\cup${s}) f)/x]". The form of the goal on LF level is like (Prf (ap_ State Prop $[\![Nu\ x\ t\ f]\!]$ $[\![s]\!]$)).
  After using the corresponding command "Refine App ? ? Nu_unfold ?", the subgoal is in a form of :
  "(Prf (ap_ State Prop (ap_ Form Form (La_ Form Form ([Z:El Form] $[\![f]\!]$) $[\![$(Nu x (t$\cup${s}) f $]\!]$))) $[\![s]\!]$)) ".
  By the definitions of $\lfloor$ - $\rfloor$ and $[\![-]\!]$, we can see the theorem holds for this case.

- Consider the command "RNubase" (i.e. Rule Nubase), we have:
  $[\![RNubase]\!]$ = Refine App ? ? Nu_base ?.
  The goal on the user level must be in a form of "s $\vdash$ Nu x t f". After using this command, the subgoal is in a form of "s $\in$ t". The form of the goal on LF level is like:

(Prf (ap_ State Prop $[\![Nu\ x\ t\ f]\!]$ $[\![s]\!]$)).
After using the corresponding command "Refine App ? ? Nu_base ?", the subgoal is in a form of (Prf (ap_ State Prop $[\![t]\!]$ $[\![s]\!]$)). By the definitions of $\lfloor$ - $\rfloor$ and $[\![-]\!]$, we can see the theorem holds for this case.

- For the command "Rinr", we have $[\![Rinr]\!]$ = Refine App ? ? p_inr ?. The goal on the user level must be in a form of "s ∈ U + V". After using this command, the subgoal is in a form of "s ∈ V". The form of the goal on LF level is like:
(Prf (ap_ State Prop $[\![U\ +\ V]\!]$ $[\![s]\!]$)).
After using the corresponding command "Refine App ? ? p_inr ?", the subgoal is in a form of : (Prf (ap_ State Prop $[\![V]\!]$ $[\![s]\!]$)).
By the definition of $\lfloor$ - $\rfloor$ and $[\![-]\!]$, we can see the theorem holds for this case.

  We can use the similar proof for the command "Rinl".

- For the command "Rbox s' h" (i.e. Rule box s' h), there is no corresponding lemma on the LF level, but a tactical which includes a relevant lemma.
$[\![Rbox\ s'\ h]\!]$ = Refine lemma_box_ccs' Then_T (Intros s') Then_T (Intros h).
The goal on the user level must be in a form of "s ⊢ [K] F". After applying this command, the subgoal is in a form of "s' ⊢ F" with a hypothesis "$s \xrightarrow{K} s'$ " named by h. The form of the goal on LF level is like (Prf (ap_ State Prop $[\![$[K] F$]\!]$ $[\![$s$]\!]$)).
After applying the corresponding tactical, the subgoal is in a form of:
(Prf (ap_ State Prop $[\![$F$]\!]$ $[\![$s'$]\!]$)) with a hypothesis (Prf (MTRANS $[\![$K$]\!]$ $[\![$s$]\!]$ $[\![$s'$]\!]$)) named by h.
By the definitions of $\lfloor$ - $\rfloor$ and $[\![-]\!]$, we can see the theorem holds for this case.

- For the command "Rinverdotl a1 a2" (i.e. Rule inverdot1 a1 a2), we have:
$[\![Rinverdotl\ a1\ a2]\!]$ = Refine lemma_dot_eq_p' a2 a1.
The goal on the user level must be in a form of "p1 = p2". After applying this command, the subgoal is in a form of "$a1.p1 \xrightarrow{a2} p2$". The form of the goal on LF level is like (Prf (Eq Process p1 p2)). After applying the corresponding command, the subgoal is in a form of (Prf (TRANS $[\![$a2$]\!]$ $[\![$a1.p1$]\!]$ $[\![$p2$]\!]$)). By the definitions of $\lfloor$ - $\rfloor$ and $[\![-]\!]$, we can see the theorem holds for this case.

  The similar proof can be used to the commands "Rinverrec a p p1 p2", "Rinverchoi a p1 p2 p " and "Rinverpar a p1 p2 p".

- For the command "Rvpair", we have [Rvpair]= Refine lemma_vpair. The goal on the user level must be in a form of "s ⊢ F1 & F2". After applying this command, the subgoals are "s ⊢ F1" and "s ⊢ F2". The form of the goal on LF level is like
(Prf (ap_ State Prop $[\![F1\&F2]\!]$ $[\![s]\!]$)).
After applying the corresponding command, the subgoals are
(Prf (ap_ State Prop $[\![F1]\!]$ $[\![s]\!]$)) and (Prf (ap_ State Prop $[\![F2]\!]$ $[\![s]\!]$)).
By the definition of $\lfloor$ - $\rfloor$ and $[\![-]\!]$, we can see the theorem holds for this case.

- For the command "Rhypchange h1 h2 h3", there is no corresponding lemma on the *LF* level, but a tactical:
  $[\![Rhypchange\ h1\ h2\ h3]\!]$ = Refine App ? ? (App ? ? h1 ?) Then_T (Refine LL Then_T (Intros h2 Then_T (Refine LL Then_T (Intros h3)))).
  This command just changes the hypothesis and does not affect the goal. So the theorem holds for this case.

- For commands such as "RModule n" and "RImport n", because there is no goal when these commands are used, so the theorem holds for these cases.

- For the command "Req", we have $[\![Req]\!]$ = Refine App ? ? Eq_refl ?. The goal on the user level must be in a form of $a1 = a2$. After applying this command, the goal is solved. The form of the goal on *LF* level is like (Prf (Eq A a1 a2)). After applying the corresponding command "Refine App ? ? Eq_refl ?", the goal is solved. So the theorem holds for this case.

- For the command "RIndn a p h", we have:
  $[\![RIndn\ a\ p\ h]\!]$ = Refine E_Nat ([n1:Nat]Prf(TRANS $[\![a]\!]$ ($[\![p]\!]$ n1) ($[\![p]\!]$ (succ n1)))) Then_T (Intros n h).
  The goal on the user level must be in a form of "Allnat a p". After applying this command, the subgoal is like:
  "p (n + 1) $\xrightarrow{a}$ p ((n + 1) + 1)" with a hypothesis "p n $\xrightarrow{a}$ p (n + 1)" named by h.
  The goal on the *LF* level is like
  (n:El Nat)El (Prf (TRANS $[\![a]\!]$ ($[\![p]\!]$ n) ($[\![p]\!]$ (succ n)))).
  After applying the corresponding command, the subgoal is
  (Prf (TRANS $[\![a]\!]$ ($[\![p]\!]$ (succ n)) ($[\![p]\!]$ (succ (succ n)))))
  with a hypothesis "(Prf (TRANS $[\![a]\!]$ ($[\![p]\!]$ n) ($[\![p]\!]$ (succ n))))".
  By the definitions of $\lfloor$ - $\rfloor$ and $[\![-]\!]$, we can see the theorem holds for this case.

Q.E.D.

8.6 Discussion

From the above description we can see that how to guarantee the forward and backward translation successfully is one of the main problem which should be solved in the approach. In this chapter we give a revised model for this problem and a concrete proof of an important theorem. We have proven the relevant lemmas and properties. The study of this chapter shows that the translations and the relevant methods and technologies are suitable to our purpose.

Chapter 9

Conclusion and Future Work

If you wish to succeed, you should use persistence as your good friend, experience as your reference, prudence as your brother and hope as your sentry.

— THOMAS EDISON, AMERICAN INVENTOR

This thesis has presented an approach to domain specific reasoning. The case studies in above chapters demonstrate the success of this approach. More specifically, the case studies demonstrate Plastic's capabilities for reasoning with many different domains. Further, they show how the capabilities can be used by presenting user friendly interfaces. The work in this thesis also lays the ground for further research into domain specific reasoning based on *LF* and its implementation.

9.1 Stocktaking

This thesis has concentrated on showing how the approach can provide a good way for domain specific reasoning. We have seen how the approach works. Our approach has the following features:

- User convenience: The approach is different from the pure proof assistant approach, the user can use an interface in their familiar way to do the verification, i.e. we provide a bridge between the domain and the underlying formalization.

- Generality: Compared to automatic model checkers, this approach can handle verifications of more complex properties, since it does not have the limitations of model checkers to finite state problems. Furthermore, by providing a simple reasoning framework where problems can be correctly decomposed and model checking used as a decision procedure on feasible sub-problems, the approach could support wider use of model checking technology.

- More convincing: Because the approach is based on a type theoretic proof assistant, a proof in our approach is a constructive proof, so it is more convincing than one arising from a model checker or from a directly programmed system. The trusted code-base is smaller (i.e. the type-checker), and the formalization will have been developed more rigorously. Plus, it is simple to obtain independent and automatic verification of results.

- Structural complexity: However, the structure of this approach is more complicated than most other approaches. But this should be balanced against the positive features, and we believe the balance is in our favor. For this kind of system, the overheads in a multi-layer approach are relatively small, so 'efficiency' there is not a concern. One may also view LF as a better (i.e. more precise or articulate) programming language for implementing such formal systems, so this use of LF is a strong advantage.

- Scalability: The examples of chapter 5 were deliberately kept simple, in order to make certain points about feasibility and translation, but the question remains of how this approach will work with larger and more realistic examples, such as security protocols. Given the generality of the underlying formalization and the result of chapter 5 and chapter 6, it appears that the only limit on trying larger examples is the performance of the underlying tools. Type theory proof assistants have been used for significant proofs, including that of the Fundamental Theorem of Algebra [Barendregt, 2005], and research is ongoing to improve the performance of the technology, so we do not envisage problems.

Note indeed that this approach is more complicated than most of other approaches and we are in a preliminary step for this approach.

Another contribution of this thesis is that we do all proofs in our case studies mechanically. We successfully generated Plastic proofs for the following theories and components:

- Set theory and fix-point theory.
- μ-calculus related theory.
- CCS related theory.
- LAZY-PCF+SHAR related theory

9.2 Evaluation

- **Evaluation related to aim 1:** In Chapter 2 we declare that the Aim 1 of the thesis is to give an analysis of requirements of domain-specific reasoning and give some criteria. Through case studies, we give a more detailed analysis of the characteristics of

domain-specific reasoning. We find that domain-specific notations and higher-level abstractions can be used directly by domain users and their corresponding translations to LF level can be done automatically. The design reuse is highly appreciated for different domains. From the case studies we also learn a lot of knowledge about how to produce domain-specific computer assisted reasoning tools. For the new approach we design the relevant architecture and construct the corresponding components, study the feasibility of it through several case studies. The relevant methodology and process are also presented and investigated. The case studies concretize the work of formalization, parser, communication protocols, translation between different levels and interfaces. They provide a good support to the suitability of the approach.

- **Evaluation related to aim 2:** The aim 2 we indicated in Chapter 2 is the analysis of LF and Plastic as a basis to support domain-specific reasoning. Our case studies suggest that LF is a suitable framework as an underlying basis for domain-specific reasoning and Plastic is a suitable system to support underlying reasoning. As indicated in the above section the theoretical and practical benefits and defects of using LF and Plastic instead of other proof assistants are also investigated in this thesis. The case studies in Chapter 5 and Chapter 6 show that LF and Plastic are powerful enough to support big applications.

- **Evaluation related to aim 3:** Many of the theoretical aspects of the approach such as the theorem of the adequacy property in Chapter 8 are proved.

- **Evaluation related to the principles in Chapter 7:**

  For Jakob Nielsen's ten general principles, to judge which points are reached by our interface, we use the following criteria:

  1. Do we follow the principles when we design the interface?
  2. Does the interface satisfy the main points of the principles?
  3. Does the interface show the main points naturally?

  According to the above criteria, our implementation of the interface accomplished most of the points of the ten principles. But we need to do more in the following aspects:

  - User control and freedom
  - Flexibility and efficiency of usage
  - Help and documentation

9.3 Future research

The work in this thesis creates a number of opportunities for future work. The most interesting are the following:

- Studying more complex examples and looking at more issues of supporting GUIs in which user commands can be more diverse, or even more powerful, such as proof-by-pointing [Bertot et al., 1997b].
- Studying the application of coercive subtyping in domain specific reasoning.
- Trying to combine model checker with our approach smoothly.
- Other issues to study include how to allow users to develop their own lemmas (i.e. to extend the formalization rather than just to work inside it), and how to improve the understandability of proofs, e.g. representing traces of computation and using Natural Language to explain proof steps.
- As the size of examples increases, we may also need to study techniques to help users organize their proofs and developments, such as allowing multiple contexts for reasoning.
- Do more case studies in dissimilar domains.

Appendix A

The proofs of the Subject Reduction theorem

Subject Reduction theorem is a very important theorem in the case study of Chapter 6. The successful proof of it in Plastic shows the power of Plastic. The proof of Subject Reduction theorem is divided to two individual proofs of the subject reduction theorem (Theorem 1.) and normal form characterization theorem (Theorem 2.). Lemma 1 and Lemma 2 are specific lemmas for supporting the proofs of Theorem 1 and Theorem 2. The following are proofs which are in our module Subjrnf. Note that the symbol ? in the following proof is a place holder(i.e. an unnamed metavariable in Plastic).

```
> module Subjrnf where;
(*************************************************************)
(*                                                           *)
(* Subjrfn.lf             This file contains the main theorem,*)
(* subjr_NF, which combines the subject reduction (Theorem 1.)*)
(* and normal form characterization theorems (Theorem 2.)     *)
(* The combination is necessary in order for the induction to *)
(* go through. This proof is followed by individual proofs of *)
(* the subject reduction theorem and normal form              *)
(* characterization theorem.                                  *)
(*************************************************************)
> import ApTypes;
> import Envprops;
> import NFprops;
> import Valid;
```

```
(*************************************************************)
(* Subject Reduction + NF                                    *)
(* <<e,A>> -> <<e',A'>>---> Valid(A)--->Domt(A)|- e:t --->    *)
(* Valid(A') /\ Domt(A')|- e':t /\ (NF e')                   *)
(*                                                           *)
(*************************************************************)


(*************************************************************)
(* lemma 1.                                                  *)
(*                                                           *)
(*************************************************************)
> Claim  subjr_NFPi: (c,c': El Config)(p1: El (Prf (OSred c c')))
>        El (Pi_ (Prf (Valid_env (cfgenv c)))
>        (Pi[t:Ty] (Pi_ (Prf (TC (OS_Dom_ty (cfgenv c)) (cfgexp c) t))
>        (Prf (and (and (Valid_env (cfgenv c')) (TC (OS_Dom_ty (cfgenv c'))
>        (cfgexp c') t)) (NF (cfgexp c')))))));
> Intros c c' p1;
> show E_OSred;
> Refine E_OSred ([c,c': El Config](Pi_ (Prf (Valid_env (cfgenv c)))
>        (Pi[t:Ty] (Pi_ (Prf (TC (OS_Dom_ty (cfgenv c)) (cfgexp c) t))
>        (Prf (and (and (Valid_env (cfgenv c')) (TC (OS_Dom_ty (cfgenv c'))
>        (cfgexp c') t))(NF (cfgexp c'))))))));
> Refine p1;
> Intros A A1 e e1 en e2 s t x pr1 pr2 pr3 pr4;
> Refine La_;
> Intros pr5;
> Refine La;
> Intros x1;
> Refine La_;
> Intros pr6;
> Refine App ? ? (App ? ? (p_pair ? ?) ?) ?;
> Claim NFen:  El (Prf (NF (cfgexp (cfg en A1))));
> Refine App ? ? (p_snd ? ? ) (ap_ ? ? (ap ? ? (ap_ ? ? pr4 ?) x1) ? );
> Refine App ? ? (p_snd ? ? ) (inv_TC_clos (OS_Dom_ty A) t x1 e e1 x pr6);
> Refine Valid_cons x t e1 A ? ?;
> Refine pr5;
> Refine App ? ? (p_fst ? ? ) (inv_TC_clos (OS_Dom_ty A) t x1 e e1 x pr6);
> ReturnAll;
```

```
> Refine NFen;
> ReturnAll;
> Refine  App ? ? (App ? ? (p_pair ? ?) ?) ?;
> Refine p_Eq_subst Ty_env (append VT (nil VT) (OS_Dom_ty A1)) (OS_Dom_ty A1)
>        ([H: Ty_env](TC H en  x1));
> 2 Refine App ? ? (Eq_refl ?) ?;
> Refine TEp_inv_nfvExt  (OS_Dom_ty (OScons x t e2 A1)) en x1 t x
>         (App ? ? (p_snd ? ? ) (App ? ? (p_fst ? ?) (ap_ ? ? (ap ? ?
>          (ap_ ? ? pr4 ?) x1) ? ))) ?  (nil VT)  (OS_Dom_ty A1) ? ;
> Refine App ? ? (Eq_refl ?) ?;
> Refine App ? ? (App ? ? (App ? ? pr3 ?) ?) ?;
> 2 Refine LL;
> 2 Intros PH1;
> Refine Snoe_notFVe en ? x;
> Refine NFenat_Snoe en ? (cons VT (pair Vari Ty x t) (OS_Dom_ty A))  ?;
> Refine p_Eq_subst Ty s nat_Ty ([nat_Ty: Ty](TC (cons VT (pair Vari Ty x t)
>        (OS_Dom_ty A)) en nat_Ty));
> Refine pr2;
> Refine PH1;
> Refine App ? ? (p_snd ? ? ) (ap_ ? ? (ap ? ? (ap_ ? ? pr4 ?) x1) ? );
> Refine App ? ? (p_snd ? ? ) (inv_TC_clos (OS_Dom_ty A) t x1 e e1 x pr6);
> Refine Valid_cons x t e1 A ? ?;
> Refine pr5;
> Refine App ? ? (p_fst ? ? ) (inv_TC_clos (OS_Dom_ty A) t x1 e e1 x pr6);
> ReturnAll;
> Refine LL;
> Intros PH1;
> Claim eneqtttorfff: Prf (or (Eq Tm en ttt) (Eq Tm en fff));
> show NFebool_TF;
> Refine ap_ ? ? (ap ? ? ( NFebool_TF en ? ) (cons VT (pair Vari Ty x t)
>        (OS_Dom_ty A))) ?;
> Refine p_Eq_subst Ty s bool_Ty ([bool_Ty: Ty](TC (cons VT (pair Vari Ty x t)
>        (OS_Dom_ty A)) en bool_Ty));
> Refine pr2;
> Refine PH1;
> Refine App ? ? (p_snd ? ? ) (ap_ ? ? (ap ? ? (ap_ ? ? pr4 ?) x1) ? );
> Refine App ? ? (p_snd ? ? ) (inv_TC_clos (OS_Dom_ty A) t x1 e e1 x pr6);
> Refine Valid_cons x t e1 A ? ?;
> Refine pr5;
> Refine App ? ? (p_fst ? ? ) (inv_TC_clos (OS_Dom_ty A) t x1 e e1 x pr6);
```

```
> ReturnAll;
> Refine App ? ? (App ? ? (App ? ? eneqtttorfff ?) ?) ?;
> Refine LL;
> Intros PH2;
> Refine p_Eq_subst Tm fff en  ([en: Tm](not (FV x en)));
> Refine inv_FV_fff x;
> Refine App ? ? (App ? ? (App ? ? (Eq_sym ?) ?) ?) PH2;
> ReturnAll;
> Refine LL;
> Intros PH2;
> Refine p_Eq_subst Tm ttt en  ([en: Tm](not (FV x en)));
> Refine inv_FV_ttt x;
> Refine App ? ? (App ? ? (App ? ? (Eq_sym ?) ?) ?) PH2;
> ReturnAll;
> Refine App ? ? (p_snd ? ? ) (inv_TC_clos (OS_Dom_ty A) t x1 e e1 x pr6);
> Refine Valid_cons x t e1 A ? pr5;
> Refine App ? ? (p_fst ? ? ) (inv_TC_clos (OS_Dom_ty A) t x1 e e1 x pr6);
> Claim validconsa1: Prf (Valid_env (cfgenv (cfg en (OScons x t e2 A1))));
> Refine App ? ? (p_fst ? ? ) (App ? ? (p_fst ? ?)
>        (ap_ ? ? (ap ? ? (ap_ ? ? pr4 ?) x1) ? ));
> Refine App ? ? (p_snd ? ? ) (inv_TC_clos (OS_Dom_ty A) t x1 e e1 x pr6);
> Refine Valid_cons x t e1 A ? ?;
> Refine pr5;
> Refine App ? ? (p_fst ? ? ) (inv_TC_clos (OS_Dom_ty A) t x1 e e1 x pr6);
> ReturnAll;
> Refine  App ? ? (p_fst ? ? ) (inv_valid_cons x t e2 A1 validconsa1);
> ReturnAll;

(* CL *)
> Intros A A1 e e1 en e2 s t x pr1 pr2 pr3 pr4;
> Refine La_;
> Intros pr5;
> Refine La;
> Intros x1;
> Refine La_;
> Intros pr6;
> Refine App ? ? (App ? ? (p_pair ? ?) ?) ?;
> Refine NF_F  (clos en x t e2) ?;
> Refine F_clos en e2 x t ?;
> Refine NFe_Fe en ?  (cons VT (pair Vari Ty x t) (OS_Dom_ty A)) s ? ?;
```

```
> Refine pr3;
> Refine pr2;
> Refine App ? ? (p_snd ? ? ) (ap_ ? ? (ap ? ? (ap_ ? ? pr4 ?) x1) ? );
> Refine App ? ? (p_snd ? ? ) (inv_TC_clos (OS_Dom_ty A) t x1 e e1 x pr6);
> Refine Valid_cons x t e1 A ? ?;
> Refine pr5;
> Refine App ? ? (p_fst ? ? ) (inv_TC_clos (OS_Dom_ty A) t x1 e e1 x pr6);
> ReturnAll;
> Refine  App ? ? (App ? ? (p_pair ? ?) ?) ?;
> Refine TC_clos  (OS_Dom_ty  A1) x  en e2 t x1 ? ?;
> Refine App ? ? (p_snd ? ? ) (App ? ? (p_fst ? ?)
>        (ap_ ? ? (ap ? ? (ap_ ? ? pr4 ?) x1) ? ));
> Refine App ? ? (p_snd ? ? ) (inv_TC_clos (OS_Dom_ty A) t x1 e e1 x pr6);
> Refine Valid_cons x t e1 A ? ?;
> Refine pr5;
> Refine App ? ? (p_fst ? ? ) (inv_TC_clos (OS_Dom_ty A) t x1 e e1 x pr6);
> Refine App ? ? (p_snd ? ? ) (inv_valid_cons x t e2 A1 ?);
> Refine App ? ? (p_fst ? ? ) (App ? ? (p_fst ? ?)
>        (ap_ ? ? (ap ? ? (ap_ ? ? pr4 ?) x1) ? ));
> Refine App ? ? (p_snd ? ? ) (inv_TC_clos (OS_Dom_ty A) t x1 e e1 x pr6);
> Refine  Valid_cons x t e1 A ? ?;
> Refine pr5;
> Refine App ? ? (p_fst ? ? ) (inv_TC_clos (OS_Dom_ty A) t x1 e e1 x pr6);
> Refine App ? ? (p_fst ? ? ) (inv_valid_cons x t e2 A1 ?);
> Refine App ? ? (p_fst ? ? ) (App ? ? (p_fst ? ?)
>        (ap_ ? ? (ap ? ? (ap_ ? ? pr4 ?) x1) ? ));
> Refine App ? ? (p_snd ? ? ) (inv_TC_clos (OS_Dom_ty A) t x1 e e1 x pr6);
> Refine  Valid_cons x t e1 A ? ?;
> Refine pr5;
> Refine App ? ? (p_fst ? ? ) (inv_TC_clos (OS_Dom_ty A) t x1 e e1 x pr6);
> ReturnAll;

(* fix *)
> Intros A A1 e e1 en t x  nx pr1 pr2 pr3 pr4;
> Refine La_;
> Intros pr5;
> Refine La;
> Intros x1;
> Refine La_;
> Intros pr6;
```

```
> Refine App ? ? (App ? ? (p_pair ? ?) ?) ?;
> Refine App ? ? (p_snd ? ? ) (ap_ ? ? (ap ? ? (ap_ ? ? pr4 ?) x1) ? );
> Refine TC_clos (OS_Dom_ty  A) nx e1 (fix x t e) t x1  ? ?;
> Refine TEp_RenExp nx x e e1 ? ? ? ? ? ?;
> 3 Refine pr2;
> Refine  App ? ? (p_snd ? ? ) (inv_TC_fix  (OS_Dom_ty A) t x1 e x ?);
> Refine pr6;
> 2 Refine p_Eq_subst Ty x1 t ([tt:Ty] (TC (OS_Dom_ty A) (fix x t e) tt)) ? ?;
> 2 Refine pr6;
> 2 Refine App ? ? (App ? ? (App ? ? (Eq_sym ?) ?) ?)
>          (App ? ? (p_fst ? ? ) (inv_TC_fix  (OS_Dom_ty A) t x1 e x ?));
> 2 Refine pr6;
> 2 Refine pr5;
> Refine App ? ? (App ? ? (App ? ? (Xmidvar nx x) ?) ?) ?;
> 2 Refine LL;
> 2 Intros PH1;
> Refine App ? ? (p_inl ? ?) ?;
> Refine PH1;
> ReturnAll;
> Refine LL;
> Intros PH1;
> Refine App ? ? (p_inr ? ?) ?;
> Refine LL;
> Intros PH2;
> Refine App ? ? (App ? ? pr1 ?) ?;
> Refine p_Eq_subst (List Vari) (TE_Dom (OS_Dom_ty A)) (OS_Dom A)
>        ([xx: (List Vari)] (member Vari nx xx)) ? ?;
> 2 Refine TEDomDomty_OSDom A;
> Refine TCHet_FVeinDomH (OS_Dom_ty A) (fix x t e) x1 pr6 nx ?;
> Refine FV_fix nx e PH2 x t PH1;
> ReturnAll;
> Refine App ? ? (App ? ? (p_pair ? ? ) ?) ?;
> Refine App ? ? (p_snd ? ? ) (App ? ? (p_fst ? ? )
>        (ap_ ? ? (ap ? ? (ap_ ? ? pr4 ?) x1) ? ));
> Refine TC_clos (OS_Dom_ty  A) nx e1 (fix x t e) t x1  ? ?;
> Refine TEp_RenExp nx x e e1 ? ? ? ? ? ?;
> 3 Refine pr2;
> Refine  App ? ? (p_snd ? ? ) (inv_TC_fix  (OS_Dom_ty A) t x1 e x ?);
> Refine pr6;
> 2 Refine p_Eq_subst Ty x1 t ([tt:Ty] (TC (OS_Dom_ty A) (fix x t e) tt)) ? ?;
```

```
> 2 Refine pr6;
> 2 Refine App ? ? (App ? ? (App ? ? (Eq_sym ?) ?) ?)
>          (App ? ? (p_fst ? ? ) (inv_TC_fix  (OS_Dom_ty A) t x1 e x ?));
> 2 Refine pr6;
> 2 Refine pr5;
> Refine App ? ? (App ? ? (App ? ? (Xmidvar nx x) ?) ?) ?;
> 2 Refine LL;
> 2 Intros PH1;
> Refine App ? ? (p_inl ? ?) ?;
> Refine PH1;
> ReturnAll;
> Refine LL;
> Intros PH1;
> Refine App ? ? (p_inr ? ?) ?;
> Refine LL;
> Intros PH2;
> Refine App ? ? (App ? ? pr1 ?) ?;
> Refine p_Eq_subst (List Vari) (TE_Dom (OS_Dom_ty A)) (OS_Dom A)
>        ([xx: (List Vari)] (member Vari nx xx)) ? ?;
> 2 Refine TEDomDomty_OSDom A;
> Refine TCHet_FVeinDomH (OS_Dom_ty A) (fix x t e) x1 pr6 nx ?;
> Refine FV_fix nx e PH2 x t PH1;
> ReturnAll;
> Refine App ? ? (p_fst ? ? ) (App ? ? (p_fst ? ? )
>        (ap_ ? ? (ap ? ? (ap_ ? ? pr4 ?) x1) ? ));
> Refine TC_clos (OS_Dom_ty  A) nx e1 (fix x t e) t x1  ? ?;
> Refine TEp_RenExp nx x e e1 ? ? ? ? ? ?;
> 3 Refine pr2;
> Refine  App ? ? (p_snd ? ? ) (inv_TC_fix  (OS_Dom_ty A) t x1 e x ?);
> Refine pr6;
> 2 Refine p_Eq_subst Ty x1 t ([tt:Ty] (TC (OS_Dom_ty A) (fix x t e) tt)) ? ?;
> 2 Refine pr6;
> 2 Refine App ? ? (App ? ? (App ? ? (Eq_sym ?) ?) ?)
>          (App ? ? (p_fst ? ? ) (inv_TC_fix  (OS_Dom_ty A) t x1 e x ?));
> 2 Refine pr6;
> 2 Refine pr5;
> Refine App ? ? (App ? ? (App ? ? (Xmidvar nx x) ?) ?) ?;
> 2 Refine LL;
> 2 Intros PH1;
> Refine App ? ? (p_inl ? ?) ?;
```

```
> Refine PH1;
> ReturnAll;
> Refine LL;
> Intros PH1;
> Refine App ? ? (p_inr ? ?) ?;
> Refine LL;
> Intros PH2;
> Refine App ? ? (App ? ? pr1 ?) ?;
> Refine p_Eq_subst (List Vari) (TE_Dom (OS_Dom_ty A)) (OS_Dom A)
>        ([xx: (List Vari)] (member Vari nx xx)) ? ?;
> 2 Refine TEDomDomty_OSDom A;
> Refine TCHet_FVeinDomH (OS_Dom_ty A) (fix x t e) x1 pr6 nx ?;
> Refine FV_fix nx e PH2 x t PH1;
> ReturnAll;

(* IfFalse *)
> Intros A A1 A2 e1 e2 e3 en pr1 pr2 pr3 pr4;
> Refine La_;
> Intros pr5;
> Refine La;
> Intros x1;
> Refine La_;
> Intros pr6;
> Refine App ? ? (App ? ? (p_pair ? ?) ?) ?;
> Refine App ? ? (p_snd ? ? ) (ap_ ? ? (ap ? ? (ap_ ? ? pr4 ?) x1) ? );
> Refine p_Eq_subst (List VT) (OS_Dom_ty A) (OS_Dom_ty A1)
>        ([tt: (List VT)] (TC  tt (cfgexp (cfg e3 A1)) x1)) ? ?;
> Refine App ? ? (p_snd ? ? ) (inv_TC_cond (OS_Dom_ty A) x1 e1 e2 e3  pr6);
> Refine Dom_pres (cfg e1 A) (cfg fff A1) pr1;
> Refine App ? ? (p_fst ? ? ) ( App ? ? (p_fst ? ? )
>        (ap_ ? ? (ap ? ? (ap_ ? ? pr3 ?) bool_Ty) ? ));
> Refine App ? ? (p_fst ? ? ) (App ? ? (p_fst ? ? )
>        (inv_TC_cond (OS_Dom_ty A) x1 e1 e2 e3  pr6));
> Refine pr5;
> Refine App ? ? (App ? ? (p_pair ? ?) ?) ?;
> Refine App ? ? (p_snd ? ? ) ( App ? ? (p_fst ? ? )
>        (ap_ ? ? (ap ? ? (ap_ ? ? pr4 ?) x1) ? ));
> Refine p_Eq_subst (List VT) (OS_Dom_ty A) (OS_Dom_ty A1)
>        ([tt: (List VT)] (TC  tt (cfgexp (cfg e3 A1)) x1)) ? ?;
> Refine App ? ? (p_snd ? ? ) (inv_TC_cond (OS_Dom_ty A) x1 e1 e2 e3  pr6);
```

```
> Refine Dom_pres (cfg e1 A) (cfg fff A1) pr1;
> Refine App ? ? (p_fst ? ? ) ( App ? ? (p_fst ? ? )
>        (ap_ ? ? (ap ? ? (ap_ ? ? pr3 ?) bool_Ty) ? ));
> Refine App ? ? (p_fst ? ? ) (App ? ? (p_fst ? ? )
>        (inv_TC_cond (OS_Dom_ty A) x1 e1 e2 e3  pr6));
> Refine pr5;
> Refine App ? ? (p_fst ? ? ) ( App ? ? (p_fst ? ? )
>        (ap_ ? ? (ap ? ? (ap_ ? ? pr4 ?) x1) ? ));
> Refine p_Eq_subst (List VT) (OS_Dom_ty A) (OS_Dom_ty A1)
>        ([tt: (List VT)] (TC  tt (cfgexp (cfg e3 A1)) x1)) ? ?;
> Refine App ? ? (p_snd ? ? ) (inv_TC_cond (OS_Dom_ty A) x1 e1 e2 e3  pr6);
> Refine Dom_pres (cfg e1 A) (cfg fff A1) pr1;
> Refine App ? ? (p_fst ? ? ) ( App ? ? (p_fst ? ? )
>        (ap_ ? ? (ap ? ? (ap_ ? ? pr3 ?) bool_Ty) ? ));
> Refine App ? ? (p_fst ? ? ) (App ? ? (p_fst ? ? )
>        (inv_TC_cond (OS_Dom_ty A) x1 e1 e2 e3  pr6));
> Refine pr5;
> ReturnAll;

(* IfTrue *)
> Intros A A1 A2 e1 e2 e3 en pr1 pr2 pr3 pr4;
> Refine La_;
> Intros pr5;
> Refine La;
> Intros x1;
> Refine La_;
> Intros pr6;
> Refine App ? ? (App ? ? (p_pair ? ?) ?) ?;
> Refine App ? ? (p_snd ? ? ) (ap_ ? ? (ap ? ? (ap_ ? ? pr4 ?) x1) ? );
> Refine p_Eq_subst (List VT) (OS_Dom_ty A) (OS_Dom_ty A1)
>        ([tt: (List VT)] (TC  tt (cfgexp (cfg e2 A1)) x1)) ? ?;
> Refine App ? ? (p_snd ? ? ) (App ? ? (p_fst ? ? )
>        (inv_TC_cond (OS_Dom_ty A) x1 e1 e2 e3  pr6));
> Refine Dom_pres (cfg e1 A) (cfg ttt A1) pr1;
> Refine App ? ? (p_fst ? ? ) ( App ? ? (p_fst ? ? )
>        (ap_ ? ? (ap ? ? (ap_ ? ? pr3 ?) bool_Ty) ? ));
> Refine App ? ? (p_fst ? ? ) (App ? ? (p_fst ? ? )
>        (inv_TC_cond (OS_Dom_ty A) x1 e1 e2 e3  pr6));
> Refine pr5;
> Refine App ? ? (App ? ? (p_pair ? ?) ?) ?;
```

```
> Refine App ? ? (p_snd ? ? ) ( App ? ? (p_fst ? ? )
>        (ap_ ? ? (ap ? ? (ap_ ? ? pr4 ?) x1) ? ));
> Refine p_Eq_subst (List VT) (OS_Dom_ty A) (OS_Dom_ty A1)
>        ([tt: (List VT)] (TC  tt (cfgexp (cfg e2 A1)) x1)) ? ?;
> Refine App ? ? (p_snd ? ? )  (App ? ? (p_fst ? ? )
>        (inv_TC_cond (OS_Dom_ty A) x1 e1 e2 e3  pr6));
> Refine Dom_pres (cfg e1 A) (cfg ttt A1) pr1;
> Refine App ? ? (p_fst ? ? ) ( App ? ? (p_fst ? ? )
>        (ap_ ? ? (ap ? ? (ap_ ? ? pr3 ?) bool_Ty) ? ));
> Refine App ? ? (p_fst ? ? ) (App ? ? (p_fst ? ? )
>        (inv_TC_cond (OS_Dom_ty A) x1 e1 e2 e3  pr6));
> Refine pr5;
> Refine App ? ? (p_fst ? ? ) ( App ? ? (p_fst ? ? )
>        (ap_ ? ? (ap ? ? (ap_ ? ? pr4 ?) x1) ? ));
> Refine p_Eq_subst (List VT) (OS_Dom_ty A) (OS_Dom_ty A1)
>        ([tt: (List VT)] (TC  tt (cfgexp (cfg e2 A1)) x1)) ? ?;
> Refine App ? ? (p_snd ? ? )  ( App ? ? (p_fst ? ? )
>        (inv_TC_cond (OS_Dom_ty A) x1 e1 e2 e3  pr6));
> Refine Dom_pres (cfg e1 A) (cfg ttt A1) pr1;
> Refine App ? ? (p_fst ? ? ) ( App ? ? (p_fst ? ? )
>        (ap_ ? ? (ap ? ? (ap_ ? ? pr3 ?) bool_Ty) ? ));
> Refine App ? ? (p_fst ? ? ) (App ? ? (p_fst ? ? )
>        (inv_TC_cond (OS_Dom_ty A) x1 e1 e2 e3  pr6));
> Refine pr5;
> ReturnAll;

(* Appl *)
> Intros A A1 A2 e1 e2 en1 en2 enf t n  pr1 pr2 pr3 pr4 pr5;
> Refine La_;
> Intros pr6;
> Refine La;
> Intros x1;
> Refine La_;
> Intros pr7;
> Refine App ? ? (App ? ? (p_pair ? ?) ?) ?;
> Refine App ? ? (p_snd ? ? ) (ap_ ? ? (ap ? ? (ap_ ? ? pr5 ?) x1) ? );
> Refine TC_clos (OS_Dom_ty  A1) n en2 e2 t x1  ? ?;
> Refine App ? ? (App ? ? (inv_TC_appl (OS_Dom_ty A) x1 e1 e2 ?) ?) ?;
> Refine LL;
> Intros r;
```

```
> Refine LL;
> Intros PH1;
> Refine App ? ? (p_fst ? ? ) (TEp_Ap e2 en1 en2 A n t pr2 ?
>        (OS_Dom_ty A1) r x1 ?);
> Refine App ? ? (p_snd ? ? ) ( App ? ? (p_fst ? ? )
>        (ap_ ? ? (ap ? ? (ap_ ? ? pr4 ?) (arr r x1)) ? ));
> Refine App ? ? (p_fst ? ?) PH1;
> Refine pr6;
> Refine La;
> Intros x;
> Refine La_;
> Intros PH2;
> Refine p_Eq_subst (List Vari) (TE_Dom (OS_Dom_ty A)) (OS_Dom A)
>        ([xx: (List Vari)] (member Vari x xx)) ? ?;
> Refine TCHet_FVeinDomH   (OS_Dom_ty A) en1  (arr r x1) ? x  PH2;
> 2 Refine TEDomDomty_OSDom A;
> Refine p_Eq_subst (List VT) (OS_Dom_ty A1) (OS_Dom_ty A)
>        ([tt: (List VT)] (TC  tt  en1 (arr r x1))) ? ?;
> Refine App ? ? (p_snd ? ? ) ( App ? ? (p_fst ? ? )
>        (ap_ ? ? (ap ? ? (ap_ ? ? pr4 ?) (arr r x1)) ? ));
> Refine App ? ? (p_fst ? ?) PH1;
> Refine pr6;
> Refine App ? ? (App ? ? (App ? ? (Eq_sym ? ) ? ) ?)
>        (Dom_pres (cfg e1 A) (cfg en1 A1) pr1);
> ReturnAll;
> Refine pr7;
> Refine App ? ? (App ? ? (inv_TC_appl (OS_Dom_ty A) x1 e1 e2 ?) ?) ?;
> Refine LL;
> Intros r;
> Refine LL;
> Intros PH1;
> Refine p_Eq_subst Ty r t ([t: Ty](TC (OS_Dom_ty A1) e2 t)) ? ?;
> Refine p_Eq_subst (List VT) (OS_Dom_ty A) (OS_Dom_ty A1)
>        ([tt: (List VT)] (TC  tt  e2 r)) ? ?;
> Refine App ? ? (p_snd ? ?) PH1;
> Refine Dom_pres (cfg e1 A) (cfg en1 A1) pr1;
> Refine App ? ? (p_snd ? ? ) (TEp_Ap e2 en1 en2 A n t pr2 ?
>        (OS_Dom_ty A1) r x1 ?);
> Refine App ? ? (p_snd ? ? ) ( App ? ? (p_fst ? ? )
>        (ap_ ? ? (ap ? ? (ap_ ? ? pr4 ?) (arr r x1)) ? ));
```

```
> Refine App ? ? (p_fst ? ?) PH1;
> Refine pr6;
> Refine La;
> Intros x;
> Refine La_;
> Intros PH2;
> Refine p_Eq_subst (List Vari) (TE_Dom (OS_Dom_ty A)) (OS_Dom A)
>        ([xx: (List Vari)] (member Vari x xx)) ? ?;
> Refine TCHet_FVeinDomH   (OS_Dom_ty A) en1  (arr r x1) ? x  PH2;
> 2 Refine TEDomDomty_OSDom A;
> Refine p_Eq_subst (List VT) (OS_Dom_ty A1) (OS_Dom_ty A)
>        ([tt: (List VT)] (TC  tt  en1 (arr r x1))) ? ?;
> Refine App ? ? (p_snd ? ? ) ( App ? ? (p_fst ? ? )
>        (ap_ ? ? (ap ? ? (ap_ ? ? pr4 ?) (arr r x1)) ? ));
> Refine App ? ? (p_fst ? ?) PH1;
> Refine pr6;
> Refine App ? ? (App ? ? (App ? ? (Eq_sym ? ) ? ) ?)
>        (Dom_pres (cfg e1 A) (cfg en1 A1) pr1);
> ReturnAll;
> Refine pr7;
> Refine App ? ? (App ? ? (inv_TC_appl (OS_Dom_ty A) x1 e1 e2 ?) ?) ?;
> Refine LL;
> Intros r;
> Refine LL;
> Intros PH1;
> Refine App ? ? (p_fst ? ? ) ( App ? ? (p_fst ? ? )
>        (ap_ ? ? (ap ? ? (ap_ ? ? pr4 ?) (arr r x1)) ? ));
> Refine App ? ? (p_fst ? ?) PH1;
> Refine pr6;
> ReturnAll;
> Refine pr7;
> Refine App ? ? (App ? ? (p_pair ? ? ) ?) ?;
> Refine App ? ? (p_snd ? ? ) (App ? ? (p_fst ? ? )
>        (ap_ ? ? (ap ? ? (ap_ ? ? pr5 ?) x1) ? ) );
> Refine TC_clos (OS_Dom_ty  A1) n en2 e2 t x1  ? ?;
> Refine App ? ? (App ? ? (inv_TC_appl (OS_Dom_ty A) x1 e1 e2 ?) ?) ?;
> Refine LL;
> Intros r;
> Refine LL;
> Intros PH1;
```

```
> Refine App ? ? (p_fst ? ? ) (TEp_Ap e2 en1 en2 A n t pr2 ?
>        (OS_Dom_ty A1) r x1 ?);
> Refine App ? ? (p_snd ? ? ) ( App ? ? (p_fst ? ? )
>        (ap_ ? ? (ap ? ? (ap_ ? ? pr4 ?) (arr r x1)) ? ));
> Refine App ? ? (p_fst ? ?) PH1;
> Refine pr6;
> Refine La;
> Intros x;
> Refine La_;
> Intros PH2;
> Refine p_Eq_subst (List Vari) (TE_Dom (OS_Dom_ty A)) (OS_Dom A)
>        ([xx: (List Vari)] (member Vari x xx)) ? ?;
> Refine TCHet_FVeinDomH   (OS_Dom_ty A) en1  (arr r x1) ? x  PH2;
> 2 Refine TEDomDomty_OSDom A;
> Refine p_Eq_subst (List VT) (OS_Dom_ty A1) (OS_Dom_ty A)
>        ([tt: (List VT)] (TC  tt  en1 (arr r x1))) ? ?;
> Refine App ? ? (p_snd ? ? ) ( App ? ? (p_fst ? ? )
>        (ap_ ? ? (ap ? ? (ap_ ? ? pr4 ?) (arr r x1)) ? ));
> Refine App ? ? (p_fst ? ?) PH1;
> Refine pr6;
> Refine App ? ? (App ? ? (App ? ? (Eq_sym ? ) ? ) ?)
>        (Dom_pres (cfg e1 A) (cfg en1 A1) pr1);
> ReturnAll;
> Refine pr7;
> Refine App ? ? (App ? ? (inv_TC_appl (OS_Dom_ty A) x1 e1 e2 ?) ?) ?;
> Refine LL;
> Intros r;
> Refine LL;
> Intros PH1;
> Refine p_Eq_subst Ty r t ([t: Ty](TC (OS_Dom_ty A1) e2 t)) ? ?;
> Refine p_Eq_subst (List VT) (OS_Dom_ty A) (OS_Dom_ty A1)
>        ([tt: (List VT)] (TC  tt  e2 r)) ? ?;
> Refine App ? ? (p_snd ? ?) PH1;
> Refine Dom_pres (cfg e1 A) (cfg en1 A1) pr1;
> Refine App ? ? (p_snd ? ? ) (TEp_Ap e2 en1 en2 A n t pr2 ?
>        (OS_Dom_ty A1) r x1 ?);
> Refine App ? ? (p_snd ? ? ) ( App ? ? (p_fst ? ? )
>        (ap_ ? ? (ap ? ? (ap_ ? ? pr4 ?) (arr r x1)) ? ));
> Refine App ? ? (p_fst ? ?) PH1;
> Refine pr6;
```

```
> Refine La;
> Intros x;
> Refine La_;
> Intros PH2;
> Refine p_Eq_subst (List Vari) (TE_Dom (OS_Dom_ty A)) (OS_Dom A)
>        ([xx: (List Vari)] (member Vari x xx)) ? ?;
> Refine TCHet_FVeinDomH   (OS_Dom_ty A) en1  (arr r x1) ? x  PH2;
> 2 Refine TEDomDomty_OSDom A;
> Refine p_Eq_subst (List VT) (OS_Dom_ty A1) (OS_Dom_ty A)
>        ([tt: (List VT)] (TC  tt  en1 (arr r x1))) ? ?;
> Refine App ? ? (p_snd ? ? ) ( App ? ? (p_fst ? ? )
>        (ap_ ? ? (ap ? ? (ap_ ? ? pr4 ?) (arr r x1)) ? ));
> Refine App ? ? (p_fst ? ?) PH1;
> Refine pr6;
> Refine App ? ? (App ? ? (App ? ? (Eq_sym ? ) ? ) ?)
>        (Dom_pres (cfg e1 A) (cfg en1 A1) pr1);
> ReturnAll;
> Refine pr7;
> Refine App ? ? (App ? ? (inv_TC_appl (OS_Dom_ty A) x1 e1 e2 ?) ?) ?;
> Refine LL;
> Intros r;
> Refine LL;
> Intros PH1;
> Refine App ? ? (p_fst ? ? ) ( App ? ? (p_fst ? ? )
>        (ap_ ? ? (ap ? ? (ap_ ? ? pr4 ?) (arr r x1)) ? ));
> Refine App ? ? (p_fst ? ?) PH1;
> Refine pr6;
> ReturnAll;
> Refine pr7;
> Refine App ? ? (App ? ? (inv_TC_appl (OS_Dom_ty A) x1 e1 e2 ?) ?) ?;
> Refine LL;
> Intros r;
> Refine LL;
> Intros PH1;
> Refine App ? ? (p_fst ? ? ) (App ? ? (p_fst ? ? )
>        (ap_ ? ? (ap ? ? (ap_ ? ? pr5 ?) x1) ? ) );
> Refine TC_clos (OS_Dom_ty  A1) n en2 e2 t x1  ? ?;
> Refine App ? ? (p_fst ? ? ) (TEp_Ap e2 en1 en2 A n t pr2 ?
>        (OS_Dom_ty A1) r x1 ?);
> Refine App ? ? (p_snd ? ? ) ( App ? ? (p_fst ? ? )
```

```
>        (ap_ ? ? (ap ? ? (ap_ ? ? pr4 ?) (arr r x1)) ? ));
> Refine App ? ? (p_fst ? ?) PH1;
> Refine pr6;
> Refine La;
> Intros x;
> Refine La_;
> Intros PH2;
> Refine p_Eq_subst (List Vari) (TE_Dom (OS_Dom_ty A)) (OS_Dom A)
>        ([xx: (List Vari)] (member Vari x xx)) ? ?;
> Refine TCHet_FVeinDomH   (OS_Dom_ty A) en1  (arr r x1) ? x  PH2;
> 2 Refine TEDomDomty_OSDom A;
> Refine p_Eq_subst (List VT) (OS_Dom_ty A1) (OS_Dom_ty A)
>        ([tt: (List VT)] (TC  tt  en1 (arr r x1))) ? ?;
> Refine App ? ? (p_snd ? ? ) ( App ? ? (p_fst ? ? )
>        (ap_ ? ? (ap ? ? (ap_ ? ? pr4 ?) (arr r x1)) ? ));
> Refine App ? ? (p_fst ? ?) PH1;
> Refine pr6;
> Refine App ? ? (App ? ? (App ? ? (Eq_sym ? ) ? ) ?)
>        (Dom_pres (cfg e1 A) (cfg en1 A1) pr1);
> ReturnAll;
> Refine p_Eq_subst Ty r t ([t: Ty](TC (OS_Dom_ty A1) e2 t)) ? ?;
> Refine p_Eq_subst (List VT) (OS_Dom_ty A) (OS_Dom_ty A1)
>        ([tt: (List VT)] (TC  tt  e2 r)) ? ?;
> Refine App ? ? (p_snd ? ?) PH1;
> Refine Dom_pres (cfg e1 A) (cfg en1 A1) pr1;
> Refine App ? ? (p_snd ? ? ) (TEp_Ap e2 en1 en2 A n t pr2 ?
>        (OS_Dom_ty A1) r x1 ?);
> Refine App ? ? (p_snd ? ? ) ( App ? ? (p_fst ? ? )
>        (ap_ ? ? (ap ? ? (ap_ ? ? pr4 ?) (arr r x1)) ? ));
> Refine App ? ? (p_fst ? ?) PH1;
> Refine pr6;
> Refine La;
> Intros x;
> Refine La_;
> Intros PH2;
> Refine p_Eq_subst (List Vari) (TE_Dom (OS_Dom_ty A)) (OS_Dom A)
>        ([xx: (List Vari)] (member Vari x xx)) ? ?;
> Refine TCHet_FVeinDomH   (OS_Dom_ty A) en1  (arr r x1) ? x  PH2;
> 2 Refine TEDomDomty_OSDom A;
> Refine p_Eq_subst (List VT) (OS_Dom_ty A1) (OS_Dom_ty A)
```

```
>       ([tt: (List VT)] (TC  tt  en1 (arr r x1))) ? ?;
> Refine App ? ? (p_snd ? ? ) ( App ? ? (p_fst ? ? )
>       (ap_ ? ? (ap ? ? (ap_ ? ? pr4 ?) (arr r x1)) ? ));
> Refine App ? ? (p_fst ? ?) PH1;
> Refine pr6;
> Refine App ? ? (App ? ? (App ? ? (Eq_sym ? ) ? ) ?)
>       (Dom_pres (cfg e1 A) (cfg en1 A1) pr1);
> ReturnAll;
> Refine App ? ? (p_fst ? ? ) ( App ? ? (p_fst ? ? )
>       (ap_ ? ? (ap ? ? (ap_ ? ? pr4 ?) (arr r x1)) ? ));
> Refine App ? ? (p_fst ? ?) PH1;
> Refine pr6;
> ReturnAll;
> Refine pr7;
> ReturnAll;

(* Var2 *)
> Intros A A1 e e1 t x y  pr1 pr2 pr3 pr4;
> Refine La_;
> Intros pr5;
> Refine La;
> Intros x1;
> Refine La_;
> Intros pr6;
> Refine App ? ? (App ? ? (p_pair ? ?) ?) ?;
> Refine App ? ? (p_snd ? ? ) (ap_ ? ? (ap ? ? (ap_ ? ? pr4 ?) x1) ? );
> Refine TC_var  (OS_Dom_ty A) y x1 ?;
> Refine Mp_inv_nfvExt  y x x1 t (nil VT) (OS_Dom_ty A) ?
>       (inv_TC_var (OS_Dom_ty (OScons x t e A)) x1 y pr6);
> Refine LL;
> Intros PH1;
> Refine App ? ? (App ? ? pr1 ?) ?;
> Refine App ? ? (App ? ? (App ? ? (Eq_sym ?) ?) ?) PH1;
> ReturnAll;
> Refine App ? ? (p_fst ? ?) (inv_valid_cons x t e A pr5);
> Refine App ? ? (App ? ? (p_pair ? ?) ?) ?;
> Refine TEp_nfvExt (append VT (nil VT) (OS_Dom_ty A1)) e1 x1 t  x ? ?
>       (nil VT) (OS_Dom_ty A1) ?;
> Refine App ? ? (Eq_refl ?) ?;
> Refine LL;
```

```
> Intros PH1;
> Refine App ? ? (App ? ? pr2 ?) ?;
> Refine p_Eq_subst (List Vari) (TE_Dom (OS_Dom_ty A)) (OS_Dom A)
>        ([xx: (List Vari)] (member Vari x xx)) ? ?;
> 2 Refine TEDomDomty_OSDom A;
> Refine TCHet_FVeinDomH (OS_Dom_ty A) e1 x1 ? x PH1;
> Refine p_Eq_subst (List VT) (OS_Dom_ty A1) (OS_Dom_ty A)
>        ([xx: (List VT)] (TC xx e1 x1)) ? ?;
> 2 Refine App ? ? (App ? ? (App ? ? (Eq_sym ?) ?) ?)
>           (Dom_pres (cfg (var y) A) (cfg e1 A1) pr3);
> Refine App ? ? (p_snd ? ? ) ( App ? ? (p_fst ? ? )
>        (ap_ ? ? (ap ? ? (ap_ ? ? pr4 ?) x1) ? ));
> Refine TC_var  (OS_Dom_ty A) y x1 ?;
> Refine Mp_inv_nfvExt  y x x1 t (nil VT) (OS_Dom_ty A) ?
>        (inv_TC_var (OS_Dom_ty (OScons x t e A)) x1 y pr6);
> Refine LL;
> Intros PH1;
> Refine App ? ? (App ? ? pr1 ?) ?;
> Refine App ? ? (App ? ? (App ? ? (Eq_sym ?) ?) ?) PH2;
> ReturnAll;
> Refine App ? ? (p_fst ? ? ) (inv_valid_cons x t e A pr5 );
> ReturnAll;
> Refine App ? ? (p_snd ? ? ) ( App ? ? (p_fst ? ? )
>        (ap_ ? ? (ap ? ? (ap_ ? ? pr4 ?) x1) ? ));
> Refine TC_var  (OS_Dom_ty A) y x1 ?;
> Refine Mp_inv_nfvExt  y x x1 t (nil VT) (OS_Dom_ty A) ?
>        (inv_TC_var (OS_Dom_ty (OScons x t e A)) x1 y pr6);
> Refine LL;
> Intros PH1;
> Refine App ? ? (App ? ? pr1 ?) ?;
> Refine App ? ? (App ? ? (App ? ? (Eq_sym ?) ?) ?) PH1;
> ReturnAll;
> Refine App ? ? (p_fst ? ? ) (inv_valid_cons x t e A pr5 );
> ReturnAll;
> Refine Valid_cons x t e A1 ? ?;
> Refine App ? ? (p_fst ? ? ) ( App ? ? (p_fst ? ? )
>        (ap_ ? ? (ap ? ? (ap_ ? ? pr4 ?) x1) ? ));
> Refine TC_var  (OS_Dom_ty A) y x1 ?;
> Refine Mp_inv_nfvExt  y x x1 t (nil VT) (OS_Dom_ty A) ?
>        (inv_TC_var (OS_Dom_ty (OScons x t e A)) x1 y pr6);
```

```
> Refine LL;
> Intros PH1;
> Refine App ? ? (App ? ? pr1 ?) ?;
> Refine App ? ? (App ? ? (App ? ? (Eq_sym ?) ?) ?) PH1;
> ReturnAll;
> Refine App ? ? (p_fst ? ? ) (inv_valid_cons x t e A pr5 );
> ReturnAll;
> Refine p_Eq_subst (List VT) (OS_Dom_ty A) (OS_Dom_ty A1)
>        ([xx: (List VT)] (TC xx e t)) ? ?;
> 2 Refine Dom_pres (cfg (var y) A) (cfg e1 A1) pr3;
> Refine App ? ? (p_snd ? ?) (inv_valid_cons x t e A pr5 );
> ReturnAll;

(* Var1 *)
> Intros A A1 e e1 t x  pr1 pr2 pr3 ;
> Refine La_;
> Intros pr4;
> Refine La;
> Intros x1;
> Refine La_;
> Intros pr5;
> Refine App ? ? (App ? ? (p_pair ? ?) ?) ?;
> Refine App ? ? (p_snd ? ? ) (ap_ ? ? (ap ? ? (ap_ ? ? pr3 ?) t) ? );
> Refine App ? ? (p_snd ? ?) (inv_valid_cons x t e A pr4 );
> Refine App ? ? (p_fst ? ?) (inv_valid_cons x t e A pr4 );
> Refine App ? ? (App ? ? (p_pair ? ?) ?) ?;
> Refine TEp_nfvExt (append VT (nil VT) (OS_Dom_ty A1)) e1 x1 t  x ? ?
>        (nil VT) (OS_Dom_ty A1) ?;
> Refine App ? ? (Eq_refl ?) ?;
> Refine LL;
> Intros PH1;
> Refine App ? ? (App ? ? pr1 ?) ?;
> Refine p_Eq_subst (List Vari) (TE_Dom (OS_Dom_ty A)) (OS_Dom A)
>        ([xx: (List Vari)] (member Vari x xx)) ? ?;
> 2 Refine TEDomDomty_OSDom A;
> Refine TCHet_FVeinDomH (OS_Dom_ty A) e1 x1 ? x PH1;
> Refine p_Eq_subst (List VT) (OS_Dom_ty A1) (OS_Dom_ty A)
>        ([xx: (List VT)] (TC xx e1 x1)) ? ?;
> 2 Refine App ? ? (App ? ? (App ? ? (Eq_sym ?) ?) ?)
>           (Dom_pres (cfg e A) (cfg e1 A1) pr2);
```

```
> Refine App ? ? (p_snd ? ? ) ( App ? ? (p_fst ? ? )
>        (ap_ ? ? (ap ? ? (ap_ ? ? pr3 ?) x1) ? ));
> Refine p_Eq_subst Ty t x1 ([t1: Ty](TC (OS_Dom_ty A) e t1)) ? ?;
> Refine App ? ? (p_snd ? ?) (inv_valid_cons x t e A pr4 );
> Refine If_T ? ? ? (inv_TC_var (OS_Dom_ty  (OScons x t e A)) x1 x pr5) ?;
> Refine App ? ? (Eq_refl ? ) ?;
> Refine App ? ? (p_fst ? ?) (inv_valid_cons x t e A pr4 );
> ReturnAll;
> Refine App ? ? (p_snd ? ? ) ( App ? ? (p_fst ? ? )
>        (ap_ ? ? (ap ? ? (ap_ ? ? pr3 ?) x1) ? ));
> Refine p_Eq_subst Ty t x1 ([t1: Ty](TC (OS_Dom_ty A) e t1)) ? ?;
> Refine App ? ? (p_snd ? ?) (inv_valid_cons x t e A pr4 );
> Refine If_T ? ? ? (inv_TC_var (OS_Dom_ty  (OScons x t e A)) x1 x pr5) ?;
> Refine App ? ? (Eq_refl ? ) ?;
> Refine App ? ? (p_fst ? ?) (inv_valid_cons x t e A pr4 );
> ReturnAll;
> Refine Valid_cons x t e1 A1 ?  ?;
> Refine App ? ? (p_fst ? ? ) ( App ? ? (p_fst ? ? )
>        (ap_ ? ? (ap ? ? (ap_ ? ? pr3 ?) t) ? ));
> Refine App ? ? (p_snd ? ?) (inv_valid_cons x t e A pr4 );
> Refine App ? ? (p_fst ? ?) (inv_valid_cons x t e A pr4 );
> Refine App ? ? (p_snd ? ? ) ( App ? ? (p_fst ? ? )
>        (ap_ ? ? (ap ? ? (ap_ ? ? pr3 ?) t) ? ));
> Refine App ? ? (p_snd ? ?) (inv_valid_cons x t e A pr4 );
> Refine App ? ? (p_fst ? ?) (inv_valid_cons x t e A pr4 );
> ReturnAll;

(* S *)
> Intros A A1 e e1 pr1 pr2 ;
> Refine La_;
> Intros pr3;
> Refine La;
> Intros x1;
> Refine La_;
> Intros pr4;
> Refine App ? ? (App ? ? (p_pair ? ?) ?) ?;
> Refine NF_Sno (suc e1);
> Refine Sno_s e1;
> Refine NFenat_Snoe e1 ?  (OS_Dom_ty A1) ?;
> Refine App ? ? (p_snd ? ? ) ( App ? ? (p_fst ? ? )
```

```
>        (ap_ ? ? (ap ? ? (ap_ ? ? pr2 ?) nat_Ty) ? ));
> Refine App ? ? (p_snd ? ? ) (inv_TC_suc (OS_Dom_ty A) x1 e pr4 );
> Refine pr3;
> Refine App ? ? (p_snd ? ? )  (ap_ ? ? (ap ? ? (ap_ ? ? pr2 ?) nat_Ty) ? );
> Refine App ? ? (p_snd ? ? ) (inv_TC_suc (OS_Dom_ty A) x1 e pr4 );
> Refine pr3;
> Refine App ? ? (App ? ? (p_pair ? ?) ?) ?;
> Refine p_Eq_subst Ty nat_Ty x1 ([x1:Ty](TC (OS_Dom_ty A1) (suc e1)  x1)) ? ?;
> Refine TC_suc ? ? ( App ? ? (p_snd ? ? )  (App ? ? (p_fst ? ? )
>        (ap_ ? ? (ap ? ? (ap_ ? ? pr2 ?) nat_Ty) ? )));
> Refine App ? ? (p_snd ? ? ) (inv_TC_suc (OS_Dom_ty A) x1 e pr4 );
> Refine pr3;
> Refine App ? ? (App ? ? (App ? ? (Eq_sym ?) ?) ?) (App ? ? (p_fst ? ? )
>        (inv_TC_suc (OS_Dom_ty A) x1 e pr4 ));
> Refine App ? ? (p_fst ? ? )  (App ? ? (p_fst ? ? )
>        (ap_ ? ? (ap ? ? (ap_ ? ? pr2 ?) nat_Ty) ? ));
> Refine App ? ? (p_snd ? ? ) (inv_TC_suc (OS_Dom_ty A) x1 e pr4 );
> Refine pr3;
> ReturnAll;

(* ZF *)
> Intros A A1 e e1 pr1 pr2 ;
> Refine La_;
> Intros pr3;
> Refine La;
> Intros x1;
> Refine La_;
> Intros pr4;
> Refine App ? ? (App ? ? (p_pair ? ?) ?) ?;
> Refine NF_fff;
> Refine App ? ? (App ? ? (p_pair ? ?) ?) ?;
> Refine p_Eq_subst Ty bool_Ty x1 ([x1:Ty](TC (OS_Dom_ty A1) fff  x1)) ? ?;
> Refine TC_fff  (OS_Dom_ty A1) ;
> Refine  App ? ? (App ? ? (App ? ? (Eq_sym ?) ?) ?)
>         (App ? ? (p_fst ? ? ) (inv_TC_is_o (OS_Dom_ty A) x1 e pr4 ));
> Refine App ? ? (p_fst ? ? )  (App ? ? (p_fst ? ? )
>        (ap_ ? ? (ap ? ? (ap_ ? ? pr2 ?) nat_Ty) ? ));
> Refine App ? ? (p_snd ? ? ) (inv_TC_is_o (OS_Dom_ty A) x1 e pr4 );
> Refine pr3;
> ReturnAll;
```

```
(* ZT *)
> Intros A A1 e  pr1 pr2 ;
> Refine La_;
> Intros pr3;
> Refine La;
> Intros x1;
> Refine La_;
> Intros pr4;
> Refine App ? ? (App ? ? (p_pair ? ?) ?) ?;
> Refine NF_ttt;
> Refine App ? ? (App ? ? (p_pair ? ?) ?) ?;
> Refine p_Eq_subst Ty bool_Ty x1 ([x1:Ty](TC (OS_Dom_ty A1) ttt  x1)) ? ?;
> Refine TC_ttt  (OS_Dom_ty A1) ;
> Refine App ? ? (App ? ? (App ? ? (Eq_sym ?) ?) ?)
>        (App ? ? (p_fst ? ? ) (inv_TC_is_o (OS_Dom_ty A) x1 e pr4 ));
> Refine App ? ? (p_fst ? ? )  (App ? ? (p_fst ? ? )
>        (ap_ ? ? (ap ? ? (ap_ ? ? pr2 ?) nat_Ty) ? ));
> Refine App ? ? (p_snd ? ? ) (inv_TC_is_o (OS_Dom_ty A) x1 e pr4 );
> Refine pr3;
> ReturnAll;

(* P *)
> Intros A A1 e e1 pr1 pr2 ;
> Refine La_;
> Intros pr3;
> Refine La;
> Intros x1;
> Refine La_;
> Intros pr4;
> Refine App ? ? (App ? ? (p_pair ? ?) ?) ?;
> Refine NF_Sno  e1;
> Refine inv_Sno_s e1;
> Refine inv_NF_Sno e1;
> Refine App ? ? (p_snd ? ? )  (ap_ ? ? (ap ? ? (ap_ ? ? pr2 ?) nat_Ty) ? );
> Refine App ? ? (p_snd ? ? ) (inv_TC_prd (OS_Dom_ty A) x1 e pr4 );
> Refine pr3;
> Refine App ? ? (App ? ? (p_pair ? ?) ?) ?;
> Refine p_Eq_subst Ty nat_Ty x1 ([x1:Ty](TC (OS_Dom_ty A1) e1  x1)) ? ?;
> Refine App ? ? (p_snd ? ?) (inv_TC_suc ? ? ?
```

```
>        (App ? ? (p_snd ? ? )  (App ? ? (p_fst ? ? )
>         (ap_ ? ? (ap ? ? (ap_ ? ? pr2 ?) nat_Ty) ? ))));
> Refine App ? ? (p_snd ? ? ) (inv_TC_prd (OS_Dom_ty A) x1 e pr4 );
> Refine pr3;
> Refine App ? ? (App ? ? (App ? ? (Eq_sym ?) ?) ?)
>        (App ? ? (p_fst ? ? ) (inv_TC_prd (OS_Dom_ty A) x1 e pr4 ));
> Refine App ? ? (p_fst ? ? )  (App ? ? (p_fst ? ? )
>        (ap_ ? ? (ap ? ? (ap_ ? ? pr2 ?) nat_Ty) ? ));
> Refine App ? ? (p_snd ? ? ) (inv_TC_prd (OS_Dom_ty A) x1 e pr4 );
> Refine pr3;
> ReturnAll;

(* PO *)
> Intros A A1 e  pr1 pr2 ;
> Refine La_;
> Intros pr3;
> Refine La;
> Intros x1;
> Refine La_;
> Intros pr4;
> Refine App ? ? (App ? ? (p_pair ? ?) ?) ?;
> Refine App ? ? (p_snd ? ? ) (ap_ ? ? (ap ? ? (ap_ ? ? pr2 ?) nat_Ty) ? );
> Refine App ? ? (p_snd ? ? ) (inv_TC_prd (OS_Dom_ty A) x1 e pr4 );
> Refine pr3;
> Refine App ? ? (App ? ? (p_pair ? ?) ?) ?;
> Refine p_Eq_subst Ty nat_Ty x1 ([x1:Ty](TC (OS_Dom_ty A1) o  x1)) ? ?;
> Refine App ? ? (p_snd ? ? ) ( App ? ? (p_fst ? ? )
>        (ap_ ? ? (ap ? ? (ap_ ? ? pr2 ?) nat_Ty) ? ));
> Refine App ? ? (p_snd ? ? ) (inv_TC_prd (OS_Dom_ty A) x1 e pr4 );
> Refine pr3;
> Refine App ? ? (App ? ? (App ? ? (Eq_sym ?) ?) ?)
>        (App ? ? (p_fst ? ? ) (inv_TC_prd (OS_Dom_ty A) x1 e pr4 ));
> Refine App ? ? (p_fst ? ? )  (App ? ? (p_fst ? ? )
>        (ap_ ? ? (ap ? ? (ap_ ? ? pr2 ?) nat_Ty) ? ));
> Refine App ? ? (p_snd ? ? ) (inv_TC_prd (OS_Dom_ty A) x1 e pr4 );
> Refine pr3;
> ReturnAll;

> Intros A e t x;
> Refine La_;
```

```
> Intros pr1;
> Refine La;
> Intros x1;
> Refine La_;
> Intros pr2;
> Refine App ? ? (App ? ? (p_pair ? ?) ?) ?;
> Refine NF_F;
> Refine F_abs;
> Refine App ? ? (App ? ? (p_pair ? ?) ?) ?;
> Refine pr2;
> Refine pr1;
> ReturnAll;

> Intros A ;
> Refine La_;
> Intros pr1;
> Refine La;
> Intros x1;
> Refine La_;
> Intros pr2;
> Refine App ? ? (App ? ? (p_pair ? ?) ?) ?;
> Refine NF_fff;
> Refine App ? ? (App ? ? (p_pair ? ?) ?) ?;
> Refine pr2;
> Refine pr1;
> ReturnAll;

> Intros A ;
> Refine La_;
> Intros pr1;
> Refine La;
> Intros x1;
> Refine La_;
> Intros pr2;
> Refine App ? ? (App ? ? (p_pair ? ?) ?) ?;
> Refine NF_ttt;
> Refine App ? ? (App ? ? (p_pair ? ?) ?) ?;
> Refine pr2;
> Refine pr1;
> ReturnAll;
```

```
> Intros A ;
> Refine La_;
> Intros pr1;
> Refine La;
> Intros x1;
> Refine La_;
> Intros pr2;
> Refine App ? ? (App ? ? (p_pair ? ?) ?) ?;
> Refine NF_Sno;
> Refine Sno_o;
> Refine App ? ? (App ? ? (p_pair ? ?) ?) ?;
> Refine pr2;
> Refine pr1;
> ReturnAll;

(*************************************************************)
(* lemma 2.                                                  *)
(*                                                           *)
(*************************************************************)

> Claim  subjr_NF: (c,c': El Config)(p1: El (Prf (OSred c c')))
>        (p2: El (Prf (Valid_env (cfgenv c))))
>        (t:Ty) (p3: El (Prf (TC (OS_Dom_ty (cfgenv c)) (cfgexp c) t)))
>        El (Prf (and (and (Valid_env (cfgenv c'))
>        (TC (OS_Dom_ty (cfgenv c')) (cfgexp c') t))
>        (NF (cfgexp c'))));
> Intros c c' p1 p2 t p3;
> Refine ap_ ? ? (ap ? ? (ap_ ? ? (subjr_NFPi c c' p1) p2) t) p3;
> ReturnAll;


(*************************************************************)
(* Theorem 1.                                                *)
(* Subject Reduction                                         *)
(* <<e,A>> -> <<e',A'>>---> Valid(A)---> Domt(A)|- e:t  --->  *)
(* Domt(A')|- e':t                                           *)
(*************************************************************)

> Claim subjr_red : (e,e': El Tm)(A,A': El OS_env)
```

```
>                   (p1: El (Prf (OSred (cfg e A) (cfg e' A'))))
>                   (p2: El (Prf (Valid_env A)))(t: El Ty)
>                   (p3: El (Prf (TC (OS_Dom_ty A) e t)))
>                   El (Prf  (TC (OS_Dom_ty A') e' t));
> Intros e e' A  A' p1 p2 t p3;
> Refine App ? ? (p_snd ? ?) ( App ? ? (p_fst ? ?)
>        (subjr_NF (cfg e A) (cfg e' A') p1 p2 t p3));
> ReturnAll;


(*************************************************************)
(* Theorem 2.                                                *)
(* Normal Forms                                              *)
(* <<e,A>> -> <<e',A'>>--->Valid<<e,A>>---> e' in NF         *)
(*************************************************************)

> Claim NormalForms : (e,e': El Tm)(A,A': El OS_env)
>                     (p1: El (Prf (OSred (cfg e A) (cfg e' A'))))
>                     (p2 : El  (Prf (Valid_config (cfg e A))))
>                     El (Prf (NF e'));
> Intros e e' A  A' p1 p2;
> Claim PH1: (Prf  (Ex Ty ([t:Ty] TC (OS_Dom_ty (cfgenv (cfg e  A)))
>                              (cfgexp (cfg e  A)) t)));
> Refine App ? ? (p_snd ? ? )  ( inv_valid_cfg (cfg e  A ) p2);
> ReturnAll;
> Refine  (App ? ? (App ? ? PH1 ?) ?);
> Refine LL;
> Intros t;
> Refine LL;
> Intros p3;
> Refine  App ? ? (p_snd ? ?) (subjr_NF (cfg e A) (cfg e' A') p1 ? t p3);
> Refine App ? ? (p_fst ? ? ) ( inv_valid_cfg (cfg e  A ) p2);
> ReturnAll;
```

From the above proof of the Subject Reduction theorem we can see that the proof in Plastic is more detailed than Lego and Coq. This is due to many reasons (such as the underlying type theory or logic framework, etc.). But we think that this does not prevent us to study the issues about domain specific reasoning.

Bibliography

[Abadi et al., 1991] Abadi, M., Cardelli, L., Curien, P.-L., and Lévy, J.-J. (1991). Explicit substitutions. *Journal of Functional Programming*, 1(4):375–416. Preliminary version in POPL 1990.

[Aczel, 1978] Aczel, P. (1978). The type theoretic interpretation of constructive set theory. In *Logic Colloquium '77 (Proc. Conf., Wrocław, 1977)*, volume 96 of *Stud. Logic Foundations Math.*, pages 55–66. North-Holland, Amsterdam.

[Aczel, 1982] Aczel, P. (1982). The type theoretic interpretation of constructive set theory: choice principles. In *The L. E. J. Brouwer Centenary Symposium (Noordwijkerhout, 1981)*, volume 110 of *Stud. Logic Found. Math.*, pages 1–40. North-Holland, Amsterdam.

[Aczel, 1986] Aczel, P. (1986). The type theoretic interpretation of constructive set theory: inductive definitions. In *Logic, methodology and philosophy of science, VII (Salzburg, 1983)*, volume 114 of *Stud. Logic Found. Math.*, pages 17–49. North-Holland, Amsterdam.

[Aczel, 1999] Aczel, P. (1999). On relating type theories and set theories. In *Types for proofs and programs (Irsee, 1998)*, volume 1657 of *Lecture Notes in Comput. Sci.*, pages 1–18. Springer, Berlin.

[Aczel and Rathjen, 2001] Aczel, P. and Rathjen, M. (2000/2001). Notes on constructive set theory. Technical Report 40, Mittag-Leffler.

[Adams, 2004] Adams, R. (2004). The consistency of large elimination. `http://WWW.cs.rhul.ac.uk/~robin`.

[Archer and Heitmeyer, 1997] Archer, M. and Heitmeyer, C. (1997). Human-style theorem proving using PVS. In *Proc. 10th International Theorem Proving in Higher Order Logics Conference*, pages 33–48.

[Archer et al., 1998] Archer, M., Heitmeyer, C., and Sims, S. (1998). TAME: A PVS interface to simplify proofs for automata models. In *User Interfaces for Theorem Provers*, Eindhoven, The Netherlands.

[Aspinall, 2000] Aspinall, D. (2000). Protocols for interactive e-proof. available at `http://zermelo.dcs.ed.ac.uk/~da/drafts/#eproof`.

[Aspinall, 2005a] Aspinall, D. (2005a). Proof general. available at `http://proofgeneral.inf.ed.ac.uk/`.

[Aspinall, 2005b] Aspinall, D. (2005b). Proof general kit. `http://proofgeneral.inf.ed.ac.uk/kit`.

[Aspinall and Lüth, 2003] Aspinall, D. and Lüth, C. (2003). Proof general meets isawin. In *User Interfaces for Theorem Provers (UITP 2003)*, Rome, Italy.

[Backhouse, 1988] Backhouse, R. (1988). On the meaning and construction of the rules in Martin-Löf's theory of types. In *et al.*, A. A., editor, *Workshop on General Logic*. LFCS Report Series, ECS-LFCS-88-52, Dept. of Computer Science, University of Edinburgh.

[Backhouse, 1998] Backhouse, R. C., editor (1998). *Informal proceedings of the Workshop on User Interfaces for Theorem Prover*. Eindhoven University of Technology.

[Bailey, 1996] Bailey, A. (1996). Lego with implicit coercions. Draft.

[Bailey, 1998] Bailey, A. (1998). *The Machine-checked Literate Formalisation of Algebra in Type Theory*. PhD thesis, University of Manchester.

[Barendregt, 2005] Barendregt, H. (2005). The fundamental theorem of algebra project. `http://WWW.cs.kun.nl/~freek/fta/xindex.html`.

[Barendregt, 1990] Barendregt, H. P. (1990). Functional programming and lambda calculus. In van Leeuwen, J., editor, *Handbook of Theoretical Computer Science, Vol. B: Formal Models and Semantics*, chapter 7, pages 321–363. Elsevier, Amsterdam, and The MIT Press, Cambridge, MA.

[Barthe et al., 2003] Barthe, G., Capretta, V., and Pons, O. (2003). Setiods in type theory. *Functional Programming*, 13(2):261–293.

[Barthe and Sørensen, 2000] Barthe, G. and Sørensen, M. H. (2000). Domain-free pure type systems. *Journal of Functional Programming*, 10(5):412–452.

[Basin and Matthews, 2002] Basin, D. and Matthews, S. (2002). Logical frameworks. In Gabbay, D. and Guenthner, F., editors, *Handbook of Philosophical Logic, second edition*, volume 9, pages 89–164. Kluwer Academic Publishers, Dordrecht.

[Berardi, 1990] Berardi, S. (1990). *Type Dependence and Constructive Mathematics*. PhD thesis, Universita di Torino, Italy.

[Bertot et al., 1997a] Bertot, J., Bertot, Y., Coscoy, Y., Goguen, H., and Montagnac, F. (1997a). User guide to the CTCOQ proof environment. Technical Report RT-0210, Inria, Institut National de Recherche en Informatique et en Automatique.

[Bertot, 1998] Bertot, Y. (1998). The ctcoq system: Design and architecture. Research report 3540, INRIA SORPHIA ANTIPOLIS.

[Bertot et al., 1997b] Bertot, Y., Kleymann-Schreiber, T., and Sequeira, D. (1997b). Implementing proof by pointing without a structure editor. Technical report ECS-LFCS-97-368, University of Edinburgh.

[Bertot and Théry, 1998] Bertot, Y. and Théry, L. (1998). A generic approach to building user interfaces for theorem provers. *Journal of Symbolic Computation*, 25(2):161–194.

[Betarte et al., 2000] Betarte, G., Cornes, C., Szasz, N., and Tasistro, A. (2000). Smart card operating system. *Lecture Notes in Computer Science*, 1956:77–93.

[Borras et al., 1989] Borras, P., Clement, D., Despeyrouz, T., Incerpi, J., Kahn, G., Lang, B., and Pascual, V. (1989). CENTAUR: The system. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments (PSDE)*, volume 24(2), pages 14–24, New York, NY. ACM Press.

[Boulton et al., 1998] Boulton, R., Slind, K., Bundy, A., and Gordon, M. (1998). An interface between CLAM and HOL. In Grundy, J. and Newey, M., editors, *Proceedings of the 11th International Conference on Theorem Proving in Higher Order Logics (TPHOLs'98)*, volume 1479 of *Lecture Notes in Computer Science*, pages 87–104, Canberra, Australia. Springer.

[Boyer and J S. Moore, 1979] Boyer, R. S. and J S. Moore (1979). *A Computational Logic.* Academic Press.

[Boyer and J S. Moore, 1997] Boyer, R. S. and J S. Moore (1997). *A Computational Logic Handbook.* Academic Press, second edition.

[Boyer and Moore, 1984] Boyer, R. S. and Moore, J. S. (1984). Proof checking the RSA public key encryption algorithm. *American Mathematical Monthly*, 91(3):181–189.

[Bradfield and Stirling, 2003] Bradfield, J. and Stirling, C. (2003). Modal logics and mu-calculi: an introduction. `http://WWW.mimuw.edu.pl/~sl/teaching/02_03/WZTPW/HANDBOOK/mu_intro.ps`.

[Bundy, 1987] Bundy, A. (1987). The use of explicit plans to guide inductive proofs. In *Conf. on Automated Deduction (CADE 9).*

[Bundy, 2001] Bundy, A. (2001). The automation of proof by mathematical induction. In Robinson, A. and Voronkov, A., editors, *Handbook of Automated Reasoning*, volume I, chapter 13, pages 845–911. Elsevier Science.

[Bundy et al., 2000] Bundy, A., Moore, J., and Zinn, C. (2000). An intelligent tutoring system for induction proofs.

[Callaghan, 1999] Callaghan, P. (1999). Plastic: an implementation of typed LF with coercions. Talk given in the Annual Conf of TYPES'99.

[Callaghan, 2000a] Callaghan, P. (2000a). The plastic proof assistant. `http://WWW.dur.ac.uk/CARG/plastic.html`.

[Callaghan and Luo, 1998] Callaghan, P. and Luo, Z. (1998). Mathematical vernacular in type theory based proof assistants. In *User Interfaces for Theorem Provers (UITP'98)*, Eindhoven.

[Callaghan and Luo, 2000a] Callaghan, P. and Luo, Z. (2000a). Implementation techniques for inductive types in plastic. *Proceedings of TYPES'99*. LNCS series.

[Callaghan, 2000b] Callaghan, P. C. (2000b). Coherence checking of coercion in plastic. APPSEM Workshop on Subtyping and Dependent Types in Programming.

[Callaghan, 2005] Callaghan, P. C. (2005). Plastic WWW page. `http://WWW.dur.ac.uk/CARG/plastic.html`.

[Callaghan and Luo, 2000b] Callaghan, P. C. and Luo, Z. (2000b). Plastic: an implementation of typed LF with coercive subtyping and universes. Journal of Automated Reasoning, special issue on Logical Frameworks.

[Callaghan and Luo, 2001] Callaghan, P. C. and Luo, Z. (2001). An implementation of LF with coercive subtyping and universes. *Automated Reasoning*, 27(1):3–27.

[Callaghan et al., 2001] Callaghan, P. C., Luo, Z., and Pang, J. (2001). Object languages in a type theoretic meta-framework. In *Proc. Workshop on Proof Transformation, Presentation and Proof Complexities*, pages 23–36, Siena, Italy.

[Church, 1932] Church, A. (1932). A set of postulates for the foundation of logic. *Annals of Mathematics*, (2) 33 and 34.

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

[Clarke and Emerson, 1981] Clarke, E. M. and Emerson, E. A. (1981). Design and synchronization skeletons using branching time temporal logic. *Lecture Notes in Computer Science*, 131:52–71.

[Clarke et al., 1986] Clarke, E. M., Emerson, E. A., and Sistla, A. P. (1986). Automatic verification of finite-state concurrent system using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263.

[Clarke et al., 1994] Clarke, E. M., Grumberg, O., and Hamaguchi, K. (1994). Another look at ltl model checking. In D.L.Dill, editor, *Proceedings of the 6th Conference on Computer Aided Verification*, volume 818 of *Lecture Notes in Computer Science*, pages 415–427, Stanford, CA. Springer Verlag.

[Cleaveland et al., 1993] Cleaveland, R., Parrow, J., and Steffen, B. (1993). The Concurrency Workbench: A semantics-based tool for the verification of concurrent systems. *ACM Transactions on Programming Languages and Systems*, 15(1):36–72.

[Cleaveland and Smolka, 1999] Cleaveland, R. and Smolka, S. (1999). Process algebra.

[Constable et al., 1986] Constable, R. et al. (1986). *Implementing Mathematics with the NuPRL Proof Development System.* Pretice-Hall.

[Consulting, 2001] Consulting, M. (2001). Domain-specific modelling: 10 times faster than uml. Technical report, MetaCase Consulting.

[Coq, 2005] Coq (2005). Coq WWW page. `http://pauillac.inria.fr/coq`.

[Coquand, 1992] Coquand, T. (1992). Pattern matching with dependent types. Talk given at the BRA workshop on Proofs and Types, Bastad.

[Coquand and Huet, 1985] Coquand, T. and Huet, G. (1985). Constructions: a higher order proof system for mechanizing mathematics. *Lecture Notes in Computer Science*, 203.

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

[Coquand and Paulin-Mohring, 1990] Coquand, T. and Paulin-Mohring, C. (1990). Inductively defined types. *LNCS*, 417.

[Crary, 1998] Crary, K. (1998). Type-theoretic methodology for practical programming languages. Technical Report TR98-1699, Cornell University, Computer Science.

[Curry, 1934] Curry, H. (1934). Functionality in combinatory logic. *Proc. Nat. Acad. Science USA*, 20:584–590.

[Curry and Feys, 1958] Curry, H. B. and Feys, R. (1958). *Combinatory Logic.* North-Holland, 1 edition.

[Dahn and Wolf, 1994] Dahn, B. I. and Wolf, A. (1994). A calculus supporting structured proofs. *Journal for Information Processing and Cybernetics (EIK)*, 5–6:261–276.

[Dahn, 1998] Dahn, I. (1998). Using ILF as an interface to many theorem provers. In *Informal proceedings of the Workshop on User Interfaces for Theorem Prover*, pages 75–86.

[Dam, 1995] Dam, M. (1995). Model checking compositional proof systems for model checking infinite state processes. In Lee, I. and Smolka, S. A., editors, *Proceedings 6th Int. Conf. on Concurrency Theory, CONCUR'95, Philadelphia, PA, USA, 21–24 Aug 1995*, volume 962 of *Lecture Notes in Computer Science*, pages 12–26. Springer-Verlag, Berlin.

[de Bruijn, 1980] de Bruijn, N. (1980). A survey of the project AUTOMATH. In Hindley, J. and Seldin, J., editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism.* Academic Press.

[Denney, 2001] Denney, E. (2001). A Prototype Proof Translator from HOL to Coq. Draft.

[Dery and Rideau, 1994] Dery, A.-M. and Rideau, L. (1994). Distributed programming environments: an example of a message protocol. Technical Report RT-0165, Inria, Institut National de Recherche en Informatique et en Automatique.

[Dybjer, 1991] Dybjer, P. (1991). Inductive sets and families in Martin-Löf's type theory and their set-theoretic semantics. In Huet, G. and Plotkin, G., editors, *Logical Frameworks.* Cambridge University Press.

[Eastaughffe, 1998] Eastaughffe, K. A. (1998). Support for interactive theorem proving: Some design principles and their application. In *Informal proceedings of the Workshop on User Interfaces for Theorem Prover*, pages 96–103.

[Emerson, 1990] Emerson, E. A. (1990). Temporal and Modal Logic. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, pages 996–1072, Amsterdam. Elsevier Science Publishers.

[Emerson and Clarke, 1980] Emerson, E. A. and Clarke, E. M. (1980). Characterizing correctness properties of parallel programs using fixpoints. *Procs ICALP'80 LNCS*, 85:169–181.

[Emerson and Lei, 1986] Emerson, E. A. and Lei, C. L. (1986). Efficient model checking in fragments of the propositional mu-calculus. In *Proc. 1th IEEE LICS.*

[Field, 1990] Field, J. (1990). On laziness and optimality in lambda interpreters: Tools for specification and analysis. In *Proceedings of the seventeenth symposium on Principles of Programming Languages*, pages 1–15, San Francisco.

[Fischer and Ladner, 1979] Fischer, M. J. and Ladner, R. E. (1979). Propositional dynamic logic of regular programs. *J. Computer and System Science*, 18:194–211.

[Gentzen, 1935] Gentzen, G. (1935). Untersuchungen über das logische schliessen. *Mathematische Zeitschrift*, 39.

[Gill and Marlow, 2005] Gill, A. and Marlow, S. (2005). Happy WWW page. `http://WWW.haskell.org/happy/`.

[Girard, 1972] Girard, J.-Y. (1972). *Interprétation fonctionelle et élimination des coupures de l'arithmétique d'ordre supérieur.* PhD thesis, Université Paris VII.

[Goguen, 1994] Goguen, H. (1994). *A Typed Operational Semantics for Type Theory.* PhD thesis, University of Edinburgh.

[Goguen, 1999] Goguen, H. (1999). Soundness of the logical framework for its typed operational semantics. In Girard, J.-Y., editor, *Proceedings of the 4th International Conference on Typed Lambda Calculi and Applications (TLCA'99)*, pages 177–197, L'Aquila, Italy. Springer-Verlag LNCS 1581.

[Gordon, 1985] Gordon, M. (1985). HOL a machine oriented formulation of higher order logic. Technical Report TR-68, Computer Laboratory, Cambridge University.

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

[Gordon et al., 1979] Gordon, M., Milner, A., and Wadsworth, C. (1979). *Edinburgh LCF: A Mechanized Logic of Computation*, volume 78 of *LNCS*. Springer-Verlag, New York.

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

[Griffin, 1988] Griffin, T. G. (1988). *Notational Definition and Top-Down Refinement for Interactive Proof Development Systems.* PhD thesis, Cornell University, Departement of Computer Science.

[Group, 2000] Group, T. C. S. (2000). Dove user manual. Technical report, Information Technology Division, The Australian Defence Science and Technology Organisation.

[Gurov, 1998] Gurov, D. B. (1998). *Specification and Verification of Communicating Systems with Value Passing.* PhD thesis, University of Victoria.

[Harper et al., 1987] Harper, R., Honsell, F., and Plotkin, G. (1987). A framework for defining logics. *Proc. 2nd Ann. Symp. on Logic in Computer Science. IEEE.*

[Hennessy, 1991] Hennessy, M. (1991). A proof system for communicating processes with value-passing. *Formal Aspects of Computing*, 3(4):346–366.

[Hennessy and Liu, 1993] Hennessy, M. and Liu, X. (1993). A modal logic for message passing processes. In *Proc. 5th International Computer Aided Verification Conference*, pages 359–370.

[Hennessy and Milner, 1980] Hennessy, M. and Milner, R. (1980). On observing nondeterminism and concurrency. In *Proc. ICALP'80 LNCS 85.*

[Hennessy and Milner, 1985] Hennessy, M. and Milner, R. (1985). Algebraic laws for nondeterminism and concurrency. *Journal of the Association for Computing Machinery*, 32(1):137–161.

[Heyd and Cregut, 1996] Heyd, B. and Cregut, P. (1996). A modular coding of unity in coq. In J. Von Wright, J. Grundy, and J. Harrison, editors, *Ninth international Conference on Theorem Proving in Higher Order Logics TPHOL*, volume 1125 of *Lecture Notes in Computer Science*, pages 251–266, Turku, Finland. Springer Verlag.

[Hickey, 1997] Hickey, J. J. (1997). Nuprl-Light: An implementation framework for higher-order logics. *Lecture Notes in Computer Science*, 1249.

[Hill and Thompson, 1995] Hill, S. and Thompson, S. (1995). Miranda in Isabelle. In Paulson, L. C., editor, *Preceedings of the first Isabelle Users Workshop*, number 397 in University Of Cambridge Computer Laboratory Technical Reports Series, pages 122–135.

[Howard, 1980] Howard, W. A. (1980). The formulae-as-types notion of construction. In Hindley, J. and Seldin, J., editors, *To H. B. Curry: Essays on Combinatory Logic*. Academic Press.

[Howe, 1989] Howe, D. J. (1989). Equality in lazy computation systems. In *Proceedings, Fourth Annual Symposium on Logic in Computer Science*, pages 198–203, Asilomar Conference Center, Pacific Grove, California. IEEE Computer Society Press.

[Hunt et al., 1992] Hunt, W. A., Jr., and Brock, B. C. (1992). A Formal HDL and its use in the FM9001 verification. In *Philosophical Transactions of the Royal Society*, pages 339(1652):35–47.

[Jackson, 1995] Jackson, P. B. (1995). *Enhancing the Nuprl Proof Development System and Applying it to Computational Abstract Algebra*. PhD thesis, Cornell University.

[Jones et al., 1998] Jones, A., Luo, Z., and Soloviev, S. (1998). Some proof-theoretic and algorithmic aspects of coercive subtyping. *Types for proofs and programs (eds, E. Gimenez and C. Paulin-Mohring), Proc. of the Inter. Conf. TYPES'96, LNCS 1512.*

[Jones and Hansen, 2002] Jones, M. A. and Hansen, T. L. (2002). Using xml in the masp client-server protocol.

[Jones, 1993] Jones, M. P. (1993). Coherence for qualified types. Technical Report YALEU/DCS/RR-989, Yale University, New Haven, Connecticut, USA.

[Jutting, 1977] Jutting, L. S. (1977). *Checking Landau's "Grndalagen" in the AUTOMATH system*. PhD thesis, Eindhoven University.

[Kahn, 1987] Kahn, G. (1987). Natural semantics. In *Proceedings of the Symposium on Theoretical Aspects of Computer Science (STACS)*, volume 247 of *Lecture Notes in Computer Science*, pages 22–39. Springer-Verlag.

[Kleene and Rosser, 1935] Kleene, S. and Rosser, J. (1935). The inconsistency of certain formal logics. *Annals Math.*, (2) 36.

[Klop, 1980] Klop, J. W. (1980). Combinatory reduction systems. *Mathematical Center Tracts 127.*

[Kozen, 1983] Kozen, D. (1983). Results on the propositional mu -calculus. *Theoretical Computer Science*, 27(3):333–354.

[Laurent, 1999] Laurent, S. S. (1999). *Inside XML DTDs: scientific and technical.* Enterprise computing. McGraw-Hill, New York, NY, USA.

[Leucker and Noll, 2000] Leucker, M. and Noll, T. (2000). Truth - A real-world application in Haskell. In Mohnen, M., editor, *Proceedings of the 12th International Workshop on Implementation of Functional Languages (IFL'00)*, number AIB-00-7 in Aachener Informatik Berichte, pages 363–380. RWTH Aachen.

[Leucker and Noll, 2001] Leucker, M. and Noll, T. (2001). Truth/SLC - A parallel verification platform for concurrent systems. In Berry, G., Comon, H., and Finkel, A., editors, *Proceedings of the 13th Conference on Computer Aided Verification (CAV '01)*, volume 2102 of *Lecture Notes in Computer Science*, pages 255–259. Springer-Verlag Inc.

[Lowe and Duncan, 1997] Lowe, H. and Duncan, D. (1997). XBarnacle: Making theorem provers more accessible. In McCune, W., editor, *Proceedings of the 14th International Conference on Automated deduction*, volume 1249 of *LNAI*, pages 404–407, Berlin. Springer.

[Luo, 2004] Luo, Y. (2004). *Coherence and Transitivity in Coercive Subtyping.* PhD thesis, University of Durham.

[Luo, 1990a] Luo, Z. (1990a). *An Extended Calculus of Constructions.* PhD thesis, University of Edinburgh. Also as Report CST-65-90/ECS-LFCS-90-118, Department of Computer Science, University of Edinburgh.

[Luo, 1990b] Luo, Z. (1990b). A problem of adequacy: conservativity of calculus of constructions over higher-order logic. Technical report, LFCS report series ECS-LFCS-90-121, Department of Computer Science, University of Edinburgh.

[Luo, 1994] Luo, Z. (1994). *Computation and Reasoning: A Type Theory for Computer Science.* Oxford University Press.

[Luo, 1997] Luo, Z. (1997). Coercive subtyping in type theory. *Proc. of CSL'96, the 1996 Annual Conference of the European Association for Computer Science Logic, Utrecht. LNCS 1258.*

[Luo, 1999] Luo, Z. (1999). Coercive subtyping. *Journal of Logic and Computation*, 9(1):105–130.

[Luo, 2003] Luo, Z. (2003). PAL+: A lambda-free logical framework. *Functional Programming*, 13(2):317–338.

[Luo and Pollack, 1992] Luo, Z. and Pollack, R. (1992). LEGO Proof Development System: User's Manual. LFCS Report ECS-LFCS-92-211, Department of Computer Science, University of Edinburgh.

[Luo and Soloviev, 1999] Luo, Z. and Soloviev, S. (1999). Dependent coercions. In *The 8th international conference on category theory and computer science (CTCS'99)*, Edinburgh, Scotland. Electronic Notes in Theoretical Computer Science 29.

[Magnusson and Nordström, 1994] Magnusson, L. and Nordström, B. (1994). The ALF proof editor and its proof engine. In *TYPES '93: Proceedings of the international workshop on Types for proofs and programs*, pages 213–237, Secaucus, NJ, USA. Springer-Verlag New York, Inc.

[Manna and Pnueli, 1981] Manna, Z. and Pnueli, A. (1981). Verification of temporal programs: the temporal framework. In Boyer, R. S. and Moore, J. S., editors, *The Correctness Problem in Computer Science*. Academic Press, New York.

[Marques, 1998] Marques, F. (1998). Program composition in coq-unity. In Grundy, J. and Newey, M., editors, *Supplementary proceedings of the 11th International Conference on Theorem Proving in Higher Order Logics (TPHOLs'98)*, pages 95–104, Canberra, Australia. Springer.

[Martin-Löf, 1975] Martin-Löf, P. (1975). An intuitionistic theory of types: predicative part. In H.Rose and J.C.Shepherdson, editors, *Logic Colloquium'73*.

[Martin-Löf, 1984] Martin-Löf, P. (1984). *Intuitionistic Type Theory*. Bibioplois, Napoli. Notes of Giowanni Sambin on a series of lectures given in Padova.

[McMillan, 1992] McMillan, K. L. (1992). *Symbolic Model Checking: An approach to the state explosion problem*. PhD thesis, Carnegie Mellon University.

[McMillan, 2005] McMillan, K. L. (2005). Smv WWW page. `http://WWW-2.cs.cmu.edu/~modelcheck/smv.html`.

[Meisels and Saaltink, 1997] Meisels, I. and Saaltink, M. (1997). The Z/EVES reference manual (for version 1.5). Technical Report TR-97-5493-03d, ORA Canada.

[Merriam and Harrison, 1997] Merriam, N. and Harrison, M. (1997). What is wrong with guis for theorem provers.

[Milner, 1980] Milner, R. (1980). *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin.

[Milner, 1989] Milner, R. (1989). *Communication and Concurrency*. International Series in Computer Science. Prentice Hall. SU Fisher Research 511/24.

[Moller and Stevens, 1999] Moller, F. and Stevens, P. (1999). *The Edinburgh Concurrency Workbench (Version 7.1)*. Laboratory for Fundations of Computer Science, University of Edinburgh.

[Monroy-Borja, 1997] Monroy-Borja, R. (1997). *Planning Proofs of Correctness of CCS system*. PhD thesis, University of Edinburgh.

[Naumov et al., 2001] Naumov, P., Stehr, M.-O., and Meseguer, J. (2001). The HOL/NuPRL proof translator: A practical approach to formal interoperability. The 14th International Conference on Theorem Proving in Higher Order Logics, Edinburgh, Scotland.

[Nesi, 1997] Nesi, M. (1997). Mechanising a modal logic for value-passing agents in HOL. *Electronic Notes in Theoretical Computer Science 5, 1997*, 5.

[Nesi, 1999] Nesi, M. (1999). Formalising a value-passing calculus in HOL. *Formal Aspects of Computing*, 11(2):160–199.

[Nielsen, 2005] Nielsen, J. (2005). Ten usability heuristics. `http://WWW.useit.com/papers/heuristic/heuristic_list.html`.

[Nordström et al., 1990] Nordström, B., Petersson, K., and Smith, J. M. (1990). *Programming in Martin-Löf's Type Theory*, volume 7 of *International Series of Monographs on Computer Science*. Oxford University Press, New York, NY.

[Nordström et al., 2000] Nordström, B., Petersson, K., and Smith, J. M. (2000). Martin-löf's type theory. In Abramsky, S., Gabbay, D. M., and Maibaum, T., editors, *Handbook of Logic in Computer Science*. Oxford University Press.

[Owre et al., 1997] Owre, S., Rushby, J., and Shankar, N. (1997). Integration in PVS: tables, types, and model checking. In Brinksma, E., editor, *Tools and Algorithms for the Construction and Analysis of Systems TACAS '97*, number 1217 in Lecture Notes in Computer Science, pages 366–383, Enschede, The Netherlands. Springer-Verlag.

[Owre et al., 1992] Owre, S., Rushby, J. M., and Shankar, N. (1992). PVS: A prototype verification system. *Lecture Notes in Computer Science*, 607:748–752.

[Pang et al., 2002] Pang, J., Callaghan, P., and Luo, Z. (2002). An approach to verification of domain properties based on LF. In *Types 2002 WORKSHOP*, Netherlands.

[Pang et al., 2005a] Pang, J., Callaghan, P., and Luo, Z. (2005a). LFTOP: An LF-based approach to domain-specific reasoning. *Journal of Computer Science and Technology*, 20(4):526–535.

[Pang and Zhao, 2005] Pang, J. and Zhao, R. (2005). The features of list comprehensions of Haskell and their applications. *Journal of Computer engineering and applications (in Chinese with English abstract)*, 41 No.4:99–101.

[Pang et al., 2005b] Pang, J., Zhao, R., and Wang, H. (2005b). The higher-order features of Haskell and their applications. *Journal of Computer Science (in Chinese with English abstract)*, 32(6):167–168,198.

[Pang et al., 2006a] Pang, J., Zhao, R., and Wang, H. (2006a). Verification of semantic properties based on logical framework LF. *Journal of Computer Science (in Chinese with English abstract)*, 33(5):12–16,69.

[Pang et al., 2006b] Pang, J., Zhao, R., and Wang, Q. (2006b). The features of lazy evaluation of Haskell and their applications. *Journal of Computer engineering and applications (in Chinese with English abstract)*, 42 No.10:97–99,122.

[Park, 1969] Park, D. (1969). Fixpoint induction and proofs of program properties. *Machine Intelligence*, 5:59–78.

[Paulson, 1999] Paulson, L. (1999). Isabelle's logics. Technical report, Computer Laboratory, Cambridge University.

[Paulson, 2005] Paulson, L. (2005). Isabelle WWW page. `http://WWW.cl.cam.ac.uk/Research/HVG/Isabelle`.

[Paulson, 1987] Paulson, L. C. (1987). *Logic and computation: interactive proof with Cambridge LCF Cambridge Tracts in Computer Science 2.* Cambridge University Press.

[Paulson, 1994] Paulson, L. C. (1994). Isabelle: A generic theorem prover. *Lecture Notes in Computer Science*, 828:xvii + 321.

[Paulson, 2000] Paulson, L. C. (2000). Mechanizing UNITY in Isabelle. *ACM Transactions on Computational Logic*, 1(1):3–32.

[Paulson, 2001] Paulson, L. C. (2001). Mechanizing a theory of program composition for UNITY. *ACM Transactions on Programming Languages and Systems*, 23(5):626–656.

[Pfenning, 1999] Pfenning, F. (1999). Logical frameworks. In Robinson, A. and Voronkov, A., editors, *Handbook of Automated Reasoning.* Elsevier Science Publishers. In preparation.

[Plotkin, 1981] Plotkin, G. D. (1981). A structural approach to operational semantics. Technical Report DAIMI FN-19, Computer Science Department, Aarhus University, Aarhus, Denmark.

[Pollack, 2005] Pollack, R. (2005). Lego WWW page. `http://WWW.dcs.ed.ac.uk/home/lego`.

[Pratt, 1976] Pratt, V. (1976). Semantical considerations of floyd-hoare logic. In *Proc. 16th IEEE FOCS.*

[Pratt, 1982] Pratt, V. (1982). A decidable mu-calculus. In *Proc. 22nd IEEE FOCS.*

[Prawitz, 1965] Prawitz, D. (1965). *Natural Deduction, a Proof-Theoretic Study.* Lmqvist and Wiksell.

[Prawitz, 1973] Prawitz, D. (1973). Towards a foundation of a general proof theory. In P. Suppes *et al.*, editor, *Logic, Methodology, and Phylosophy of Science IV.*

[Prawitz, 1974] Prawitz, D. (1974). On the idea of a general proof theory. *Synthese*, 27.

[Project, 2002] Project, T. C. D. T. L. (2002). *The Coq Proof Assistant Reference Manual (version 7.3).* INRIA-Rocquencourt and CNRS-ENS Lyon.

[Project, 2004] Project, T. C. D. T. L. (2004). *The Coq Proof Assistant Reference Manual (version 8.0).* INRIA-Rocquencourt and CNRS-ENS Lyon.

[Ranta, 1994] Ranta, A. (1994). *Type-Theoretical Grammar*, volume 1 of *Indices.* Clarendon Press, Oxford.

[Ranta, 2005] Ranta, A. (2005). Gf WWW page. `http://WWW.cs.chalmers.se/~aarne/GF`.

[Rudnicki, 1992] Rudnicki, P. (1992). An overview of the Mizar project. In *1992 Workshop on Types for Proof and Programs.* Chalmers University of Technology.

[Saaltink, 1997] Saaltink, M. (1997). The Z/EVES user's guide. Technical Report TR-97-5493-06, ORA Canada.

[Safranek, 2001] Safranek, D. (2001). Verification tools for concurrent processes. Technical report, `http://WWW.fi.muni.cz/paradise/acp_tools`.

[Saibi, 1997] Saibi, A. (1997). Typing algorithm in type theory with inheritance. *Proc of POPL'97.*

[Saibi, 1999] Saibi, A. (1999). *Outils Génériques de Modélisation et de Démonstration pour la Formalisation des Mathématiques en Théorie des Types. Application ?la Théorie des Catégories.* PhD thesis, INRIA.

[Seaman and Felty, 1993] Seaman, J. and Felty, A. (1993). Proving properties about a lazy functional language with the coq proof development system. `http://WWW.site.uottawa.ca/~afelty/bib.html`.

[Seaman and Iyer, 1996] Seaman, J. and Iyer, S. (1996). An operational semantics of sharing in lazy evalutation. *Journal of Computer Programming*, 27(3):289–322.

[Sellink, 1994] Sellink, M. P. A. (1994). Verifying process algebra proofs in type theory. In Andrews, D. J., Groote, J. F., and Middelburg, C. A., editors, *Proceedings of the International Workshop on Semantics of Specification Languages (SOSL'93)*, Workshops in Computing, pages 315–339, London, UK. Springer.

[Shankar, 1986] Shankar, N. (1986). *Proof Checking Metamathematics.* PhD thesis, University of Texas at Austin.

[Slind and Boulton, 2000] Slind, K. and Boulton, R. (2000). Iterative dialogues and automated proof. In Gabbay, D. M. and de Rijke, M., editors, *Frontiers of Combining Systems 2 (Proceedings of the Second International Workshop, FroCoS'98, Amsterdam, The Netherlands, October 1998)*, volume 7 of *Studies in Logic and Computation*, pages 317–335. Research Studies Press.

[Smith, 1988] Smith, J. (1988). The independence of Peano's fourth axiom from Martin-Löf's type theory without universes. *Journal of Symbolic Logic*, 53(3).

[Soloviev and Luo, 2000] Soloviev, S. and Luo, Z. (2000). Coercion completion and conservativity in coercive subtyping. In *Annals of Pure and Applied Logic.*

[Sørensen and Urzyczyn, 1998] Sørensen, M. H. and Urzyczyn, P. (1998). Lectures on the Curry-Howard isomorphism. Technical report 98/14 (= TOPPS note D-368), Univ. of Copenhagen.

[Steve Bishop and et al., 2005] Steve Bishop, M. F. and et al., M. N. (2005). Tcp,udp,and sockets:rigorous and experimentally-validated behavioural specification volume 1: Overview. Technical Report UCAM-CL-TR-624, University of Cambridge.

[Stirling, 1992] Stirling, C. (1992). Modal and temporal logics. In Abramsky, S., Gabbay, D. M., and Maibaum, T. S. E., editors, *Handbook of Logic in Computer Science. Volume 2. Background: Computational Structures*, pages 477–563. Oxford University Press.

[Tarski, 1955] Tarski, A. (1955). A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 25(2):285–309.

[Thery et al., 1992] Thery, L., Bertot, Y., and Kahn, G. (1992). Real theorem provers deserve real user-interfaces. Technical Report RR-1684, Inria, Institut National de Recherche en Informatique et en Automatique.

[Trybulec, 2005] Trybulec, A. (2005). Mizar WWW page. `http://mizar.org/`.

[van Daalen, 1980] van Daalen, D. T. (1980). *The Language Theory of Automath.* PhD thesis, Technological University, Eindhoven.

[Vardi and Wolper, 1986] Vardi, M. and Wolper, P. (1986). An automata-theoretic approach. to automatic program verification. In *Proc. 1986 IEEE Symp. Logic Comput. Sci.*, pages 322–331, Cambridge.

[Vardi and Wolper, 1983] Vardi, M. Y. and Wolper, P. (1983). Yet another process logic. *LNCS*, 164:501–512.

[Victor, 1994] Victor, B. (1994). *A Verification Tool for the Polyadic π-Calculus.* Licentiate thesis, Department of Computer Systems, Uppsala University, Sweden. Available as report DoCS 94/50.

[Walukiewicz, 1993] Walukiewicz, I. (1993). *A Complete Deductive System for the μ-Calculus.* PhD thesis, Warsaw University.

[Walukiewicz, 1995] Walukiewicz, I. (1995). Notes on the propositional μ-calculus: Completeness and related results. Technical Report NS-95-1, BRICS, Dept. of Computer Science, Univ. of Århus, Denmark.

[Wenzel, 2002] Wenzel, M. M. (2002). *Isabelle/Isar – a versatile environment for human-readable formal proof documents.* PhD thesis, TU München.

[Wiedijk, 2001] Wiedijk, F. (2001). Mizar light for HOL light. *Lecture Notes in Computer Science*, 2152:378–393.

[Winskel, 1989] Winskel, G. (1989). A note on model checking the modal nu-calculus. In Ausiello, G., Dezani-Ciancaglini, M., and Rocca, S. R. D., editors, *Automata, Languages and Programming, 16th International Colloquium*, volume 372 of *Lecture Notes in Computer Science*, pages 761–772, Stresa, Italy. Springer-Verlag.

[Yu, 1999] Yu, S. (1999). *Verification of Concurrent Programs Based on Type Theory.* PhD thesis, University of Durham.

[Yu and Luo, 1997] Yu, S. and Luo, Z. (1997). Implementing a model checker for LEGO. In Fitzgerald, J., Jones, C. B., and Lucas, P., editors, *FME'97: Industrial Applications and Strengthened Foundations of Formal Methods (Proc. 4th Intl. Symposium of Formal Methods Europe, Graz, Austria, September 1997)*, volume 1313 of *Lecture Notes in Computer Science*, pages 442–458. Springer-Verlag. ISBN 3-540-63533-5.

Index