Machine Assisted Proofs for Generic Semantics to Compiler Transformation Correctness Theorems

Saif Ullah Khan

Master of Philosophy
Department of Computer Science
University of Edinburgh
1997

To My Parents

Abstract

This thesis investigates the issues involved in the creation of a “general theory of operational semantics” in LEGO, a type-theoretic theorem proving environment implementing a constructionist logic. Such a general theory permits the ability to manipulate and reason about operational semantics both individually and as a class. The motivation for this lies in the studies of semantics directed compiler generation in which a set of generic semantics transforming functions can help convert arbitrary semantic definitions to abstract machines. Such transformations require correctness theorems that quantify over the class of operational semantics. In implementation terms this indicates the need to ensure both the class of operational semantics and the means of inferring results thereon remain at the theorem prover level. The endeavour of this thesis can be seen as assessing both the requirements that general theories of semantics impose on proof assistants and the efficacy of proof assistants in modelling such theories.

Acknowledgements

First and foremost I would like to thank Kevin Mitchell who supervised me for my first four years, supplying me with many helpful hints and constructive criticisms. He also bore with me at a period of my life when my mental health deteriorated for which I am eternally grateful. Secondly I would like to thank Stuart Anderson an ever present of my life at the University since I first arrived in 1988, for taking over the supervision of my work when it was seemingly near its conclusion. The help and encouragement I received meant I was able to (finally!) complete this thesis. Special mention must go to Rod Burstall, my mentor through the entirety of my postgraduate studies. My all too brief encounters with him lifted my spirits at a time when they were desperately in need of a boost. I would also like to especially thank Thomas Kleymann (formerly Schreiber) for the many times he aided me in my Lego miseries. I also thank James Hugh McKinna, Randy Pollack and other members of the Lego club for their helpful ideas, various helpful office-mates Pietro Cenciarelli, Andrew Wilson, Dilip Sequeira and Masahito Hasegawa, Claudio Russo my long term friend whom I've known ever since I moved to Edinburgh, all the secretarial staff: Eleanor Kerse, Tracy Combe, Sam Falconer, Margaret Davis, Monika Lekuse, Lucinda McGill, Mairi McLennan, Bill Orrok and Angela Riddell and everyone else I met at LFCS.

On a personal note I must thank the following: Lisa Charlotte Bolton for lots of good advice and saving my life on numerous occasions, Nick Robinson for being a Gooner who it was alright to be "different" around, his charming wife Tomoko, Vicki Clayton my loving, gullible and tolerant friend — don't know where I'd be without you mate, Beaded de Mowbray for all those pinky perky beef jerkey bubbly times and breaking my Arsenal mug, Sally Mayne my partner and soul mate on many trips and escapades for her layed back yet enthusiastic lovely voice and e-mails and breaking my Arsenal mug, Elaine - Oh Elaine, you're beautiful, Moira MacKay for general revelry and letting me be a moustached lecher, Richard Elphee my one time room-mate and companion who breaks the records in smiling, dancing, laughing and general consumptious behaviour, Meg his darling darling sister, Tom "Come at ME!" Elphee his brother, Udi the interesting and very congenial friend who didn't learn enough sensible Urdu from me to go to India — Kya baat hai?, Shlair Teimourian the lovely tea drinker who was so nice to me on so many occasions, Alison Kjellström her infinitely energetic friend, Davva Angel the lanky vegan who let me star in his film and helped me in my maddest tempest, Sally his lovely partner and Margaret his voluptuous sister, Lucy Ketchin the bees knees and a charming little girl, Sarah "Listen — Take care" Eckersley for being one of the few people I loved as a sister (Awww), Iggy Smith and Penny the mad and thin people, John Wallace the man with the Wok, Tom "Don't call me doughboy" Hirons just for the smile on his face, Ruth Kingshott for her wit and gorgeous Banoffie Pie, Nick Florey and Alison Blackhall for just about everything else, Sarah Jessica Longley for letting me slowly uncoil when her seeing eye was far next to happier times, Miss Gina Ward for so many insane and yet disturbingly hilarious comments, Miss Amelia Davies for laughing when I was joking, Andy Clamp and Keith "scratch card" Butler, Leah Bain and Vicki Hageman, Chris Binnie and Duncan Moore for their help in my homeless hardships, Donna and Paul Douglas for "sorting us out" with a roof over my head, Bruce Danraj for his . . . well yes, Mark McCauley for his supremely captivating style and being one of the funniest people I have ever met, Shane "Chancellor" Gorman, Helen Cairns for an interesting time in Paris and a lovely slap up meal, Beth, Julia Lappin, Maggie Loach for all those "conversations", Julie Abrahams for being a friend I desperately needed when times were bad and for letting me stay in her flat when I was homeless (not forgetting Lucy Head!), Thom McKean for letting me stay at his place in San Francisco, Julie Doak who I sadly broke down in front of but who gave me some of the best days of my life, not forgetting the Mancunian and oh so pretty Anita, Rita, Sue, Bob too, and all the rest of the Forrest People and their parents in no particular order, Rachel Louise Wilson — We'd fallen out of the sky the day before and kissed on the sandy mile with the clean sea behind. We walked across fields of soft earth that fed our bellies and I realised why my body was a smile. Finally, Dr. Blackwood, Dr. Tim Brown, Dr. Shah, Dr. Cunliffe, Dr. Fletcher, Dr. Last, everyone at the Richard Verney Centre, Ward 1 and 2 of the Royal Edinburgh Hospital, Mum and Dad who kept me going in some of the darkest days of my life, brothers Khurshid, Saad, Fraaz, sisters Romisa, Rubina, Memoona, nieces Afifa, Maaria all who supported me in every way possible and for their priceless patience with me, the rest of the family and rabble of Barking people, the close friends I haven't mentioned already: Muazzam, Muazzamel, Rehan, Jay, Asad, Saad, and finally Perry Farrell, Seaman, Dixon, Winterburn, Bould, Adams, Keown, Viera, Merson, Bergkamp, Wright, Davis, Michael Thomas, Lukic, Wenger, Rioch, Graham, Bertie Mee and Herbert Chapman may he rest in peace.

Declaration

I declare that this thesis was composed by myself and that the work contained therein is my own, except where explicitly stated otherwise in the text.

(*Saif Ullah Khan*)

Table of Contents

Chapter 1

Introduction

Several proof checkers [GM93, Des88, BB$^+$96a] exist today that allow reasoning about operational semantics, viewed as inductively defined rules, in a style consistent with the presentation of inductive reasoning in the literature. Typically a proof checker is supplied with a package allowing one to specify semantics, construct derivations and use rule induction. One aspect these packages do not cover is reasoning generalized over the class of inductively defined operational semantics. This is not a common necessity — unless we wished to provide support for functions taking semantics as arguments and returning them as results and then prove theorems of such functions that quantify over the whole class. The motive behind this seemingly obscure circumstance lies in the studies of semantics directed compiler generation. There, generic functions are used to convert arbitrary operational semantics to abstract machine equivalents — formalisms that are intermediate between semantics and compilers. Such translation functions require "consistency theorems" to ensure that the antecedent and consequent semantics elicit the same behaviours — both being equivalent in some sense. These theorems quantify over the class of operational semantics due to the generic nature of the transformations. Facilitating the proof of such theorems motivates this study.

We shall explore issues in the development of a feasible implementation of a package, in the LEGO [LP92] proof assistant, to specify and logically reason about operational semantics both individually (as in other packages) and collectively as a class. In specifying semantics, they must, as a prerequisite, be represented as first order objects within the framework. In other words, there must be a means of syntactically enumerating the elements of this class in the proof assistant. However, having only a lexicographic description, semantics are rendered meaningless unless a notion of their connotations in computation is supplied alongside to give a mathematical basis upon which reasoning can occur. There is a generic quality inherent in this notion of meaning as it must be applicable to any semantics as well as providing a foundation upon which to reason about the class of semantics as a whole in the sense that theorems quantifying over all semantics can be both specified and proven in the context of a proof assistant.

Before continuing further we need to embellish the concepts introduced above. In the next section we discuss the forms of operational semantics that are the focus of this thesis, section 1.2 describes proof checkers in general and our choice, Lego [LP92], for its expressive and flexible type theory as well as an exercise in its suitability as a plausible environment within which to implement the concepts above. Section 1.3 illuminates the work that motivates our own — general transformations converting operational semantics closer to abstract machine equivalents in a provably correct fashion. In section 1.4 we highlight the shortcomings of the traditional implementations of inductive rule reasoning mechanisms with regard to supporting the kind of generalization mentioned above as well as discussing features necessary to do so. Section 1.5 introduces and explains the aspects of Lego that can be used to realise such features and section 1.6 provides an outline of the rest of the thesis.

1.1 Operational Semantics

Operational Semantics [Plo81] is an established medium for expressing the meaning of programming languages in use today. The term is used loosely to refer to the class of formalisms in which the behaviour of programming languages is defined by how programs are evaluated to results. In time a style of operational semantics developed in which inference rules were used as the defining mechanism — inspired by [Plo81] where the term “structural operational semantics” was coined to refer to inference rules providing an inductive definition of some relation defined as the phrases of the language. The structural aspect of the semantics is in the use of inductive rules, guided by the abstract syntax of the programming language, to define the relation. Since the relation defined was single step, the style of semantics is referred to as *transitional*.

In Kahn’s work on Natural Semantics [Kah87, Kah88] a proof-theoretic vein is taken. Programming languages are defined in terms of deductive systems in the form of a sequent calculus [Sza69] where a deductive system is a set of natural deductive rules [Pra65] of sequents. The conclusions of natural semantics rules were, in general, statements relating program states to canonical forms. For this reason this style of operational semantics is referred to as *relational*. Using inference rules to represent such relations was inspired by Martin-Löf in [ML84].

Relational semantics are highly abstract and have been successfully used to define real languages [MTH90]. As a paradigm it has been particularly fitted to specify programming languages where evaluation is an inherent factor.

The relational style is also expressed by inductive definitions, which are used extensively in computer science to define sets having a natural formulation in terms of an inductive closure condition [Acz77]. Several forms of inductive definition exist including classical (or positive), co-inductive, bi-inductive and negative [CC92]. We shall concentrate on the classical form because it is the most frequently used. Plotkin's structured operational semantics and Kahn's natural semantics have a common representation as inductive definitions. In view of this, we shall henceforth use the terms operational semantics and inductive definition synonymously.

1.2 Lego: A Proof Assistant for Constructionist Logics

Proof development systems (alternatively *proof assistants*, *proof checkers* or *theorem proving assistants*) are computer implementations of logical reasoning mechanisms within which true propositions can be rigorously verified with machine assistance. Formulae proved in this manner become reliable and undeniably correct since every aspect of a proof is formalized. With progressive use of such tools, theories of whole sections of mathematics can be coded within a proof assistant. A myriad of implementations exist equipped for a variety of logics and applications including HOL [GM93, BCG91] for higher order logic and hardware verification, LAMBDA [MH91] for the same purpose and Isabelle [Pau94, Pau93] a generic theorem prover supporting a wide variety of logics.

Lego [LP92] is an interactive proof checker designed and written in Edinburgh in the functional programming language ML [HMM86]. A number of type systems are implemented by it including the Edinburgh Logical Framework [HHP87], the Calculus of Constructions [CH88] and the Extended Calculus of Constructions [Luo90]. Proofs are developed in a natural deduction system by *refinement*. Lego provides an expressive language with which to formalize mathematical notions. That language is a framework for higher-order type theories with dependent types (including strong sum types, useful for natural representations of abstract data types and mathematical theories [Luo91a, LPT89]), inductive types, type universes and universal polymorphism[1].

[1]For a summary of type theoretic concepts see Appendix A

Applications of Lego include a proof of the Chinese Remainder Theorem [McK92], a formalization of the Z specification notation [Mah91] as well as program specification and program correctness proofs [BM91, Luo91b, Hof91, McK92]. More recently, work on issues such as verification calculi for imperative programs [Sch97] and representing modular specification languages [Mah96] has taken place. A brief summary of the system is provided in Appendix B. A more detailed exposition on the theory and implementation of Lego can be found in [Pol95].

The expressiveness of Lego's type theory is one of the reasons for the choice of Lego in this thesis. This exercise also provides an assessment of the suitability of Lego for the development of a theory of operational semantics.

1.3 Generic Semantics Transformations

The problem of constructing correct compilers from precise semantic definitions has become increasingly familiar in recent years [Jon80]. During this time a trade-off has been realised between the efficiency of the eventual compiler and the difficulty of the correctness proof [HM91].

On the one hand a hand crafted compiler may be implemented by techniques specialized to a language and machine — the implementor having a degree of flexibility as a result. The correctness of the compiler can be difficult to show however since the resulting program may have an obscure relationship to the original semantics. On the other hand an implementation can be coded directly from the semantics — affording mechanization in both the compiler's construction and the proof process. Unfortunately, the performance of an implementation produced in this way is usually far outshone by a hand crafted compiler since the flexibility available in the hand crafted case have given way to the rigours of the direct encoding.

In the light of this, work by Hannan and Miller [HM91, HM90] has focussed on bridging this gap by founding a set of semantics *transformations* general enough to convert a significant proportion of operational semantics closer to a representation intermediate between high level mathematical description and low level compiler, keeping results abstract from any particular machine architecture. The chosen intermediate is the abstract machine (examples of which are Landin's SECD machine [Lan64] and Milner's SMC machine [Mil76]) typically used as a low level architecture for a wide range of programming languages since it is a paradigm that facilitates portability, code optimizations and machine code generation [Car84].

The generic transformations in [HM91] preserve the original qualities of semantics in the sense that they are each accompanied by theorems stating a relationship (usually equivalence) between input and output. All the examples of such transformations can be seen in [HM91], one of which (Branch Elimination) is discussed in detail in chapter 6. To illustrate the idea we shall introduce an artificial example of a semantic transformation here. Suppose we have a semantics *LessSem* describing the less-than relation

$$\frac{}{0 \ < \ suc\, n}$$

$$\frac{n \ < \ m}{suc\, n \ < \ suc\, m}$$

and we wish to convert this to a semantics to describe the greater-than relation written as

$$\frac{}{suc\, n \ > \ 0}$$

$$\frac{m \ > \ n}{suc\, m \ > \ suc\, n}$$

We could apply the function *Perm2* defined in the following way. For any semantics *Sem* specifying a two place predicate P, *Perm2* takes all rules in *Sem* mentioning $P\,x\,y$ (where x and y are well-formed arguments to P) and substitutes the new formula $P'\,y\,x$ (where P' is a new predicate not mentioned in *Sem* with the reversed arity of P). The result is a semantics *Perm2*(*Sem*) related by the transformation theorem

$$\forall Sem.\, \forall P.\, \forall x, y.\, Sem \vdash \ P\ x\ y \iff Perm2(Sem) \vdash \ P'\ y\ x$$

where $\vdash$ denotes derivability in the proof theoretic sense. The theorem above quantifies over the set of semantics and so our notion of meaning must also span to this wider context. In terms of the theorem above, the symbol $\vdash$ should be taken to denote a *generic* notion of derivability in the sense that it is a relation applicable to any semantics and one that we can reason about.

The transformations in [HM91] are meant to be used in stages to automatically build abstract machine equivalent semantics. The correctness theorems ensure that the results are immediately correct. Demonstrations are provided in the paper of this process, converting simple evaluation semantics to abstract machines implementing those evaluators. The application of *Perm2* to *LessSem* is an example of one stage in the kind of process in Hannan and Miller's work. From the general theorem for *Perm2* we can infer the theorem

$$LessSem \vdash \ x \ < \ y \iff Perm2(LessSem) \vdash \ y \ > \ x$$

The Hannan and Miller transformations are small in number and can readily be automated. Our interest is in the machine assisted proofs of the attached correctness theorems. In such circumstances, it would be possible to automate both compiler construction and correctness proof. With transformations used in stages and each stage being machine verified, the proof immediately follows. It is to this end to be able to blend all aspects of operational semantics reasoning, that we wish to create a workbench within a proof assistant context for this task. Taking the issues in this section into account, the mandatory features of such a package include

- A syntactic formalism to express operational semantics. This allows us to define the elements of the class of operational semantics and transformations thereon. Furthermore, it must make quantification over this class possible.
- The syntactic descriptions must be furnished with their intended meaning as inference rules (including an induction principle) in some shape or form. This permits the usual kinds of reasoning to take place including proof tree construction and the proofs of theorems connected to individual semantics such as monogenicity and the like.
- The characterization of meaning must extend to the general class of operational semantics. We must be able to define and reason about a generic relation over this class to express the fact that a formula f is derivable from a semantics Sem. With this we are able to specify and prove the transformation correctness theorems we are interested in.

The points above necessitate a separation between the idea of a semantics as syntactic notation and a semantics as meaningful entity while the notion of meaning itself must be generalised to the extent that theorems can be proven of functions on semantics. We shall refer to inductive definitions expressed as syntactic entities as *inductive specifications*. We shall refer to the foundation of a general workbench for operational semantics in Lego as a "*General Theory of Operational Semantics*" or simply as a *general theory*.

1.4 Formalizing at Different Levels

Schema already exist in a number of proof assistants in which semantics can be specified syntactically where meaning is derived from functions that produce theorem prover rules. Whilst this gives us a way of defining transformations for inductive definitions as well as a means of reasoning with them in the proof assistant, we cannot prove or even define the kinds of transformation correctness theorems we are interested in. This is a consequence of the absence of the general relation of derivability we mentioned in the previous section. In other words, the functions that provide the meanings to inductive specifications do not themselves provide a way of reasoning about inductive rules in general. Their purpose is purely imperative in asserting inference rules in a proof assistant's context for specific inductive specifications rather than providing a generic derivability relation so essential to a machine formalization of Hannan and Miller's transformation theorems. This is related to work in "shallow" and "deep" embeddings in [BG$^+$92]. There, a distinction was made between representing the syntax of hardware description languages and their semantic functions explicitly within the logic of the theorem prover (deep embeddings) and representing them in a shallow manner by only defining the semantic structures within the logic. The syntactic aspects being parsed directly into semantic features by the user-interface. The implementation suggested here is closer to a shallow embedding. We provide an illustration of the points above with reference to the HOL system [GM93].

In HOL we can represent an inductive specification as a list of pairs. Each pair represents a rule. The first object of the pair is a list of strings (the premisses) and the second object is a string (the conclusion). We are then provided a procedure that takes these structures and produces the corresponding inference rules at the theorem prover level with an associated rule induction principle for them. The function is called `new_inductive_definition` in HOL [Mel92, CM92, Mel88]. We can cater for Hannan and Miller transformations by writing functions in HOL to manipulate the string representations of inductive definitions. To reason about these syntactic entities we can invoke `new_inductive_definition` to give us the appropriate inference rules in the global HOL context. To demonstrate the process, take the following specification for the *LessSem* semantics of the previous section

```
[                    ([],
            (* --------------*)
              "lt 0 (Suc n) "),



               (["lt m n"],
            (*---------------*)
          "lt (Suc m) (Suc n)")]
```

We can reason about this by applying `new_inductive_definition` to obtain the rules

$$\frac{}{lt\;0\;(Suc\,n)}$$

$$\frac{lt\;m\;n}{lt\;(Suc\,m)\;(Suc\,n)}$$

$$\frac{R\;0\;(Suc\,m) \quad \forall m,n.\;\; R\;m\;n \;\rightarrow\; (R\;(Suc\,m)\;(Suc\,n))}{\forall m,n.\;\; lt\;m\;n \;\;\rightarrow\;\; R\;m\;n}$$

in the HOL context. We can define the *Perm2* transformation of the previous section by defining a function that takes a string specification like the one above and performs the appropriate manipulations to construct a new string consistent with the action of *Perm2*. Let us say that the result of applying such an implementation of *Perm2* to our representation of *LessSem* yields the new list

```
[                    ([],
            (* --------------*)
              "gtr (Suc n) 0"),



               (["gtr n m"],
            (*---------------*)
          "gtr (Suc n) (Suc m)")]
```

where the new predicate name `gtr` stands for the greater-than relation. We can now again supply HOL with a means of reasoning with the new rules by applying `new_inductive_definition` to give the new rules

$$\frac{}{gtr\;(Suc\,n)\;0}$$

$$\frac{gtr\;n\;m}{gtr\;(Suc\,n)\;(Suc\,m)}$$

$$\frac{R\;(Suc\,n)\;0 \quad \forall n,m.\;\; R\;n\;m \;\rightarrow\; (R\;(Suc\,n)\;(Suc\,m))}{\forall n,m.\;\; gtr\;n\;m \;\;\rightarrow\;\; R\;n\;m}$$

This kind of set-up in HOL allows both a syntacticly manipulable representation for inductive definitions as well as a ready means of reasoning with individual inductive definitions. Top down derivations can be constructed using HOL's basic operators by continuous rule applications until axioms are reached. The rule induction principle is immediately available allowing a straightforward way of proving theorems of a set of inductive rules. The presentation of the inference rules in HOL is also fairly close to the notation of the literature.

However it falls short of providing a general theory of operational semantics since it precludes the possibility of defining a general derivability relation in HOL. The `new_inductive_definition` command cannot itself be reasoned about at the theorem proving level (its soundness for example is a meta theorem). It operates as a command which takes a string specification and simply asserts the right inductive rules and rule induction principle in the theorem proving context. Derivations may be built using HOL's basic reasoning mechanisms but derivations of *arbitrary* semantics cannot be related. The consequence being that there is no means of specifying a transformation correctness theorem such as

$$\forall \mathit{Sem}.\, \forall P.\, \forall x, y.\, \mathit{Sem} \vdash\; P\; x\; y \quad\Longleftrightarrow\quad \mathit{Perm2}(\mathit{Sem}) \vdash\; P'\; y\; x$$

because $\vdash$, the generic derivability relation has no representation in HOL. In terms of supplying machine checked proofs of correctness for programming language transformations, the best that can be done is a proof in HOL of a relation between two specific semantics. In the demonstration above this amounts to a machine assisted proof of the theorem

$$\forall n, m.\, (\mathit{lt}\; n\; m) \quad\Longleftrightarrow\quad (\mathit{gtr}\; m\; n)$$

which is not quantified over the set of semantics but an instance of the general theorem. The effect of an inability to state and prove general transformation correctness theorems is that when using transformations for a particular inductive specification *Sem* say, we do not immediately have a proof that the meaning of *Sem* is related to the meaning of the result of applying a transformation to *Sem*. Full mechanization of the proof process becomes impossible.

Despite this drawback, some machine assistance can be given to prove specific theorems like the one above. Tactics are useful accompaniments to each transformation aiding the automation of the correctness proofs at this level. For example, a general tactic to solve the theorem above would be given the rules and predicate names for the two semantics involved and then first apply the $\forall$-introduction rule (to strip off the quantifiers) continuously until it is no longer applicable. It would then simplify the double implication to turn the goal into a conjunction of implications. For each implication goal it would apply the relevant rule induction rules (for `lt` and `gtr` in the example) and proceed thereon.

By providing tactics in this way, we are able to provide some machine assistance for proofs of specific theorems but they are not guaranteed to succeed. In fact, the tactic discussed above *is* guaranteed to succeed and its success amounts to a proof of the general theorem — but we cannot reason about tactics in HOL. In general, the best that can be done is to design tactics in such a way that they can be used to prove the widest subset of semantics covered by a general transformation correctness theorem. Although we may have meta theorems that a given tactic always succeeds, there is no means of verifying this within HOL since tactics themselves cannot be reasoned about at the theorem proving level.

Having understood the problems of a ground level implementation for inductive definitions, what are the essential features of a generalized one? To begin with we need a language to represent inductive specifications as first order objects. Rules and rule sets are easy to represent since the format in different inductive rule sets is relatively uniform. However there is a range of notation in the atomic formulae in different rule sets. The grammar for formulae in *LessSem* would differ from that of a semantics for lists for example. We need a formalism to account for this and first order term algebras are a simple yet effective solution. A first order term algebra consists of a set of sorts, function names and signatures coupled with a simple set of term forming rules. A signature acts similarly to a type signature and the term forming rules mirror typing rules. The signature for *LessSem* looks like

$$\{0 : \rightarrow nat,\ suc : \ nat \rightarrow nat,\ <: \ nat \rightarrow nat \rightarrow \phi\}$$

where ϕ is understood as the sort for propositions. In this way inductive definitions with differing notations can all be specified in a universal formalism. One last point worth bearing in mind is that meta-variables in inductive definitions can be represented as variables in the first order term algebra.

Once we have a first-order representation of inductive definitions, we need to assign a way of reasoning with them in a generic fashion, obtaining a means of constructing proof trees and an induction principle, both ranging over the class of inductive definitions. One of the main contentious issues confronting us is in representing in some form the set defined by an inductive definition. An inductive set of rules defines the least set of formulae closed under the rules in a rule set. Closure is typically simple to codify but a notion of "leastness" always proves trickier to supply in machine implementations. We now focus our attention on integrating these ideas into the concrete structures of Lego.

1.5 Representation in Lego

Expressiveness in Lego is manifested in its type theory which includes nested type universes as well as dependent and inductive types. Such rich types are ideal for abstract and recursive structure representations, which may be particularly helpful in representing the set of inductive specifications. The notions of set in set theory have a natural equivalence with notions of type in type theory and so the set of inductive specifications can be represented by a type in Lego. Furthermore, the structures in an inductive definition such as premisses, conclusions, rules and rule sets can all be defined in some form of inductive manner. A formula is composed of sub-terms, themselves recursively made up from sub-terms for example. There are several ways of using dependent types, especially strong sum types, in Lego. Abstract data types are naturally expressed using dependent sums. Semantic specifications fit this category. Another use for strong sum types is a consequence of the Curry-Howard isomorphism [How80]. Types can be thought of as propositions in the underlying intuitionistic logic. In summary, the dependent sum $\Sigma x : A.P$ corresponds to the existential formula $\exists x : A.P$ in constructive logic where the first element x is a well typed term and the second element P (mentioning x) is a proposition. Furthermore, a proof of the proposition $\exists x.P$ is equivalent to the construction of a pair (a, p) where the types of a and p are respectively A and $[a/x]P$ (substituting a for x in P)[2]. This is a powerful tool that can be used to describe complex theories and structures — an induction principle and derivability relation perhaps.

Assessing the suitability of Lego is an important issue in this thesis. Features such as the expressiveness and flexibility of the type theory, how easily inductive definitions can be read and operated upon and the space and time efficiencies possible are all factors in the assessment.

One of the most contentious questions is how flexible Lego is. We need to define inductive specifications, reason about specific inductive definitions thereon, define transformations, prove correctness theorems and reason about the class of operational semantics in general all within one package. It seems as though to focus on one aspect must mean a detraction from another. It is important to combine all of the above in a cohesive manner so that any deficiencies are kept to a minimum and support is provided to cover for them where necessary.

For example one of the problems we may expect is a certain amount of verbosity in encodings of inductive specifications. This is due to the necessitation of a

[2]cf. Appendix A

specification language (first order algebra) for them, the implementation of which will probably be more complex than the notation normally used in a description in the literature. There are certain features of Lego that help remove some of the more tedious aspects of the theorem proving process. One of these is type synthesis which allows the user to omit certain arguments in a term which can be inferred from its context. Aside from this Lego provides a fairly sparse interface in this respect.

1.6 Outline of the Thesis

The next chapter introduces the basic concepts of first order signatures and their implementation in Lego. It then discusses the suitability of the various types available for the definition of well formed terms in a first order term algebra. This involves a comparison between dependent sum and product types. The resulting notation for well formed terms in the general theory of operational semantics is cumbersome and makes opaque reading compared with the usual presentations of terms in the literature — A consequence of Lego's primitive interface.

Chapter 3 sets out to redress the balance and improve the readability of notation. It starts by analyzing the ways in which Lego's in-built features can be exploited to allow clearer representations of terms. Taking the intrinsic limitations of this solution into account, we adopt the approach of adding a quotation parser for well formed terms to Lego. This allows users to write terms as quotations[3] and the system translates these to Lego terms. The parser itself must be generic, accommodating all first order signatures, and so new Lego commands are given and documented allowing users to define their own grammars for their signatures. The printing routines built into Lego are also augmented to display terms as quotations.

In chapter 4 we complete the formalisation of inductive specifications by extending our general theory with the syntactic formalisms for side conditions, rules, rule sets and inductive specifications. A generic basis for the semantics of inductive definitions is then given. This includes notions of substitution, the meaning attributed to side conditions, how they are proved, and the main semantic constructions: a type denoting the set of formulae derivable from inductive specifications (the constructors of which enable, among other things, proof tree development) and a generic induction and recursion operator for inductive definitions. From this basic material we define a library of routines and theorems useful in the

[3]streams of characters parenthesised by quotation marks.

domain of normal inductive definition reasoning similar to those provided in other theorem proving environments [CM92]. These include proof tree de-constructors, a case analysis facility (for "backward" reasoning), automatic substitution (for interpreting), as well as a means of performing induction on the depth of inference in an obvious manner. In section 4.4 derivation construction is found to be very slow in Lego. This sluggishness is explained and we describe how to rectify the problem using Lego's in-built facilities. The chapter concludes with an illustration of how the general theory can be extended to cope with a larger variety of side conditions.

In chapter 5, all the elements of the general theory of operational semantics are brought together by specifying and reasoning about an example inductive definition, a small functional programming language `ExpSem`, in Lego. We go on to explain how derivations can be built for `ExpSem` in both top down and bottom up fashion. Induction on the depth of inference is exemplified by proving monogenicity for `ExpSem` and an example of case analysis is performed in a proof of the equivalence of the `let` and function application constructs of functional languages. Examples of the use of the various library utilities introduced in chapter 4 are also provided.

We show that the class of operational semantics can be reasoned about in chapter 6 by defining one of Hannan and Miller's transformations and proving its correctness theorem within our general theory. The construction of the transformation and the subsequent theorem and proof also act as a template for future encodings of transformations on operational semantics.

Finally chapter 7 discusses the various issues arising from the work in this thesis including an assessment of Lego and the capability of the general theory within it. Future directions of the work such as interface extensions and the implementations of other transformations are also discussed.

Chapter 2

Well-Formed Terms in a General Theory

In this chapter we discuss the implementation of some of the basic constituents of operational semantics in a theorem proving context. The implementation allows quantification over all these basic constituents. As described in the previous chapter, we seek to set up a foundation in the proof assistant which allows us to specify and enumerate the set of operational semantics, and also to define and reason about the meaning attributed to such constructions. Since Lego is a type-based theorem prover, defining the set of operational semantics amounts to providing a type for them. The implementation in this chapter is developed to the point of describing terms and formulae, the basic fabric of the theory of operational semantics.

The sections below are structured in the following way. Section 2.1 outlines the preliminary definitions we need to be able to represent terms as objects in a first order term algebra, and section 2.2 explores the ways in which we may specify these concepts in Lego.

2.1 Sorts, Identifiers and First Order Signatures

The descriptions provided below are based on the work on operational semantics transformations by Hannan and Miller's [HM91]. We assume knowledge of universal algebra [ST87]. Before being able to specify the terms mentioned above it is necessary to define a notion of first order signature, and before this we need a notion of sorts and function names.

2.1.1 Sorts and Function Names

Let $\mathcal{S}$= $\{s_0, s_1, \dots\}$ be a countable set of names or sorts. Let $\mathcal{F}$= $\{f_0, f_1, \dots\}$ be a countable set of function names. Let o stand for the distinguished sort of (atomic) logical formulae. As we shall see later, we must be able to distinguish between sorts and similarly between function names. That is we must be able to define equality for sorts (function names). We can simply define them in Lego as record types (a variation of an inductive type) where there is only one constructor which takes a natural number as its sole argument. In Lego we have

```
Record [Sort:Type(0)]
Fields
 [sort:nat];

[Formula = make_Sort (suc zero)];

Record [FIdent:Type(0)]
Fields
 [id:nat];
```

where the type `nat` is the expected inductive type for natural numbers and `Formula` represents the sort o in our Lego formulation. Equality between sorts (or identifiers) is simply natural number equality. In the example operational semantics provided in chapter 1 we make the following definitions.

```
[Natural = make_Sort (suc (suc zero))];

[Zero = make_FIdent zero];
[Suc  = make_FIdent (suc zero)];
[LessThan = make_FIdent (suc (suc zero))];
```

2.1.2 First Order Signatures

The purpose of a first order signature is to ensure the well-formedness of terms in a similar way that type signatures are used for type checking. The definition of a first order signature we shall use is a simplification of the one given in [HM91]. For a set S, let S^* denote the set of sequences $<s_1, \dots, s_n>$ such that $s_1, \dots, s_n \in S$ for $n \geq 0$, where $<>$ is the empty sequence.

Definition 2.1 (First Order Signatures)

A First Order Signature *is a finite map from $\mathcal{F}$ to $\mathcal{S}^* \rightarrow \mathcal{S}$. Let us write $f\!:<s_1, s_2, \ldots, s_n> \rightarrow s$ to represent a binding in such a map. Then $<s_1, s_2, \ldots, s_n>$ is the* domain *of f and s is the* range *of f.*

Let $\mathcal{SIG}$ stand for the set of all first order signatures. We now need functions to interrogate signatures for a given function name. For any $Sig \in \mathcal{SIG}$, and any $f \in \mathcal{F}$, let $dom_{Sig}(f)$ return the domain of f in Sig and let $rng_{Sig}(f)$ return the range of f in Sig. In Lego we can define $\mathcal{SIG}$ as a list of bindings where a binding is a triple: a function name, its domain and its range:

```
[FSig = list | (prod FIdent
                     (prod  (list|Sort)
                            Sort
                     )
               )
];
```

We could also represent $\mathcal{SIG}$ more directly as the type of functions from `FIdent` to `list|Sort` to `Sort` but using lists and products make it straightforward to manipulate and reason about signatures by utilizing the respective elimination operators. For instance, the *dom* and *rng* functions above can be defined very simply.

Using a list to represent a finite map as we do means we allow multiple bindings for the same function name. We must be able to handle the constraint that a signature is effectively a finite map in the theorem prover. Failure to do so would be synonymous with giving a function two possibly different functional types in a type signature, which is clearly possible with this loose definition for signatures. The anomaly is curtailed by ensuring that the implementations of the two projection functions dom_{Sig} and rng_{Sig} for any given signature consistently return only one set of values. Two functions in Lego fulfil these requirements given any signature. Their types are

```
IDSort1 : FSig -> FIdent -> (list | Sort)

IDSort2 : FSig -> FIdent -> Sort
```

and are both defined using list recursion on the signature. It is at this point where we need to test for equality between function names. Bindings closer to the head of the list effectively over-write ones further down. This guarantees the fact that a signature is effectively treated as a finite map.

As with all functions in Lego these projections must be total. The functions they realize are naturally partial. If a function name does not appear in a signature then no value should be returned. Partiality can be achieved in Lego using dependent sum-types, but as we shall discuss in section 2.2.1 we wish to avoid their use. As an alternative, the functions return default values. The empty list of sorts `(nil|Sort)` in the case of `IDSort1` and a special sort `botSort` in the case of `IDSort2`. This exception value is defined as

```
[botSort = make_Sort zero];
```

This is not a major shortcoming but it is an inconvenience. As long as one is aware of and avoids including terms whose sort is this exception value, no problems are posed. In practice, dealing with partiality becomes tedious. Following the running example, the appropriate signature is

```
[NatSig = (cons (Pair Zero
                      (Pair (nil|Sort)
                            natural))
          (cons (Pair Suc
                      (Pair (cons natural (nil|Sort))
                            natural))
          (cons (Pair LessThan
                     (Pair (cons natural (cons natural (nil|Sort)))
                           Formula))
          (nil|(prod FIdent (prod (list|Sort) Sort))))))];
```

so for example the final entry denotes the fact that the function `LessThan` takes a list of two `naturals` to form a `Formula` term. With these preliminary building blocks we shall now be able to define well formed terms.

2.2 Well Formed Terms

This section compares and contrasts the various ways in which the specification of the class of well-formed terms can be described in Lego's type scheme. For a given signature $Sig \in \mathcal{SIG}$, we define a well-formed term to be an object in the first order term algebra of *Sig*. Such an algebra is described as follows.

Definition 2.2 (Well Formed Terms)

Assume $Sig \in \mathcal{SIG}$ is a signature and $\mathcal{V}$ is an $\mathcal{S}$-sorted set of variables. Then the set of Well Formed Terms *with respect to Sig is the least set closed under the following rules.*

1. *A variable $v \in \mathcal{V}_s$ is a* Well Formed Term *of sort $s \in \mathcal{S}$*

2. *A function application $f(x_1, x_2, \ldots, x_n)$ is a* Well Formed Term *of sort $s \in \mathcal{S}$, where $f \in \mathcal{F}$, $dom_{Sig}(f) = < s_1, s_2, \ldots, s_n >$, $rng_{Sig}(f) = s$, and $s_1, s_2, \ldots, s_n, s \in \mathcal{S}$, $n \geq 0$, and each x_i is a* Well Formed Term *(with respect to Sig) of sort s_i for $1 \leq i \leq n$*

From this specification we can see that the type for *Well Formed Term* in Lego will take a signature as a parameter. We can also see this definition is inherently inductive so we may expect to use inductive types to describe it. The critical issue is how to encapsulate the final condition in the second rule above using Lego's type system. Conditional constraints on types can be dealt with by using dependent types. Lego has two such type constructs. Dependent sum types and dependent product types [Luo92]. We start by investigating the sufficiency of the former.

2.2.1 Well Formed Terms Using Dependent Sum Types

We begin by giving an introduction to types, dependent sum types and how we can use them for our purposes. For a summary of type theoretic concepts used here see Appendix A. However in the main text we introduce those concepts we need as they are used to make the exposition self-contained.

Let $\mathcal{TYPE}$ be a universe of types, i.e. some collection with types as members. For types T and T' in $\mathcal{TYPE}$ we write $a : T$ to mean a is a member of T and $T \to T'$ for the type of total functions mapping members of T to members of T'. The universe $\mathcal{TYPE}$ is closed under function formation (functional values). In addition to types we consider families of types indexed by a particular type. For example for some type T we may have a family F such that $F(t)$ is a type for every $t : T$. Using this notion we can define a new type constructor called a dependent sum or *sigma type* written $\Sigma x : T.\, F(x)$ whose members are pairs (a, b) where $a : T$ and $b : F(a)$. For any type T and family F indexed by T, $\Sigma x : T.\, F(x)$ is a type. Henceforth we abuse notation by writing $T : \mathcal{TYPE}$ to mean that T is a type.

We can utilize this to specify well-formed terms by defining a sigma type where the first argument is an object x whose type defines the set of all terms (any

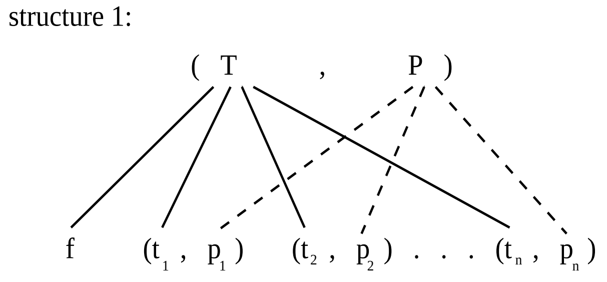

Figure 2.1: The Structure of Well Formed Function Applications

variable and any function application) regardless of any signature, and the second argument is a type that represents a "proof" that x is well-formed. Such a proof can be constructed in Lego by taking advantage of the Curry-Howard isomorphism [How80] [Luo91c] that states that types can be thought of as propositions in the calculus of constructions. The set of well-formed terms would be defined as the set of pairs (x, P) where x is a term (variable or function application) and P is a proof that x is well-formed according to a given signature. What we are essentially doing is defining the universal set of terms, and then defining well-formed terms (for a given signature) as pairs of a term and a witness to the fact that the term is in the subset of this universal set that includes all well-formed terms.

The next question is how do we define such a set? We need two types, a type $Term$ and a family of types $\mathit{WF}(t)$ (for $t : \mathit{Term}$) for well-formedness proofs. We can break this problem down further by looking at definition 2.2. We need to know how to define well-formed variables and well-formed function applications. In the case of variables we simply have to provide a variable v and prove it is a member of the set of variables $\mathcal{V}_s$ for known s. From definition 2.2, we see that the set of well-formed terms is an inductive set and that function applications are built recursively from their sub-terms. The structure in figure 2.1 is a graphic representation of the abstract syntax tree of a well-formed function application (T, P) where T is the term $f\, t_1 \cdots t_n$ (for $n \geq 0$) and P the well-formedness proof. Both T and P are built recursively from their sub-terms t_i and sub-proofs p_i (for $0 \leq i \leq n$) respectively.

Whilst this gives a systematic way in which to construct well-formed terms it is a tedious process to manually provide well-formedness proofs. Ideally proof building would be automated. This means providing functions that act as *constructors* for well formed terms — with the basic components we have separate constructors for terms and proofs. The new constructor for well-formed function applications for example would be used to build the term in figure 2.1 by taking the function name f and the appropriate list of well-formed terms $(t_1, p_1) \cdots (t_n, p_n)$ to give the well-formed term (T, P). Having these "well forming" constructors means well-formed terms can be treated as having a recursive structure rather than being dependent pairs — provided we complement this recursiveness with an elimination operator for well-formed terms to give the ability to define recursive functions for them as well as the ability to prove theorems thereon in an inductive manner. The summary of this enquiry is the acknowledgement that types of both terms and proofs are most naturally some form of inductive type, and well-formed terms themselves can be represented as if they were also built inductively.

One question remains. Which proposition should be used to express well formedness? If we again look at definition 2.2, we see that the formula needed can be expressed as "Term t is a well-formed term of sort s with respect to signature *Sig*". We shall now describe the types we need for terms, proofs and well formed terms.

Definition 2.3 (Terms)

The set $\mathcal{T}$, of Terms *is the least set closed under the following rules.*

1. *A variable $v \in \mathcal{V}_s$ is in $\mathcal{T}$, for $s \in \mathcal{S}$*
2. *If $f \in \mathcal{F}$, and $t_1, t_2, \ldots t_n$ are in $\mathcal{T}$, then $f(t_1, t_2, \ldots t_n)$ is in $\mathcal{T}$, for $n \geq 0$.*

Let the symbols $\mathcal{S}$, $\mathcal{F}$, $\mathcal{V}$, $\mathcal{SIG}$, $\mathcal{T}$ and $\mathcal{TYPE}$ denote the types of the sets they represent. We use the notation $\Pi x : t.y$ to stand for the dependent product type for type variable x, and types t and y and assume we have the inductive type $list : t \rightarrow Type$ (where $t : Type$) of lists whose constructors are *nil* for empty lists (where t is clear from the context) and :: the infix list concatenation constructor. The type $\mathcal{T}$ can be defined as an inductive type with two constructors

$$\begin{aligned} var &: \Pi s : \mathcal{S}.\, \mathcal{V}_s \rightarrow \mathcal{T} \\ fun\_app &: \mathcal{F} \rightarrow (list\ \mathcal{T}) \rightarrow \mathcal{T} \end{aligned}$$

Note that there is no mention of any signature, so there is no notion of well-formedness. Let us now turn to the types of proofs. It is simplest to describe this as two sets.

Definition 2.4 (Well Formed Term Proofs)

For $s : \mathcal{S}$ and $Sig : \mathcal{SIG}$, let $\mathcal{W}^s_{Sig}$ stand for the set of functions from $\mathcal{T}$ to proofs of well-formedness of the given term as detailed in definition 2.2. Also for $n \geq 0$, let $\mathcal{WL}^{<s_1, s_2, \ldots, s_n>}_{Sig}$ stand for the set of functions taking a sequence of terms $< t_1, t_2, \ldots, t_n >$ and returning the sequence

$$< \mathcal{W}^{s_1}_{Sig}, \mathcal{W}^{s_2}_{Sig}, \ldots, \mathcal{W}^{s_n}_{Sig} >$$

If we use the list type to represent sequences, let the symbols $\mathcal{W}^{s}_{Sig}$ and $\mathcal{WL}^{<s_1,s_2,\ldots,s_n>}_{Sig}$ again denote the type of the sets they represent and in addition use $dom_{Sig} : \mathcal{F} \rightarrow (list\ \mathcal{S})$ and $rng_{Sig} : \mathcal{F} \rightarrow \mathcal{S}$ to represent the functions for dom_{Sig} and rng_{Sig} respectively, we can define the sets

$$\mathcal{W}^{s}_{Sig} : \mathcal{T} \rightarrow\ Type$$

$$\mathcal{WL}^{<s_1,s_2,\ldots,s_n>}_{Sig} : (list\ \mathcal{T}) \rightarrow\ Type$$

by mutual induction using the constructors

$$\begin{array}{rcl}
WFvar & : & \Pi Sig : \mathcal{SIG}. \\
 & & \quad \Pi s : \mathcal{S}. \\
 & & \quad\quad \Pi v : \mathcal{V}_s. \\
 & & \quad\quad\quad \mathcal{W}^{s}_{Sig}\ (var\ s\ v) \\
WFfun\_app & : & \Pi Sig : \mathcal{SIG}. \\
 & & \quad \Pi f : \mathcal{F}. \\
 & & \quad\quad \Pi tl : list\ \mathcal{T}. \\
 & & \quad\quad\quad (\mathcal{WL}^{(dom_{Sig}\ f)}_{Sig}\ tl) \rightarrow \\
 & & \quad\quad\quad\quad \mathcal{W}^{(rng_{Sig}\ f)}_{Sig}\ (fun\_app\ f\ tl) \\
WFnil & : & \Pi Sig : \mathcal{SIG}. \mathcal{WL}^{<>}_{Sig}\ nil \\
WFcons & : & \Pi Sig : \mathcal{SIG}. \\
 & & \quad \Pi s : \mathcal{S}. \\
 & & \quad\quad \Pi sl : list\ \mathcal{S}. \\
 & & \quad\quad\quad \Pi t : \mathcal{T}. \\
 & & \quad\quad\quad\quad \Pi tl : list\ \mathcal{T}. \\
 & & \quad\quad\quad\quad\quad (\mathcal{W}^{s}_{Sig}\ t) \rightarrow \\
 & & \quad\quad\quad\quad\quad\quad (\mathcal{WL}^{sl}_{Sig}\ tl) \rightarrow \\
 & & \quad\quad\quad\quad\quad\quad\quad \mathcal{WL}^{<s,sl>}_{Sig}\ (t :: tl)
\end{array}$$

The interesting constructor here is *WFfun_app*. Given a signature *Sig*, function name f, and list of terms $t_1, t_2, \ldots, t_n$, the application is well-formed and of sort $rng_{Sig}(f)$ if you can supply a proof that the terms $t_1, t_2, \ldots, t_n$ are of sort $dom_{Sig}(f) = s_1, s_2, \ldots s_n$ respectively. Such a proof is obtained using the constructors *WFnil* and *WFcons*. Finally we can express the type of well-formed terms and lists thereon as the types

$$\begin{array}{rcl}
WFTerm & = & \Pi Sig : \mathcal{SIG}.\ \Pi s : \mathcal{S}.\ \Sigma t : \mathcal{T}.\ \mathcal{W}^{s}_{Sig}\ t \\
WFList & = & \Pi Sig : \mathcal{SIG}.\ \Pi sl : list\ \mathcal{S}.\ \Sigma tl : list\ \mathcal{T}.\ \mathcal{WL}^{sl}_{Sig}\ tl
\end{array}$$

Once we have these types we need the "well forming" constructors mentioned earlier to automate proof construction. We can easily provide a proof that a variable is well-formed using *WFvar*. To construct well-formed function applications automatically one would want to provide a function name f and the well-formed terms $(t_1, p_1), (t_2, p_2), \ldots, (t_n, p_n)$ of the right arity and construct the term part of the application (using *fun_app*, f *and* $t_1, t_2, \ldots t_n$). The proof for the application would use all the proofs $p_1, \ldots, p_n$ as its sub-proofs in the application of the constructor *WFfun_app*. This is essentially what is depicted in the structure of figure 2.1. It is helpful to look at the types of two functions that would perform the operations above, one for variables and one for function applications.

$$\begin{aligned}
\mathit{varWF} &: \Pi \mathit{Sig} : \mathcal{SIG}.\ \Pi s : \mathcal{S}.\ \Pi v : \mathcal{V}_s.\ \ \mathit{WFTerm\ Sig}\ s \\
\mathit{appWF} &: \Pi \mathit{Sig} : \mathcal{SIG}. \\
&\qquad \Pi f : \mathcal{F}. \\
&\qquad\quad (\mathit{vector}\ (\mathit{map}\ (\lambda s : \mathcal{S}.\ \ \mathit{WFTerm\ Sig}\ s)\ (\mathit{dom}_{\mathit{Sig}}\ f))) \rightarrow \\
&\qquad\qquad \mathit{WFTerm\ Sig}\ (\mathit{rng}_{\mathit{Sig}}\ f)
\end{aligned}$$

We can see that *varWF* can simply be defined to take arguments pertaining to a variable and use *var* and *WFvar* to make a well-formed term. Well-formed function applications are more complicated. In the type above, we assume we have the map function for lists *map* and we assume we have the type *vector* whose objects are list-like structures that allow one to concatenate objects of different types. The type of `vector` is `(list|Type(0))->Type(1)`. The types of the elements of the vector are given in the argument applied to the type *vector* above. In this case the vector contains the elements whose types are successively $\mathit{WFTerm\ Sig}\ s_1, \ldots, \mathit{WFTerm\ Sig}\ s_n$ where $s_1 :: \cdots :: s_n$ is equal to $\mathit{dom}_{\mathit{Sig}}\ f$.

We can understand the operation of *appWF* if we again refer to the diagram in figure 2.1. For a given signature *Sig* and function name f, the third argument to *appWF* is the vector containing the well-formed sub-terms $(t_1, p_1), \ldots, (t_n, p_n)$ where the sorts of $t_1, \ldots, t_n$ equal respectively $s_1, \ldots, s_n$ equal $\mathit{dom}_{\mathit{Sig}}(f)$. With these functions the user is freed from providing well-formedness proofs and they can view well-formed terms as if the proofs are not present.

As mentioned previously, we should provide an elimination operator to complement *varWF* and *appWF*. Such a construction, like other elimination rules for inductive types, includes a quantification over all functions from well-formed terms to types. It also takes two functions — one from well-formed variables to the chosen type and a similar function for well-formed function applications. The type of such an operator could be written as

$$\begin{array}{llll}
\mathit{WFElim} & : & \Pi \mathit{Sig} : \mathcal{SIG}. & \\
& & \quad \Pi P:\ \Pi s:\mathcal{S}.\ (\mathit{WFTerm}\ \mathit{Sig}\ s) \rightarrow \mathcal{TYPE}. & \\
& & \quad\quad \Pi PL:\ \Pi sl: \mathit{list}\ \mathcal{S}.\ (\mathit{WFList}\ \mathit{Sig}\ sl) \rightarrow \mathcal{TYPE}. & \\
& (1) & \quad\quad\quad (\Pi s:\mathcal{S}. \Pi v:\mathcal{V}_s.\ P\ s\ (\mathit{varWF}\ \mathit{Sig}\ s\ v)) \rightarrow & \\
& (2) & \quad\quad\quad\quad (\Pi f:\mathcal{F}. & \\
& & \quad\quad\quad\quad\quad \Pi tl:\ \mathit{WFList}\ \mathit{Sig}\ (\mathit{dom}_{\mathit{Sig}}\ f). & \\
& & \quad\quad\quad\quad\quad (PL\ (\mathit{dom}_{\mathit{Sig}} f)\ tl) \rightarrow & \\
& & \quad\quad\quad\quad\quad\quad (P\ (\mathit{rng}_{\mathit{Sig}} f)\ (\mathit{appWF}\ \mathit{Sig}\ f\ tl_v))) \rightarrow & \\
& & \quad\quad\quad\quad\quad\quad\quad \Pi s:\mathcal{S}. \Pi \mathit{wft}:\ \mathit{WFTerm}\ \mathit{Sig}\ s.\ \ P\ s\ \mathit{wft} &
\end{array}$$

where PL is a function from *WFList*s to a chosen type and tl_v is a *WFList* converted to a vector. The parenthesized type expression (1) is the type of the variable case function and expression (2) is the type for the function application one. One notable advantage with well-formed terms defined in terms of dependent sum types is that since we have a type for all terms (well and ill formed) we can prove theorems of all well-formed terms by proving the case for all terms since the former is a subset of the latter.

There is however a practical drawback with this type specification. If we define well-formed terms in this way, we are constructing them by including objects whose sole purpose is to be a witness to well formedness. Note that in figure 2.1 each node of the term tree has a proof along with each actual term. It is important to keep the size of objects to a minimum since the larger objects get, the slower it is to process them in a machine. Recall we are using sigma types here to define pairs, where the first element is an object in the larger set $\mathcal{T}$ of both well and ill formed terms, and the second argument is essentially a witness to the fact that the given object is in the subset $\mathcal{W}^s_{Sig}$ of $\mathcal{T}$ of well-formed terms of sort s with respect to signature *Sig*. Ideally we would wish to be able to build terms in such a way so that all and only all elements of $\mathcal{W}^s_{Sig}$ can be constructed without the need for any witnesses. We discuss how this can be done in the next section.

2.2.2 Well Formed Terms From Dependent Product Types

We can use another dependent type in Lego to define well-formed terms in a more direct manner. The dependent product types (or Π types) mentioned in the previous section can be used to define this set. To start, recall that the definition 2.2 for such objects is inductive. So we would expect the type we require to be an inductive one. Again we need two constructors for variables and function applications. If we define these using Π-types, we no longer need witness constructions, providing a means for creating compact terms. We specifically use them so that they effectively implement the constraint on function applications in definition 2.2. Let us define the inductive type $\mathcal{TERM}$. It is parametric on signatures and sorts and has two constructors both exploiting Π-types:

$$\begin{array}{rcl} \mathit{var} & : & \Pi \mathit{Sig} : \mathcal{SIG}. \\ & & \quad \Pi s : \mathcal{S}. \\ & & \quad\quad \Pi v : \mathcal{V}_s.\ \mathcal{TERM}^{s}_{\mathit{Sig}} \\ \mathit{fun\_app} & : & \Pi \mathit{Sig} : \mathcal{SIG}. \\ & & \quad \Pi f : \mathcal{F}. \\ & & \quad\quad (\mathit{vector}\ (\mathit{map}\ (\lambda s : \mathcal{S}.\ \mathcal{TERM}^{s}_{\mathit{Sig}})\ (\mathit{dom}_{\mathit{Sig}}\ f))) \rightarrow \\ & & \quad\quad\quad \mathcal{TERM}^{\mathit{rng}_{\mathit{Sig}}\ f}_{\mathit{Sig}} \end{array}$$

Note that now we parameterize the constructors with a signature. Note also that *fun_app* is now defined in such a way that the arguments supplied in the function application are the terms of the right sort with respect to the signature. To see this more clearly, recall that the vector type allows objects of different types to be concatenated, and that the types of the elements of a vector are dictated by the argument that the type *vector* is applied to. So in this case, the types of the elements of the vector specified above are successively

$$\mathcal{TERM}^{s_1}_{\mathit{Sig}}, \mathcal{TERM}^{s_2}_{\mathit{Sig}}, \ldots, \mathcal{TERM}^{s_n}_{\mathit{Sig}},$$

where $s_1 :: s_2 :: \cdots :: s_n$ is equal to $\mathit{dom}_{\mathit{Sig}}(f)$. Defining well-formed terms in this way obviates the need for extra constructions. In Lego we define the type above in a slightly different way since the `Inductive` command in Lego does not allow vectors to be used in this way for defining constructors. Rather than using vectors, we define two mutually inductive types

```
[Term : FSig -> Sort -> Type(0)]
[Tlist: FSig -> (list|Sort) -> Type(0)]
```

of well-formed terms and well-formed term lists. We can think of the type `Tlist` as being a vector type made specific for terms (terms of different sort still have a different type). There are four constructors. The two for `Terms` and two for `Tlists`. They are

```
[var:{FS|FSig}
     {n:nat}
     {s:Sort}Term|FS s]

[fa:{FS|FSig}
    {f:FIdent}
    {tl:(Tlist|FS (IDSort1|FS f))} Term|FS (IDSort2|FS f)]

[tnil:{FS|FSig}Tlist|FS (nil|Sort)]

[tcons:{FS|FSig}
       {s|Sort}
       {sl|(list|Sort)}
       {t:(Term|FS s)}
       {lt:(Tlist|FS sl)} (Tlist|FS (cons s sl))];
```

The set of variables is defined using natural numbers (again because we wish to be able to distinguish between such objects).

The two constructors `tnil` and `tcons` define the type `Tlist` of term vectors, and `fa` takes a function name with an appropriate term vector (as dictated by the signature via the function `IDSort1`) to return a well-formed function application. Note this definition allows us to build terms of sort `botSort`. This is a consequence of the totality of Lego functions, imposing exception results on the functions `IDSort1` and `IDSort2`. The use and abuse of this fact is left as a user prerogative. Such values do need to be accounted for in certain circumstances as we shall see in chapter 6.

In our example we can construct terms such as

```
fa Zero (tnil|NatSig)


fa Suc  (tcons (fa Suc (tcons (fa Zero (tnil|NatSig))
                       (tnil|NatSig)))
        (tnil|NatSig))
```

to represent the natural numbers 0 and 2. It should be appreciated from the above that although we can now describe terms relatively concisely, such terms are very hard to comprehend as syntactic objects. Even for very simple terms with the helpful spacing above it is hard to recognize the second term as a representation for the number 2.

This is one of the effects of defining objects in the general theory. Here we are able to define terms from many different signatures. This accounts for the proliferation of the signature parameter `NatSig` above, and the fact that we have to make the structure of terms explicit. If we now extrapolate and imagine we are constructing more complex terms from a larger signature than `NatSig`, we can expect to come across terms that become unreadable. The consequence of this is that using a system where the terms look so complex would become a very laborious and frustrating task. The next chapter is devoted to studying ways in which we can provide systems support to make it easier to read and write the kinds of terms introduced in this chapter.

Chapter 3

Object Language Support, Parsing and Pretty Printing

It is an immediate conclusion from the last chapter that if we expect our general encoding of operational semantics to be effective, we must present the terms of this encoding in a notation closer to that used in mathematical representations of such semantics. For example, we would wish to be able to write the inequality $0 < (suc\, n)$ directly into Lego rather than the cumbersome and unreadable

```
[n = var|Nat zero natural]
fa LessThan (tcons (fa Zero  (tnil|Nat))
            (tcons (fa Succ  (tcons n
                              (tnil|Nat)))
            (tnil|Nat)))
```

This problem arises as a by-product of the generality and manipulability of the encoding. If terms must be defined at a level in which they are first order objects in a theorem prover (and specifically Lego) we must define them as objects of some form of datatype with given constructors. It follows that we must expect to include these constructors (`fa`, `var`, `tnil` and `tcons` in our case) in the constitution of a term. If we also add the proviso that the datatype is generic parameterized on a signature, we must also presume that this signature will appear in some part of the body of a term. In the coding of terms in Lego, the signature is an argument to each constructor of the type for terms. But due to the type synthesis facility in Lego, we may omit stating this information for certain constructors (`fa` and `tcons`) when it can be inferred from the sub-terms involved. Sorts are another parameter to term constructors hidden in this way. To appreciate the expediency of type synthesis, the term above written without exploiting this utility would be

```
fa Nat LessThan (tcons Nat
                       natural
                       (cons Sort natural (nil Sort))
                       (fa Nat Zero (tnil Nat))
                (tcons Nat
                       natural
                       (nil Sort)
                       (fa Nat Succ (tcons Nat
                                           natural
                                           (nil Sort)
                                           (var Nat zero natural)
                                    (tnil Nat)))
               (tnil Nat)))
```

The next sections discuss the ways in which a clearer representation of terms can be written and presented in Lego's user interface.

3.1 Approximation 1: Lego's In-Built Utilities

Lego has two main features we can use to build terms in a more succinct manner. We have seen that type synthesis can be used to a great extent to hide parameters such as sorts and signatures as much as possible. The next step we can take is to extend the context of Lego with a set of definitions, each being a macro for each possible function application with respect to a given signature. In this way, we can essentially abbreviate a term as a macro where any parameters and constructors are omitted. In our example, we would provide the signature `Nat` with the macros

```
[zro = fa Zero (tnil|Nat)]
[succ = [n:Term|Nat natural]fa Succ (tcons n (tnil|Nat))]
[lessthan = [n1,n2:Term|Nat natural]
             fa LessThan (tcons n1 (tcons n2 (tnil|Nat)))]
```

Given these definitions we can then write $0 < (suc\, 0)$ as the clearer term

```
lessthan zro (succ zro)
```

Having to repeat this process for every signature is time consuming so Lego can be extended with a new command that adds these macros every time a new

```
Signature Nat = zro      :()->natural,
                succ     :(natural)->natural,
                lessthan:(natural,natural)->Formula ;
```

Figure 3.1: Example Signature Command in Lego

signature is defined. The command takes a sugared representation of a signature. Its form is

$$\begin{array}{rcl} \textbf{Signature} <\!name\!> & ::= & \\ f_1 & : & (s_1^1, s_1^2, \ldots, s_1^{n_1}) \longrightarrow s_1 \\ f_2 & : & (s_2^1, s_2^2, \ldots, s_2^{n_2}) \longrightarrow s_2 \\ \vdots & & \vdots \\ f_m & : & (s_m^1, s_m^2, \ldots, s_m^{n_m}) \longrightarrow s_m \end{array}$$

where $<\!name\!>$ is the name of the signature, s_i is a sort for $0 \leq i \leq m$, each s_j^k is a `Sort` for $0 \leq j \leq m$ and $0 \leq k \leq n_j$, and each f_i is an `FIdent` for $0 \leq i \leq m$. The example would be written in Lego as shown in figure 3.1.

A call to **Signature** invokes an in-built Lego function that adds the definitions of the signature and macro functions to the global Lego context. If the identifiers and sorts are not already defined in Lego, the command automatically adds them as new ones.

There are still a number of problems if we rely on this facility. The macro

```
lessthan zro (succ zro)
```

is still not as succinct or clear as $0 < (suc\, 0)$. The form and notation of terms is still dictated by Lego. Only alphanumerical symbols can be used to build terms, all operator-like symbols (`zro`, `succ` and `lessthan`) are prefix ones, variables are still verbose and the only kind of parentheses are "(" and ")". This can in turn lead to further reading difficulties. If we were to extend our example signature with a conditional functional

```
ifthenelse:(bool,natural,natural)->natural
```

where `bool` is a new sort with two functionals `True:()->bool` and `False:()->bool`, we would represent the mathematical term

$$suc\, suc\, n \;<\; (\mathit{if}\; \mathit{true}\; \mathit{then}\; (suc\; (suc\; (suc\; 0)))\; \mathit{else}\; (suc\; 0))$$

as

```
lessthan (succ (succ (var|Nat zero natural)))
         (ifthenelse  True
                      (succ (succ (succ zro)))
                      (succ zro))
```

losing any non-prefix non alphanumeric notation and any helpful keywords. Another complication can be foreseen in that if a macro term were to be broken down and operated on by a Lego function the system will return the fully expanded object and not the equivalent macro. Any advantage gained before would be lost.

This is in essence, a limitation of Lego's interface. It has no facility with which to support customized notations for Lego constructions. A way to relax these restrictions is necessary. The next sections provide details of how other theorem proving assistants have dealt with similar problems, and how we achieve the same in Lego.

3.2 Object Language Support in Other Theorem Provers

Similar problems to those above, arise in the HOL theorem proving system [BCG91]. The primitive objects in HOL are well typed ML (see [HMM86]) terms representing either variables, function abstractions or function applications. Other operators are supplied to enrich the logic and are also themselves well typed ML terms. Before describing the structure of terms it is necessary to define the notion of type in HOL.

Definition 3.1 *There are two primitive constructor functions for HOL* `type`*s:*

- *Type variables:* `mk_vartype: string->type` *where the string is a succession of asterisks followed by a number or identifier.*
- *Compound types:* `mk_type: (string # type list)-> type` *such that for a type expression* `mk_type(‘name‘,[`$\sigma_1; \ldots; \sigma_n$`])` *the name is an identifier and type operator in the current theory whose arity is n.*

For example, the type *nat* → *bool* would be written as

`mk_type(‘fun‘,[mk_type(‘nat‘,[]); mk_type(‘bool‘,[])]).`

As can be seen, this is a very complicated expression defining a fairly simple type. As types become more complex, readability suffers. To this end HOL is supplied with a *quotation parser* allowing the user to write concrete HOL expressions using the usual abstract notation. *Quotations* are streams of ASCII characters enclosed by a pair of quotation marks. The parser translates quotations into concrete HOL expressions in the following way:

Definition 3.2 *Type quotation forms and their translation:*

- *Type variables are of the form* `‘‘:*...’’` *and translate to* `mk_vartype(‘*...‘)`.
- *Function type quotations* `‘‘:`$\sigma_1 \rightarrow \sigma_2$`’’` *translate to* `mk_type(‘fun‘,[`$\tau_1$`;` τ_2`])` *where* τ_1 *and* τ_2 *are the translations of* `‘‘:`σ_1`’’` *and* `‘‘:`σ_2`’’` *respectively.*
- *Type constant quotations* `‘‘:op’’` *translate to* `mk_type(‘op‘,[])`

Terms are built and quoted in a similar manner to types. We start by defining the primitive constructors for terms, give an example term and then define the forms of quotations for terms.

Definition 3.3 *Primitive terms are built using four constructors:*

Variables: `mk_var: (string # type)-> term` *where* `mk_var(x,`σ`)` *evaluates to a variable* `x` *of type* σ*.*

Constants: `mk_const:(string # type)-> term` *where* `mk_const(c,`σ`)` *evaluates to a constant* `c` *with type* σ *if* `c` *is a valid constant and* σ *is its generic type.*

Function abstractions: `mk_abs: (term # term)-> term` *where* `mk_abs(`x, t`)` *evaluates to a term representing the abstraction* $\lambda x.\, t$ *provided* x *is a variable.*

Combinations: `mk_comb:(term # term)-> term` *where* `mk_comb(`t_1, t_2`)` *is a combination* $t_1\ t_2$ *provided* t_1 *is of type* $\sigma_1 \rightarrow \sigma_2$ *and* t_2 *is of type* σ_1 *for types* σ_1 *and* σ_2*.*

An example term highlights the need for a better interface for reading and writing terms. The function application $(\lambda x : bool.\, x)F$ where F is the boolean false constant is written as

```
mk_comb (mk_abs (mk_var(‘x‘, mk_type(‘bool‘,[])),
                 (mk_var(‘x‘, mk_type(‘bool‘,[])))),
            mk_const (‘F‘,mk_type(‘bool‘,[])))
```

Again we could assign identifiers to parts of the expression and work with these macros as we did in section 3.1 for Lego but as before this technique has its limitations. To absolve the user from this problem, quotations can be used to abstract away from the concrete syntax.

Definition 3.4 *Quotations for terms:*

Variables: ``x:σ''

Constants: ``c:σ''

Abstractions: ``\x. t''

Combinations: ``t1 t2''

More operators in the logic exist with their quoted form so that formulae such as

$$\forall x\, y.\, x \ <\ y \ \Rightarrow\ \exists z.\, x\ +\ z\ =\ y$$

are written as

```
``!x y. x < y  ==>  ?z. x + z = y''
```

In addition to a quotation parser, the system presents its output in quoted form wherever possible. This *pretty printing* sets the user in an environment almost free from the underlying representations of HOL objects. It is an important part of the utility since the output of a theorem prover must be sufficiently understandable for the reader to be able to reason within the system fluently.

The final significant facility provided by HOL in this respect is the ability to intermingle concrete terms within quotations. This is called *anti-quotation* and a concrete term is written as such by prefixing it with a "^" (caret) mark. For example we can write the quotation

```
``\x. x + ^(mk_comb (....))''
```

It is sometimes necessary to make use of anti-quotation since some complex concrete expressions have no quoted form. As we can see, writing complicated terms as anti-quotations can begin to detract from any advantages gained from this utility but we can tidy such quotations up by first making use of a quoted form for `let` expressions that binds the concrete term to an identifier and then using the identifier in the body of the `let` expression:

```
``let a = ^(mk_comb (.....))
  in
      \x. x + a''
```

The HOL experience is useful in that it can provide a basis for object language support facilities to abstract away from the concrete syntax for the Lego `Terms` introduced in the previous chapter. The main difference in our case is that our generic framework allows one to define an innumerable number of signatures for which different notational customizations may apply. A typical user may expect the notation for one operational semantics to use different symbols and structures from another. In HOL, the language of the parser can be thought of as fixed and in our case there is no one convenient syntax to cater for the multitude of definable semantics. As a result it seems we need to generalize the notion of a parser to allow for differing customizations of terms for different users and/or signatures.

3.3 Object Language Support via Grammar and Lexicon

Reviewing the material in the previous section we can affirm that our goal is to add a generic quotation parser and a compatible pretty printer to Lego. The range and domain of these respective utilities are the `Term` type constructs introduced in the previous chapter. The parser must be general in the sense that the range of the parser is not a single set T of terms as in HOL, but a $T^{\mathcal{S}}_{\mathcal{SIG}}$ indexed set ($\mathcal{SIG}$ and $\mathcal{S}$ being the set of signatures and sorts defined in the previous chapter). This immediately suggests the parser should be a function on signatures and sorts as well as quotations.

Furthermore, we would also expect the syntax of a quotation corresponding to one set T^{s}_{Sig} of well formed terms to look different from a quotation associated to a different set $T^{s'}_{Sig'}$. Quotations for the set of well formed naturals for instance would look unlike quotations for boolean expressions. They are formed in accordance to distinct *languages*. A language can be thought of as being composed of a lexicon of symbols and a grammar of sentence forming rules. In our example we could define them as in figure 3.2 where the elements of the lexicon are the *terminals* of the language and each production in the grammar is composed of a *non-terminal* on the left hand side and either terminals or non-terminals on the right. Note the parallel between this language and the definition of the signature in figure 3.1. The functional map

```
lessthan:(natural,natural)->Formula
```

has a simile in the grammatical rule

$$Formula \longrightarrow natural \text{ ``<'' } natural$$

$$\begin{array}{rcl}
\textit{Lexicon} & = & \{\text{“0”},\ \text{“suc”},\ \text{“<”}\} \\
\\
\textit{Grammar} & = & \\
\textit{Formula} & \longrightarrow & \textit{natural}\ \text{“<”}\ \textit{natural} \\
\textit{natural} & \longrightarrow & \text{“0”} \\
\textit{natural} & \longrightarrow & \text{“suc”}\ \textit{natural}
\end{array}$$

Figure 3.2: Language for natural numbers

which suggests a close relationship between signatures and context free grammars; something we shall exploit in section 3.5.2.

Another requirement of the quotation parser is to provide a means of defining variables succinctly. Recall the constructor `var`

```
var : {FS|FSig}natural->{s:Sort}(Term|FS s)
```

We would prefer the syntax of variables to be an alphanumeric quantity as normally portrayed in the literature rather a term built using constructors.

A further necessity is to support the antiquotation facility described in the previous section. This allows one to write concrete Lego constructions within the scope of a quotation. This is an important facility in Lego. It allows quotations to reference terms outside their scope (defined in the global context for example).

Parsing sentences to concrete Lego expressions is not the only requirement on the system. It must also *unparse* such expressions and print them in their quoted form. In this way the interface is completed as the user only reads and writes terms as quotations. On implementing the parser and unparser it would be best to show that for any quotation q, the formula

$$Parse(Unparse(Parse(q))) \ = \ Parse(q)$$

holds to ensure no information is lost in either procedure. However, this theorem is difficult to prove because of the complexity of the functions involved.

The final requirement is related to the transformations discussed in chapters 1 and 6. We would expect that we should be able to preserve the quoted representations of terms after transformations have been applied to supplied semantics. Informally we can define this to be the commutativity of the diagram in figure 3.3. If we have a semantics Sem and transformation T and a notion of a function $Pretty$ mapping Sem to its user friendly parallel, then applying T then $Pretty$

$$\begin{tikzcd}
\text{Sem} \arrow[r, "T"] \arrow[d, "\text{Pretty}"] & T(\text{Sem}) \arrow[d, "\text{Pretty}"] \\
\text{Pretty}(\text{Sem}) \arrow[r, "T_{\text{Pretty}}"] & \text{Pretty}(T(\text{Sem}))
\end{tikzcd}$$

Figure 3.3: Transformation Commutativity Diagram

to Sem is equivalent to applying $Pretty$ then T_{Pretty} (The pretty equivalent of T) to Sem. This congruence is one that we would want to demonstrate informally if not at the theorem prover level. Given these objectives the next sections discuss the various ways in which parsing and unparsing can be implemented.

3.4 Typed Parsing Versus Parsing as Preprocessing

There are two options available to effect object language support in Lego. The first choice is to extend Lego with the capability to assign a type to quotations (strings) and then build the lexicon and grammar building commands as well as the parsing routines at the theorem prover level. The benefit here would be that we could reason about the parser and more specifically prove the commutativity diagram in figure 3.3 in Lego. There are a number of technical and practical problems with this approach however which detract from any of these benefits. If parsing is built into Lego's machinery a new construction and type for quotations must be added to Lego with the parser presumably defined by reduction strategies. On top of this a necessary amount of meta theorems would need to be proved to ensure the consistency of the type theory. This amounts in total to a significant addition to the implementation of Lego.

The second choice is to design the parser as a preprocessing procedure that passes the translation of a quotation to the machinery of Lego. The immediate implication of this is that quotations and parsing are placed outside the theorem prover level and so the commutativity of the diagram of figure 3.3 cannot be demonstrated in Lego. An informal proof must suffice. A second point worth noting on this subject is that the function T_{Pretty} also cannot be defined in Lego. If we wish to obtain $Pretty(T(Sem))$ from $Pretty(Sem)$ as in figure 3.3, we must follow the other arrows in the diagram. This can be achieved if for a semantics Sem the user provides the informal map $Pretty$ via the definition of the language for that semantics, and then for a given transformation T the user again supplies the appropriate informal map. This second alternative has advantages over the first in that it is a conservative extension of Lego's type theory and a parsing algorithm written in ML will inevitably be much faster than one written in Lego.

3.5 Extending Lego with Object Language Support

Four additions must be made to the Lego system to provide a quotation supporting mechanism. A means of defining a language lexicon, a grammar, a parsing facility and an unparsing facility. The next four subsections outline the implementation of these respective modules.

3.5.1 Defining the Lexicon

A new command is added to Lego that allows one to define a lexicon for a new language. The form of the command is

$$\textbf{Terminals}\ \langle\textit{name}\rangle\ = s_1,\ s_2\ \ldots\ s_n$$

where s_1, s_2 ... s_n are strings for $n \geq 0$ and <*name*> is the name of the language being defined. This set of symbols is stored within a special area outside the Lego context. It is essential to define this set before a context free grammar using this set can be defined (since the grammar rules will include these terminals). The parser and unparser access this information. For the example, the command is

```
Terminals Nat = ``0'' ``suc'' ``<''
```

3.5.2 Defining the Grammar and Signature

In section 3.3 it was observed that a signature and a language bore a close resemblance. In fact, if we extend the format of a grammar to provide a name for each production then the grammar can be effectively treated as a signature definition as well. The reasons are clear if we realize that the non-terminals of the grammar correspond to the sorts in a signature and the names for productions correspond to the function names of a signature. If we extend the grammar in figure 3.2 to include the appropriately named productions we get

$$\begin{array}{rcl} Formula & \rightarrow & lessthan \;:\; natural \text{ ``<'' } natural \\ natural & \rightarrow & zro \;:\; \text{``0''} \\ natural & \rightarrow & succ \;:\; \text{``suc'' } natural \end{array}$$

from which we can immediately see the relationship between the functional arity

```
lessthan : (natural,natural) -> Formula
```

in the signature of figure 3.1 and the extended production labelled by *lessthan* in the grammar above. Each production is a parallel of a functional arity map in a signature. We can take advantage of this loose equivalence and encode a signature into the context free grammar. This allows the user to perceive terms as objects well formed according to their own notation (the grammar they supplied) rather than according to the implementation of terms in Lego. It also provides a straightforward means of translating a quotation into the appropriate Lego term as shall be demonstrated in section 3.5.3.

A new Lego command **Productions** is defined for the creation of a language grammar (and thus also a signature). Its form is

Productions <*name*> **::=** <*prodns*> **;**

where the form of <*prodns*> is either a <*prodn*> or

<*prodns*> **,** <*prodn*>

and where a <*prodn*> has the form

$$\begin{array}{rcl} S & = & \\ f_1 & \textbf{:} & \alpha_1^1 \;\alpha_1^2 \;\cdots \alpha_1^{n_1} \;\;| \\ f_2 & \textbf{:} & \alpha_2^1 \;\alpha_2^2 \;\cdots \alpha_2^{n_2} \;\;| \\ \vdots & & \vdots \ldots | \\ f_m & \textbf{:} & \alpha_m^1 \;\alpha_m^2 \;\cdots \alpha_m^{n_m} \end{array}$$

where S is a sort name, each f_i is a function name for $0 \leq i \leq m$, each $n_i \geq 0$ for each $0 \leq i \leq m$, each α is either a non-terminal (sort name) or a terminal symbol of the language and $m \geq 0$. Each such <*prodn*> defines the subset of the relevant signature Sig characterized as the set of functional maps

$$f \;:\; (s_1,\, s_2 \,\ldots\,,\, s_n) \rightarrow s$$

for sort s in *Sig*. The information needed to define the implied signature is extracted from each production of the grammar. The provided <*name*> is bound to the signature in the global Lego context, and the grammar is stored in a separate context along with the <*name*> for future reference as data for the parsing and unparsing processes. This <*name*> is a reference for both signature and language (terminals and productions). As in the **Signature** command in section 3.1, if the sort or function names are not already defined as such in the global Lego context then they are added as new ones. The example command call is shown below

```
Productions Nat ::=
      natural = zro      : "0" |
                succ     : "suc" natural,
      Formula = lessthan : natural "<" natural;
```

3.5.3 The Parser

There are a plethora of parsing algorithms applicable to the task of parsing a quotation to a base term. The Earley Algorithm is one of the most general algorithms available. It can be used to parse a wider variety of languages than other parsers. In fact, the algorithm is so general that even sentences of an ambiguous grammar may be parsed; every possible parse tree being returned. In order to specify which particular parse tree is intended when working with an ambiguous grammar, the user is expected to make use of parenthesis constructs included explicitly as part of the language. For instance, if we intended to extend the example with addition, we would need two extra productions

```
natural =  ... |
           ... |
           pls  : natural ‘‘+’’ natural |
           natp : ‘‘(’’ natural ‘‘)’’,
           ...
```

the first to introduce addition and the second to be able to enforce the intended form of associativity for addition. Ambiguous sentences can still be parsed so it is up to the user to use parentheses to disambiguate them.

Since the parser is essentially a pre-processing procedure, it is written in Standard ML as an extension to the implementation of Lego. As mentioned earlier, the parser is a general function capable of parsing any number of different languages. In order to invoke it correctly for a given quotation, the name of its language must be supplied. In addition to this we must also supply the relevant non-terminal (sort) name because a language may allow a quotation to be a term of two different sorts. Say we were to allow identifiers in `Nat`:

```
natural = ... |
          ... |
          Id,


Id      = x | y | z, ...
```

then the parser may parse the quotation `‘‘x’’` as an `Id` when the intention of the user is to parse it as a `natural`. Therefore an invocation to the parser has the form

$$<\!\mathit{Sig}\!>\ \textbf{!}\ <\!\mathit{sort}\!><\!\mathit{string}\!>$$

being the language/signature name, sort and quotation in double quotes respectively. Typing an invocation sets off the parser which converts the quote to a Lego term that is in turn passed to the evaluator and type checker. Parsing and translation is performed simultaneously in the following way. When an invocation is detected in Lego, the parser is executed with the given language terminals and the productions associated with the given sort. The algorithm then builds a succession of *states*, sets of *configurations* where a configuration is a 5-tuple

$$A \ \rightarrow\ \alpha\, .\, \beta\,, n,\, T$$

where $A \ \rightarrow\ \alpha\beta$ is a production in the language, the dot separating α and β is taken to mean that all the symbols in α to the left of it have been recognized and all the symbols in β are yet to be recognized, the *prediction number* n represents the number of the state in which this configuration was initially *predicted* (see below) and T represents the current translation fragment of the configuration.

A state is synonymous with a point in the parsing process. For natural number i, state i represents the point in the parse where the first i tokens in the input string have been recognized. So if we have a quotation invocation

```
Nat!Formula ‘‘0 < suc 0’’
```

The parser adds the initial configurations for `Formula` productions to state 0, since this prediction is in the initial state the prediction number attached to this configuration is 0 to give

$$config1 \ = \ Formula \ \longrightarrow\ \bullet\, natural \text{ “<” } natural,\ \ 0,\ \ lessthan$$

The translation fragment *lessthan* comes from the name of the production used for this configuration. It is the initial form of the translation signifying the main function name of the Lego construction to be built. After a number of prediction and reduction steps in the algorithm we arrive at the final state:

$$\begin{array}{rclclll} config9 & = & natural & \longrightarrow & \text{``suc''}\ natural\ \bullet, & 2, & succ\ zro \\ config10 & = & Formula & \longrightarrow & natural\ \text{``<''}\ natural\ \bullet, & 0, & lessthan\ zro \\ &&&&&& \qquad (succ\ zro) \end{array}$$

The last configuration signals the fact that a `Formula` has been recognized. Note that the translation result is the macro form for the Lego term of the quotation `‘‘0 < suc 0’’`. Correctness is ensured since the grammar is also essentially encoding the signature it describes.

The parser is also provided with an ability to handle variables so that they may be written succinctly. It must essentially code up a concrete term such as `var|Sig n sort` in a more abbreviated form like `e1` or `e’` etc. But we have to be able to parse a variable to its intended sort and distinct natural number as well as determine which signature it is a variable of. Since it must be expected that there will be many sorts involved in many languages, abbreviated forms of variables may become confusing to read. We provide a means by which a variable is close in definition to a concrete variable term. The form is

$$<\!sortname\!>\tilde{}<\!N\!>$$

where N is a number. The translation is simple. The number translates to the Lego natural number representation of N, the sort is directly translated and the signature can be obtained from the parser invocation. An example of a variable quotation is

```
Nat!natural ‘‘natural~1’’
```

Antiquotation is also supported as in HOL except that any concrete HOL term can be antiquoted, but this can again lead to problems of readability. In our case, this would mean we could for example write a quotation such as

```
Nat!Formula ‘‘0 < ^(fa succ (tcons (fa zro (tnil|Nat))
                               (tnil|Nat)))’’
```

but this only serves to reintroduce the confusion that quotations set out to eradicate. A compromise can be reached in that only Lego identifiers can be antiquoted and if more complex concrete terms are to be included in a quotation, then they should be bound to a new identifier in the current Lego context and then this identifier should be antiquoted in their place. Thus the quotation above would be written in Lego as

```
Lego> [x = (fa succ (tcons (fa zro (tnil|Nat)) (tnil|Nat)))];
value = ...
type  = ...

Lego> Nat!Formula ‘‘0 < ^x’’
```

Obviously the concrete term here can immediately be written as a quotation anyway but there may be more complex terms that cannot be directly parsed into a quotation.

3.5.4 The Unparser

Unparsing is the inverse function of translation. It is an extension of the output procedures of Lego in that wherever possible, quotations are substituted as output for Lego constructions of type `Term|Sig s` (where `Sig` is a signature, `s` is a sort). In this way, the concrete structure of such objects is to some extent hidden from the user. At most, only quotations are read or written at the theorem prover level. A typical session in Lego without this unparsing facility would look like this:

```
(* User input *)
Lego> Nat!natural ‘‘suc 0’’

(* Lego output *)
value = fa succ (tcons (fa zro (tnil|Nat)) (tnil|Nat))
type  = Term Nat natural
```

where we would ideally like the system to return the value field as the quotation provided.

Once the parser routines are implemented, it is a simple procedure to augment the printing routines of Lego so that whenever a quotable `‘‘Term’’` type object is to be outputted, the appropriate quotation is printed instead. Such objects can be identified as those built entirely from either

- The two constructors for the type of `Terms`: `fa` and `var`
- OR Antiquotable Lego identifiers.

We shall call such constructions *quotable terms*. Once a quotable term is recognized, the relevant lexicon and grammar are retrieved from storage by looking within the structure of the term to locate the signature (and hence language) name. Once this information is produced the reverse of the translation process is applied to give a quotation. The following is an example of unparsing. The quotation `Nat!natural ‘‘suc 0’’` is written in concrete Lego syntax as

```
fa succ (tcons (fa zro (tnil|Nat))
        (tnil|Nat))
```

The output stream is initially empty. The term is immediately quotable so the unparsing routines locate the production of the main function name, `succ` in the grammar of `Nat`, which is

```
natural = ... |
          succ: ‘‘suc’’ natural
```

The current output stream becomes

```
‘Nat!natural ‘‘suc ’’ ’.
```

The hole at the end of the string represents the position of the rest of the quotation yet to be filled in. The first and only argument of the term is

```
fa zro (tnil|Nat)
```

and again the production for `zro` is

```
natural = ... |
          zro: ‘‘0’’
```

appending “0” to the output stream completing it to become

```
‘Nat!natural ‘‘suc 0’’ ’.
```

Variables and antiquotation are handled slightly differently and the details are straightforward.

The final concern is to ensure that quotations are printed to the screen in an appropriate format so that larger terms become easier to read. When the length of a quotation is longer than the width of the screen it is printed to, it can become difficult to identify the different parts or blocks of the sentence. Pretty printing routines are supplied in the Lego source code, and it is expedient to make use of them here. Since the object language support utilities here are general and allow for any form of context free grammar, and aesthetics for formatting term fragments differ, it is difficult to provide one ideal formatting style to suit all or most languages and tastes. A convenient compromise is therefore to make use of any in-built utilities of the system. The formatting programs in Lego are very imposing in that it is difficult to provide a separate formatting module without major changes in the source code for Lego, and this then in turn means it is problematic to keep such a system up to date with newer implementations.

3.6 Lego Inductive Types Specified with Object Language Support

The object language support facilities described in this chapter can also be directly applied to cater for a small subset of Lego's inductive types. This subset can be defined as the set of inductive types specified with the following form

$$\begin{array}{rl}
\textbf{Inductive} & [T_1,\, T_2,\, \ldots,\, T_j : \textbf{Type}(x)] \\
\textbf{Constructors} & \\
& [\ c_1\!: t_1^1 \!\to\! t_1^2 \!\to\ \cdots\ \to\! t_1^{n_1}] \\
& [\ c_2\!: t_2^1 \!\to\! t_2^2 \!\to\ \cdots\ \to\! t_2^{n_2}] \\
& \qquad\quad \vdots \quad\ \vdots \\
& [\ c_m\!: t_m^1 \!\to\! t_m^2 \!\to\ \cdots\ \to\! t_m^{n_m}]
\end{array}$$

where $x \geq 0$, $j \geq 1$, $m \geq 0$ and each $t \in \{T_1, \ldots, T_j\}$ for $1 \leq i \leq n_i$, each $n_i \geq 1$ and $0 \leq i \leq m$. Obviously this is a major restriction since no types outside the set $\{T_1, \ldots, T_j\}$ are permitted in the type declarations of the constructors. Significantly we have to abandon polymorphism due to the absence of Π-types. As an example we may define a lexicon and grammar for the inductive type of natural number lists, which in normal Lego is defined using the command

```
Inductive [natural,nlist:Type(0)]
Constructors
          [zero:natural]
          [succ:natural->natural]
          [nil: nlist]
          [cons:natural->nlist->nlist];
```

and which in the extension to Lego covered in this chapter can be defined using the commands

```
Terminals = ‘‘0’’ ‘‘suc’’ ‘‘,’’ ‘‘[’’ ‘‘]’’;
Productions NatList Ind ::=
            natural = zero : ‘‘0’’ | succ : ‘‘suc’’ ‘‘natural’’,
            nlist   = nil  : ‘‘[’’ ‘‘]’’ |
                      ne   : ‘‘[’’ ‘‘nlst’’ ‘‘]’’,
            nlst    = sngl : ‘‘natural’’ |
                      cons : ‘‘natural’’ ‘‘,’’ ‘‘nlst’’;
```

Given these, we can express `nlists` as quotes such as

```
NatList!nlist ‘‘[suc 0, 0, suc suc 0]’’
```

Chapter 4

Formalizing Operational Semantics

In the previous chapters we introduced the basic terms in our formalization of language semantics. We now extend the theory to include operational rules which we choose to describe as inductive definitions. Once we have this set, we can ascribe the intended meaning to its elements. Henceforth we use the terms "operational semantics" and "inductive definition" interchangeably. We commence in section 4.1 with a complete outline of the components and syntax of inductive definitions in terms of Lego types, section 4.2 describes the meaning attached to them, section 4.3 gives some useful functions and theorems thereon, section 4.4 provides a preliminary evaluation of the system and section 4.5 outlines an extension to the theory of the preceding sections.

4.1 Types for Inductive Definitions

An inductive definition is a set of inference rules of the form

$$\frac{P_1 \quad P_2 \quad \cdots \quad P_n}{C} \; \sigma_1,\, \ldots,\, \sigma_m$$

for $n, m \geq 0$, where the atomic formulae P_1, P_2, ...,P_n are *premisses* or relational assumptions, $\sigma_1, \ldots, \sigma_m$ are *side conditions* not directly connected with the relations being defined and C is an atomic formula, the *conclusion* of the rule. The rule represents the Horn clause

$$\forall x_1.\forall x_2.\ldots\forall x_k.(P_1 \wedge P_2 \wedge \ldots \wedge P_n) \;\rightarrow\; (\sigma_1 \wedge \sigma_2 \wedge \ldots \wedge \sigma_m) \;\rightarrow\; C$$

as expected where the *meta-variables* $x_1, x_2, \ldots, x_k$ are the free variables in the succeeding formulae. In our theory, premisses and conclusions are `Formula` `Terms`. The meta-variables in rules are `Terms` built using the constructor `var`.

Universal quantification can be kept implicit in the syntactic representation of rules — as in their presentations in the literature. It must be borne in mind that semantically, the quantification is present and is treated as such. We must remember that semantic operations should respect this. In particular, substitution must be applied to a rule uniformly to preserve the implied binding in rules. This leaves us to define the notion of side condition in our theory to complete the encoding of the syntactic representations of inductive rules.

4.1.1 Side Conditions

Side conditions are present in rules where a predicate/relation being defined cannot be described using positive induction rules. The most common such relation is inequality (for terms), a minimum requirement. Other inductively definable relations are also sometimes presented as side conditions making proof trees more concise. A common such relation being evaluation under natural number addition. Take $\Rightarrow$ to be the relation denoting evaluation for addition for natural numbers. The formula

$$n + m \Rightarrow k$$

could equally be expressed as a side condition or by the inference rules for Peano arithmetic. The consequence of expanding the set of side condition relations amounts to a transfer of complexity from the proof tree to the side condition. We gain nothing in terms of expressibility so for now we disregard providing relations other than inequality for terms. An account of extending our theory for other relations is given in section 4.5. With this restriction in mind we can express side conditions in rules as a sequence of pairs of terms

$$(x_1, y_1), \ldots, (x_n, y_n)$$

where each x_i, y_i are `Term`s of sort s_i for $0 \leq i \leq n$. Since each s_i may differ, the type of each element in a sequence may differ. In Lego we can express the type of this sequence as an inductive type

```
SCList: FSig -> Type(0)
```

whose constructors are

```
SCnil  : {FS|FSig}SCList|FS
SCcons : {FS|FSig}
         {s|Sort}
         (Term|FS s)->(Term|FS s)->(SCList|FS)->SCList|FS
```

where an object

```
SCcons t1 t2 scl
```

represents the inequality $t_1 \neq t_2$ appended to the list *scl* of side conditions. The `vector` type introduced in section 2.2.1 could have been used here but we opt to use a special inductive type for side conditions to make the sub-theory easier to reason about.

4.1.2 Rules

We can now express the type of rules, rule sets and inductive definitions. A rule is a triple $(P,\ \sigma,\ c)$ of a list of formulae terms P denoting a rule's premisses, σ denoting its side conditions and a formula c, the conclusion of the rule. In Lego we use the `product` type to obtain the type for rules as

```
[Rule = {FS|FSig}
        prod (list|(Term|FS Formula))
        (prod (SCList|FS)
              (Term|FS Formula))]
```

With this we provide projection functions to access each triplet in a rule

```
[Prems     : {FS|FSig}(Rule|FS)->(list|(Term|FS Formula))]
[SideConds : {FS|FSig}(Rule|FS)->(SCList|FS)]
[Conc      : {FS|FSig}(Rule|FS)->(Term|FS Formula)]
```

returning respectively the premisses, side conditions and conclusions of a rule. We can express a rule's components in a readable manner using the parsing facilities of the previous chapter, but we cannot yet do this for a rule as a whole. The rule

$$\frac{x \ < \ y}{suc\,x \ < \ suc\,y}$$

is written as

```
Pair (cons (Nat!Formula "natural~1 < natural~2")
           (nil|(Term|Nat Formula)))
     (Pair (SCnil|Nat)
           (Nat!Formula "suc natural~1 < suc natural~2"))
```

We have reintroduced raw Lego syntax into the objects of our theory. In chapter 3 we saw how to extend Lego with parsing capabilities for `Terms`. The appropriate solution is to extend this further for a wider variety of types (those occurring in rules).

In effect we are extending Lego's user interface to provide a better means of understanding the objects in our theory. An extension of the language grammar command `Productions` from the previous chapter is provided for the new elements of our theory. Four new groups of productions are permissible in language grammars. The templates for them are

$$\begin{array}{rcl}
\texttt{"Prems"} & \texttt{=} & \\
\texttt{"prems"} & \texttt{:} & \alpha_1 \ \texttt{"Prms"} \ \alpha_2 \ | \\
\texttt{"nilPrems"} & \texttt{:} & \alpha_3, \\
\texttt{"Prms"} & \texttt{=} & \\
\texttt{"snglPrms"} & \texttt{:} & \alpha_4 \ \texttt{"Formula"} \ \alpha_5 \ | \\
\texttt{"consPrems"} & \texttt{:} & \alpha_6 \ \texttt{"Formula"} \ \alpha_7 \ \texttt{"Prms"} \ \alpha_8,
\end{array}$$

for premisses,

$$\begin{array}{rcl}
\texttt{"SCList"} & \texttt{=} & \\
\texttt{"nilSC"} & \texttt{:} & \alpha_9 \ | \\
\texttt{"consSC"} & \texttt{:} & \alpha_{10} \ \texttt{"Formula"} \ \alpha_{11} \ \texttt{"Formula"} \ \alpha_{12} \ \texttt{"SCList"} \ \alpha_{13} \ | \ldots
\end{array}$$

for side conditions (one production for each sort in the signature) and

$$\begin{array}{rcl}
\texttt{"Rule"} & \texttt{=} & \\
\texttt{"rule"} & \texttt{:} & \alpha_{14} \ \texttt{"Prems"} \ \alpha_{15} \ \texttt{"SCList"} \ \alpha_{16} \ \texttt{"Formula"},
\end{array}$$

for rules, where each α is a sequence of terminal symbols of the user's choosing. In our example, we could add the following rules to our grammar for the example language `Nat` from section 3.5.2:

```
"Prems"
       = "prems"        : "[" "Prms" "]" |
         "nilPrems"     :     "[" "]",
"Prms"
       = "snglPrms"     : "Formula" |
         "consPrems"    : "Formula" "," "Prms",
"SCList"
       =
         "consSC"      : "natural" "<>" "natural" "," "SCList" |
         "consSC"      : "Formula" "<>" "Formula" "," "SCList" |
         "nilSC"       : ".",
"Rule"
       = "rule"        : "Prems" "[" "SCList" "]" "|-" "Formula";
```

The parser is invoked in the same way, so for example we can now write our two example rules as

```
[
ZeroRule = Nat!Rule "[][.]|- 0 < suc natural~1"
]
[
SuccRule = Nat!Rule "[natural~1 < natural~2]
                      [.]
                      |- suc natural~1 < suc natural~2"
]
```

The keywords `Prems`, `Prms`, `SCList`, `Rule`, `prems`, `nilPrems`, `snglPrms`, `consPrems`, `consSC`, `nilSC` and `rule` are special and should not be used in a `Productions` command for any other purpose.

The form of the grammatical rules is restricted. The user is only free to choose the terminal symbols they require, but this helps the user by accommodating their own notation. As an alternative, with a few changes to the grammar we could express the second rule above as

```
Nat!Rule "natural~1 < natural~2
          --------------------- ()
          suc natural~1 < suc natural~2 "
```

4.1.3 Rule Sets

The final syntactic object to categorize is the rule set. One could use lists to represent them but it would be naive to do so since we can expect to make use of functions that return elements of rule sets. This cannot be done for the empty list of rules if Lego functions are total, which they are. We circumvent this by dictating that rule sets be non-empty. We lose nothing of interest as a result. Non-empty lists are defined as a Lego inductive type

```
NElist:Type(0)->Type(0)
```

with the expected constructors `Nnil` and `Ncons` for singleton and larger lists respectively. The gives us the type for rule sets:

```
[RuleSet = [FS|FSig]NElist|(Rule|FS)]
```

Three functions are supplied with this definition to refer to the head rule in a rule set, the tail rules and the i_th rule. They are simple instantiations of the elimination rule for non-empty lists and their respective types are

```
HdRule  : {FS|FSig}(RuleSet|FS)->Rule|FS
(* In the singleton case, TlRule is the identity function *)
TlRule  : {FS|FSig}(RuleSet|FS)->(RuleSet|FS)
(* If nat out of range, RuleNum returns the last rule *)
RuleNum : {FS|FSig}(RuleSet|FS)->nat->Rule|FS
```

The third function `RuleNum` is particularly important in our theory as it is the primary means of referring to specific elements of a rule set. Our example rule set is written in our theory as

```
[NatRules = (Ncons ZeroRule (Nnil SuccRule)) :RuleSet|Nat]
```

In order to complete the type of inductive definitions, we can define them as a pair: A signature with a rule set over that signature. The right type for this is a sigma type: "`[Spec = sigma|FSig|RuleSet];`" Our example semantics becomes `(Nat,NatRules):Spec`. Up to this point we have discussed the syntactic properties of operational semantics to give us a first order representation of them in our theory. We now provide the mathematical substance for them.

4.2 The Meaning of Semantic Specifications

We begin with the notion of ground term instances of rules. A *rule instance* is a rule in which all meta-variables have been substituted with ground terms.

Definition 4.1 (Closure)

For a set of inductive rules R, a set S of formulae is closed *under R if and only if for all rule instances $r \in R$:*

$$\frac{P_1 \quad P_2 \quad \cdots \quad P_k}{C}\ \sigma_1,\ \ldots,\ \sigma_m$$

if each $P_i \in S$ for each $1 \leq i \leq k$ and each of $\sigma_1, \ldots, \sigma_m$ is true, then $C \in S$.

Definition 4.2 (Inductive Sets)

A set of inductive rules R defines an Inductive Set

$$Ind(R) \;=\; \bigcap\{S \mid S\, is\, closed\, under\, R\}$$

and as such this set is the least *such set closed under the rules.*

The assertion $x \in Ind(R)$ can be thought of as formalizing the fact that "x is derivable by the rules of R." Any inductive set has an associated induction principle, and one described by an inductive definition has an accompanying rule induction principle.

Definition 4.3 (Rule Induction)

For a set S inductively defined by a set of rules R, the property P holds of S by Rule Induction *if P is closed under R.*

In rule induction, we only have to prove properties hold for any set closed under the rules, not just the least set. From the definition of closure above we see that instantiations of the premisses and conclusions of rules are the elements of inductive sets. The example rule set `NatRules` defines the inductive set

$$\begin{aligned} Ind(NatRules) \;&=\; \{(0 \,<\, suc\, 0), (0 \,<\, suc\, suc\, 0), \ldots, \\ &\qquad (suc\, 0 \,<\, suc\, suc\, 0), \ldots\} \end{aligned}$$

A set is naturally equated to a type in constructive logic with set membership translating to type membership. To define inductive sets therefore is to define a type for them in Lego. Our basic aim in this section is to provide a statement in our theory of the general type for inductive sets (the elements of which we call *judgements*) provided with a general operator for rule induction over this type. Other useful lemmas and theorems on such sets can then be built on these foundations. The generality reflects the fact that we need a type equipped to delineate the inductive sets defined by *any* rule set. The rule induction operator must be similarly generic. In our Lego theory, this means the type for judgements and rule induction will be parametric on `RuleSet`s and (since `RuleSet`s are parametric on `FSig`) signatures.

A type encoding judgements is a type that must encode the notions of closure given in definition 4.1 and *leastness* in definition 4.2. We immediately come across two types similar to the two type choices in chapter 2, powerful enough to describe inductive sets. We could use a parameterized sigma type $\Pi RSet.\, \Sigma x.\, P(x,\, RSet)$ where elements of this set are pairs of a formula x and a proof $P(x,\, RSet)$ that

x is in the inductive set $Ind(RSet)$. This is akin to the method of defining the set of well formed terms in chapter 2 using sigma types. Only a subset of `Formula`e `Term`s are in $Ind(RSet)$ so a dependent pair delineates this by dictating judgements are pairs (x, P) where the proof P that x is a judgement encodes the closure and leastness criteria.

However taking the lessons of chapter 2 into account, it is more expedient to be able to define types without this bureaucracy. It can be avoided again by using Lego's inductive types. The significant feature this time is that we can use the inductive property of inductive types to encode the leastness we require and encode closure in the constructor types. The elimination operator for an inductive type gives the induction principle.

Taking a closer look at definition 4.1 and the inductive set defined by `NatRules` notice that the definition above hides the fact that substitutions for the meta-variables in rules occur implicitly via rule instances. It is more precise to point out that we are interested in closure modulo substitution. This leaves us with two concepts to define before we can encode closure and therefore the type of judgements. Firstly substitution and secondly the meaning attached to side conditions.

4.2.1 Substitution

Substitutions can be represented in two equivalent senses. Firstly as a function from variables to terms and secondly as a list of variable term pairs. The former is relevant to the process of substitution and the latter is relevant to the information within one. It is clear that the theory needs a substitution function but it is also essential to construct and reason about the class of substitutions in a simple manner. For this reason we represent a substitution as a list whose elements are variable term pairs, and furnish this with a function from substitutions and variables to terms. This way we can easily build and reason about substitutions (via list recursion) as well as apply them as a function.

Recall that variables are `Term`s formed using the constructor

```
var : {FS|FSig}nat->{s:Sort}Term|FS s
```

which means the defining criteria of a variable is its sort and numbering index. A function from variables to terms is then a function from these two values. The type of substitutions as lists of such mappings is parametric on signatures and sorts and is written as

```
[Subst = {FS|FSig}{s:Sort}list|(prod nat (Term|FS s))]
```

in our theory. For a given signature `FS` and sort `s`, we have a list of pairs representing bindings of variables of sort `s` (distinguished by a numbering parameter) to terms of type `Term|FS s`. If a variable is mentioned twice in a list, it is the mapping closest to the front that takes precedent. Since substitutions are parameterized lists we have to provide building functions for them analogous to list constructors. An empty substitution is formed by

```
[initSub = ([FS|FSig]
            [s:Sort]nil|(prod nat (Term s)) ):Subst|FS
]
```

and we can add to substitutions by using a function that overrides a given substitution with a new mapping. The function

```
updateSub : {FS|FSig}{s|Sort}nat->(Term s)->(Subst|FS)->Subst|FS
```

does this via a conditional construct that finds the appropriate list of mappings for the given sort and plants the new mapping at its head. The conditional depends on Martin-Löf equality as opposed to the normal Leibniz equality used in Lego. This gives us the substitutions as lists slant to our theory. To complete the picture and ensure the intended overriding takes place we define a substituting function

```
SubFn : {FS|FSig}(Subst|FS)->{s:Sort}nat->Term s
```

taking a substitution and a variable's attributes to return the right (most recently mapped to) term. As we shall be making use of substitutions in rules, it is necessary to provide functions that apply them to the premiss, side condition and conclusion components of rules. The function

```
TSubFn :{FS|FSig}(Subst|FS)->{s|Sort}(Term|FS s)->(Term|FS s)
```

applies a substitution to a `Term` and therefore can be used to apply one to the conclusion of a rule. It is defined by recursion on terms. Any variables that are sub-terms of a term are replaced by calling `SubFn`. The function

```
TlistSubFn:
          {FS|FSig}
          (Subst|FS)->
          {s|Sort}(list|(Term|FS s))->(list|(Term|FS s))
```

applies a substitution to a list of terms by list recursion. `TSubFn` is applied to each element of the list. It can be applied to the premisses of a rule. Finally the function

```
SCSub: {FS|FSig}(Subst|FS)->(SCList|FS)->(SCList|FS)
```

applies a substitution to a list of side conditions by recursion on the specialist `SCList` type.

We return to the subject of legibility. As in section 4.1.2 we extend our parsing capabilities with a facility for objects of type `Subst`. The new production class has the form

$$\begin{array}{rcl} \text{"}\mathtt{Subst}\text{"} & = & \\ \text{"}\mathtt{nilSub}\text{"} & : & \alpha_1 \ | \\ \text{"}\mathtt{consSub}\text{"} & : & \alpha_2 \ \text{"}\mathtt{Formula}\text{"} \ \alpha_3 \ \text{"}\mathtt{Formula}\text{"} \ \alpha_4 \ \text{"}\mathtt{Subst}\text{"} \ \alpha_5 \ | \ldots \end{array}$$

similar to the production class for side conditions, there should be one production for every sort in the language. In our example language `Nat` we have

```
 "Subst"
        =
          "consSub"     : "natural" "|-->" "natural" "," "Subst" |
          "consSub"     : "Formula" "|-->" "Formula" "," "Subst" |
          "nilSub"      : "nil"                      ,
```

To appreciate the perspicuity here, the substitution

$$\{x \mapsto 0,\ y \mapsto suc\,0\}$$

is expressed in raw Lego as

```
updateSub one (Nat!natural "0")
 (updateSub two (Nat!natural  "suc 0")
   (initSub|Nat))
```

but in its pretty form as

```
Nat!Subst "natural~1 |--> 0, natural~2 |--> suc 0, nil"
```

4.2.2 Side Conditions and their Proofs

Recall that in section 4.1.1 we introduced side conditions as sequences of pairs of terms. Each pair denoting an inequality. To determine whether the inequality holds, a simple function can be defined from terms to `Prop` (the proposition type in Lego). Its functionality is

```
TermNeq : {FS|FSig}
          {s|Sort}
          (Term|FS s)->(Term|FS s)->Prop
```

where TermNeq is an object with the appropriate instantiation of the elimination operator for Terms. To determine the same for a list of side conditions then requires a function

```
SCHold : {FS|FSig}(SCList|FS)->Prop
```

constructed by primitive recursion on SCLists essentially conjoining all the inequalities therein. With this, two auxiliary functions are provided

```
NilSCPrf  : {FS|FSig}SCHold (SCnil|FS)
ConsSCPrf : {FS|FSig}
            {s|Sort}
            {l:SCList|FS}
            {t1,t2:Term|FS s}(TermNeq t1 t2)->(SCHold l)->
            (SCHold (SCcons t1 t2 l))
```

the first provides an automatic proof that an empty list of side conditions hold trivially, and the second breaks the proof that a non-empty list is true down to proving the first inequality holds and the rest of the inequalities hold. By using these two functions, a user can assert the validity of a list of side conditions by the continual application of these functions coupled with proofs of the individual inequalities.

4.2.3 Judgements: Inductively Defined Sets

We are now ready to describe the type of judgements using an inductive type with a constructor defining closure which must be the type equivalent of the informal proposition

> "For any signature FS, rule set RSet over FS, rule r in RSet and substitution Sub over FS, if the premisses of r under Sub are judgements and if the side conditions of r under Sub hold, then the conclusion of r under Sub is a judgement."

Notice that the substitution is applied uniformly throughout the rule r. This fulfils the need to model the implicit universal binding in rules when applying substitution. Note also the inductive part of the closure property is propagated through the premisses. In fact, we can ascribe a type for judgements analogous to the type for well formed terms in chapter 2. In Lego we have a type for `Judgement`s with a type `Jlist` (for judgement lists) defined by mutual induction:

```
Judgement: {FS|FSig}(RuleSet|FS)->(Term|FS Formula)->Type(0)
Jlist    : {FS|FSig}(RuleSet|FS)->(list|(Term|FS Formula))->Type(0)
```

where the type `Judgement RSet f` can be thought of as asserting "`f` is in the inductive set defined by `RSet`," and `Jlist RSet fl` is equivalent to "The formulae `fl` are in the inductive set defined by `RSet`". Recalling the functions described in this chapter and using natural numbers to refer to the elements of rule sets, the constructor for the type of `Judgement`s is

```
ruleAp :
        (* For all signatures *)
  {FS|FSig}
        (* For all rule sets *)
  {RSet:RuleSet|FS}
        (* For all naturals *)
  {i:nat}
        (* For all substitutions *)
  {Sub:Subst|FS}
        (* If the premisses are judgements *)
  (Jlist RSet    (TlistSubFn Sub (Prems (RuleNum RSet i))))->
        (* If the side conditions hold *)
  (SCHold|FS     (SCSub      Sub (SideConds (RuleNum RSet i))))->
        (* Then the conclusion is a judgement *)
  Judgement RSet (TSubFn     Sub (Conc (RuleNum RSet i)))
```

encoding closure in the sense of the quoted proposition above. In proof theoretic terms it can be seen as the rule applying function for building proof trees for a given set of inductive rules. An object inhabiting the type `Judgement RSet f` can be thought of as being a proof tree of the proposition $f \in Ind(RSet)$. The constructors for `Jlist`s are used to build lists of `Judgement`s corresponding to the list of formulae denoted in the type information for `Jlist`s:

```
jnil :
  {FS|FSig}
  {RSet:RuleSet|FS}
  Jlist RSet (nil|(Term|FS Formula))
```

```
J_Induction :
  (* For all signatures *)
  {FS|FSig}
  (* For all rule sets *)
  {RSet:RuleSet|FS}
  (* For all properties P of judgements *)
  {P:{f|Term|FS Formula}(Judgement RSet f)->Prop}
  (* If the property preserves closure *)
  (    {i:nat}
       {Sub:Subst|FS}
       {p:Jlist (TlistSubFn Sub (Prems (RuleNum RSet i)))}
       {x1:SCHold (SCSub Sub (SideConds (RuleNum RSet i)))}
       (Conj P p)->
     P (ruleAp RSet i Sub p x1)
  )->
  (* If P holds the empty list of judgements *)
  (  Conj P (jnil|FS RSet)  )->
  (* If P holds for all elements of a non-empty judgement list *)
  (    {f|Term|FS Formula}
       {fl|list|(Term|FS Formula)}
       {jh:Judgement RSet f}
       {jt:Jlist RSet fl}
       (P jh)->
       (Conj P jt)->
     Conj P (jcons RSet jh jt))->
 (* Then P holds for all judgements *)
 {x1|(Term|FS Formula)}{z:Judgement|FS RSet x1}P z
```

Figure 4.1: The Induction Operator for Judgements

the empty formula list being a judgement list and

```
jcons :
  {FS|FSig}
  {RSet:RuleSet|FS}
  {f|(Term|FS Formula)}
  {fl|list|(Term|FS Formula)}
  (Judgement RSet f)->(Jlist RSet fl)->
  Jlist RSet (cons f fl)
```

to concatenate a judgement to a judgement list to make a judgement list.

The elimination operator `Judgement_elim` provides the recursive function and inductive operator for judgements. The relevant instantiation for induction gives us the functionality in figure 4.1 where the result of the function

```
Conj :
  {FS|FSig}
  {RSet|(RuleSet|FS)}
  {P:{f|(Term|FS Formula)}(Judgement RSet f)->Prop}
  {fl|list|(Term|FS Formula)}
  (Jlist RSet fl)-> Prop
```

is the conjunction of `P` applied to every element in the judgement list. The version of induction we acquire is tighter than the notion of rule induction in that instead of having to prove a property is preserved by any closed set, you need to prove it holds for exactly the elements of the least closed set. The property `P` does not quantify over all formulae but over the subset of them that are judgements. This is the same situation as we discovered for the type of well formed `Term`s in chapter 2 where the elimination operator only quantified for the subset of terms that were well formed. We lose nothing however if our only interest is in proving theorems of the least closed set. In fact, the induction operator in figure 4.1 is close to a rule for the induction on the depth of inference. With some extra machinery, outlined in section 4.3.4, we can indeed perform this kind of reasoning in our general theory. We now have a foundation upon which to reason about inductive definitions. The next section details some helpful functions and theorems for inductive rules, sets and judgements.

4.3 Utilities for Inductively Defined Sets

The constructions that follow are provided as corollaries to the work already provided in the general theory. They can be used to enhance the theorem proving interface for the class of judgements and its associated induction principle.

4.3.1 Named Rules

The way rules are referenced in our theory of `Judgement`s contributes to the relative illegibility of proof tree construction. When using `ruleAp` we must provide the position of the rule being applied in the rule set, which can often mean manually determining this index beforehand. We provide an alternative function to `ruleAp` which allows one to give instead the rule or the name of the rule one wishes to apply at a point in a derivation. The type of the function is

```
ruleAp' :
  {FS|FSig}
  {RSet:RuleSet|FS}
  {r:Rule|FS}
  [i=RuleNameToNum r RSet]
  {Sub:Subst|FS}
  (Jlist RSet (TlistSubFn Sub (Prems (RuleNum RSet i))))->
  (SCHold (SCSub Sub (SideConds (RuleNum RSet i))))->
  Judgement RSet (TSubFn Sub (Conc (RuleNum RSet i)))
```

which is identical to `ruleAp` except one provides an object of type `Rule` in place of a natural. Lego automatically searches the rule set `RSet` for the rule and determines the relevant index by calling `RuleNameToNum`, the rest of the arguments are supplied as they are to `ruleAp`. If the provided rule is not in the rule set then the index of the last rule is returned by `RuleNameToNum`. This allows us for example to write

```
ruleAp' NatRules ZeroRule ...
```

and frees us from the lower levels of detail in our theory. As such it provides a form of abstraction for rule referencing. Examples and an assessment of the use of `ruleAp'` are provided in the next chapter.

4.3.2 HeadPremiss, TailPremisses

The functions `HeadPremiss` and `TailPremisses` extract the head and tail respectively of a non-empty list of judgements. If we have such a list, its type (disregarding parameter information) must be `Jlist...(cons f fl)`. The judgement list must be non-empty since it must at least have a judgement for the formula `f` at its head. The functions are derived from a specialization of the recursive operator for `Jlist`s that caters specifically for non-empty judgement lists. The types of the functions are

```
(* Returns the judgement at the head of a Jlist *)
HeadPremiss :
  {FS|FSig}
  {RSet:RuleSet|FS}
  {f|(Term|FS Formula)}
  {fl|list|(Term|FS Formula)}
  (Jlist RSet (cons f fl))->Judgement RSet f
```

```
(* Returns the tail of a non-empty Jlist *)
TailPremisses :
  {FS|FSig}
  {RSet:RuleSet|FS}
  {f|(Term|FS Formula)}
  {fl|list|(Term Formula)}
  (Jlist RSet (cons f fl))->Jlist RSet fl
```

4.3.3 Case Analysis

Its often the case that in order to prove that if a formula `f` is a judgement with respect to a rule set `RSet`, it is so only by virtue of the fact that one of the rules in `RSet` can be used to derive it. This allows us to be able to assert that if a formula holds (is a judgement) then the premisses and side conditions of one of the rules hold. We can think of the theorem we are about to prove as the generic analogue of `derive_cases_thm` in HOL [CM92]. In logical terms, if for a rule set $RSet = \{r_1, r_2, \ldots, r_n\}$ the formula f is a judgement then we can show

$$\begin{array}{ll} & (\exists Sub\colon \mathbf{Subst}.\ \mathbf{Jlist}\ RSet\,(\mathbf{Prems}\ r_1)_{Sub}\ \ \wedge\ \ f \equiv (\mathbf{Conc}\ r_1)_{Sub}) \\ \vee & (\exists Sub\colon \mathbf{Subst}.\ \mathbf{Jlist}\ RSet\,(\mathbf{Prems}\ r_2)_{Sub}\ \ \wedge\ \ f \equiv (\mathbf{Conc}\ r_2)_{Sub}) \\ \vee & \vdots \\ \vee & (\exists Sub\colon \mathbf{Subst}.\ \mathbf{Jlist}\ RSet\,(\mathbf{Prems}\ r_n)_{Sub}\ \ \wedge\ \ f \equiv (\mathbf{Conc}\ r_n)_{Sub}) \end{array}$$

where terms subscripted by substitutions represent those terms with the substitution applied to them. The logical connectives range over the appropriate type universes. The existentials and equalities are needed when applying this formula in practice to either derive contradictions or ensure the correct bindings for variables. We can render a type in Lego to represent this formula using the function

```
JlistSum : {FS|FSig}
           (RuleSet|FS)->
           (RuleSet|FS)->
           (Term|FS Formula)->Type(0)
```

where `JlistSum RSet RSet f` represents the formula above. Two copies of a rule set are passed to this function since on closer inspection of the formula, we must both recurse along and keep a full copy of the rule set within the body of the type. More importantly, the function must have this form to make a crucial induction further on in the theory possible. The body of the function is

```
[JlistSum =
  [FS|FSig][RSet:RuleSet|FS]
  [RSet':RuleSet|FS][f:Term|FS Formula]
  NElist_elim ([_:RuleSet|FS]Type(0))
              ([r:Rule|FS]
               sigma|(Subst|FS)
                    |([Sub:Subst|FS]
                      prod (Jlist RSet (TlistSubFn Sub (Prems r)))
                           (Eq f (TSubFn Sub (Conc r)))))
              ([r:Rule|FS]
               [rl:NElist|(Rule|FS)]
               [rl_ih:Type(0)]
               sum (sigma|(Subst|FS)
                     |([Sub:Subst|FS]
                       prod (Jlist RSet (TlistSubFn Sub (Prems r)))
                            (Eq f (TSubFn Sub (Conc r)))))
                   rl_ih)
              RSet'];
```

where the first rule set is kept fixed and the second is recursed upon. We use sigma types to represent existentials, sum types for disjunction and product types for conjunctions since we are combining objects from `Prop` and `Type(0)` and these types generalize over both universes.

Before we can continue we have to define a theorem that states that if we know that the ith element in the disjunction of the formula above holds, then we can show that `JlistSum RSet RSet f` holds. We start with the type

```
[Jl_exists =
   [FS|FSig][RSet:RuleSet|FS]
   [RSet':RuleSet|FS][f:Term|FS Formula][i:nat]
   sigma|(Subst|FS)|([sub:Subst|FS]
      prod (Jlist RSet (TlistSubFn sub (Prems (RuleNum RSet' i))))
           (Eq f (TSubFn sub (Conc (RuleNum RSet' i)))))];
```

where the type `Jl_exists RSet RSet f i` is isomorphic to the formula

$$\begin{aligned} &\exists Sub\colon \mathbf{Subst}.\ \mathbf{Jlist}\ \mathit{RSet}\ (\mathbf{Prems}\ (\mathbf{RuleNum}\ \mathit{RSet}\ i))_{Sub} \\ \wedge\quad & f \equiv (\mathbf{Conc}\ (\mathbf{RuleNum}\ \mathit{RSet}\ i))_{Sub} \end{aligned}$$

With this we can write the theorem

```
Goal ith_sum :
        {FS|FSig}
        {f:Term|FS Formula}
        {RSet:RuleSet|FS}
        {i:nat}
        (Jl_exists RSet RSet f i)->
          (JlistSum RSet RSet f);
```

which essentially states that if we can construct the `i`th element in the summation defined by `JlistSum` then we can derive an object of type `JlistSum` itself. The proof of this theorem involves an intricate induction. Given the proof state

```
  FS | FSig
  f : Term|FS Formula
  RSet : RuleSet|FS
  ?1 : {i:nat}
         (Jl_exists RSet RSet f i)->
           JlistSum RSet RSet f
```

we need to refine by

```
NElist_elim ([RSet':RuleSet|FS]
             {i:nat}(Jl_exists RSet RSet' f i)->
             JlistSum RSet RSet' f)
```

to make sure that the correct type on induction takes place and that the right form of induction hypothesis is formed in the inductive step of the proof. Notice how the induction is conducted upon the second copy of `RSet`. The correct form of induction hypothesis would not be formed if we define `JlistSum` and `Jl_exists` with a single rule set argument.

Having proven `ith_sum`, we can now prove our main result

```
JRSum :
     {FS|FSig}
     {RSet:RuleSet|FS}
     {f|(Term|FS Formula)}
     {j:Judgement RSet f}
     JlistSum RSet RSet f;
```

by induction on `Judgements`. In the main `ruleAp` case we get the proof state

```
  i : nat
  Sub : Subst|FS
  jl : Jlist RSet (TlistSubFn Sub (Prems (RuleNum RSet i)))
  sch : SCHold (SCSub Sub (SideConds (RuleNum RSet i)))
  jl_ih : Jlist RSet (TlistSubFn Sub (Prems (RuleNum RSet i)))
  ?16 : JlistSum RSet RSet (TSubFn Sub (Conc (RuleNum RSet i)))
```

at which point we refine by

```
Refine ith_sum (TSubFn Sub (Conc (RuleNum RSet i))) RSet i;
```

to get

```
  ?22 : Jl_exists RSet RSet (TSubFn Sub (Conc (RuleNum RSet i))) i
```

which can be easily shown by the steps

```
Refine dep_pair Sub ?;
Refine Pair jl (Eq_refl ?);
```

JRSum is usually used in conjunction with the case function in Lego

```
case : {s,t|SET}{u|TYPE}(s->u)->(t->u)->(sum s t)->u
```

Given that the user assumes a formula `f` is a `Judgement`, we can apply `JRSum` to it to get a summation over the rule set. If their goal involves inferring the premisses of the last rule application for `f` then they apply `case` recursively to this summation until it diminishes. The effect is that each rule is posited as the rule from which we derived `f` as a `Judgement`. This will either generate a contradiction (due to the form of the rule combined with the imposed equality in each summand of `JlistSum`) or the rule will indeed be one of the possible rules the formula could be derived from and we proceed to derive the premisses. In such a case, the presence (as assumptions) of the existential substitution and the equality in that summand of `JlistSum` make sure that the correct substitutions for the variables can be inferred. An example of the use of `JlistSum` is shown in the next chapter.

4.3.4 Properties of Rule Sets

The function `RuleRec` is used to prove a property is true for every rule in a rule set. Its an instantiation of the recursive function for rule sets (non-empty lists) and its type is

```
RuleRec :
  {FS|FSig}
  (RuleSet|FS)->
  ((Rule|FS)->Prop)->
  Prop
```

We can derive a corollary of this stating for all signatures `FS` and rule sets `RSet` that if for a property `P:(Rule|FS)->Prop`, the proposition

```
RuleRec RSet P
```

holds, then for all `i:nat`

```
P (RuleNum RSet i)
```

holds. The theorem

```
RLemma :
  {FS|FSig}
  {RSet:RuleSet|FS}
  {P:(Rule|FS)->Prop}
  (RuleRec RSet P)->
  {i:nat} P (RuleNum RSet i)
```

effectively states the same thing as `RuleRec` but it is significant in that it is in terms of the way rules are referenced using the constructor `ruleAp` for the type `Judgement`. This means that whenever we wish to prove a property

```
R : {FS|FSig}
    {RSet:RuleSet|FS}
    {f|(Term|FS Formula)}(Judgement RSet f)->Prop
```

using `J_Induction` our representation of induction for judgements, we can, with the right instance of the function `P` above, prove that `R` preserves closure for every rule in `RSet` by induction on non-empty lists. This form of reasoning is referred to as *induction on the depth of inference* and is a common means of reasoning about inductive rules. This is obvious once one realizes that the type `Judgement` can also be seen as the type of proof trees. The constructors of the type apply rules, and concatenate proof trees together using `Jlist` objects, themselves formed of `Judgement` objects. An example application of induction on the depth of inference can be seen in the next chapter.

4.3.5 Automated Substitutions and Interpreting

With the current state of the theory in Lego, a goal directed derivation of an inductive definition demands the user define the exact substitution to be used for every rule application since it is one of the arguments of `ruleAp`. There are times however when we may wish to leave it partly or wholly unspecified. There are two main reasons for doing so.

Firstly, in certain circumstances the substitution we need can be entirely derivable from our goal on the one hand and the conclusion of the rule we wish to apply on the other. For example if our goal is

```
Judgement ... Nat!Formula ‘‘suc 0 < suc suc 0’’
```

and we wish to apply the rule

$$\frac{x \ < \ y}{suc\, x \ < \ suc\, y}$$

the substitution we need is

$$\{x \mapsto 0,\ y \mapsto suc\, 0\}$$

which can be obtained from the information in the goal and the conclusion of the rule. In such cases it is expedient to be able to let the system automatically create the right instantiation.

Secondly we may wish to leave some variables unbound to a ground term in a substitution and allow them to become bound further up the proof tree, in effect using the derivation process to act as an interpreting machine. The inductive definition for addition in Peano arithmetic

$$\frac{}{0 \ + \ y \ \Rightarrow \ y}\ (1)$$

$$\frac{x \ + \ suc\, y \ \Rightarrow \ z}{suc\, x \ + \ y \ \Rightarrow \ z}\ (2)$$

is a good example. Starting off with a goal

$$suc\, 0 \ + \ suc\, 0 \ \Rightarrow \ v$$

we can apply rule (2) and then (1) to get the proof tree

$$\cfrac{\cfrac{}{0 \ + \ suc\, suc\, 0 \ \Rightarrow \ suc\, suc\, 0}\ \textit{by rule } 1}{suc\, 0 \ + \ suc\, 0 \ \Rightarrow \ suc\, suc\, 0}\ \textit{by rule } 2$$

thus instantiating the variable v.

To this end we supply a function

```
unifSub :
  {FS|FSig}
  {s|Sort}(Term|FS s)-> (Term|FS s)-> (Subst|FS)
```

that takes two terms and creates a substitution for the variables in each such that if applied to both, it would make them equal if this is possible. The function's strategy is to recurse through both terms constructing the substitution as it does so. At each point in the double recursion there are four possibilities.

- One sub-term is a variable and the other is a function application of the same sort: a mapping from the variable to the function application is added to the substitution.
- Both are variables of the same sort: one is mapped to the other.
- Both are function applications of the same sort: the function recurses through their sub-terms collating all the resulting substitutions.
- The two sub-terms are of different sorts: no substitution is returned.

The user is free to attempt to use this function on two terms that cannot be unified. The result would be a useless construction. An example invocation of `unifSub` is

```
unifSub (Nat!Formula "natural~1 < suc 0")
        (Nat!Formula "0 < natural~2")
```

which produces the substitution

```
Nat!Subst "natural~1 |--> 0,  natural~2 |--> suc 0"
```

In a goal directed proof, `unifSub` can be used instead of a written substitution by supplying it the conclusion formula of the rule we wish to apply and the formula of the current goal. To apply the rule `SuccRule`

```
Nat!Rule "[natural~1 < natural~2]
          [.]
          |- suc natural~1 < suc natural~2"
```

from section 4.1.2 to the goal

```
Judgement NatRules "suc 0 < suc suc 0"
```

one would write

```
ruleAp NatRules
       one
       (unifSub (Nat!Formula "suc natural~1 < suc natural~2")
                (Nat!Formula "suc 0 < suc suc 0"))
```

or even

```
ruleAp NatRules one (unifSub (Conc (RuleNum NatRules one))
                             (Nat!Formula "suc 0 < suc suc 0"))
```

Examples of the use of all the utilities in this section appear in the next chapter.

4.4 Preliminary Evaluation

Before continuing it is instructive to analyse in part our theory up to this point. Proof trees of semantics are built using the constructors for `Judgements`. They can be built top down or bottom up in Lego. To construct the equivalent proof tree to

$$\frac{0 \ < \ suc\, 0}{suc\, 0 \ < \ suc\, suc\, 0}$$

in a bottom up fashion, we would enter the following list of definitions in Lego:

```
[sub1 = Nat!Subst "natural~1 |--> 0"];
[j1   = ruleAp NatRules zero sub1 (jnil NatRules) (NilSCPrf|Nat)];

[sub2 = Nat!Subst "natural~1 |-> 0,  natural~2 |--> suc 0"];
[j2   = ruleAp NatRules one  sub2 (jcons NatRules j1
                                          (jnil NatRules))
                                   (NilSCPrf|Nat)]
```

A straightforward process, but it is at this point where a curious phenomenon occurs. Constructing `j2` takes an infeasible amount of time. With such a small language and semantics the time in real terms is negligible but if we extrapolate this to the semantics in the next chapter, a thirty step proof (thirty rule applications) takes approximately five hours to compute using steps similar to those above. This is a problem that reappears in top down proofs. It can be traced back to the type checking invoked when applying

```
ruleAp NatRules one sub2
```

which expects something of type

```
(* type number (1) *)
Jlist...(TlistSubFn sub2 (Prems (RuleNum NatRules one)))
```

that is something consistent with a list of judgements for the premisses of rule `one` of `NatRules`, and it is supplied with `(jcons j1 (jnil...))` which has type

```
(* type number (2) *)
Jlist...(cons (TSubFn sub1 (Conc (RuleNum NatRules zero)))
              (nil|(Term|FS Formula)))
```

The type of the former does not match the type of the latter so an amount of reducing is necessary before the types conflate to the same term. Looking at the two types `(1)` and `(2)` above, we can see that the earliest possible point at which the two types become identical is when they both reduce to

```
(* type number (3) *)
Jlist ... (cons (Nat!Formula "0 < suc 0")
                (nil|(Term|Nat Formula)))
```

something that can be confirmed if we try to reduce them so in Lego using the `Equiv` command, but the unification strategy of Lego fails to find this point and reduces the two terms further. In fact its heuristics virtually normalize both terms fully before type checking succeeds. In the example here, this means both terms evaluate to the raw Lego terms for `Judgement...(Nat!Formula "0 < suc 0")` that is

```
Judgement...
  (fa ltn (tcons (fa zro (tnil|Nat))
          (tcons (fa succ (tcons (fa zro (tnil|Nat)) (tnil|Nat)))
          (tnil|Nat)))
```

before they are unified. Furthermore, in reducing `(1)` and `(2)` to these concrete terms, they themselves are type checked to see that their sub-terms are well-formed with respect to the signature `Nat`. In the term above it is done four times since there are four function applications in it. Larger terms require more and more of this well-formedness checking proportional to the size of the abstract syntax tree of the term. The time taken is also a function of the size of the signatures and rule sets involved since they too are processed in the unification. This adds up to contribute to the intractable performance.

We can observe this at the top level of Lego by attempting to unify types `(1)` and `(2)` when we `Freeze` the signature `Nat`. Doing so signals to the system that

`Nat` is not to be expanded. This results in the same amount of delay succeeded by an type error message confirming the failure of the well formed check. But as we have seen, this processing is neither correct nor necessary since both types `(1)` and `(2)` are reducible to type `(3)`. In essence, this inexpediency is a consequence of an aberration in Lego's reduction heuristics. To attempt to rectify them for the sake of speeding up the type checking above may be to the detriment of the other reductions. Work is currently being undertaken to improve Lego's run-time performances.

The problem can be circumvented by forcing Lego's unification process to prohibit expanding and reducing terms to the full extent. Whenever a term is passed to the Lego engine it should be type casted to its normalized type. This can be automatically done at the parsing stage for objects of type `Term`. Doing so means that all the necessary type checking is done beforehand. If this is the case then the signature can be frozen in Lego when proof trees are constructed thus forcing the unification process to unify types `(1)` and `(2)` earlier. The solution is not an extension of Lego but an effective use of its pre-existing mechanisms. It yields an approximate one hundred fold improvement in run-time.

4.5 Other Side Conditions

We can expand the library of side conditions by extending and modifying the existing tools in the theory defined in section 4.1.1. Presently there is only one kind of side condition: term inequality with type

```
{FS|FSig}{s|Sort}(Term|FS s)->(Term|FS s)->Prop
```

The type of side conditions can be generalized to allow for properties taking any number of `Terms` as arguments. A side condition becomes a pair (P, a) of a property P and a list of arguments a of the right arity. The type of such properties would be

```
{FS|FSig}{sl|list|Sort}(Tlist|FS sl)->Prop
```

and one can get an instance of this for term inequality `TermNeq` if its type is changed equivalently to

```
{FS|FSig}{s|Sort}(Tlist (cons s (cons s (nil|Sort))))->Prop
```

and if we refer to it as `TermNeq|s` for given `s`.

With this change, the constructor `SCcons` for `SCList`, the inductive type of side condition lists of section 4.1.1 is transformed to

```
SCcons: {FS|FSig}
        {sl|list|Sort}
        ((Tlist|FS sl)->Prop)->  (** property **)
        (Tlist|FS sl)->          (** arguments **)
        (SCList|FS)->            (** rest of side conditions **)
        (SCList|FS)
```

so that the side condition function and the arguments of the side condition are supplied as the data of each element of the list.

As a result of this change, we also have to amend the type of `ConsSCPrf` (section 4.1.1) to

```
ConsSCPrf : {FS|FSig}
            {sl|list|Sort}
            {f:(Tlist|FS sl)->Prop}
            {a:Tlist|FS a}
            {scl:SCList|FS}
            (f a)->                 (** Proof of head **)
            (SCHold scl)->          (** Proof of tail **)
            SCHold (SCcons f a scl) (** Proof of whole **)
```

One would be expected to add any extra side condition properties to the theory where necessary. Libraries of such properties could then be accumulated for properties of common sorts of terms such as natural numbers and booleans.

The list of side conditions

$$[0 \neq suc\, 0]$$

would be expressed in the new style as

```
SCcons (TermNeq|Nat) (tcons (Nat!natural "0")
                     (tcons (Nat!natural "suc 0")
                     (tnil|Nat)))
       (SCnil|Nat)
```

The extensions in this section would be complete if the parsing facilities were extended likewise.

Chapter 5

An Example Semantics

In this chapter we demonstrate all aspects of defining and reasoning with a semantics in our general theory of inductive specifications. The example we choose is a simple functional language and semantics `ExpSem`. In section 5.1 we show how to specify its syntax in terms of our parsing facilities, in section 5.2 we declare the semantic rules of `ExpSem` and section 5.3 exhibits the various methods and provisions available for proof tree construction using some derivations of `ExpSem` as examples. Subsequent sections quote various theorems and proofs of properties of the semantics of `ExpSem`, demonstrating how to reason about semantics in our general theory using such techniques as rule induction and case analysis discussed in the previous chapter. Theorems include the proof of the monogenicity of `ExpSem` as well as a proof that the `let` construction and the function application construction are equivalent.

5.1 The Syntax of `ExpSem`

The basic entities of the semantics for `ExpSem` are natural number constants, variables, expressions and declarations. The BNF grammar for these is shown in figure 5.1. We can encode this grammar in terms of the commands `Terminals` and `Productions` specified in chapter 3. We commence with the tokens of the language for `ExpSem`, written in Lego as in figure 5.2. The first three lines are included for the basic entities in figure 5.1. The rest of the tokens pertain to the pretty language for the components of rules and substitution.

Before we describe `Exp`, the grammar of the terms of `ExpSem` it is necessary to understand the nature of the relations being specified, to give us an insight into the form of the formulae in its rules. The inductive sets of `ExpSem` relate environments (mappings of identifiers to values) and phrases to values. A phrase being either an expression or a declaration. Values are either natural number expressions (in

Basic Entities		**Generic Name**
Constants **0**, **1**, $\ldots$,	**true**, **false**	n
Identifiers x, y, $\ldots$		x
Phrase Class	**Generic Name**	**Phrase Forms**
Expressions	e ::=	$n \,\vert\, x \,\vert\, e_1\, e_2 \,\vert e_1 + e_2 \vert$ **if** e_0 **then** e_1 **else** $e_2 \,\vert$ **fn** x **.** $e \,\vert$ **let** d **in** e **end**
Declarations	d ::=	$x = e$

Figure 5.1: The Basic Entities of `ExpSem`

```
Terminals "Exp" =
                 "0" "suc"  "+" "++"  "true"  "false"  "if"
                 "then" "else" "x" "y"  "z" "let"  "=" "in"
                 "in" "end"  "fn"  "."  "@"  "("  ")"  "{}"
                 "," "::" ":" "Nat" "==>"  "=:=>" "|-" "<>"
                 "[" "]" "|="  "|-->" "nil";
```

Figure 5.2: The Terminals of `Exp`

terms of Peano arithmetic), function expressions, or environments. The form of the relations can be written as

$$E \vdash \mathit{phrase} \;\Rightarrow\; \mathit{value}$$

where E is an environment and the 'result' value is an expression if the phrase is an expression and an environment if the phrase is a declaration. With this information, we are ready to define the language of the terms of `ExpSem` using the `Productions` command. We start with the portion of the grammar for well formed terms, described in figure 5.3.

There are a number of points to be made about this grammar. Note that the productions labelled under `"ID"`, the ones for describing identifiers, are a simplification of the original declaration of the basic entities of `ExpSem`. There are only three possible identifiers `x`, `y` and `z`. The set can obviously be extended to an infinite one if for example, we define them as an indexed set, using natural numbers as the indexing set for instance.

There are two productions for addition in figure 5.3 since we shall distinguish between the addition of natural numbers and the addition of expressions in the rules of `ExpSem` later. The productions labelled under `"Formula"` give us the forms

```
Productions "Exp"
      FirstOrder  ::=
 "expr" = "zro"        : "0"  |
          "succ"       : "suc" "expr"         |
          "pl"         : "expr" "+" "expr"    |
          "tr"         : "true"       |
          "fls"        : "false"      |
          "Idr"        : "ID"         |
          "ifte"       : "if" "expr" "then" "expr" "else" "expr" |
          "let"        : "let" "decl" "in" "expr" "end"          |
          "abs"        : "fn" "ID" "." "expr" |
          "app"        : "expr" "@" "expr"    |
          "eplus"      : "expr" "++" "expr"   |
          "brexp"      : "(" "expr" ")",
 "ID"   = "x_ID"       : "x"  |
          "y_ID"       : "y"  |
          "z_ID"       : "z" ,
 "decl" = "dcl"        : "ID" "=" "expr",
 "Env"  = "enil"       : "{}" |
          "econs"      : "(" "ID" "," "expr" ")" "::" "Env",
 "Formula"
        = "statNat"    : "expr" ":" "Nat"               |
          "evalE"      : "Env" "|=" "expr" "==>" "expr" |
          "evalD"      : "Env" "|=" "decl" "=:=>" "Env" ,
```

Figure 5.3: The Grammar of Terms in Exp

of the three types of formulae to be defined, the first statNat can be thought of as a relation defining the static semantics for the portion of expressions that are natural numbers. Finally note that a bracketing production has been added to the grammar for expressions since there is no facility in the parser to deal with precedence or associativity information.

The rest of the grammar for ExpSem (describing the form of rules and substitution) will be given later in this section. We shall now concentrate on the effects of this first part of the grammar on the Lego context. Assuming that no sorts have been specified by the user already, the definitions

```
[expr = make_Sort (suc (suc zero))]
[ID   = make_Sort (suc (suc (suc zero)))] ...
...[Env  = make_Sort (suc (suc (suc (suc (suc zero)))))]
```

are added to the current context, where the numerical index is distinct for each newly defined sort, in turn making each sort distinct. A similar treatment is given to the names `zro, succ, ...` labelling each production in the grammar in figure 5.3. Each production name `id` necessitates the addition of a new function name definition `idF` to the current context. For `Exp`, this means we add

```
[zroF = make_FIdent zero]
[succF = make_FIdent (suc zero)]...

...[evalEF = make_FIdent (suc (suc ....(suc zero)...))];
```

The next addition to Lego's context is the signature that the grammar denotes. The name of the signature is the name of the language. The sort and function name information is also supplied within the grammar as explained in section 3.5.2.

```
[Exp =
   cons (Pair zroF   (Pair (nil|Sort) expr))
  (cons (Pair succF  (Pair (cons expr (nil|Sort)) expr))
    .      .      .                .
    .      .      .                .
    .      .      .                .
  (cons (Pair evalEF
              (Pair (cons Env (cons expr (cons expr (nil|Sort))))
                    Formula))
  (nil|(prod FIdent (prod (list|Sort) Sort))))...)];
```

Additionally to these basic definitions, a set of abbreviating functions are added to Lego's context. These are the same abbreviations that were described in section 3.1, and the information needed to create them can again be obtained from the details in the signature `Exp`. In this example we get the functions

```
[zro = Exp!expr "0"]
[succ = [expr0:Term|Exp expr]Exp!expr "suc ^expr0"]...
.
.
.
[evalE =
  [Env0:Term|Exp Env]
  [expr0:Term|Exp expr]
  [expr1:Term|Exp expr]
  Exp!Formula "^Env0 |= ^expr0 ==> ^expr1"];
```

```
"Prems"
       = "prems"      : "[" "Prms" "]" |
         "nilPrems"   : "[" "]",
"Prms"
       = "snglPrms"   : "Formula" |
         "consPrems"  : "Formula" "," "Prms",
"SCList"
       = "consSC"     : "expr" "<>" "expr" "," "SCList" |
         "consSC"     : "Env"  "<>" "Env"  "," "SCList" |
             .              .     .     .     .      .
             .              .     .     .     .      .
             .              .     .     .     .      .
         "nilSC"      : "nil",
"Rule"
       = "rule"       : "Prems" "[" "SCList" "]" "|-"  "Formula",
"Subst"
       = "consSub"    : "expr" "|-->" "expr" "," "Subst" |
             .              .      .      .     .     .
             .              .      .      .     .     .
             .              .      .      .     .     .
         "nilSub"     : "nil";
```

Figure 5.4: The Grammar for Rules and Substitution for `ExpSem`

These intermediary definitions are sometimes a helpful substitute for quotations when reasoning about well formed terms, their components and the effect of functions on them. As we shall see in later sections, it is sometimes hard to understand and reason about the make-up of terms in a quoted form especially when functions such as substitution are being applied to them. We wish to write terms in a comprehensible manner in such situations but we need to make sure we can reason about them clearly at the same time. These abbreviations provide a convenient medium since they are written as Lego expressions but are not so concrete as to become unreadable and not as opaque as a quote in that the structure of the term is evident and readily accessible.

We can now complete the grammar of `ExpSem` by providing its productions for rules and substitution. They are similar to the productions in the previous chapter for the language `Nat` and are shown in figure 5.4.

Note that we need a production in the `SCList` and `Subst` groups for every sort (including `Formula`) in the grammar of figure 5.3. A rule in `ExpSem` has the form

```
Exp!Rule "[ p1, p2, ..., pN][ sc1, sc2, ..., scM] |- c"
```

Expressions e

Constant: $$\frac{}{E \vdash n \Rightarrow n}\; n : natural$$

Identifier: $$\frac{}{E \vdash x \Rightarrow n}\; x \mapsto n \in E$$

Addition: $$\frac{E \vdash e_1 \Rightarrow n_1 \quad E \vdash e_2 \Rightarrow n_2}{E \vdash e_1 + e_2 \Rightarrow n}\; n_1 + n_2 \mapsto n$$

Conditional: (1) $$\frac{E \vdash e_1 \Rightarrow n}{E \vdash \textbf{if true then}\, e_1 \,\textbf{else}\, e_2 \Rightarrow n}$$

(2) $$\frac{E \vdash e_2 \Rightarrow n}{E \vdash \textbf{if false then}\, e_1 \,\textbf{else}\, e_2 \Rightarrow n}$$

Let: $$\frac{E \vdash d \Rightarrow E' \quad E' \vdash e \Rightarrow n}{E \vdash \textbf{let}\, d \,\textbf{in}\, e \,\textbf{end} \Rightarrow n}$$

Function: $$\frac{}{E \vdash \textbf{fn}\, x.e \Rightarrow \textbf{fn}\, x.e}$$

Application: $$\frac{E \vdash e_1 \Rightarrow \textbf{fn}\, x.e \quad E \vdash e_2 \Rightarrow n \quad \{x \mapsto n\} \oplus E \vdash e \Rightarrow n'}{E \vdash e_1(e_2) \Rightarrow n'}$$

Declarations d

Simple: $$\frac{E \vdash e \Rightarrow n}{E \vdash x = e \Rightarrow \{x \mapsto n\} \oplus E}$$

Figure 5.5: The Semantics for `ExpSem`

where `p1, ... pN` and `c` are `Formula`e quotes and `sc1, ..., scM` are `SCList` quotes.

5.2 The Semantics of ExpSem

The semantic rules of `ExpSem` are shown in figure 5.5. It has the basic attributes of a functional language, but demonstrates fully the complexity of our general theory. It is important to point out that environments E in `ExpSem` can be thought of as functions upon which overriding can take place. This is represented as terms of the form $E_1 \oplus E_2$ meaning that the maps in E_1 override the maps in E_2, so for example if we have $(x \mapsto n) \in E_1$ and $(x \mapsto n') \in E_2$ then we have $\{x \mapsto n\} \in E_1 \oplus E_2$. We shall now iterate through the rules in figure 5.5 and show their counterparts in our theory in Lego.

The Constant rule can be expressed as two rules that in effect define the subset of expressions that are natural numbers (in Peano arithmetic). Before we provide these, we need to define the side condition attached to this rule in terms of two inductive rules:

```
[NatRule1 =
Exp!Rule "[][nil] |- 0 : Nat"
];

[NatRule2 =
Exp!Rule "[expr~1 : Nat][nil] |- suc expr~1 : Nat "
];
```

These rules can be seen as defining a part of the *static* semantics of `ExpSem` in that the relation being defined is one concerning the types of expressions. With the side condition defined, we can represent the Constant rule with two rules. The first for zero and the second for higher numbers:

```
[ZeroRule =
Exp!Rule "[][nil] |- Env~1 |= 0 ==> 0"
];

[SuccRule =
Exp!Rule "[expr~1 : Nat]
          [nil] |-
          Env~1 |= suc expr~1 ==> suc expr~1 "
];
```

The Identifier rule in figure 5.5 is subtle in the fact that environments are functions upon which functional overriding takes place as stated in the rules Let and Application. This can be implemented by representing environments as stacks, or FIFO lists. Whenever a binding takes place (in either Let or Application), the bound identifier together with the attached value is pushed onto the stack. The most recent mapping for a particular identifier is the one nearest the top of the stack and is therefore the correct map to refer to when its value is needed. This feature can be realized by the two rules:

```
[IdentRule1 =
Exp!Rule "[][nil] |-
          (ID~1,expr~1)::Env~1 |= ID~1 ==> expr~1"
];
```

```
[IdentRule2 =
Exp!Rule "[Env~1 |= ID~1 ==> expr~2]
          [ID~1 <> ID~2, nil ] |-
          (ID~2,expr~1)::Env~1 |= ID~1 ==> expr~2 "
];
```

where the first rule is for the case when the identifier to reference is at the front of the environment list, and the second when it is not. Note that the second rule has to have a side condition asserting an inequality between identifiers. If the identifier we are interested in is not equal to the identifier involved in the mapping at the top of the environment stack, it is "popped" and the premiss effectively lets us us recurse with the rest of the environment.

The Addition rule has again a side condition we must define in terms of inductive rules. The addition in the side condition for this rule pertains to addition for natural numbers whereas the addition in the conclusion of the rule corresponds to the addition of expressions. We distinguish the two in our language by using the symbol "+" to denote the former and "++" to denote the latter. The former can be defined using two rules

```
[PlusRule1 =
Exp!Rule "[expr~1 : Nat]
          [nil] |-
          Env~1 |= 0 + expr~1 ==> expr~1 "
];


[PlusRule2 =
Exp!Rule "[Env~1 |= expr~1 + suc expr~2 ==> expr~3]
          [nil] |-
          Env~1 |= suc expr~1 + expr~2 ==> expr~3 "
];
```

and the latter, the representation of the Addition rule is written

```
[ExpPlusRule =
Exp!Rule "[Env~1 |= expr~1 ==> expr~4,
           Env~1 |= expr~2 ==> expr~5,
           Env~1 |= expr~4 + expr~5 ==> expr~3]
          [nil] |-
          Env~1 |= expr~1 ++ expr~2 ==> expr~3 "
];
```

The conditional rules are a straightforward translation:

```
[IfRule1  =
Exp!Rule "[Env~1 |= expr~1 ==> expr~3]
          [nil] |-
          Env~1 |= if true then expr~1 else expr~2 ==> expr~3"
];

[IfRule2  =
Exp!Rule "[Env~1 |= expr~2 ==> expr~3]
          [nil] |-
          Env~1 |= if false then expr~1 else expr~2 ==> expr~3"
];
```

The only boolean terms in `ExpSem` are true and false but we could extend `ExpSem` to have more complex boolean terms by defining operators such as and, or and implication by adding the inductive rules for them as we did for natural number addition.

The Let rule in figure 5.5 provides the definition of a construct familiar from many functional languages. We must somehow make sure that the new environment E' created by the declaration d overrides the current environment in the body of the expression e. The rule we write in Lego is:

```
[LetRule =
Exp!Rule "[Env~1 |= decl~1 =:=> Env~2, Env~2 |= expr~1 ==> expr~2]
          [nil] |-
          Env~1 |= let decl~1 in expr~1 end ==> expr~2"
];
```

where the variable `Env~2` is taken to be `Env~1` overridden by the mapping in the declaration `decl~1`. To guarantee that this is so, we define the rule for declarations to be:

```
[DeclRule =
Exp!Rule "[Env~1 |= expr~1 ==> expr~2]
          [nil] |-
          Env~1 |= ID~1 = expr~1  =:=> ( ID~1 , expr~2 ) :: Env~1"
];
```

so that the mapping of the identifier `ID~1` to the expression `expr~2` is placed at the head of the environment. The overriding is effectively taking place in the rule for declarations. The rule for functions and function applications are straightforward:

```
[FnRule =
Exp!Rule "[]
          [nil] |-
          Env~1 |= fn ID~1.expr~1 ==> fn ID~1.expr~1"
];
[AppRule =
Exp!Rule "[Env~1 |= expr~1 ==> fn ID~1. expr~3,
           Env~1 |= expr~2 ==> expr~4,
           (ID~1,expr~4 )::Env~1 |= expr~3 ==> expr~5]
          [nil] |-
          Env~1 |=  expr~1 @ expr~2  ==> expr~5"
];
```

Finally, we need a rule for bracketing expressions so that the parser returns the parse tree of the user's choosing. The grammar for expressions is ambiguous and so explicit parenthesizing is necessary. The rule for brackets is:

```
[BracRule =
Exp!Rule "[Env~1 |= expr~1 ==> expr~2]
          [nil] |-
          Env~1 |= ( expr~1 ) ==> expr~2"
];
```

With all the rules defined in Lego, the rule set `ExpRules` of `ExpSem` is the list in figure 5.6. Now, we can define `ExpSem` itself as the specification

```
[ExpSem = (dep_pair|FSig|RuleSet Exp ExpRules):Spec];
```

which is the pair of the signature and rule set.

```
[ExpRules =
        (Ncons ZeroRule   (* zero *)
        (Ncons SuccRule
        (Ncons NatRule1   (* two *)
        (Ncons NatRule2
        (Ncons IdentRule1 (* four *)
        (Ncons IdentRule2
        (Ncons PlusRule1  (* six *)
        (Ncons PlusRule2
        (Ncons ExpPlusRule (* eight *)
        (Ncons IfRule1
        (Ncons IfRule2     (* ten *)
        (Ncons LetRule
        (Ncons DeclRule   (* twelve *)
        (Ncons FnRule
        (Ncons AppRule    (* fourteen *)
        (Nnil  BracRule))))))))))))))):RuleSet|Exp];
```

Figure 5.6: The Rule Set for `ExpSem`

5.3 Example derivations

We shall now demonstrate how proof trees can be constructed in both bottom up and top down fashion. We will use the proof tree for the assertion

$$\begin{aligned}\{\} \vdash\ &\mathbf{let}\, x\ = (suc\, suc\, 0)\\ &\mathbf{in}\\ &\quad 0\ +\ x\\ &\mathbf{end}\ \ \Rightarrow\ \ suc\, suc\, 0\end{aligned}$$

as an example. In formal terms, the derivation for this formula looks like

$$\frac{\dfrac{\dfrac{\dfrac{\overline{0 : Nat}}{suc\, 0\ :\ Nat}}{\{\} \vdash\ suc\, suc\, 0\ \Rightarrow\ suc\, suc\, 0}}{\{\}\vdash\ x = suc\, suc\, 0\ \Rightarrow\ \{x \mapsto suc\, suc\, 0\}} \qquad \dfrac{\begin{array}{c}\overline{\{x \mapsto\ suc\, suc\, 0\}\vdash\ x\ \Rightarrow\ suc\, suc\, 0}\\ \vdots\end{array}}{\{x \mapsto suc\, suc\, 0\}\vdash\ 0 + x\ \Rightarrow\ suc\, suc\, 0}}{\{\}\vdash\ let\, x =\ (suc\, suc\, 0)\ in \ldots\ \Rightarrow\ suc\, suc\, 0}$$

This section is divided into three parts. The first shows how to construct the proof tree in a top down fashion, the second in a bottom up manner and the third discusses a derivation that includes proving a side condition.

5.3.1 Top Down Derivations

We start by executing the command `Freeze Exp` in Lego. This makes sure that it cannot expand the definition of the signature `Exp` during the course of its evaluations of terms and types. As explained in section 4.4, this is necessary to save time especially when as here, derivations are being built. We then declare our Lego goal to be that the assertion above is a `Judgement`:

```
Goal
Judgement ExpRules
          (Exp!Formula " {} |= let x = suc suc 0
                                in
                                   0 ++ x
                                end ==>
                               suc suc 0"
          );
```

The first rule we wish to apply is the `LetRule` with the substitution

```
[sub1 = Exp!Subst "Env~1  |--> {},
                   decl~1 |--> x = suc suc 0,
                   expr~1 |--> 0 ++ x,
                   expr~2 |--> suc suc 0,
                   Env~2  |--> ( x , suc suc 0 ) :: {},
                   nil"
];
```

The rule has no side conditions, so we can break the goal down by refining it with

```
ruleAp ExpRules (suc ten) sub1 ? (NilSCPrf|Exp)
```

which gives us the subgoal

```
Jlist ExpRules (TlistSubFn sub1
                           (Prems (RuleNum ExpRules (suc ten))))
```

which we know pertains to the premisses of the let rule under the substitution `sub1`. We know therefore that the judgement list we require has two elements since `LetRule` has two premisses, so we can refine this subgoal by `jcons` since we know the list must be non-empty. Going further, we can be more specific and refine it by

```
jcons ExpRules ? (jcons ExpRules ? (jnil ExpRules))
```

using the Lego wildcard symbol `?` since we know the list must have exactly two judgements in it. It is easier to use this "template" expression rather than simply `jcons` since refining by the latter gives us a more complicated looking (although equivalent) subgoal to prove. Refining by the template expression gives us the new subgoals

```
  ?6  : Judgement ExpRules
                (TSubFn sub1
                      (Exp!Formula "Env~1 |= decl~1 =:=> Env~2"))
  ?10 : Judgement ExpRules
                (TSubFn sub1
                      (Exp!Formula "Env~2 |= expr~1 ==> expr~2"))
```

These relate to the two premisses of `LetRule` under substitution `sub1`. As can be seen it is not at all clear what the actual instantiations of the formulae are since a substitution is applied to the quoted formulae. The problem is exacerbated since the next step in the proof is to apply different rules to the new subgoals which means making use of another substitution in order to refine by `ruleAp`. This can make the construction of proof trees hard to manage correctly. There are no ways of influencing Lego's reduction strategies on parts of expressions other than using the command `Equiv`, where the user postulates an equivalent goal to the one for them to prove. In this case, we can assert that the goal `?6` is equivalently

```
?6 : Judgement ExpRules
               (Exp!Formula "{} |= x = suc suc 0 =:=>
                                   (x, suc suc 0)::{}");
```

in that the substitution `sub1` has been applied to the previous expression of `?6`. We can carry on and prove this goal using the rule for declarations and so on.

As can be seen, supplying a substitution every time a rule is applied can make derivations exceedingly prolix. The function `unifSub` defined in section 4.3.5 can be used in its place to make Lego find a substitution that unifies the goal formula with a conclusion of a provided rule. Consider our initial goal. Let us make a definition

```
[goal_formula = Exp!Formula " {} |= let x = suc suc 0 in ..."];
```

being the formula of the judgement that is our goal. We could refine the goal by

```
Refine ruleAp ExpRules
              (suc ten)
              (unifSub (Conc LetRule) goal_formula)
              ?
              (NilSCPrf|Exp);
```

to get the subgoal

```
?5 : Jlist ExpRules
           (TlistSubFn (unifSub (Conc LetRule) goal_formula)
                       (Prems (RuleNum ExpRules (suc ten))))
```

and then refining by

```
Refine jcons ExpRules ? (jcons ExpRules ? (jnil ExpRules));
```

as before, we get the subgoals

```
  ?9 : Judgement ExpRules
             (TSubFn (unifSub (Conc LetRule) goal_formula)
                     (Exp!Formula "Env~1 |= decl~1 =:=> Env~2"))
  ?13 : Judgement ExpRules
              (TSubFn (unifSub (Conc LetRule) goal_formula)
                      (Exp!Formula "Env~2 |= expr~1 ==> expr~2"))
```

Although this does not aid our understanding of the instantiations of the formulae involved, we are at least spared the trouble of determining the concrete values of the goals and defining the substitution beforehand. Again, if we were to want to know the actual value of the formula in subgoal ?9, we could use Equiv to postulate a value for the formula and reduce it so. Another drawback to using this function is that it is time intensive due to the unification process taking place. A consequence of this is that derivations take longer to construct if unifSub is used for each successive rule application. This is because the function has to deal with increasingly complex terms — unless of course the user intervenes regularly to reduce each successive goal using an Equiv step.

As mentioned at the beginning of this section, we must make sure that the signature Exp is frozen at the start of the proof. Indeed, if we left it unfrozen, then a derivation would take approximately sixty times longer to construct. Unfortunately, once all the steps in a top down derivation are executed, the system requires that the signature be unfrozen before the goal can be proven. It then executes some necessary type checks that involve expanding the value of the signature and evaluations thereon similar to the ones we avoided during the interactive steps of the proof. This virtually wipes out the speed-up gained by freezing, but at least it can be said that the interactive part of the process is completed by this time.

5.3.2 Bottom Up Proofs

We now demonstrate the construction of the same proof tree in a bottom up manner. In this case we need to provide each substitution for every step in the derivation since there is no one initial goal as our focus. Let us start by building the sub-proof for the subgoal

```
Judgement ExpRules (Exp!Formula " {} |= x = suc suc 0 =:=>
                                        {x,suc suc 0}::{} ")
```

We build the proof tree for this with a succession of definitions in Lego. We first start with the command "`Freeze Exp;`" so that Lego avoids any superfluous evaluation. The first step in the proof of this branch of the proof tree (given at the beginning of section 5.3) is to construct the derivation of `0: Nat`. This is realized by the definition

```
Lego> [J1 = ruleAp ExpRules
                   two
                   (Exp!Subst "nil")
                   (jnil ExpRules)
                   (NilSCPrf|Exp)];
defn  J1 : Judgement ExpRules (TSubFn (Exp!Subst "nil")
                                      (Conc (RuleNum ExpRules two)))
(*    J1 :Judgement ExpRules (Exp!Formula "0 : Nat")             *)
```

where the type added in `(*comments*)` for intelligibility is equivalent to the type returned by Lego since it refers to the conclusion of `NatRule1` with the nullary substitution applied to it. From `J1` we can now build the derivation for `suc 0 : Nat` with

```
Lego> [subst1 = Exp!Subst "expr~1 |--> 0, nil"];
Lego> [J2 = ruleAp ExpRules
                   three
                   subst1
                   (jcons ExpRules J1 (jnil ExpRules)) (NilSCPrf|Exp)];
```

```
defn  J2 : Judgement ExpRules (TSubFn subst1
                                         (Conc (RuleNum ExpRules three)))
(*    J2 : Judgement ExpRules (Exp!Formula "suc 0 : Nat")         *)
```

although it is not obvious from the output of Lego that the judgement `J1` fits in the rule application for `J2`. Carrying on, the definitions

```
Lego> [subst2 = Exp!Subst "Env~1 |--> {}, expr~1 |--> suc 0, nil"];
Lego> [J3 = ruleAp ExpRules
                   one
                   subst2
                   (jcons ExpRules J2 (jnil ExpRules))
                   (NilSCPrf|Exp)];
defn  J3 : Judgement ExpRules (TSubFn subst2
                                         (Conc (RuleNum ExpRules one)))
(*    J3 : Judgement ExpRules
                    (Exp!Formula "{} |= suc suc 0 ==> suc suc 0") *)
```

give us the proof tree equivalent to an application of the Constant rule for the number two. Finally we need

```
Lego> [subst3 = Exp!Subst " Env~1  |--> {},
                            expr~1 |--> suc suc 0,
                            expr~2 |--> suc suc 0,
                            ID~1   |--> x,
                            nil "]
Lego> [J4 = ruleAp ExpRules
                   (suc (suc ten))
                   subst3
                   (jcons ExpRules J3 (jnil ExpRules))
                   (NilSCPrf|Exp)];
defn  J4 :
  Judgement ExpRules
            (TSubFn subst3
                    (Conc (RuleNum ExpRules (suc (suc ten)))))
(*    J4 :
  Judgement ExpRules
           (Exp!Formula "{} |= x = suc suc 0
                              =:=>
                              (x, suc suc 0)::{}") *)
```

to get the desired proof-tree. As can be seen, the types returned by Lego do not aid our understanding of the derivations we are building. Indeed it can be harder to build derivations bottom up in Lego because the information returned by the system is less helpful than the information provided during a top down proof.

5.3.3 Proving Side Conditions

We shall conclude this section by looking at a top down derivation of a proof tree involving the proof of a side condition. Assume we have the following goal somewhere in a top down derivation:

```
Judgement ExpRules
  (Exp!Formula "(x, 0)::(y, suc 0)::{} |= y ==> suc 0")
```

We need to use the rule `IdentRule2` which has a side condition, an inequality between the meta-variables for identifiers. We can refine this goal by the rule application

```
Refine ruleAp ExpRules five (unifSub ...)
             (jcons ExpRules ? (jnil ExpRules))
             (ConsSCPrf x y ? (NilSCPrf|Exp))
```

since we know there is one premiss (hence one wildcard symbol in the judgement list) and one side condition (hence one `?` in the list of side conditions) in `IdentRule2`, so the refinement spawns two new subgoals equivalent to

```
 n?   : Judgement ExpRules
                 (Exp!Formula "(y, suc 0)::{} |= y ==> suc 0")
 n+1? : TermNeq (Exp!ID "x") (Exp!ID "y")
```

The first goal can be refined by the rule application of `IdentRule1` and the second is solved purely through expanding and reducing the value of `TermNeq x y` which reduces to `trueProp`, whose proof is is trivially true.

5.3.4 Named Rules

Until now we have been applying rules in derivations and the like by referring to their position in the list `ExpRules`. This is an inconvenience when every time a rule is applied, the definition of `ExpRules` has to be consulted to find this numerical index. The alternative is the rule applying function `ruleAp'` from section 4.3.1. Recall that the function determines the position of the named rules in the given rule set and applies `ruleAp` with the appropriate number. If we also take into account that the function `unifSub` can automatically create substitutions for us, top down derivations can be built with a minimum of effort on the user's part. What follows is an example of a rule application in a top down derivation in which we maximize the amount of automation available to us. We start with the goal

```
Judgement ExpRules (Exp!Formula "{} |= suc suc 0 ==> suc suc 0")
```

which we can refine by the term

```
Refine ruleAp' ExpRules
               SuccRule
               (unifSub (Conc SuccRule)
                        (Exp!Formula "{} |= suc suc 0 ==>
                                                 suc suc 0"))
               (jcons ExpRules ? (jnil ExpRules))
               (NilSCPrf|Exp);
```

as can be seen, we are applying `SuccRule` with a substitution to match the goal formula with the conclusion of the rule. Lego returns with the new goal

```
  ?8 : Judgement ExpRules
                 (TSubFn (unifSub (Conc SuccRule)
                   (Exp!Formula "{}|= suc suc 0 ==> suc suc 0"))
                 (Exp!Formula "expr~1 :Nat"))
```

which if we work this out is equivalent to

```
Judgement ExpRules (Exp!Formula "suc 0 : Nat")
```

but again, as when we exploited `unifSub` in section 5.3.1 to save effort, we find that the output of the theorem prover becomes difficult to understand. Another problem is the time taken to infer the rule number and substitution using `ruleAp'` and `unifSub` respectively is over a hundred times slower than the time taken using `ruleAp` and an explicit substitution.

5.4 Monogenicity Theorem for ExpSem

In this section we demonstrate the use of rule induction in a proof of the monogenicity of the semantics of `ExpSem`. The theorem represents the assertion that every expression phrase evaluates to exactly one value. It is expressed in Lego

```
Goal {e1,e2,e3:Term|Exp expr}
     {E:Term|Exp Env}
     {t|Term|Exp Formula}
     {j:Judgement ExpRules t}
     {t'|Term|Exp Formula}
     {j':Judgement ExpRules t'}
     (Term_Eq t  (Exp!Formula " ^E |= ^e1 ==> ^e2 "))->
     (Term_Eq t' (Exp!Formula " ^E |= ^e1 ==> ^e3 "))->
     (Term_Eq e2 e3);
```

Figure 5.7: Monogenicity Theorem for `ExpSem`

as in figure 5.7 The proof proceeds by double rule induction, first on `j` and then on `j'`. Applying this breaks the goal down so that we can prove the property is closed under all the rules in `ExpRules`. After applying `J_Induction` for all judgements `j` above, we get the goal

```
  ?11 : {i:nat}
        {Sub:Subst|Exp}
        {p:Jlist ExpRules (... (RuleNum ExpRules i)))}
        (SCHold (SCSub Sub (SideConds (RuleNum ExpRules i))))->
        (Conj (....) p)->
        {t'|Term|Exp Formula}
        (Judgement ExpRules t')->
        (Term_Eq (TSubFn Sub (Conc (RuleNum ExpRules i)))
                 (Exp!Formula "^E |= ^e1 ==> ^e2"))->
        (Term_Eq t'
                 (Exp!Formula "^E |= ^e1 ==> ^e3"))->
        Term_Eq e2 e3
```

Note that this goal is a statement of closure that is inductive on natural numbers but we wish to transform this somehow to something inductive on the list `ExpRules`. Doing so would mean we could iterate through `ExpRules` to show the property preserves closure. At this point it is necessary to convert goal `?11` using `Rlemma`. If we have a goal such as the above of the form `{i:nat}T` and we refine it by `Rlemma ExpRules ([r:Rule|FS]T')` where `T'` is `T` with all instances of `RuleNum ExpRules i` substituted with `r`, we get a new goal which is effectively the conjunction that `T` is closed under all the rules in `ExpRules`. The appropriate instantiation of `Rlemma` is:

```
Refine Rlemma ExpRules
              ([r:Rule|Exp]
               {Sub:Subst|Exp}
               {p:Jlist ExpRules (TlistSubFn Sub (Prems r))}
               (SCHold (SCSub Sub (SideConds r)))->
               (Conj (...) p)->
               {t'|Term|Exp Formula}(Judgement ExpRules t')->
               (Term_Eq (TSubFn Sub (Conc r))
                        (Exp!Formula "^E |= ^e1 ==> ^e2"))->
               (Term_Eq t'
                        (Exp!Formula "^E |= ^e1 ==> ^e3"))->
                Term_Eq e2 e3);
```

from which we get the the new goal

```
  ?43 : RuleRec ExpRules
              ([r:Rule|Exp]{Sub:Subst|Exp}
               {p:Jlist ExpRules (TlistSubFn Sub (Prems r))}
               (SCHold (SCSub Sub (SideConds r)))->
               (Conj (...) p)->
               {t'|Term|Exp Formula}
               (Judgement ExpRules t')->
               (Term_Eq (TSubFn Sub (Conc r))
                        (Exp!Formula "^E |= ^e1 ==> ^e2"))->
               (Term_Eq t' (Exp!Formula "^E |= ^e1 ==> ^e3"))->
               Term_Eq e2 e3)
```

to which we can successively apply and-introduction iterating through the list `ExpRules`. The rest of the proof amounts to comparing the possible forms of `j` and `j'` and showing that if

```
j  = Judgement ExpRules (Exp!Formula "^E |= ^e1 ==> ^e2 ")
j' = Judgement ExpRules (Exp!Formula "^E |= ^e1 ==> ^e3 ")
```

then either `e2` equals `e3` or we can infer a contradiction. We can examine all the possible forms and permutations of these judgements by looking at the last rule application used to derive them. For instance once we have refined both `j` and `j'` by `J_Induction`, the first case we come to is the one where both `j` and `j'` represent derivations that last applied the rule `ZeroRule`:

```
...
Sub : Subst|Exp
Sub' : Subst|Exp
h_eq :
  Term_Eq (TSubFn Sub (Conc ZeroRule))
          (Exp!Formula "^E |= ^e1 ==> ^e2")
h_eq' :
  Term_Eq (TSubFn Sub' (Conc ZeroRule))
          (Exp!Formula "^E |= ^e1 ==> ^e3")
?101 : Term_Eq e2 e3
```

from which we can trivially prove the goal since both must be 0. The next case is the one where `j` is a derivation using `ZeroRule` and `j'` uses `SuccRule`. The proof state at this point is:

```
...
Sub : Subst|Exp
Sub' : Subst|Exp
h_eq :
  Term_Eq (TSubFn Sub (Conc ZeroRule))
          (Exp!Formula "^E |= ^e1 ==> ^e2")
h_eq' :
  Term_Eq (TSubFn Sub' (Conc SuccRule))
          (Exp!Formula "^E |= ^e1 ==> ^e3")
?121 : Term_Eq e2 e3
```

which is a contradiction since `e1` cannot be both “0” and “suc n” (for some expression “n”) as implied by the hypotheses above. The rest of the proof continues along the same vein.

5.5 Let and Function Application Equivalence Theorem

The theorem in this section is the simple equality between the evaluations of let and function application expressions. We also show how to apply case analysis when we wish to infer premisses from given conclusions. The theorem is

$$\forall E : Env. \forall e_1, e_2, e_3 : expr.$$
$$E \vdash \textbf{let}\, x = e_1 \,\textbf{in}\, e_2 \,\textbf{end} \Rightarrow e_3 \quad \textit{iff} \quad E \vdash (\textbf{fn}\, x.\, e_2)\, e_1 \Rightarrow e_3$$

and its proof proceeds in two steps, proving the implication in both directions. We show the first direction, the second is similar. We must prove the goal

$$\begin{array}{l} \forall E : Env. \forall e_1, e_2, e_3 : expr. \\ \qquad E \vdash \ \mathbf{let}\, x = \ e_1 \,\mathbf{in}\, e_2 \,\mathbf{end} \ \Rightarrow \ e_3 \quad \textit{implies} \quad E \vdash \ (\mathbf{fn}\, x.\, e_2)\, e_1 \Rightarrow \ e_3 \end{array}$$

In essence, the proof proceeds by building a top down proof tree for the function application from a set of "backwards derived" premisses in the proof tree for the let construct. These premisses must be built from the assumption that the formula "**let** $x = e_1$ **in** e_2 **end** $\Rightarrow e_3$" is a `Judgement`. In mathematical terms, if we know this formula is a `Judgement` then we know we must have derived the proof tree

$$\dfrac{\dfrac{\dfrac{\vdots}{E \vdash \ e_1 \ \Rightarrow \ e_4}}{E \vdash \ x = e_1 \ \Rightarrow \ \{x \mapsto e_4\} \oplus E}\ \textit{Decl.} \qquad \dfrac{\vdots}{\{x \mapsto e_4\} \oplus E \vdash \ e_2 \ \Rightarrow \ e_3}}{E \vdash \ \mathbf{let}\, x = e_1 \,\mathbf{in}\, e_2 \,\mathbf{end} \ \Rightarrow \ e_3}\ \textit{Let.}$$

and so we must know that the premisses above are all `Judgements`. More importantly, we know that an e_4 exists such that those premisses may hold. If we now take a look at the proof tree we need to build for the function application:

$$\dfrac{\dfrac{}{E \vdash (\mathbf{fn}\, x.\, e_2) \ \Rightarrow \ (\mathbf{fn}\, x.\, e_2)}\ \textit{Fn.} \quad E \vdash e_1 \ \Rightarrow \ e_4 \quad \{x \mapsto \ e_4\} \oplus E \vdash \ e_2 \Rightarrow e_3}{E \vdash \ (\mathbf{fn}\, x.\, e_2)\, e_1 \ \Rightarrow \ e_3}\ \textit{App.}$$

we can see that once we have inferred the premisses of the **let** construction, we can plug them into this proof tree making sure that e_4, the existential variable is correctly bound.

We shall start with the proof of the theorem that derives the premisses of the Let rule from the assumption of its conclusion. The formula in figure 5.8 states that given a `Judgement` involving the `let` construct, there must exist an expression `e4` such that the premisses of the Let rule under the implied substitution are `Judgements`. We can then use this existential object to bind the `expr~4` variable of the `AppRule` in our main theorem. The theorem is proved using `JRSum` from section 4.3.3 and a long case analysis. After introducing our assumptions we get the proof state

```
Let_derive :
  {E:Term|Exp Env}
  {e1,e2,e3:Term|Exp expr}
  (Judgement ExpRules
             (Exp!Formula "^E|= let x=^e1 in ^e2 end ==> ^e3"))->
  sigma|(Term|Exp expr)
       |([e4:Term|Exp expr]
       Jlist ExpRules
             (cons (Exp!Formula "^E |= x = ^e1 =:=> (x,^e4)::^E")
             (cons (Exp!Formula "(x,^e4)::^E |= ^e2 ==> ^e3")
             (nil|(Term|Exp Formula)))));
```

Figure 5.8: The Let_derive Lemma

```
E : Term|Exp Env
e1 : Term|Exp expr
e2 : Term|Exp expr
e3 : Term|Exp expr
J : Judgement ExpRules
              (Exp!Formula "^E|= let x =^e1 in ^e2 end ==> ^e3")
?1 : sigma|(Term|Exp expr)
     |([e4:Term|Exp expr]
     Jlist ExpRules
           (cons (Exp!Formula "^E |= x = ^e1 =:=> (x,^e4)::^E")
           (cons (Exp!Formula "(x,^e4)::^E |= ^e2 ==> ^e3")
           (nil|(Term|Exp Formula)))));
```

at which point we make the definition "`[JLs = JRSum ExpRules J]`" to provide us with a summation. We can derive our goal from this sum by refining by the term "`case ? ? JLs`". This then gives us two proof obligations. The first being

```
Je_H : sigma|(Subst|Exp)
            |([Sub:Subst|Exp]
         prod (Jlist ExpRules (TlistSubFn Sub (Prems ZeroRule)))
              (Eq (Exp!Formula "^E|= let x=^e1 in ^e2 end ==>^e3")
                  (TSubFn Sub (Conc ZeroRule))))
?7 : sigma|(Term|Exp expr)
     |([e4:Term|Exp expr]
     Jlist ExpRules (cons (Exp!Formula "^E |= ...") ...)
```

which denotes the case where we must derive our goal from the assumption that the premisses of the `ZeroRule` hold and that its conclusion under some substitution matches "`let x = e1 ...`". We can derive a proof by contradiction from the second part of this assumption in the following way. Firstly we can refine by

```
empty_elim ([_:empty]
            sigma|(Term|Exp expr)
                 |([e4:Term|Exp expr]Jlist ExpRules ... ))
```

the elimination operator for the empty type. This means having to provide an object of type `empty` which we do by applying the theorem

```
absurd_impl_empty : absurd -> empty
```

which holds in our theory since we implement Martin-Löf equality. This leaves us to derive `absurd` which we do from the equality in `Je_H` above. It implies that the function name for "`let`" equals the function name for "`0`". The term

```
true_not_false (Eq_sym (fst (fst
                        (snd (snd (Term_Eq_character' ? ?
                                    (Snd (sig_pi2 Je_H))))))));
```

gives us an object of type `absurd` from this implication of `Je_H`. Refining by this finishes the proof of this first case. The second proof obligation mentioned above is

```
 ?6 : (JlistSum ExpRules
              (Ncons SuccRule (Ncons ... (Nnil BracRule)))
              (Exp!Formula "^E|= let x=^e1 in ^e2 ==> ^e3"))->
      sigma|(Term|Exp expr)
           |([e4:Term|Exp expr]
             Jlist ExpRules (cons (Exp!Formula "...") ...)
```

where now we can show our goal holds from a smaller summation than before: a sum over `ExpRules` minus the `ZeroRule`. We continue exactly as before proving each case by contradiction until we reach the `LetRule` case. In this instance we do not have a contradiction but must derive our goal from our immediate assumption. The proof state can be reduced to the one shown in figure 5.9. Notice that the assumption `Je_H` has had to be expanded and stated in the more verbose manner in order to be able to apply rewrites later on. The list in `Je_H` is the expansion of the term `TlistSubFn Sub (Prems LetRule)`. It is from the `Jlist` part of `Je_H`

```
 Je_H : sigma|(Subst|Exp)
            |([Sub:Subst|Exp]
         prod (Jlist ExpRules
                 (cons (evalD (SubFn Sub Env  (suc zero))
                              (SubFn Sub decl (suc zero))
                              (SubFn Sub Env  (suc (suc zero))))
                 (cons (evalE (SubFn Sub Env  (suc (suc zero)))
                              (SubFn Sub expr (suc zero))
                              (SubFn Sub expr (suc (suc zero))))
                 (nil|(Term|Exp Formula)))))
          (Eq (Exp!Formula "^E |= let x  = ^e1 in ^e2 end ==> ^e3")
              (TSubFn Sub (Conc LetRule))))
 ?259 : sigma|(Term|Exp expr)
            |([e4:Term|Exp expr]
             Jlist ExpRules
                   (cons (Exp!Formula "^E|= x=^e1 =:=> (x,^e4)::^E")
                   (cons (Exp!Formula "(x,^e4)::^E|= ^e2 ==> ^e3")
                   (nil|(Term|Exp Formula)))))
```

Figure 5.9: The `LetRule` case in the proof of `Let_derive`

that we shall prove our goal. From the equality in `Je_H` we can impose equalities on the terms of the `Jlist` so that it matches our goal formula list. We can derive the equalities

```
Env1_Eq  : Eq E (SubFn (sig_pi1 Je_H) Env (suc zero))
Decl1_Eq : Eq (Exp!decl "x  = ^e1")
              (SubFn (sig_pi1 Je_H) decl (suc zero))
e2_Eq =  : Eq e2 (SubFn (sig_pi1 Je_H) expr (suc zero))
e3_Eq =  : Eq e3 (SubFn (sig_pi1 Je_H) expr (suc (suc zero)))
```

from the equality in `Je_H` but we cannot define an equality for the `Env~2` variable in the `LetRule` under the substitution of `Je_H`, i.e. for the term

```
SubFn (sig_pi1 Je_H) Env (suc (suc zero))
```

in `Je_H`. It is at this point that we need to backwards derive the premisses of the `DeclRule` as pointed out in the proof tree for the let construction above. We need the theorem `Decl_derive` shown in figure 5.10 which is proven in exactly the same way as we prove the current theorem. With this we can introduce the definition

```
Decl_derive :
  {E,E':Term|Exp Env}
  {e1:Term|Exp expr}
  (Judgement ExpRules (Exp!Formula "^E|= x=^e1 =:=> ^E'"))->
  sigma|(Term|Exp expr)
       |([e2:Term|Exp expr]
         prod (Judgement ExpRules
                         (Exp!Formula "^E|= ^e1 ==> ^e2"))
              (Eq E' (Exp!Env "(x,^e2)::^E")));
```

Figure 5.10: The theorem Decl_derive

```
[J_L2_eH = (Decl_derive E
                        (SubFn (sig_pi1 Je_H) Env (suc (suc zero)))
                        e1
                        (HeadPremiss ExpRules J_L2))
         : sigma|(Term|Exp expr)
                |([e2:Term|Exp expr]
              prod (Judgement ExpRules
                              (Exp!Formula "^E|= ^e1 ==> ^e2"))
                   (Eq (SubFn (sig_pi1 Je_H) Env (suc (suc zero)))
                       (Exp!Env "(x,^e2)::^E")))];
```

giving us not only the equality on Env~2 we desire but also the existential term we require in our goal in figure 5.9. We shall set the value of this existential to be "[e' = sig_pi1 J_L2_eH]" and we now successively rewrite the Jlist term in Je_H shown in figure 5.9 until we reach the proof state

```
...
  e'    : Term|Exp expr
  JL_5 : Jlist ExpRules
               (cons (Exp!Formula "^E|= x=^e1 =:=> (x,^e')::^E")
               (cons (Exp!Formula "(x,^e')::^E |= ^e2 ==> ^e3")
               (nil|(Term|Exp Formula)))
  ?259 : sigma|(Term|Exp expr)
              |([e4:Term|Exp expr]
            Jlist ExpRules
                  (cons (Exp!Formula "^E|= x=^e1 =:=> (x,^e4)::^E")
                  (cons (Exp!Formula "(x,^e4)::^E|= ^e2 ==> ^e3")
                  (nil|(Term|Exp Formula)))))
```

from which we can immediately derive our goal by refining by

```
Refine dep_pair e' ?;
Refine JL_5;
```

The rest of the proof's cases are proved by contradiction as before.

Having proven `Let_derive` and `Decl_derive` we can prove our main result

```
Goal {E:Term|Exp Env}
     {e1,e2,e3:Term|Exp expr}
     (Judgement ExpRules
              (Exp!Formula "^E|= let x=^e1 in ^e2 end ==> ^e3"))->
     (Judgement ExpRules
              (Exp!Formula "^E|= (fn x.^e2) @ ^e1 ==> ^e3"));
```

in Lego. After introducing our assumptions we reach the proof state

```
  E : Term|Exp Env
  e1 : Term|Exp expr
  e2 : Term|Exp expr
  e3 : Term|Exp expr
  J : Judgement ExpRules (Exp!Formula "^E|= let x=^e1
                                            in ^e2 end ==> ^e3")
  ?1 : Judgement ExpRules (Exp!Formula "^E|= (fn x.^e2)@^e1 ==> ^e3")
```

At which point we need to build the proof tree for the function application bearing in mind that the variable `expr~4` in `AppRule` must be determined from our assumption `J`. To do this we introduce the definitions

```
[JL = Let_derive E e1 e2 e3 J];
[e4 = sig_pi1 JL];
```

and then define our first substitution

```
[sub1 = Exp!Subst "Env~1 |--> ^E ,
                   expr~1 |--> ( fn x. ^e2 ) ,
                   expr~2 |--> ^e1 ,
                   expr~3 |--> ^e2 ,
                   expr~4 |--> ^e4 ,
                   expr~5 |--> ^e3 ,
                   ID~1   |--> x,
                   nil"];
```

```
...
  JL : sigma|(Term|Exp expr)
           |([e4:Term|Exp expr]
            Jlist ExpRules
                  (cons (Exp!Formula "^E|= x=^e1 =:=> (x,^e4)::^E")
                  (cons (Exp!Formula "(x,^e4)::^E|= ^e2 ==> ^e3")
                  (nil|(Term|Exp Formula)))))
  e4 = ... : Term|Exp expr
  sub1 = ... : Subst|Exp
  sub2 = ... : Subst|Exp
  sub3 = ... : Subst|Exp
  ?28 : Judgement ExpRules
                 (Exp!Formula "(x,^e4)::^E|= ^e2 ==> ^e3")
  ?22 : Judgement ExpRules (Exp!Formula "^E|= ^e1 ==> ^e4")
  ?29 : Jlist ExpRules (nil|(Term|Exp Formula))
```

Figure 5.11: The Premisses of `AppRule`

Note that `expr~4` is set to the existential term in our backwards derived list of `Judgements` by the substitution `sub1`. After applying `ruleAp` using `AppRule` and `sub1` we need to prove that the premisses of `AppRule` under `sub1` are `Judgements`. The first premiss is easily proved by the `BracRule` and then the `FnRule` — exactly proving the Fn branch of the function application proof tree shown at the start of this section. This leaves us with a proof state reducible to the one in figure 5.11. The last goal can be proved by `jnil`. The first goal can be proved using our `Let_derive` assumption `JL` — more particularly the second `Judgement` in `JL`.

```
Refine HeadPremiss ExpRules
      (TailPremisses ExpRules (sig_pi2 JL));
```

but the second goal must be backwards derived using `Decl_derive` and `JL`. The first `Judgement` in `JL` must have been derived using the `DeclRule`, and it is the premiss of *this* that we need. The definition

```
[J_D_eH = Decl_derive E
                        (Exp!Env "(x,^e4)::^E")
                        e1
                        (HeadPremiss ExpRules (sig_pi2 JL))];
```

Gives us a `Jlist` of the form we require and also an equality we need to impose on `e4`, so that we eventually reach the proof state

```
  J_D_eH : sigma|(Term|Exp expr)
                |([e2:Term|Exp expr]
             prod (Judgement ExpRules
                             (Exp!Formula "^E|= ^e1 ==> ^e2"))
                  (Eq (Exp!Env "(x,^e4)::^E")
                      (Exp!Env "(x,^e2)::^E")))
  e4_Eq : Eq e4 (sig_pi1 J_D_eH)
  J_D1 : Judgement ExpRules (Exp!Formula "^E |= ^e1 ==> ^e4")
  ?22 : Judgement ExpRules (Exp!Formula "^E |= ^e1 ==> ^e4")
```

where `e4_Eq` is defined from `J_D_eH` and `J_D1` is part of `J_D_eH` with the rewrite of `e4_Eq` applied to it. The last goal in our proof is immediate from `J_D1`.

We glossed over many of the Lego specific parts of the proof above. For instance, in the proofs by contradiction we made use of term equality using the theorem

```
Term_Eq_character' :
     {FS|FSig}
     {s|Sort}
     {t1,t2:Term s}
     (Eq t1 t2)->
     (Term_Eq t1 t2)
```

that gave us a `Term_Eq` object. Such constructions are terms in `Prop` consisting of numerous nested conjunctions equating various elements of the terms supplied. To get the right conjunction that produces `absurd` presupposes the knowledge of how `Term_Eq` is structured. The user would either have to study this function (and hence the implementation of terms) or extra theorems and definitions would have to be added to the theory to make such things easier to reason about. In effect, hiding more of the concrete code of Lego from the user.

Another issue worthy of mention is term rewriting. In the proofs above, we could not apply the usual rewrite tool `Qrepl` to the goal. Type check errors would be created as a result. Instead the function `Eq_subst` has to be used to effectively rewrite assumptions. The type of this function is

```
Eq_subst : {t|SET}
           {m,n|t}
           (Eq m n)->
           {P:t->TYPE_minus1}
           (P m)->P n
```

and it is used by defining a new term

```
[T = Eq_subst E F t]
```

where `E` is an equality, `F` is a suitable template function and `t` is the assumption we wish to apply the rewrite (implied by `E`) to. Usually, we can get Lego to help us by leaving the function parameter uninstantiated and let it fill in the gap for us. In this case however, all parameters must be quoted explicitly.

Issues such as these make it harder to prove all but the simplest of theorems about semantics in our general theory without a fairly thorough knowledge of both the underlying theorem prover and the implementation of our general theory. By simple inspection of the proofs above it is clear that the user must be conversant with Lego's libraries of definitions and theorems. It is also an advantage to know the variety of tricks necessary to prove many goals in Lego. Supplements to the theory that help the user avoid implementation specific issues are only helpful if all the possible requirements of a user are covered. Even if such a global library could be provided, using user-friendly theorems is likely to greatly expand the size of the underlying terms. This then makes theorem proving slower as we showed in section 5.3.4 where one could avoid having to reference rules by numbers by supplying their names instead. This coupled with another function that built substitutions automatically resulted in a hundred-fold increase in compute time.

Chapter 6

An Example Transformation and Correctness Proof

This chapter provides an example transformation from Hannan and Miller's work [HM91] discussed in chapter 1 together with its correctness theorem. In our theory, a transformation is a function from semantics[1] to semantics. The theorem relates an arbitrary semantics to the result of applying the transformation to it so that if you can derive a judgement for the former then you can derive a related one for the latter. The next section describes the general transformation and the correctness theorem we shall be implementing in this chapter. The successive sections describe how to realize them in our general theory as well as providing a template for implementing other transformations and correctness theorems.

6.1 Branch Elimination

The transformation we shall be looking at is one from Hannan and Miller's work [HM91, HM90] called *Branch Elimination*. Its effect is to eliminate branches in the proof trees of semantics by making sure that all rules in a rule set have at most one premiss. It is an important transformation in Hannan and Miller's work since it introduces determinism into the rules and therefore produces a semantics one step closer to a parallel of an abstract machine. It does this by taking inference rules and replacing them with rules in which multiple premisses are added to a stack of "formulae to be proven". This stack eliminates branching in inference rules and imposes an order in which formulae are proved. The transformation is defined as follows.

[1]Recall that a semantics is a pair `(FS,RSet)` of a signature `FS` and a rule set `RSet`

Definition 6.1 Branch Elimination Transformation: *Let* Sem *be a semantics. Let* nil *and* :: *be the term constructors for lists of formulae and let* prove *be a one place predicate taking a formula list as its argument. Define* BESem *to be the proof system consisting of the following axioms and inference rules:*

- $\overline{\text{prove nil}} \in \text{BESem}$
- *if* $\dfrac{A_1 \cdots A_n}{A_0}\,\sigma \in$ Sem *for* $n \geq 0$, *then* $\dfrac{\text{prove } A_1{::}\ \cdots\ {::}A_n{::}G}{\text{prove } A_0{::}G}\,\sigma \in$ BESem *for formula list meta-variable* G *not free in any* A_i *for* $i \geq 0$.

The semantics *Sem* and *BESem* are related by the following theorem

Theorem 6.1 *Let* Sem *be a semantics. Then*

$$x \in Ind(\text{Sem}) \ \ iff \ \ prove(x{::}nil) \in Ind(\text{BESem})$$

The proof is by elementary induction on the depth of inference. To see the effect of eliminating branches, take an example semantics *AndSem*

$$\overline{\Gamma,\, A,\, \Gamma' \vdash\, A}$$

$$\frac{\Gamma \vdash\, A \quad \Gamma \vdash\, B}{\Gamma \vdash\, A \wedge B}$$

$$\frac{\Gamma,\, A,\, B \vdash\, C}{\Gamma, A \wedge B \vdash\, C}$$

consisting of the axiom, introduction and elimination rules for conjunction. We can transform *AndSem* to *BEAndSem* to get

$$\overline{prove\ nil}$$

$$\overline{prove\ (\Gamma,\, A,\, \Gamma' \vdash\, A) :: G}$$

$$\frac{prove\ (\Gamma \vdash\, A) :: (\Gamma \vdash\, B) :: G}{prove\ (\Gamma \vdash\, A \wedge B) :: G}$$

$$\frac{prove\ (\Gamma,\, A,\, B \vdash\, C) :: G}{prove\ (\Gamma, A \wedge B \vdash\, C) :: G}$$

where the introduction rule for conjunction is turned into a rule with one premiss by creating a stack from the formulae of the rules in *AndSem*.

6.2 The Transformation in Lego

A transformation in the general theory consists of a function from `Spec` to `Spec`. Recall an object of such a type in our theory is a pair `(FS,RSet)` of a signature and complimentary rule set. In a typical transformation this means both elements of a `Spec` are transformed. We can think of the function as consisting of two main sub-functions. One to transform the signature and the other, the rule set. This is indeed the case for Branch Elimination as we have to add a new sort for formula lists and the function arity definitions for *nil*, :: and *prove* to the signature of a semantics.

The act of adding sorts and function identifiers so that they are indeed new to the signature given can be tackled in a number of ways — not all of them desirable. Recall that sorts and function identifiers (whose types are `Sort` and `FIdent` in the general theory respectively) are essentially objects in a natural number indexed set. Adding a new object be it sort or function identifier to the information of a signature means ensuring that its index is not mentioned in the existing signature. The obvious solution seems to be to iterate through a signature `FS` and find the highest index for function names and sorts within it (`topFIdent FS` and `topSort FS` say) and then to use a higher index than that for the new object. Unfortunately this is undesirable in the detail of our general theory. To see why, recall the type `Sort` of sorts where

```
[botSort = make_Sort zero]
```

is defined as a global sort. If we also closely inspect the `Term` constructor for function applications

```
[fa:{FS|FSig}
    {f:FIdent}
    (Tlist|FS (IDSort1 FS f))->Term|FS (IDSort2 FS f)]
```

we can see from the definitions of `IDSort1` and `IDSort2` in section 2.1.2 that it is possible to use any function identifier `f` not mentioned in a signature `FS` to make a term whose sort is `botSort`. If we now add a new entry to `FS` for the arity of `f` to (to create a new signature `FS'` say) we have a situation in which a term built using `f` has sort `botSort` with respect to `FS`, but which could potentially have a different sort in `FS'`. This is undesirable because we expect that any additions to a signature should be a conservative extension in the sense that if terms have a certain sort `s` with respect to signature `FS` then they should have the same sort with respect to `FS'`, the extension of `FS`. This is especially crucial when proving properties of transformations with respect to substitutions such as `BEConvSubstLemma` in section 6.3.

The same problem arises if we add new sorts to a signature `FS` by mapping their indices to ones higher than `topSort FS`. We could build a term

```
[ T = fa f (tcons (var zero s) (tnil|FS)) ]
```

where the indices of `s` and `f` are `suc (topSort FS)` and `suc (topFIdent FS)` respectively. The term itself has sort `botSort`. But now if we extend `FS` with a new entry, which in the notation of figure 3.1 is

```
f : s -> s
```

to make a new signature `FS'`, then the term above translated over for `FS'` would have sort `s`. This means that theorems such as `BEConvSubstLemma` (essential later on) no longer hold true.

The solution lies in making sure any additions be they sorts, function names (and later on rule sets) are added at the beginning of a sequence. For sorts and function identifiers this means that their indices in a signature are shifted along the appropriate number of places to accommodate for the additions. On top of this we must make sure that when a term is translated from being well formed with respect to a signature to being well formed with respect to the extension of that signature, the index shift is accounted for. Doing so means we no longer have the problem where adding more information to a signature results in the possibility of translating terms of sort `botSort` to terms of a different sort.

Getting back to the transformation example, we need to add one new sort to stand for the list of formulae (it is a specific kind of list since we do not support polymorphism) called `flist`. We also need three new function identifiers: one for the empty formula list `fnil`, one for the `::` constructor in the definition of the transformation (call it `fcons`) and finally the new predicate `prove`. The new entries again in the notation of figure 3.1 are

```
fnil : flist,
fcons : Formula->flist->flist,
prove : flist->Formula
```

Before we can define a function to transform a signature, we need the functions

```
suc_sort  : Sort->Sort
suc_sortL : (list|Sort)->list|Sort
add3_id   : FIdent->FIdent
```

to increase the index(indices) of a sort, sort list and function identifier respectively. The first is a conditional that is the identity function for `botSort` and `Formula` (since they are global values that should not be changed) and adds one to the index of all other sorts. The second function is defined using list recursion and `suc_sort` and the third function adds three to the index of any function identifier (since three new function identifiers are being added).

The next function takes a signature and shifts the sort and function name indices in it by one and three respectively. It is defined using the functions above and its name and functionality are

```
BESig' : FSig->FSig
```

With this function we can now write the signature transformation as the function

```
[BESig = [FS:FSig]
          [flist=make_Sort two][fnil=make_FIdent zero]
          [fcons=make_FIdent one][prove=make_FIdent two]
          cons (Pair fnil (Pair (nil|Sort) flist))
         (cons (Pair fcons
                  (Pair (cons Formula (cons flist (nil|Sort))) flist))
         (cons (Pair prove (Pair (cons flist (nil|Sort)) Formula))
               (BESig' FS)))]
```

In addition to this we prove a couple of theorems that relate a function identifier arity in `FS` to its arity in "`BESig FS`." These are essential to the process of translating terms. They are

```
BESigThm1 :
    {FS:FSig}
    {f:FIdent}
    Eq (suc_sort (IDSort2|FS f))
       (IDSort2|(BESig FS) (add3_id f))
BESigThm2 :
    {FS:FSig}
    {f:FIdent}
    Eq (suc_sortL (IDSort1|FS f))
       (IDSort1|(BESig FS) (add3_id f))
```

and are proven by list induction on the length of signatures.

To retain legibility in the general theory we must state a new set of productions for each new signature. The parsing and printing facilities in chapter 3 do not account for functions on signatures so it is up to the end user to produce an updated grammar for the transformed subject. In our example *AndSem* this means adding the following grammar[2] to the general theory's context:

```
Productions FirstOrder BEAndSig ::=
     "prop"    = tr  : "true" |
                 fls : "false"|
                 aNd : "prop" "/\" "prop",
     "props"   = pnl : "pnil" |
                 pcn : "prop" "," "props",
     "Formula" = prv : "prove" "flist" |
                 frm : "props" "|-" "prop",
     "flist"   = fnl : "nil" |
                 fcn : "Formula" "::" "flist",
     ...
```

To complete the transformation we need a function from rule sets to rule sets. This means writing translation functions for terms, term lists, side condition lists and rules. With the theorems `BESigThm1` and `BESigThm2` above it is possible to write translation functions for terms and term lists by primitive recursion on those types. The functions

```
BEConv :
    {FS|FSig}
    {s|Sort}(Term|FS s)->
            Term|(BESig FS) (suc_sort s)
BEConvTL :
    {FS|FSig}
    {sl|list|Sort}
    (Tlist|FS sl)->
    Tlist|(BESig FS) (suc_sortL sl)
```

convert a term (term list) well formed for an arbitrary signature `FS` to the appropriate term (term list) well formed for `BESig FS`. Similarly the function

```
BEConvSc : {FS|FSig}(SCList|FS)->SCList|(BESig FS)
```

Converts side condition lists for `FS` to ones for `BESig FS` by recursion on `SCLists`.

The next step is to introduce abbreviation functions for the new constructors `fnil`, `fcons` and `prove` to relieve us of writing such terms out in the longhand of concrete Lego much the way we did so in section 3.1. If we define the functions

[2]omitting the productions for rules and substitutions for brevity

```
(* FNIL  : ...Term|BS flist *)
[FNIL  = [FS|FSig][BS = BESig FS]
         [fnil = make_FIdent zero]
         fa fnil (tnil|BS)];

(* FCONS : ...(Term|BS flist)->(Term|BS Formula)->Term|BS flist *)
[FCONS = [FS|FSig][BS = BESig FS]
         [flist = make_Sort two]
         [fcons = make_FIdent one]
         [f:Term|BS Formula]
         [fl:Term|BS flist]
         fa fcons (tcons f (tcons fl (tnil|BS)))];

(* PROVE : ...(Term|BS flist)->Term|BS Formula *)
[PROVE = [FS|FSig][BS = BESig FS]
         [flist = make_Sort two]
         [prove = make_FIdent two]
         [fl:Term|BS flist]
         fa prove (tcons fl (tnil|BS))];
```

in Lego we can write terms involving the new function names more concisely. This helps us not only in writing the transformation but also in proving the transformation theorem.

The next function takes a list of formulae and converts it into a term of sort `flist` whose last element is a variable not occurring in the given list. It's effectively the part of the overall transformation that is responsible for taking the premisses of a rule

$$A_1 \cdots A_n$$

in definition 6.1 and converting them to the term

$$A_1 :: \cdots :: A_n :: G$$

It is defined by list recursion and its functionality is

```
flistify :
    {FS|FSig}
    [BS=BESig FS]
    [flist = make_Sort two]
    (list|(Term|FS Formula))->
    Term|BS flist
```

Bringing all of the functions in this section together we can now define a transformation on rules as

```
(* BERule : {FS|FSig}[BS=BESig FS](Rule|FS)->Rule|BS *)
[BERule =
    [FS|FSig]
    [BS=BESig FS]
    [flist = make_Sort two]
    [r:Rule|FS]
     Pair (cons (PROVE (flistify (Prems r)))     (*1*)
                (nil|(Term|BS Formula)))
    (Pair (BEConvSc (SideConds r))               (*2*)
          (PROVE (FCONS (BEConv (Conc r))        (*3*)
                        (var|BS zero flist))))   (*4*)
];
```

where line (*1*) indicates the transformation on the premisses of a rule r, line (*2*) is the transformation for its side conditions and the subsequent lines represent the translation for its conclusion. The transformation on rule sets is the function

```
BERuleSet : {FS|FSig}[BS=BESig FS](RuleSet|FS)->RuleSet|BS
```

that converts each rule in `RSet` using `BERule` and adds the additional rule

```
(Pair (nil|(Term|BS Formula))
(Pair (SCnil|BS)
      (PROVE (FNIL|FS))))
```

analogous to the rule

$$\overline{\textit{prove nil}}$$

in the transformation's definition to the front of the converted `RSet`. New rules are added to the front of a new rule set for similar reasons as to why new sort and function name indices were mapped to the start of their enumerating sequences. If a rule set `RSet` is of length `i` then `RuleNum RSet (suc i)` returns the same rule as `RuleNum RSet i` (cf. the definition of `RuleNum` in section 4.1.3) by default. So if we add a new rule to the end of `RSet` to get `RSet'` and we call `RuleNum` with `RSet'` and `suc i` we get that new rule and not the `i`th one. So the equality

```
Eq (RuleNum RSet i) (RuleNum RSet' i)
```

does not hold for all `RSet` and `i`. We have no way of relating rules from `RSet` to `RSet'` — which is something necessary in a transformation proof. If we instead add the new rule to the front to get the rule set `RSet''`, we can relate it to `RSet` by the equality

```
Eq (RuleNum RSet i) (RuleNum RSet'' (suc i))
```

In the next section we will see that the `ith` rule in the old semantics is related to the `(suc i)`th rule in the new semantics in this way although the relationship is not the equality mentioned in this specific example.

Finally the complete transformation `BETrans` is defined as a function taking a dependent pair of a signature and rule set and returning a new dependent pair of a transformed signature and a transformed rule set:

```
(* BETrans : Spec -> Spec *)
  [BETrans = [Sem:Spec](
             (dep_pair (BESig (sig_pi1 Sem))
                        (BERuleSet (sig_pi2 Sem))):Spec)]
```

6.3 The Branch Elimination Theorem

This section outlines the steps up to and including the proof of the transformation theorem for Branch Elimination. The theorem itself amounts to showing that if for a given rule set `RSet` we can derive a `Judgement` for a formula `f` then we can also derive an appropriate `Judgement` from `BERuleSet RSet` for the transformation of `f`. The structure of the transformation theorem proving process here acts as an example template for proving other such theorems for transformations in our general theory.

Before we quote and prove the theorem it is imperative that a number of functions and theorems be proved. Most of the sub-theorems we need here involve substitution. This is the prominent function in the definition of the `Judgement` type. To begin with we define the function

```
flistify2 :
    {FS|FSig}
    [BS=BESig FS]
    [flist=make_Sort two]
    (list|(Term|FS Formula))->
    Term|BS flist
```

identical to `flistify` in the previous section except that instead of appending a variable to the end of the `flist` term, it adds an `FNIL` term. From this we define a function that takes the formula list $f_1, \ldots, f_n$ and transforms it into the formula $\mathit{prove}\,(f_1 :: \cdots :: f_n :: \mathit{nil})$. The function

```
[BEConc =
        [FS|FSig]
        [lt:list|(Term|FS Formula)]
        PROVE (flistify2 lt)];
```

does this using `flistify2`. The function will be used below in representing the effect of Branch Elimination on a ground instance of the conclusion of a rule. The effect of Branch Elimination on the premisses of a rule is captured by the function

```
[BEFlist =
     [FS|FSig]
     [BS=BESig FS]
     [lt:list|(Term|FS Formula)]
     cons (PROVE (flistify2 lt))
          (nil|(Term|BS Formula))]
```

converting a list $[f_1, \ldots, f_n]$ to the list $[\mathit{prove}\,(f_1 :: \cdots :: f_n :: \mathit{nil})]$, a ground instance of the premiss of a rule where the free variable added by `BERule` is instantiated to `FNIL`. We now begin to look at the effect of Branch Elimination on substitutions. Begin by defining a function

```
BESub : {FS|FSig}
        (Subst|FS)->
        (list|(Term|FS Formula))->
        Subst|(BESig FS)
```

that takes a substitution `Sub` and a list of formulae `fl` and converts `Sub` into one well formed for the transformed signature with an additional entry that maps the free variable introduced in `BEConv` to `flistify2 fl`. The reason being that in our main theorem we shall be taking a rule

$$\frac{\mathit{prove}\ f_1 :: \cdots :: G}{\mathit{prove}\ f :: G}$$

and substituting an arbitrary list of formulae for the variable G. This makes the main theorem stronger but at the same time provable by induction. We must

$$\begin{tikzcd}
C \arrow[r, "Be"] \arrow[d, "Sub"'] & C_{Be} \arrow[d, "Sub_{Be}"] \\
C^{Sub} \arrow[r, "BEConc"'] & C_{Be}^{Sub_{Be}}
\end{tikzcd}$$

Figure 6.1: Commutativity Diagram for theorem `BEHom2`

now prove a theorem that states it is equivalent to either apply a substitution `Sub` to a term `t` and then transform that term or to transform `t` first and then apply `BESub Sub` to it. The form of the theorem is

```
BEConvSubstLemma:
        {FS|FSig}
        {fl:list|(Term|FS Formula)}
        {Sub:Subst|FS}
        {s|Sort}
        {t:Term|FS s}
        Eq (BEConv (TSubFn Sub t))
           (TSubFn (BESub Sub fl) (BEConv t))
```

and is proven by induction on the `Term` type. From this important theorem we can prove three similar theorems that say that to apply the transformation to a rule set and then apply a substitution to a constituent of one of the rules is equivalent to applying the substitution to the rule and then transforming that constituent. The equality we require for the conclusions of rules can be represented as demonstrating the commutativity of the diagram in figure 6.1 where subscripted terms are ones with the transformation applied to them, superscripted terms are ones with substitution applied and where the arrow marked `Be` takes the conclusion of a rule

```
C = Conc (RuleNum RSet i)
```

to the conclusion of the transformed rule

```
C_be = Conc (RuleNum (BERuleSet RSet) (suc i))
```

It is vital to prove these sub-theorems since the transformation proof commences by induction on judgements and it is here that substitution is applied to the constituents of rules. The equality theorems are used for rewriting the

```
BEHom1 :
    {FS|FSig}
    {fl:list|(Term|FS Formula)}
    {Sub|Subst|FS}
    {RSet:RuleSet|FS}
    {i:nat}
Eq (BEFlist (append (TlistSubFn Sub (Prems (RuleNum RSet i))) fl))
   (TlistSubFn (BESub Sub fl)
               (Prems (RuleNum (BERuleSet RSet) (suc i))));
BEHom2 :
    {FS|FSig}
    {fl:list|(Term|FS Formula)}
    {Sub:Subst|FS}
    {RSet:RuleSet|FS}
    {i:nat}
Eq (BEConc (cons (TSubFn Sub (Conc (RuleNum RSet i))) fl))
   (TSubFn (BESub Sub fl)
           (Conc (RuleNum (BERuleSet RSet) (suc i))));
BEHom3 :
    {FS|FSig}
    {fl:list|(Term|FS Formula)}
    {Sub:Subst|FS}
    {RSet:RuleSet|FS}
    {i:nat}
(SCHold (SCSub Sub (SideConds (RuleNum RSet i))))->
(SCHold (SCSub (BESub Sub fl)
               (SideConds (RuleNum (BERuleSet RSet) (suc i)))))
```

Figure 6.2: BEHom Theorems

formulae in `Judgement` types to shift the transformation to entirely within the scope of the substitution. Terms matching the left hand side of the equality are rewritten to the terms on the right hand side. This is essential when in the proof of the main theorem we need to apply `ruleAp`. The outermost function in the types of the main arguments to this constructor is substitution.

The sub-theorems we need in our theory are stated in figure 6.2 and are proven by induction on the length of rule sets as well as induction on natural numbers.

Each of the theorems in figure 6.2 rely on the proof of three respective lemmas. These lemmas prove the properties of the `BEHom` theorems not in terms of a `RuleNum` index `i` but in terms of an arbitrary rule `r`. For example the lemma we need in the proof of `BEHom2` is

```
BEHomLemma2 : {FS|FSig}
              {fl:list|(Term|FS Formula)}
              {Sub:Subst|FS}
              {r:Rule|FS}
              Eq (BEConc (cons (TSubFn Sub (Conc r)) fl))
                 (TSubFn (BESub Sub fl) (Conc (BERule r)));
```

and is proven (as indeed `BEHomLemma1` and `BEHomLemma3` are) by breaking down the structure of rules and making use of `BEConvSubstLemma`. They are used in the proofs of the `BEHom` theorems once induction (for both rule sets and natural numbers) has been applied.

We can now quote and prove the transformation theorem for Branch Elimination. The theorem we write in Lego is

```
BETheorem : {FS:FSig}
            {RSet:RuleSet|FS}
            [BRSet = BERuleSet RSet]
            {f|Term|FS Formula}
            (Judgement|FS RSet f)->
            {fl'|list|(Term|FS Formula)}
            (Jlist BRSet (BEFlist fl'))->
            (Judgement BRSet (BEConc (cons f fl')));
```

which states that given any formula f that is a `Judgement` of a rule set `RSet` and given any list of formulae *fl* for which we can show *prove fl*, we can show *prove* $(f :: fl)$. This is slightly more general than we desire but it makes the proof by induction on the depth of inference possible. The proof proceeds as follows. After the relevant assumption introductions we proceed by induction on judgements using the term

```
Judgement_elim RSet ([f|Term|FS Formula]
                      [_:Judgement|FS RSet f]
                      {fl'|list|(Term|FS Formula)}
                      (Jlist BRSet (BEFlist fl'))->
                      Judgement BRSet (BEConc (cons f fl')))
                     ([fl|list|(Term|FS Formula)]
                      [jl:Jlist RSet fl]
                      {fl'|list|(Term|FS Formula)}
                      (Jlist BRSet (BEFlist fl'))->
                      (Jlist BRSet (BEFlist (append fl fl'))));
```

```
  i : nat
  Sub : Subst|FS
  jl : Jlist RSet (TlistSubFn Sub (Prems (RuleNum RSet i)))
  sch : SCHold (SCSub Sub (SideConds (RuleNum RSet i)))
  jl_ih :
    {fl'|list|(Term|FS Formula)}
    (Jlist BRSet (BEFlist fl'))->
    Jlist BRSet
       (BEFlist (append (TlistSubFn Sub (Prems (RuleNum RSet i)))
                        fl'))
  fl' | list|(Term|FS Formula)
  Jl_fl' : Jlist BRSet (BEFlist fl')
  ?17 : Judgement BRSet
              (BEConc (cons (TSubFn Sub (Conc (RuleNum RSet i)))
                            fl'))
```

Figure 6.3: Proof state for `ruleAp` case

to give us three subgoals, the `ruleAp`, `jnil` and `jcons` cases. Notice how applying induction for `Judgements` entails additionally writing a template function for judgement lists which in this instance means taking a list of judgements for the formulae `fl` in `RSet` together with a list of Judgements for the formula list `BEFlist fl'` and deriving a singleton judgement list in `BRSet` where the `fl` and transformed `fl'` are put into a "formulae to prove" stack by the transformation.

We start with the important `ruleAp` case. The proof state we get is displayed in figure 6.3. At this point we need to use the `BEHom` theorems in figure 6.2 since we need to rewrite the goal and some of the assumptions so that the transformation is "moved" to within the application of substitution to both the premisses and conclusion of rule number `i`. The theorems `BEHom1` and `BEHom2` in figure 6.2 allow us to perform such rewrites in conjunction with Lego's `Qrepl` command and `Eq_subst` function. The theorem `BEHom3` is different. It lets us derive a proof of the side conditions for a branch eliminated rule set from a derivation of the side conditions of the original rule set. By entering the commands

```
[JL = jl_ih Jl_fl'];
[JL1 = Eq_subst (BEHom1 fl' RSet i) ? JL];
[SCH = BEHom3 fl' Sub RSet i sch];
Qrepl BEHom2 fl' Sub RSet i;
```

we can get a `Jlist` from the induction hypothesis (and then rewrite it), a side condition list using `BEHom3` and rewrite the goal to obtain the proof state

```
...
  JL1 :
    Jlist BRSet
          (TlistSubFn (BESub Sub fl')
                      (Prems (RuleNum BRSet (suc i))))
  SCH :
    SCHold (SCSub (BESub Sub fl')
                  (SideConds (RuleNum BRSet (suc i))))
  ?28 : Judgement BRSet
                  (TSubFn (BESub Sub fl')
                          (Conc (RuleNum BRSet (suc i))))
```

which is easily derivable by refining by

```
Refine ruleAp BRSet (suc i) (BESub Sub fl') JL1 SCH;
```

Since we had refined by `Judgement_elim` earlier we must prove the two cases for `Jlists`. The `jnil` case gives us the proof state

```
...
  fl' | list|(Term|FS Formula)
  Jl_fl' : Jlist BRSet (BEFlist fl')
  ?31 : Jlist BRSet (BEFlist (append (nil|(Term Formula)) fl'))
```

which is simply refined by `Jl_fl'`. Finally, the proof state for the `jcons` case looks like

```
...
  jh_ih :
    {fl'|list|(Term|FS Formula)}
    (Jlist BRSet (BEFlist fl'))->
    Judgement BRSet (BEConc (cons f fl'))
  jt_ih :
    {fl'|list|(Term|FS Formula)}
    (Jlist BRSet (BEFlist fl'))->
    Jlist BRSet (BEFlist (append fl fl'))
  fl' | list|(Term|FS Formula)
  Jl_fl' : Jlist BRSet (BEFlist fl')
  ?32 : Jlist BRSet (BEFlist (append (cons f fl) fl'))
```

We can see that the goal here is equivalent to

```
Jlist BRSet (BEFlist (cons f (append fl fl')));
```

so we can use our induction hypotheses. The goal can be shown by refining by the term

```
jcons BRSet (jh_ih (jt_ih Jl_fl')) (jnil BRSet);
```

completing the proof of the theorem.

This is a general theorem from which we can obtain a corollary equivalent to theorem 6.1 quoted at the beginning of this chapter. Such a theorem in our theory would look like

```
BranchElimTheorem :
     {FS|FSig}
     {RSet:RuleSet|FS}
     [BRSet = BERuleSet RSet]
     {f|Term|FS Formula}
     (Judgement RSet f)->
     (Judgement BRSet
                (BEConc (cons f (nil|(Term|FS Formula)))));
```

and be proven quite simply by using `BETheorem` above. If we refine by `BETheorem` in the proof of this corollary we still have a proof obligation reducible to

```
 ?2 : Jlist BRSet (cons (PROVE (FNIL|FS))
                        (nil|(Term|(BESig FS) Formula)))
```

a trivially derivable list of `Judgements` since we added a rule specifically for "`PROVE (FNIL|FS)`" in `BERuleSet`. This corollary is perhaps the usual theorem we would use in practice since it relates a formula in one semantics directly to its counterpart in a branch eliminated semantics.

Chapter 7

Conclusion and Further Work

Thus far we have concentrated on documenting the construction of a "general theory of operational semantics" in Lego, but have to a large extent refrained from assessing it as an effective tool for the purposes for which it was made. In this final chapter we bring all the individual issues raised in the preceding chapters together in our appraisal of the whole. We commence by analyzing the basic syntactic fabric and then continue with a discussion of our enhancements to the user interface. We then assess the formalization of our general theory in Lego in the light of the case studies in chapters 5 and 6. We point to the future directions of our work at each stage of its development.

7.1 Well Formed Terms

In chapter 2 we introduced the basic elements of our general theory — the terms that make up and include the atomic formulae in inference rules. In Hannan and Miller's work [HM91], the meta-logic comprised simply typed $\lambda-$terms over a given finite set of base types and constants *particular to each application.* For us the issue developed into the pursuit of a suitable formalism for our general theory in which the whole range of such base types and constants could be expressed.

We found first order term algebras were an adequate notation although in some instances simple expressions have large representations. This is evident in the use of Peano arithmetic in representing natural numbers. In this case however it can be argued that this is merely a consequence of modelling a particularly simple basis of a meta-logic. A certain amount of prolixity arises when such an algebra is used due to the lack of polymorphic sorts. For example, there can be no generic sort for all lists. They must be particularized.

First order term algebras comprise of sets of sorts, function names, signatures and a set of well forming rules for the terms in the algebra. The first three entities can be expressed in Lego as fairly simple inductive types. The well forming rules are a little trickier to implement. Lego *is* supplied with an expressive type system although one has to be mindful of matters concerning efficiency. The lack of partiality is an additional shortcoming.

These points are best illustrated in the two type choices for well formed terms in chapter 2. Elaborate $\Sigma-$types could be used to express the formation rules for well formed terms at the expense of clarity and practicability. All terms in such a scenario are appended with witnesses: well formedness proofs that are a weight to handle in the context of functions and theories built on top of these foundations. Since terms are the basic elements of operational semantics it is clear that a significant amount of space and run-time complexity must be sacrificed to accommodate witnesses. Construction of well formed terms themselves does not have to be as tedious as expected since functions can be defined to construct well formed proofs automatically. But because of the prevalence of terms in operational semantic theories, leaner representations would be a significant improvement.

Indeed the robustness of Lego's type system allows us a type that can essentially combine all aspects of well formed term construction into a small set of mutually inductive constructors utilizing simpler $\Pi-$types to express the constraints of well formedness.

However one limit of Lego's type theory is in its handling of partiality. Functions in Lego must be total[1]. This poses a technical anomaly in that the functions involved in the constructors for well formed terms are most naturally partial — a trait that must be mimicked by mapping all undefined objects to a global error (exception) value. As a result, inclusion in the type `Term` does not technically mean the term is well typed unless its sort is not the exception value for the set of sorts. Pragmatism suggests that these values should be avoided except where necessary. Subsequent parts of a theory ultimately must take account of them when dealing with functions where the error value may prove significant in a function or proof (as we found in chapter 6 in the proof of the important `BESig` theorems). Partiality could only be approximated for both type choices in chapter 2.

The final issue in chapter 2 was more practical. It concerned the clarity of the notation used to describe (well formed) terms in the general theory at that point in its development. Even with a type aiding the formation of compact terms, their form as presented in Lego is relatively incomprehensible. This is partly due to the generic nature of the theory being defined and partly a result of the proof

[1]Totality is essential for Lego functions since their results must be well-typed.

assistant's basic interface.

7.2 Object Language Support

Chapter 3 addresses this problem by at first exploring the extent to which Lego's in-built functions can be exploited to yield a notation for terms closer to that used in the literature. A feature like type synthesis helps rid Lego notation of extraneous syntax, terms that can be inferred from context, but is not useful for the notational clarity needed. We turned to using Lego definition as a device to provide macros in the Lego global context that abbreviate the concrete syntax to the notation of basic first order term algebra. Whilst this allowed the user a clearer and more practical way of writing terms, the limits of the basic term algebra, restricting terms to alphanumerics with all operators prefixing their arguments, results in confusion especially when the size of terms increases. This is exacerbated by printing routines that do not necessarily print terms in their macro form. If a term is passed to a function that outputs a term, then that output will typically consist of a marriage between macro and concrete syntax.

Our generalization of a HOL style quotation parser gives the user a much freer reign on the notation they desire. We elicited a correspondence between context free grammars and first order signatures. This allowed us to extend Lego's interface in a natural manner. Rather than defining a signature in concrete syntax, it can be declared in terms of a language grammar. Each signature entry for a function name's arity

$$f : (s_1, s_2, \ldots, s_n) \to s$$

can be written in terms of a production in a grammar by punctuating the sort references with lexical tokens as

$$\begin{array}{lll} s & \to & \cdots \quad | \\ & & f: \ \alpha_1 \, s_1 \, \alpha_2 \, s_2 \, \ldots \, s_n \quad | \\ & & \vdots \end{array}$$

where each α_i is a sequence of strings. Turning to our choice of parser, various alterations should include dispensing with the Earley algorithm in favour of an LR type parsing strategy. The accommodation of ambiguity can lead to parses not consistent with the user's expectations leading to errors in theory creation. If we choose an LR-type parser there would be no possibility of parsing ambiguous grammars but it can be argued that they do not on the whole exist in typical program language specifications. An important extension in this respect to an acceptable grammar and parser would be the inclusion of associativity and precedence information for infix operators, facilitating an implementation closer to Unix's YACC and other parsers in common use. Such a modification frees the user from the explicit parenthesizing currently obligatory as seen in chapter 5.

Several minor improvements can be made to the grammar syntax and parser including changing the way variables are referenced in quotations of the general theory. Currently a variable in a quotation of sort `expr` say is quoted as `"expr~`ν`"` where ν is a sequence of one or more numerical digits. This is clearly at odds with the usual notation often as simple as `"e"` which may be suffixed by numerics or any number of prime symbols. To take account of this, the grammar could include the users preference for the form of variables in each non-terminal's production as in

```
expr e = ... |
         suc : "suc" "expr" | ...
```

where the `e` is the expected token/prefix for variables. It would be left to the system to map each variable to the numerical index in the implementation.

On the pretty printing side, the in-built Lego utilities we use do not give the user an indication of where line breaks may take place. The user has no control over the way Lego presents terms but may well wish to do so as format could be crucial to their notation. For instance inference rules written as

```
     E |- n ==> n
 ---------------------- ()
  E |- suc n ==> suc n
```

may well be displayed as

```
 E |- n ==> n ---------------------- () E |- suc n ==> suc n
```

Since we added extra routines to the pretty printer to take account of the parser and as a result, consult language grammars, we could add even more complexity to the grammar by allowing the user to specify where breaks are necessary, forbidden or permissible. This information would be processed only by the pretty printing routines.

The concepts in chapter 3 and above are only the beginning of an adequate user interface. For instance, currently we have parsing facilities for only the syntactic aspects of the general theory. All reasoning within it is still expressed in terms of Lego notation. Extending Lego with a full quotation parsing facility is a massive task considering its behaviours and the plethora of applications. An intermediate improvement would be an extension of the class of Lego types the quotation parser can cope with. In section 3.6 we showed an implementation of parsing for a class of simple mutually inductive types. Extending the class means fitting in a facility for more complex types most importantly dependent types and type universes.

Type universes are relatively simple to handle. Any grammatical object `Prop` or `Type` is treated as a special non-terminal. Dependent types are trickier. Take the type for the function application constructor for `Term`s (ignoring the signature parameter for now):

```
 fa : {f:FIdent}(Tlist (IDSort1 f))->Term (IDSort2 f)
```

If we were to express this dependent type information in a grammar rule it follows that the dependencies must be accounted for, and so grammars must become more complex. One grammar representation for the `Term` type could be

```
Term = ... |
       fa : "FIdent" "(" "Tlist (IDSort1 FIdent#1)" ")"
               : Term (IDSort2 FIdent#1) |
              ...
```

where the `FIdent#1` is a yacc-style reference to the type variable of the dependent product. With this promulgation of the parser, we would be able to define all inductively defined constructs (including `Judgements`) in terms of quotations.

The next step is the introduction of a graphical user interface, one which may include several windows each for differing purposes such as entering commands, displaying the global context or a library of proven theorems, editing and showing the current proof state. In Nuprl [C$^+$86], four windows comprise the interface. A command/status window for overall control of the interaction with NuPrl, a Library or current global context window, a text editor for general input and a proof editor where proofs are formed. Each window can be moved and resized.

A more up to date version along the same lines is the CtCoq interface [BB96b], where a multipaned window is used. Each pane comes with a scroll bar which can be resized within the window. The window itself may also be moved and resized on the screen. The purpose of using panes is to avoid the overuse of many windows on a screen at the same time and yet to maintain some kind of proximity between each facet of the theorem proving environment. The top pane functions as the command window recording the script of commands sent to the proof engine. The pane in the middle displays the current state of the proof including the premisses and conclusion divided by a visible bar and the bottom pane shows the theorem prover's database of definitions and theorems. Coupled with this is a method called *proof by pointing* in which subgoals of the current goal can be selected with the aid of a mouse to bring the selected subgoal to the surface of the goal. Many bookkeeping steps in a proof are simplified with this utility and the proof becomes more mouse oriented.

This is particularly suitable style of graphical user interface for our general theory. With it we could for instance build top down derivations by stating a goal formula in a command/proof state pane running in Lego. We would then consult a global context pane containing the sets of operational semantics and general theory theorems and definitions in the global context. Using a proof by pointing strategy we could then select a rule to apply and let the system search the appropriate rule set to get the correct numerical index for `ruleAp`, use `unifSub` to elicit the right substitution and apply `ruleAp` with this information to the current goal formula. Using the same idea of proof by pointing we could help prove other theorems of a semantics by for instance selecting the induction on the depth of inference principle when it is applicable to the current goal.

7.3 Types and Semantics for Inductive Definitions

Chapter 4 concluded the syntactic description of operational semantics — which we chose to represent as inductive definitions — by providing the form of premisses, side conditions, rules and rule sets in terms of Lego types. It could have been possible to define these as special kinds of the `Term` type, making the whole syntactic structure of inductive definitions accessible to the parser. Grammars could then include productions for `Terms` of sort `Prems`, `SideConds`, `Rule` and so on. However this increases the number of reserved sorts (such as `botSort` and `Formula`) already in the global context, leads to a necessity to account for more and more special cases in subsequent functions and theorems[2]. It is also difficult to process and provide inferences upon rules and their components directly if those components are nested within a `Term`-like structure. In our encoding, rules are simple to reason about. This is at the expense of including ad-hoc extensions to the parser to simplify their format to the user. In the light of chapters 5 and 6 this is not such a bad decision since it makes sense to present such entities whose

[2]cf. the `BESig` theorems in chapter 6

form is a global constant, in a standard and readily accessible form as we do.

As far as semantics were concerned, we concentrate on providing the inductive aspects of inductive definitions at the expense of certain logical concerns. A type for the set of inductive sets and an induction principle thereon provides the mathematical basis for inductive definitions in our general theory. While this gives us an immediate and satisfactory way of dealing with their inductive nature, some of the more logic related aspects must be approached indirectly.

Inductive definitions can be directly encoded in Lego as an inductive type, albeit in Lego a mutual one, in which the secondary type `Jlist`, the type for lists of `Judgement`s tends to become obstructive in functions and theorems since they must take account of `Jlists`. These factors add to the technical complexity of an implementation rendering a level of imposed sophistication to the whole that is both distracting and bureaucratic.

An obvious omission in our treatment of inductive definitions is the connection between the rules and Horn clauses, their logical denotation. At the syntactic level, they are merely unquantified triples with no logical connectives involved. Their only presence being implicit in the inductive sub-theory. Variables in rules are left unbounded in our general theory both in syntax and semantics, so no direct link between rules and horn clauses is made. As a result, it is difficult to emulate intrinsically logic oriented reasoning in the general theory as we have organized it. Case analysis inferences (see section 5.5) where we have to use a certain amount of indirect and complex inferences to obtain what is essentially a propositional conclusion, exemplify this problem. The proof steps are made harder mainly by the absence of quantified rules (and hence binding) in the logic.

The type of `Judgement`s themselves had a fairly natural denotation as a Lego inductive type (following the design of the `Term` type). The "leastness" of inductive definitions was the leastness in the inductive type. The closure principle encoded in the constructor type. One point of note was the employment of mutual induction. This made it awkward to reason about `Judgement`s as one would also have to take account of the `Jlist` (lists of `Judgements`) type every time a Judgement function was written, a refinement by `ruleAp` took place or when one applied induction on the depth of inference.

The next step of our evaluation is an assessment of the functionality of the general theory for reasoning with inductive definitions with regards to similar features in HOL's inductive definition package [CM92]. The general theory facilitated both top-down and bottom-up styles of derivation construction as well as the principle of induction on the depth of inference but in a comparatively indigestible manner. Both refinement terms and Lego's outputs verge on the incomprehensible. Recall that applying a rule in a derivation means supplying substitutions and numerical references to rules. The latter is something we could conceal in the user interface whereas the former is an inextricable part of the platform we intend to model. Furthermore, during derivation-time construction, the system does not help the user by returning goal/derivative values that do not correspond to our perceptions of the inferences taking place but as unreadable expressions. This is obvious in the examples of derivations in chapter 5 where the consequence of applying the rule for `let` expressions to the goal formula in section 5.3.1 did not yield the two new subgoals we would normally expect but an expression

```
Jlist ExpRules (TlistSubFn sub1
                           (Prems (RuleNum ExpRules (suc ten))))
```

which with a little extra inference gave the subgoals

```
  ?6  : Judgement ExpRules
                (TSubFn sub1
                      (Exp!Formula "Env~1 |= decl~1 =:=> Env~2"))
  ?10 : Judgement ExpRules
                (TSubFn sub1
                      (Exp!Formula "Env~2 |= expr~1 ==> expr~2"))
```

admittedly closer in some respects to the expected subgoals at that point in the derivation but still obscured by an unapplied substitution. In comparison, substitution in the HOL package is almost completely transparent. A similar problem arose in uses of our version of induction on the depth of inference where the object terms we applied hardly resemble the usual inference rules we associate with such depth of inference inductions. Again, the values Lego returns as a result of the proof steps applied tend to obscure the nature of the inference taking place.

These observations can partly be explained by the generic nature of our platform. As we mentioned in the introductory chapter, we must expect a certain degree of complexity with any "deep embedding" style activity that intends to model a set of objects in both syntactic and semantic domains. At the same time however, two points are prevalent. The first is the obtrusive nature of substitution and the second is the difficulty Lego faces in reducing terms to the canonical forms we expect. The first issue accounts for much of the illegibility in applications of the general theory for inferences upon specific inductive definitions as compared to HOL's package. It becomes obstructive in top down derivations and even if we use unifSub to impel the system to unify a goal formula with the conclusion of a specified rule, we cannot expect to omit some form of substitution information. In bottom up proofs, substitution terms are similarly unreduced contributing to the confusion in building derivations as the user cannot see immediately the direction in which they have taken a derivation. It also a hindrance in proofs of theorems of semantics such as monogenicity in section 5.4 and the case analysis proofs in section 5.5. Continuing further, inferring properties of substitution constitutes the trickiest part of transformation correctness proofs as demonstrated in chapter 6 where a great deal of the work involved proving the transformation applied to a substitution expression could be wholly moved within the body of that expression.

By comparison, substitution in HOL's inductive definition package is kept implicit by the natural representation of rules as universally quantified formulae. Clearly this was not an option for us as we showed in chapter 1. In our general theory, substitution must be accounted for explicitly since rules at the representational level need to be understood as objects of syntax. Nevertheless we should hope that this, like the numerical referencing of rules, can be hidden by a layer of Lego between the user and the system that combines reduction strategies to simplify terms involving substitution wherever possible — leading us to our second observation.

Whenever we apply ruleAp in a derivation the substitution is not automatically applied to the current formula(e) by the system and so it returns an unreduced expression. It is up to the user to use the Equiv command to reduce the results to the intuitive result they expect as demonstrated in section 5.3.1. Lego makes it extremely difficult to automate this process. It provides normalizing and expanding routines but the nature of the calculation makes it difficult to reach the ideal result somewhere in between an unevaluated expression and a canonical form (where everything is normalized and expanded as far as possible). If this were in fact possible to do simply then we could expect to use its reduction facilities to apply substitutions and the like wherever they can be used to simplify terms. Indeed, this would not only make derivations significantly more comprehensible, it would also contribute to a better run-time performance. In chapter 4 we found that the due to the lazy attitude of the Lego evaluator coupled with the nature of derivations being built successively upon one another, each time a proof was advanced one step when a substitution had not been applied as a result, the system took longer to compute the next proof step than if we used Equiv to manually reduce the goal formula.

The final point worth mentioning in this section is the speed of operations in general. With Lego in its current form, reasoning with inductive definitions can become a relatively compute intensive task particularly when building derivations where the base language and the set of rules is large. Couple this with shortcut routines such as named rules and automated substitutions, the system becomes increasingly slower especially if we further compound this with the run-time observation above. This is a typical trait of Lego's reduction and normalization module. As the complexity of tasks increase, it takes significantly longer to run. In section 4.4 we experienced the brunt of Lego's intractabilities when relatively simple derivations took extraordinary times to evaluate. Work has taken place to improve some aspects of its run time performance by improving its reduction heuristics. However, this does not give a guarantee that computations will be faster in all cases.

With these criticisms in mind, we can see that future developments of this portion of our theory would include

- exploring means of making applications such as case analysis and depth of inference induction easier for the user to use and understand. This may include adding strategic tacticals to automate more and more of the book-keeping tasks involved. For instance, it may be advisable to combine the two main theorems used in depth of inference induction: `J_Induction` and `RuleRec` in a more comprehensible manner,

- eliciting ways of partially evaluating terms involving substitution in an attempt to conceal its explicit nature as much as possible from the user. This would require an in depth analysis of the possibilities of Lego's reduction features

- and a general consideration of the ways in which the general theory's run-time performance can be improved — a wide ranging problem that touches on many of the aspects of the general theory as a whole as well as the internal features of Lego itself.

7.4 Transformation Correctness Proofs

In chapter 6 we saw how to apply our general theory to implement a simple Hannan and Miller transformation [HM91] and correctness proof. The transformation itself was relatively easy to render into the system. This set down a clear structured means of defining transformations. Stating this structure in order, we need to define the effect of the transformation on an inductive specification's signature, proofs that the transformation preserves well formed terms, the extra features needed for the parsing facilities (new terminals and productions), the term transforming functions, rule transformers and finally the whole combined transformation of the rule set and signature.

The Branch Elimination transformation described in the previous chapter is a very simple example of a Hannan and Miller transformation however, and we must expect to cater for more complex examples that may impose conditions on the structural aspects of operational semantics or may add and delete a varying number of sorts and/or function names. Branch elimination imposed no conditions on its input. Other transformations require certain syntactic features of a semantics before a transformation is applicable. Such conditions would not be difficult to implement but they are likely to be verbose and run-time expensive. In addition, recall that Lego allows only the definition of total functions. The consequence being that transformations involving partiality must be lifted to total functions. We can imagine the pseudo code description of such transformations as

$$\textbf{if } b \textbf{ then } Trans(Sem) \textbf{ else } \textbf{Id}(Sem)$$

where **Id** is the identity function. As mentioned above, Branch Elimination adds only a constant number of new sorts and function names no matter which semantics it is given. Typically this is not the case. A transformation may add and/or delete any number of such entities depending on the form of the rules it is passed. (They may also replace rules with two or more rules — in the transformation in chapter 6, each rule was mapped to one new rule.) This means using more complex mapping functions for the indices of sorts and function names to avoid any inconsistencies and to preserve the well formed nature of terms passed into a transformation suite.

In chapter 6 we laid out guidelines for adding new sort and function names to signatures if well formedness is to be maintained. The indices of all additions must be added to the beginning of the respective index list and all pre-existing indices shifted up the total number of additions. If we now consider deleting sorts or function names, we must simply ensure that the new signature does not contain a sort/function name whose indexing number is equal to the number of the deleted item had it been shifted the appropriate number of places. An example of the idea can be seen in figure 7.1. The function name whose index is four in the as yet untransformed signature is to be deleted from the new signature. This is achieved by making sure that no function name in the latter has the corresponding index

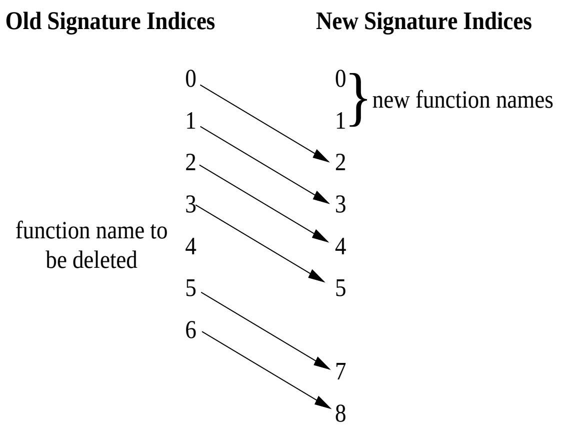

Figure 7.1: Mapping Indices for Function Names.

(six in this case). Not knowing how many additions or deletions to a signature are necessary beforehand is not a problem if a transformation is effected in two passes through a rule set — one to determine the number new sort/function names and the second to perform the transformation. A similar solution can be applied to the problem of adding variable numbers of new rules. Again this means transformation implementations become invariably complex.

Similar to the structural form of the transformation, the correctness proof in chapter 6 also provided the reader with a template of a methodological means of proving such theorems. After any necessary extra definitions (native to the transformation), we must prove a succession of theorems that state that the transformation applied to a substitution can be wholly taken within the body of the substitution term. These are exemplified by the proofs of the "commutativity" theorems (`BEConvSubstLemma` and the like) in section 6.3. The proof of transformation correctness itself proceeds by induction using the elimination operator for `Judgement`s. At this point the commutativity theorems are applied. It then remains for the user to finish off the proof for `Jlist`s, requiring either a set of extra lemmas proven beforehand or the induction hypotheses.

A couple of points are worth mentioning. As before, inferring properties about substitution comprises the major portion of the proof process. This vindicates us in our choice of implementation of substitution as parameterized lists of variable-value pairs, which affords easy reasoning. Nevertheless, these theorems are formidable to prove, even in simple circumstances. The second point is the consequence of using a mutually inductive type for our main inference mechanism. All inferences with `Judgement`s must invariably take account of the effect of inferences with `Jlist`s, lists of judgements, which can mean concerting a great deal of energy into defining functions and proving theorems for judgement lists.

In chapter 6 this was not as trivial as we may have expected. More intricate transformations may pose significantly greater problems.

As far as conditional transformations are concerned, any attached conditions will be manifested in extra hypotheses in the proof. Ones we should expect to use. It is up to the user to make their presence as accessible to reasoning as possible. The fact that we must also now deal with sorts, function names and rules being added and deleted arbitrarily means the proofs of well formedness and transformation correctness must employ complex mapping functions such as shown in figure 7.1 to actuate their results and as such become more complicated themselves.

Further developments in this area would include defining more of Hannan and Miller's transformations in an attempt to explore the issues outlined above in greater depth. In this way we shall be able to fully assess the merits and faults of a general theory of operational semantics designed with the aim of providing an efficacious platform supporting transformation correctness proofs.

7.5 Future Directions – Theorem Prover Support

In the light of evidence from previous sections we can foresee a need for supplementary theorem prover support for our general theory of operational semantics. The arguments put forward suggest that as the theory currently stands, reasoning about individual semantics and their class requires developed understanding of Lego and the implementation of the theory.

To begin with we can bring tacticals and functions into the theory that handle some of the more mundane aspects of proof checking. We already mentioned the possibility of combining `J_Induction` and `RuleRec` to provide a more cohesive version of induction on the depth of inference. Other feasible routines include bringing the `JlistSum` and `case` functions together into a tactic that, when confronted with a case analysis on a rule set, the user can call upon it to immediately list all the necessary proof obligations.

More advanced additions may be possible or even feasible. Just as we described extra features to enhance the interface of our theory, so we can also perceive of supplements to the basic reasoning mechanism to hide the low level aspects within it.

In case analysis proofs for example, one has to delve into the implementation of the `Term` type to elicit contradictions. It is possible that inferences such as these can be automated. For instance, a function that takes two terms and an obviously contradictory equality between them would recurse through them to root it out. Other examples of our theory that may afford some form of mechanization include transformation definitions. A library of functions and theorems that help add or delete sorts and function names in a consistent manner. In this way, signatures could be safely transformed so that no inconsistencies arise when `Term`s are formed thereon.

We have seen examples of such automation previously in the functions `ruleAp'` and `unifSub` in chapters 4 and 5. We can draw the conclusion from their implementation that evaluation times increase significantly. Particularly when such functions are combined. The challenge then becomes acquiring a balance between the need to hide low level elements of the theory and maintaining an efficient system. Unfortunately, the primitive evaluation mechanisms in Lego make this extremely difficult.

Furthermore, it seems that certain aspects of Lego and our theory must be kept largely explicit. Term rewrites are a case in point. Where normally one would expect to be able to use the command `Qrepl` to rewrite terms in a goal, sometimes assumptions must be taken to be rewritten laboriously using the `Eq_subst` function. Other situations arise where quite legitimate terms have to be type casted to avoid type check errors. Lego does allow such shortcuts as applying theorems with parameters uninstantiated although in some of our proofs we found that even this was not possible.

Generally, although Lego has a versatile type system for representations, its inference engine can be frustratingly inflexible. At some points, Lego will only allow theorems to be proven in one very specific way although the same theorem could be proved in many ways mathematically.

In our general theory of operational semantics, accommodating substitution presents the greatest challenge. It is pervasive in all of the main proofs in the previous chapters. It is due to the inherent nature of the theory we model that this is so. Nevertheless we would hope to conceal its existence in some instances such as proof tree construction. In section 5.3.1 we saw how the system leaves substitutions unapplied. The value of terms was thus not immediately obvious. Sadly not much can be done about this at the theorem prover level as the user can only control the evaluation of terms very bluntly. Some form of complex ad-hoc extension to Lego's reduction routines would be necessary — and this is likely to be difficult.

On the whole however, substitution is an integral part of the theory that one cannot avoid in proof checking. One would have to be conversant in its implementation if attempting to prove theorems with the theory. Its generic aspects restrict the freedom with which to afford optimization in all areas.

7.6 Conclusion

This thesis has studied the impact of the generalization of a theory on Lego as a model of a proof assistant. Explicitly setting up an algebra and reasoning mechanisms for this theory leads to an implementation much more visible at the theorem prover level. This is manifested in a syntax, semantics and modus operandi that can be both diffuse and awkward to handle.

There are various ways and levels in which these can problems can be alleviated. From the introduction of parsing facilities to enhance the proof assistant's interface to functions and theorems at the theorem prover level to hide implementation specific details of the theory. It must borne in mind that such enhancements are limited in range and may adversely slow down a proof assistant's processes.

It could be argued that at least some of the problems encountered in this thesis are due to Lego. Perhaps it is too basic a tool in its foundation upon a few elementary notions such as types, type universes, proof by refinement, the Curry-Howard isomorphism and induction. Everything is built up successively from first principles, which in the course of a session, often have to be returned to. When a generalization of a theory takes place we can expect that an implementation built in this environment will involve overtly complex forms. This was mirrored in both the incomprehensible concrete syntactic encoding of operational semantic terms and the awkward nature of reasoning about them in our theory. Of these concerns the former can be allayed albeit with additions to the implementation of the proof assistant. The latter poses a harder problem for us. If we try to hide the low level elements of our general theory we must be mindful of the implications of automating implementation-specific areas on the run-time performance of the system as a whole. It has been acknowledged that Lego's evaluation routines can be improved in this respect.

Our study then suggests that a successful realization of a general theory such as ours imposes criteria on the features of proof assistants. Firstly, the notion of expressiveness should be coupled with the ideas of elegance and notational freedom. Such ideas should be integrated more comprehensively within the fabric of the theorem prover so that the abstract syntax of user created terms can be used seamlessly throughout it. Elegant representations of the syntax and semantics of general theories should give the elements of a theory a more natural look. Simpler to read and cleaner to use. Secondly, the whole evaluation processes of a proof assistant must afford degrees of flexibility, efficiency and if possible, user interaction. In this way, functions in a general theory running at optimal speeds can be written. More of a general theory can be automated without detriment to the performance of the system. Such mechanizations are desirable since they free the user from many of the inevitable bureaucracies of a proof checker. Finally, if a proof system allows more control over evaluation then more of the prominent features of a general theory may be hidden via notions like partial evaluation. If proof assistants combine the features suggested above we can expect the successful realizations of the general theories we desire.

Bibliography

[Acz77] Peter Aczel. An introduction to inductive definitions. In K. J. Barwise, editor, *The Handbook of Mathematical Logic*, Studies in Logic and Foundations of Mathematics. North Holland, 1977.

[BB$^+$96a] B. Barras, S. Boutin, et al. *The Coq Proof Assistant Reference Manual.* INRIA, Rocquencourt, November 1996.

[BB96b] J. Bertot and Y. Bertot. The CtCoq experience. In N. Merriam, editor, *The Second International Workshop on User Interfaces for Theorem Provers*. Dept. of Computer Science, University of York, July 1996.

[BCG91] G. Birtwistle, S.K. Chin, and B. Graham. *An Introduction to Hardware Verification in Higher Order Logic.* University of Cambridge, 1991.

[BG$^+$92] Richard Boulton, Andrew Gordon, et al. Experience with embedding hardware description languages in HOL. In T.F. Melham V. Stavridon and R.T. Bourke, editors, *Theorem Provers in Circuit Design: Theory, Practice and Experience*, pages 129–156. North-Holland, June 1992.

[BM91] R. Burstall and J. McKinna. Deliverables: an approach to program development in the Calculus of Constructions. Technical Report ECS-LFCS-91-133, LFCS, Dept. of Computer Science, University of Edinburgh, 1991.

[C$^+$86] R.L. Constable et al. *Implementing Mathematics with the Nuprl Proof Development System.* Prentice Hall, 1986.

[Car84] L. Cardelli. Compiling a functional language. In *1984 ACM Symposium on LISP and Functional Programming*, pages 208–217. ACM, August 1984.

[CC92] P. Cousot and R. Cousot. Inductive definitions, semantics and abstract interpretation. In *The Ninteenth Annual SIGPLAN-SIGACT*

Symposium on Principles of Programming Languages, volume 247. Springer-Verlag, January 1992. Albuquerque, New Mexico, USA.

[CH88] Th. Coquand and G. Huet. The Calculus of Constructions. *Information and Computation*, 76:96–120, 1988.

[CM92] J. Camilleri and T.F. Melham. Reasoning with inductively defined relations in the HOL theorem prover. Technical Report 265, University of Cambridge Computing Laboratory, 1992.

[Des88] T. Despeyroux. TYPOL: A formalism to implement natural semantics. Technical Report RT-0094, INRIA, Sophia-Antipolis, March 1988.

[GM93] M. Gordon and T. F. Melham. *Introduction to HOL.* Cambridge University Press, 1993.

[HHP87] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. In *Symposium on Logic in Computer Science, Ithaca, NY*, pages 194–204. IEEE, June 1987.

[HM90] J. Hannan and D. Miller. From operational semantics to abstract machines preliminary results. In *ACM Conference on LISP and Functional Programming*, pages 323–332. acm Press, June 1990.

[HM91] J. Hannan and D. Miller. From operational semantics to abstract machines. Technical report, Department of Computer and Information Science, University of Pennsylvania, 1991.

[HMM86] R. Harper, D. MacQueen, and R. Milner. Standard ML. Technical Report ECS-LFCS-86-2, Department of Computer Science, University of Edinburgh, 1986.

[Hof91] M. Hofmann. Verifikation von ML-programmen mit dem beweisprüfer Lego. Master's thesis, Diplomarbeit an der Universität Erlangen Nürnberg, 1991.

[How80] W. A. Howard. The Formulae-as-Types notion of construction. In J. Hindley and J. Seldin, editors, *To H. B. Curry: Essays on Combinatory Logic.* Academic Press, 1980.

[Jon80] N. Jones, editor. *Semantics Directed Compiler Generation*, volume 94 of *Lecture Notes in Computer Science.* Springer-Verlag, 1980.

[Kah87] G. Kahn. Natural Semantics. In G. Goos and J. Hartmanis, editors, *The Fourth Annual Symposium on Theoretical Aspects of Computer Science*, volume 247, pages 22–39. Springer-Verlag LNCS, February 1987. Passau, Germany.

[Kah88] G. Kahn. Natural Semantics. In K. Fuchi and M. Nivat, editors, *Programming of Future Generation Computers*, pages 237–258. North-Holland Publishing Company, 1988.

[Lan64] P. J. Landin. The mechanical evaluation of expressions. *The Computer Journal*, 6(4):308–320, January 1964.

[LP92] Z. Luo and R. Pollack. LEGO proof development system: A user's manual. Technical Report ECS-LFCS-92-211, Laboratory for Foundations of Computer Science, University of Edinburgh, 1992.

[LPT89] Z. Luo, R. Pollack, and P. Taylor. How to use LEGO: a preliminary users manual. Technical Report LFCS-TN-27, LFCS, Dept. of Computer Science, University of Edinburgh, 1989.

[Luo90] Z. Luo. *PhD. Thesis, An Extended Calculus of Constructions.* PhD thesis, LFCS, Department of Computer Science, University of Edinburgh, 1990. Also as technical report CST-65-90/ECS-LFCS-90-118, Dept. of Computer Science, University of Edinburgh.

[Luo91a] Z. Luo. A higher-order calculus and theory abstraction. *Information and Computation*, 90(1):107–137, January 1991.

[Luo91b] Z. Luo. Program specification and data refinement in type theory. In *Proceedings of the International Joint Conference on the Theory and Practice of Software Development*, Brighton, 1991. Also LFCS report ECS-LFCS-91-131, Dept. of Computer Science, University of Edinburgh.

[Luo91c] Z. Luo. Type theory, logic and computer science. Lecture Notes for Post-Graduate Theory course, LFCS Edinburgh University, January 1991.

[Luo92] Z. Luo. A unifying theory of dependent types: The schematic approach. In A. Nerode and M. Taitslin, editors, *The Second International Symposium on Logical Foundations of Computer Science*, pages 98–145, July 1992. Tver, Russia.

[Mah91] S. Maharaj. Implementing Z in Lego. Master's thesis, Dept. of Computer Science, University of Edinburgh, 1991.

[Mah96] S. Maharaj. *PhD. Thesis, A Type-theoretic Analysis of Modular Specifications.* PhD thesis, LFCS, The University of Edinburgh, 1996. Report No: ECS-LFCS-97-354.

[McK92] J. McKinna. *Deliverables: a categorical approach to program development in type theory.* PhD thesis, Dept. of Computer Science, University of Edinburgh, 1992.

[Mel88] T.F. Melham. Using recursive types to reason about hardware in higher order logic. Technical Report 135, University of Cambridge Computing Laboratory, 1988.

[Mel92] Thomas F. Melham. A package for inductive relation definitions in HOL. In M. Archer, J. J. Joyce, K. N. Levitt and P. J. Windley, editors, *Proceedings of the 1991 International Workshop on the HOL Theorem Proving System and its Applications, Davis, August 1991*, pages 350–357. IEEE Computer Society Press, 1992.

[MH91] E. Maygar and R.L. Harris. *User Guide for the LAMBDA System.* Abstract HardWare Ltd, 1991.

[Mil76] R. Milner. Program semantics and mechanized proof. *Foundations of Computer Science II*, pages 3–44, 1976. Math. Centre Amsterdam Tracts 82.

[ML84] P. Martin-Löf. *Intuitionistic Type Theory. Studies in Proof Theory Lecture Notes.* BIBLIOPOLIS, Napoli, 1984.

[MTH90] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML.* MIT Press, 1990.

[Pau93] L.C. Paulson. Introduction to Isabelle. Technical Report 280, University of Cambridge, Computer Laboratory, 1993.

[Pau94] L.C. Paulson. *Isabelle: A Generic Theorem Prover.* Springer-Verlag LNCS 828, 1994.

[Plo81] G. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Aarhus University, Aarhus, Denmark, September 1981.

[Pol95] R. Pollack. *The Theory of Lego: a Proof Checker for the Extended Calculus of Constructions*. PhD thesis, Dept. of Computer Science, University of Edinburgh, 1995.

[Pra65] D. Prawitz. *Natural Deduction*. Almqvist and Wiksell, 1965.

[Sch97] T. Schreiber. Auxiliary variables and recursive procedures. In *TAPSOFT '97: The Theory and Practice of Software Development*, volume 1214, pages 697–711. Springer-Verlag, April 1997. Lille, France.

[ST87] D. Sannella and A. Tarlecki. *Algebraic Specifications in Theory and Practice*. LFCS, Department of Computer Science, University of Edinburgh, 1987.

[Sza69] M. E. Szabo, editor. *The collected papers of Gerhard Gentzen*. North-Holland, Amsterdam, 1969.

Appendix A

A Type Theory Primer

In this primer we shall introduce some of the type-theoretic notations used in this thesis. Consider for now a single universal type of types called *Type*. The operator ":" denotes type inhabitance. Throughout this discourse read "$x : T$" as "x is of type T" and where T is a type universe, as "x is a type". Let $A, B : \mathit{Type}$ and let Γ represent a type context. Let us now consider the simply typed lambda calculus. This can be described by the formation, introduction and elimination rules:

$$\frac{\Gamma \vdash A : \mathit{Type} \quad \Gamma \vdash B : \mathit{Type}}{\Gamma \vdash A \to B \; : \; \mathit{Type}}$$

$$\frac{}{\Gamma, x : A, \Gamma' \vdash x : A} \qquad \frac{\Gamma, x : A \vdash b : B}{\Gamma \vdash \lambda x{:}A.b \; : \; A \to B}$$

$$\frac{\Gamma \vdash f : A \to B \quad \Gamma \vdash a : A}{\Gamma \vdash f\,a \; : \; B}$$

We can describe base types by introducing new constants. For example the rules

$$\frac{}{\Gamma \vdash \mathcal{N} : \mathit{Type}} \qquad \frac{}{\Gamma \vdash \mathbf{0} : \mathcal{N}} \qquad \frac{\Gamma \vdash n : \mathcal{N}}{\Gamma \vdash \mathbf{suc}(n) \; : \; \mathcal{N}}$$

describe the type formation and introduction rules for natural numbers. Combinatory types such as disjoint sum types can be described by the rules

$$\frac{\Gamma \vdash A : \mathit{Type} \quad \Gamma \vdash B : \mathit{Type}}{\Gamma \vdash A + B \; : \; \mathit{Type}}$$

$$\frac{\Gamma \vdash a : A}{\Gamma \vdash \mathbf{i}(a) : A + B} \qquad \frac{\Gamma \vdash b : B}{\Gamma \vdash \mathbf{j}(b) : A + B}$$

and the product type with the rules

$$\frac{\Gamma \vdash A : \mathit{Type} \quad \Gamma \vdash B : \mathit{Type}}{\Gamma \vdash A \times B \; : \; \mathit{Type}} \qquad \frac{\Gamma \vdash a : A \quad \Gamma \vdash b : B}{\Gamma \vdash (a, b) : A \times B}$$

$$\frac{\Gamma \vdash c : A \times B}{\Gamma \vdash \pi_1(c) : A} \qquad \frac{\Gamma \vdash c : A \times B}{\Gamma \vdash \pi_2(c) : B}$$

where $\pi_1(a,b) = a$ and $\pi_2(a,b) = b$ for $(a,b) : A \times B$. We can generalize the notion of types to include type variables and dependent product types and dependent sum types. The formation rule for dependent product types (or Π-types) is

$$\frac{\Gamma \vdash A : \mathit{Type} \quad \Gamma, x : A \vdash B : \mathit{Type}}{\Gamma \vdash \Pi x\!:\!A.B \ : \ \mathit{Type}}$$

We may write $A \to B$ for $\Pi x : A.B$ when x is not free in B. An object of a Π-type is a functional construction which can be applied to an object of its domain type to form an object in its range type:

$$\frac{\Gamma, x : A \vdash b : B}{\Gamma \vdash \Lambda x\!:\!A.b \ : \ \Pi x\!:\!A.B} \qquad \frac{\Gamma \vdash f : \Pi x\!:\!A.B \quad \Gamma \vdash a : A}{\Gamma \vdash f\ a \ : \ [a/x]B}$$

where the term $[a/x]B$ represents the term B with all free occurrences of a replaced by x. Dependent sum types (Σ-types, strong sum types) have the formation rule:

$$\frac{\Gamma \vdash A : \mathit{Type} \quad \Gamma, x\!:\!A \vdash B : \mathit{Type}}{\Gamma \vdash \Sigma x\!:\!A.B : \mathit{Type}}$$

where we may write $A \times B$ for $\Sigma x : A.B$ when x does not occur free in B. A Σ-type object is a pair whose components can be extracted by using projections:

$$\frac{\Gamma \vdash a : A \quad \Gamma \vdash b : [a/x]B}{\Gamma \vdash (a,b) \ : \ \Sigma x\!:\!A.B} \qquad \frac{\Gamma \vdash c : \Sigma x\!:\!A.B}{\Gamma \vdash \pi_1(c) : A} \qquad \frac{\Gamma \vdash c : \Sigma x\!:\!A.B}{\Gamma \vdash \pi_2(c) \ : \ [\pi_1(c)/x]B}$$

where $\pi_1(a,b) = a$ and $\pi_2(a,b) = b$ for $(a,b) : \Sigma x\!:\!A.B$.

The use of a universal type of types as we did above leads to paradoxes within the type theory. The work in this theory is based in an impredicative type system: The Extended Calculus of Constructions (ECC). In such a system, paradoxes are avoided by setting up a hierarchy of type universes:

$$\mathit{Prop} : \mathit{Type(0)} : \mathit{Type(1)} : \cdots$$

where *Prop* is the type of propositions. In this context, the rules above elicit a correspondence between formulae and types in an intuitionistic logic. Summarily, to show a type is inhabited is to prove a formula holds in the underlying intuitionistic logic. Furthermore, the types above correspond to logical operators. The function type corresponds to implication as is borne out by the rules above. Disjoint sums correspond to disjunction and product types to conjunction. The Π-type $\Pi x\!:\!A.B$ corresponds to the formula $\forall x\!:\!A.B$. Similarly the Σ-type $\Sigma x : A.B$ corresponds to the existential formula $\exists x : A.B$ whose proof is a pair (a,b) of a witness a and a proof of $B(a)$. This correspondence is known as the Curry-Howard isomorphism.

Appendix B

Lego Syntax and Commands

The following is taken from the Lego reference card available on-line available in Lego in its two states. The modules and tactical facilities are also shown. Notice that all commands need to be terminated by a semicolon. `Drop` will enable you to exit LEGO and return to UNIX.

The Syntax of Terms

`Prop`, `Type`	kinds
{id:term }term, **{ id|term } term**, **term->term**	Π abstraction
[id:term]term, **[id|term]term**	λ abstraction
term term	application
term symb term	infix application
term.term	postfix application
<id:term>term, **<id|term>term**, **term#term**	Σ type
(term,term,...)	tuple
(term,term,...:term)	annotated tuple
term.1, **term**.2	projections
(term:term)	type cast
[id=term]term	local definition

LEGO State and Proof State

After loading LEGO, the system enters a state called LEGO state, in which you may extend the context. Using the command `Goal` you enter the proof state. The system returns to LEGO state, if a proof is finished. The following commands apply to both states

`[`**id=term**`]`	persistent definition
`$[`**id=term**`]`	non-persistent definition
`Cd "`**directory**`"`	change directory
`Ctxt`, `Ctxt` **n**, `Ctxt` **id**	display the current context
`Decls`, `Decls` **n**, `Decls` **id**	display the declarations in the current context
`(Dnf` **term**`)`	compute display-normal form of **term**
`Dnf TReg`	compute display-normal form of the current type
`Dnf VReg`	compute display-normal form of the current term
`echo "`**comment**`"`	print **comment**
`Equiv TReg` **term**	replace current type with equivalent type **term**
`Equiv VReg` **term**	replace current term with equivalent term **term**
`ExpAll TReg` **n**	expand all definitions in the current type to a depth of **n**
`ExpAll VReg` **n**	expand all definitions in the current term to a depth of **n**
`Expand TReg` $\mathbf{id}_1$ $\mathbf{id}_2$ ...	use definitions $\mathbf{id}_1$ $\mathbf{id}_2$ ... to expand the current type
`Expand VReg` $\mathbf{id}_1$ $\mathbf{id}_2$ ...	use definitions $\mathbf{id}_1$ $\mathbf{id}_2$ ... to expand the current term
`ExportState "`**filename**`"`	save the current LEGO process in **filename**
`Help`	print help message
`(Hnf` **term**`)`	compute (weak) head normal form of **term**
`(Hnf` **n term**`)`	compute (weak) head normal form of **term** for **n** toplevels
`Hnf TReg`	compute (weak) head normal form of the current type
`Hnf` **n** `TReg`	compute (weak) head normal form of the current type for **n** toplevels
`Hnf VReg`	compute (weak) head normal form of the current term
`Hnf` **n** `VReg`	compute (weak) head normal form of the current term for **n** toplevels
`Init` **theory**,	initialise the context where **theory** is one of `LF`, `PCC`, `XCC`, `XCC'`, `XCC_s` or `XCC'_s`
`line`	insert a blank line
`Normal TReg`	normalise the current type
`Normal VReg`	normalise the current term
`(Normal` **term**`)`	Normalise **term**
`Pwd`	print the current working directory

LEGO State

`[`**id**`:`**term**`]`	non-persistent declaration
`$[`**id**`:`**term**`]`	persistent declaration
`Discharge` **id**	Abstract and remove all non-persistent entries in the context back to and including **id**
`DischargeKeep` **id**	Abstract all non-persistent entries in the context back to and including **id** and append them to the context
`Forget` **id**	Remove all entries in the context back to and including **id**
`Goal` **term**, `Goal` **id**`:`**term**	start refinement proof with goal **term**
`$Goal` **term**, `$Goal` **id**`:`**term**	start refinement proof with non-persistent goal **term**

Syntax for Inductive Types

`Inductive`	$[\mathbf{T_1}:\mathbf{M_1}]$ ... $[\mathbf{T_m}:\mathbf{M_m}]$	
`Double Inversion NoReductions Relation Theorems`		*(optional)*
`ElimOver` **Universe**		*(optional)*
`Parameters`	$[\mathbf{a_1}:\mathbf{N_1}]$ ... $[\mathbf{a_n}:\mathbf{N_n}]$	*(optional)*
`Constructors`	$[\mathbf{cons_1}:\mathbf{L_1}]$ ... $[\mathbf{cons_o}:\mathbf{L_o}]$	
`Record` [**id**:**kind**]		
`Fields` $[\mathbf{cons_1}:\mathbf{L_1}]$ ... $[\mathbf{cons_o}:\mathbf{L_o}]$;		

Proof State

Certain commands in proof state take the number of a goal as argument, which may be either an absolute value, or a value relative to the number of the current goal. We denote this by [+|-]**n**.

`Assumption` [+|-]**n**	close goal by an assumption
`Claim` **term**	create a new goal (lemma) in a proof
`Cut` $[\mathbf{id_1}=\mathbf{id_2}]$	apply substitution lemma
`Dnf`	compute the display-normal form of the current goal
`ExpAll` **n**, `ExpAll` **+n**	expand all definitions in a goal
`Expand` $\mathbf{id}_1$ $\mathbf{id}_2$ ...	use definitions $\mathbf{id}_1$ $\mathbf{id}_2$ ... to expand the current goal
`Equiv` **term**	replace current goal with equivalent goal **term**
`Hnf`	compute the (weak) head normal form of the current goal
`Hnf` **n**	compute the (weak) head normal form of the current goal for **n** toplevels
`Immed`	remove all goals which unify with some type in the context
`Invert` **id**	invert premise **id**
`Induction` **id**	induction on premise **id**
`Induction` **n**	induction on **n**th unnamed premise
`Induction` **term**	abstract **term** from the goal, then do induction on it
`intros` $\mathbf{id}_1$ $\mathbf{id}_2$ ...	named Π introduction on the current goal
`intros` [+|-]**n** $\mathbf{id}_1$ $\mathbf{id}_2$ ...	named Π introduction on the goal `?`[+|-]**n**
`intros`	Π introduction on the current goal
`intros` [+|-]**n**	Π introduction on a goal
`Intros` $\mathbf{id}_1$ $\mathbf{id}_2$ ...	named Π introduction on head normal form of the current goal
`Intros` [+|-]**n** $\mathbf{id}_1$ $\mathbf{id}_2$ ...	named Π introduction on head normal form of the goal `?`[+|-]**n**
`Intros`	Π introduction on head normal form of the current goal
`Intros` [+|-]**n**	Π introduction on head normal form of a goal
`Intros #`	single introduction on Σ types
`KillRef`	kill the current refinement proof
`Next` [+|-]**n**	focus on another goal
`Prf`	show current proof context
`Normal`	normalise the current goal

`Qnify` **n**	unify, considering leftmost **n** equational premises
`Qnify`	unify, considering all equational premises
`Qnify` **id**	unify equational premise **id**
`Qrepl` **term**	use the type of **term** as a conditional equation to rewrite the current goal
`Refine` **term**	refine the current goal by **term**
`Refine` [+|-]**n** **term**	refine the goal `?`[+|-] **n** by **term**
`Save`, `Save` **id**	save persistent proof term
`$Save`, `$Save` **id**	save non-persistent proof term
`Undo` **n**	undo **n** proof steps

The Module mechanism

`ForgetMark` **id**	forget back to the mark **id**
Load **id**	load module **id** if not yet loaded
`Make` **id**	load object file corresponding to module **id**
`Marks`	list current marks
`Module` **id**	module header which creates the mark **id**
`Module` $\mathbf{id}_0$ `Import` $\mathbf{id}_1$ $\mathbf{id}_2$ ...	module header with imported modules $\mathbf{id}_1$ $\mathbf{id}_2$ ... creating the mark $\mathbf{id}_0$
`Reload` **id**	is an abbreviation for `ForgetMark` **id**`;` `Load` **id**
`Reload` $\mathbf{id}_0$ `From` $\mathbf{id}_1$	is an abbreviation for `ForgetMark` $\mathbf{id}_1$`;` `Load` $\mathbf{id}_0$

Tacticals

$\mathbf{exprsn_1}$ `Then` $\mathbf{exprsn_2}$	evaluate $\mathbf{exprsn_1}$, if evaluation succeeds, evaluate $\mathbf{exprsn_2}$
$\mathbf{exprsn_1}$ `Else` $\mathbf{exprsn_2}$	evaluate $\mathbf{exprsn_1}$, if evaluation fails, backtrack and evaluate $\mathbf{exprsn_2}$
`Repeat` $\mathbf{exprsn_1}$	evaluate $\mathbf{exprsn_1}$ `Then` $\mathbf{exprsn_1}$ `Then` ...
`For` **n** $\mathbf{exprsn_1}$	evaluate $\mathbf{exprsn_1}$ `Then` $\mathbf{exprsn_1}$ `Then` ... (**n** times)
`Succeed`	this tactical always succeeds
`Fail`	this tactical always fails
`Try` **exprsn**	evaluate **exprsn** and backtrack if evaluation fails

Appendix C

Code for the Lego General Theory of Operational Semantics

```
[Set = Type(0)];
Include lib_start_up;
Include lib_unit;
Include lib_list;
Include lib_prod;
Include lib_sigma;
Include lib_NElist;
Include lib_sum.l;
Include lib_empty.l;
Include lib_ML.l;
[Q = Eq];

(*      Specification Section           *)
(*      *********************           *)

(* Set of sorts S *)
(* This can be done by declaring the sorts directly as (S)Types *)
(* Or by sigma type with functional and relational signatures   *)

Record [Sort:Type(0)]
Fields
 [sort:nat];
[Formula = (make_Sort one)];
```

```
Goal sort_eq : Sort->Sort->bool;
intros s1 s2;
Refine nat_eq;
Refine sort s1;
Refine sort s2;
Save;

(* Sort equality theorem *)
Goal Sort_Eq : {s1,s2:Sort}
               iff (is_true (sort_eq s1 s2))
                   (Eq s1 s2);
intros;
andI;
(* ==> *)
Refine Sort_elim [s1:Sort]
                 (is_true (sort_eq s1 s2))->(Eq s1 s2);
intros _;
Refine Sort_elim [s2:Sort]
                 (is_true (sort_eq (make_Sort sort1) s2))->
                 (Eq (make_Sort sort1) s2);
intros;
Refine Eq_resp;
Refine fst (nat_eq_character ? ?);
Immed;
(* ==> *)
intros;
Qrepl H;
Refine nat_eq_refl;
Save;

(* Identifiers *)
(* Exactly as above *)

Record [FIdent:Type(0)]
Fields
 [id:nat];
```

```
Goal id_eq : FIdent->FIdent->bool;
intros f1 f2;
Refine nat_eq;
Refine id f1;
Refine id f2;
Save;

Goal Id_Eq : {f1,f2:FIdent}
               iff (is_true (id_eq f1 f2))
                   (Eq f1 f2);
intros;
andI;
(* ==> *)
Refine FIdent_elim [f1:FIdent]
                   (is_true (id_eq f1 f2))->(Eq f1 f2);
intros _;
Refine FIdent_elim [f2:FIdent]
                   (is_true (id_eq (make_FIdent id1) f2))->
                   (Eq (make_FIdent id1) f2);
intros;
Refine Eq_resp;
Refine fst (nat_eq_character ? ?);
Immed;
(* ==> *)
intros;
Qrepl H;
Refine nat_eq_refl;
Save;

[FSig = list|(prod FIdent (prod  (list|Sort) Sort))];
[FS|FSig];

(* First Order Signature is the triple (S,F,P) *)
(* May or may not include S as mentioned before *)

[botSort = make_Sort zero];
```

```
[IDSort1 [f:FIdent] =
       (listiter (nil|Sort)
                ([h:(prod FIdent (prod (list|Sort) Sort))]
                 [tl:(list|Sort)]
                       bool_rec  (Fst (Snd h))
                                tl
                                (id_eq f (Fst h)))
       ) FS];

[IDSort2 [f:FIdent] =
       (listiter botSort
                ([h:(prod FIdent (prod (list|Sort) Sort))]
                 [so:Sort]
                       bool_rec  (Snd (Snd h))
                                so
                                (id_eq f (Fst h)))
       ) FS];


(* Terms *)

(* The sort of a meta-variable is a component of the
   definition of Terms *)

Inductive [Term:Sort->Type(0)]
          [Tlist:(list|Sort)->Type(0)]
Constructors
 [var:{n:nat}{s:Sort}Term s]
 [fa:{f:FIdent}
     {tl:(Tlist (IDSort1 f))}
     Term (IDSort2 f)]
 [tnil:Tlist (nil|Sort)]
 [tcons:{s|Sort}
        {sl|(list|Sort)}
        {t:(Term s)}
        {lt:(Tlist  sl)}(Tlist  (cons s sl))];
```

```
[termrec  = [s|Type]
            [t|Type]
            Term_elim ([a|Sort][_:Term a]s)
                      ([b|list|Sort][_:Tlist b]t)];
[tlistrec = [s|Type]
            [t|Type]
            Tlist_elim ([a|Sort][_:Term a]s)
                       ([b|list|Sort][_:Tlist b]t)];

(* Boolean and Prop equality inequality for terms *)
Include "term_eq.l";

(* Rules: *)

Inductive [SCList:Type(0)]
Constructors
[SCnil:SCList]
[SCcons:{s|Sort}(Term s)->(Term s)->SCList->SCList];


[Rule = (prod (list|(Term Formula))
        (prod (SCList)
              (Term Formula)))];

[Conc      = [r:Rule]Snd (Snd r)];
[SideConds = [r:Rule]Fst (Snd r)];
[Prems     = [r:Rule]Fst r];



[RuleSet = (NElist|Rule)];
[RSet:RuleSet];

(* Rule Selection *)
(* If n is larger than the length of the list,
   the last element is returned              *)
```

```
[HdRule = NElistiter ([r:Rule]r)
                     ([r:Rule][_:Rule]r)];
[TlRule = NElistrec ([r:Rule]Nnil r)
                    ([_:Rule]
                     [rt:NElist|Rule]
                     [_:NElist|Rule]rt)];

[RuleNum [n:nat] =
   [z [rl:NElist|Rule] = HdRule rl]
   [s [_:nat]
      [f:(NElist|Rule)->Rule]
      [rls:(NElist|Rule)] = f (TlRule rls)]
   nat_rec z s n RSet];


(*      Semantic Section        *)
(*      ****************        *)
(* Substitution *)

[Subst = {s:Sort}list|(prod nat (Term s))];

[initSub = ([s:Sort](nil|(prod nat (Term s)))):Subst];

Goal updateSub :
        {s|Sort}{n:nat}{t:Term s}{Sub:Subst}Subst;
Intros s n t recSub s';
Refine bool_elim [cond:bool](Eq (sort_eq s' s) cond)->
                     (list|(prod nat (Term s')));
Refine +3 Eq_refl;
intros;
Qrepl fst (Sort_Eq ? ?) H;
Refine (cons (Pair n t) (recSub s));
intros;
Refine recSub s';
Save;
```

```
(* Type of SubFn: Subst->Sort->nat->(Term s) *)
[SubFn = [Sub:Subst][s:Sort][n:nat]
         listiter (var n s)
                  ([h:(prod nat (Term s))]
                   [recn:Term s]
                   bool_rec (Snd h)
                            recn
                            (nat_eq n (Fst h)))
                  (Sub s)];

[TSubFn = [Sub:Subst]
    Term_elim ([s|Sort][t:Term s]Term s)
              ([sl|list|Sort][t:Tlist sl]Tlist sl)
              ([n:nat][s:Sort]SubFn Sub s n)
              ([f:FIdent][tl:(Tlist (IDSort1 f))]
               [res:(Tlist (IDSort1 f))]
               ((fa f res):(Term (IDSort2 f))))
              tnil
              ([s|Sort][sl|list|Sort]
               [t:Term s][tl:Tlist sl]
               [resT:Term s][resTl:Tlist sl]
               (tcons resT resTl))];

(* This one is useful for proving transformations *)
[tlistSubFn = [Sub:Subst]
    Tlist_elim ([s|Sort][t:Term s]Term s)
               ([sl|list|Sort][t:Tlist sl]Tlist sl)
               ([n:nat][s:Sort]SubFn Sub s n)
               ([f:FIdent][tl:(Tlist (IDSort1 f))]
                [res:(Tlist (IDSort1 f))]
                ((fa f res):(Term (IDSort2 f))))
                tnil
                ([s|Sort][sl|list|Sort]
                 [t:Term s]
                 [tl:Tlist sl]
                 [resT:Term s]
                 [resTl:Tlist sl](tcons resT resTl))];
```

```
[TlistSubFn = [s|Sort]
              [Sub:Subst]
              listiter (nil|(Term s))
                       ([h:Term s]
                        [recn:list|(Term s)]
                        (cons (TSubFn Sub h) recn))];

(* Side Condition Functions *)
(* Function checks all side conditions in a list hold *)

Goal SCHold : SCList->Prop;
Refine SCList_elim ([SC:SCList]Prop);
Refine trueProp;
intros s t1 t2 SC recn;
Refine (TermNeq t1 t2) /\ recn;
Save;


(* Function applying substitution to a list of side conditions *)

Goal SCSub : Subst->SCList->SCList;
intros sub;
Refine SCList_elim ([SC:SCList]SCList);
Refine SCnil;
intros s t1 t2 SCTail SCrecn;
Refine SCcons (TSubFn sub t1) (TSubFn sub t2) SCrecn;
Save;

(* Automated Proof for when there are no side conditions *)

Goal NilSCPrf :SCHold SCnil;
Intros;
Refine H;
Save;

(* Automated Proof for when there are many side conditions *)
```

```
Goal ConsSCPrf : {s|Sort}
                 {l|SCList}
                 {t1,t2:Term s}
                 (TermNeq t1 t2)->
                 (SCHold l)->
                 (SCHold (SCcons t1 t2 l));
Intros;
Refine H2;
Refine H;
Refine H1;
Save;

(* Meanings in terms of Judgements *)
Inductive [Judgement:(Term Formula)->Type(0)]
          [Jlist:(list|(Term Formula))->Type(0)]
Constructors
        [ruleAp:{i:nat}
                {Sub:Subst}
                {p:(Jlist (TlistSubFn Sub (Prems (RuleNum i))))}
                (SCHold (SCSub Sub (SideConds (RuleNum i))))->
                Judgement (TSubFn Sub (Conc (RuleNum i)))]
        [jnil:Jlist (nil|(Term Formula))]
        [jcons:{f|(Term Formula)}
               {fl|(list|(Term Formula))}
               {jh:(Judgement f)}
               {jt:(Jlist fl)}Jlist (cons f fl)];



(* Elimination rule for a non-empty list of premisses *)
Goal {P:{f|Term Formula}{fl|list|(Term Formula)}
         (Jlist (cons f fl))->Type(0)}
     {pl:{f|Term Formula}{fl|list|(Term Formula)}
         {jh:Judgement f}{jt:Jlist fl}P (jcons jh jt)}
     {f|Term Formula}
     {fl|list|(Term Formula)}
     {jl:Jlist (cons f fl)}P jl;
Intros;
```

```
(* Type {l|list|(Term Formula)}(Jlist l)->Type(1) *)
[PP = list_elim ([fl:list|(Term Formula)](Jlist fl)->Type)
                ([_:Jlist (nil|(Term Formula))]Unit)
                ([f:Term Formula]
                 [fl:list|(Term Formula)]
                 [_:(Jlist fl)->Type]
                 [jl:(Jlist (cons f fl))]P jl)];
Refine Jlist_elim ([f|Term Formula][_:Judgement f]Unit)
            PP
            ([i:nat]
             [sub:Subst]
             [jl:Jlist (TlistSubFn sub (Fst (RuleNum i)))]
             [SC:SCHold (SCSub sub (Fst (Snd (RuleNum i))))]
             [_:PP (TlistSubFn sub (Fst (RuleNum i))) jl]void)
            void
            ([f|Term Formula]
             [fl|list|(Term Formula)]
             [jh:Judgement f]
             [jt:Jlist fl]
             [_:Unit]
             [_:PP fl jt]pl jh jt)
            jl;
Save Jlist_cons_elim;

(* Extracting the head premiss of
   a non-empty list of premisses *)
Goal HeadPremiss :
     {f|Term Formula}
     {fl|list|(Term Formula)}
     {jl:Jlist (cons f fl)}(Judgement f);
Refine Jlist_cons_elim ([f|Term Formula]
                        [fl|list|(Term Formula)]
                        [jl:Jlist (cons f fl)](Judgement f));
Intros;
Refine jh;
Save;
```

```
(* Extracting the tail premisses of
   a non-empty list of premisses *)
Goal TailPremisses :
     {f|Term Formula}
     {fl|list|(Term Formula)}
     {jl:Jlist (cons f fl)}(Jlist fl);
Refine Jlist_cons_elim ([f|Term Formula]
                        [fl|list|(Term Formula)]
                        [jl:Jlist (cons f fl)](Jlist fl));
Intros;
Refine jt;
Save;



Goal (NElist|Rule)->Rule;
Refine NElistiter;
intros r;
Refine r;
intros hr lastr;
Refine lastr;
Save NElistLast;

Goal (NElist|Rule)->Rule;
Refine NElistiter;
intros r;
Refine r;
intros headr tr;
Refine headr;
Save NElistHead;

Goal (NElist|Rule)->(NElist|Rule);
Refine NElistrec;
intros r;
Refine (Nnil r);
intros hr tailrs tlrs;
Refine tailrs;
Save NElistTail;
```

```
Discharge FS;


(* Spec is a pair: First Order Signature and a Rule Set over it *)

[Spec = sigma|FSig|RuleSet];

[jlistrec = [FS|FSig][RSet:RuleSet|FS][s|Type][t|Type]
            Jlist_elim RSet  ([f|(Term|FS Formula)]
                               [_:Judgement RSet f]s)
                              ([fl|(list|(Term|FS Formula))]
                               [_:Jlist RSet fl]t)
];


[RuleRec = [FS|FSig]
           [RSet:RuleSet|FS]
           [f:(Rule|FS)->Prop]
           NElistiter ([r:Rule|FS](f r))
                      ([r:Rule|FS]
                       [p:Prop](f r) /\ p)
                      RSet];


Goal Rlemma : {FS|FSig}
              {RSet:RuleSet|FS}
              {P:(Rule|FS)->Prop}
              (RuleRec RSet P)->
               {i:nat}P (RuleNum RSet i);
intros FS;
Refine NElist_elim ([RSet:RuleSet|FS]
                    {P:(Rule|FS)->Prop}
                    (RuleRec RSet P)->
                    {i:nat}P (RuleNum RSet i));
intros r P H;
Refine nat_elim ([i:nat]P (RuleNum (Nnil r) i));
Refine H;
intros i H1;
Equiv P (RuleNum (Nnil r) i);
Refine H1;
intros HeadRule TailRules H P H1;
Refine nat_elim ([i:nat]
                 P (RuleNum (Ncons HeadRule TailRules) i));
Refine fst H1;
intros i H2;
Refine H;
Refine snd H1;
Save;

Goal Conj : {FS|FSig}
            {RSet|(RuleSet|FS)}
            {P:{f|(Term|FS Formula)}(Judgement RSet f)->Prop}
            {fl|list|(Term|FS Formula)}(Jlist RSet fl)-> Prop;
intros FS RSet P;
Refine jlistrec RSet
          ([i:nat]
           [Sub:Subst|FS]
           [jl:Jlist RSet
                    (TlistSubFn Sub
                                (Prems (RuleNum RSet i)))]
           [scl:SCHold (SCSub Sub
                             (SideConds (RuleNum RSet i)))]
           [_:Prop]P (ruleAp RSet i Sub jl scl))
          trueProp
          ([f|(Term|FS Formula)]
           [fl|(list|(Term|FS Formula))]
           [j:Judgement RSet f]
           [jl:Jlist RSet fl]
           [J:Prop]
           [JL:Prop]J /\ JL);
Save;
```

```
[J_Induction =   [FS|FSig]
                 [RSet:RuleSet|FS]
                 [P:{f|(Term|FS Formula)}(Judgement RSet f)->Prop]
                 Judgement_elim RSet P
                                ([fl|(list|(Term|FS Formula))]
                                 [jl:Jlist RSet fl]Conj P jl))];



(* Append Substitutions *)

Goal appendSubs : {FS|FSig}(Subst|FS)->(Subst|FS)->(Subst|FS);
Intros FS Sub1 Sub2 s;
Refine append (Sub1 s) (Sub2 s);
Save;

(* This is a very naive function *)
Goal unifSub :  {FS|FSig}
                {s|Sort}
                {conc:Term|FS s}
                {goal_f:Term|FS s}
                (Subst|FS);
intros FS;
(* Term elim on conclusion of given rule *)
Refine Term_elim|FS ([s|Sort]
                     [t:Term|FS s]
                     (Term|FS s)->Subst|FS)
                    ([sl|list|Sort]
                     [tl:Tlist|FS sl]
                     (Tlist|FS sl)->Subst|FS);

(* var case for conc *)
intros n s t;
Refine updateSub|FS n t (initSub|FS);
(* function app case *)
intros _ _ _;
Refine Term_elim|FS ([s|Sort][t:Term|FS s]Subst|FS)
                    ([sl|list|Sort][tl:Tlist|FS sl]Subst|FS);
```

```
intros; Refine initSub|FS;
intros f1;
Refine bool_elim [cond:bool]
                 (Eq (id_eq f1 f) cond)->
                 ((Tlist|FS (IDSort1|FS f1))->
                 (Subst|FS)->Subst|FS);
Refine +3 Eq_refl;
intros H;
Qrepl fst (Id_Eq ? ?) H;
intros;
Refine tl_ih H1;
intros; Refine initSub|FS;
Refine initSub|FS;
intros; Refine initSub|FS;

(* tnil case *)
intros;
Refine initSub;

(* tcons case *)
intros;
Refine appendSubs (t_ih (HeadTerm|FS H))
                  (lt_ih (TailTerms|FS H));
Save;

[rAP =  [FS|FSig]
        [RSet:RuleSet|FS]
        [i:nat]
        [f:(Term|FS Formula)]
        [r = RuleNum RSet i]
        [conc = Conc|FS r]
        [scs  = SideConds|FS r]
        [prems= Prems|FS r]
        [sub = unifSub|FS conc f]
        [jl: Jlist RSet (TlistSubFn sub prems)]
        [sc: SCHold (SCSub sub scs)]ruleAp RSet i sub jl sc];
```

```
Goal sclist_eq : {FS|FSig}(SCList|FS)->(SCList|FS)->bool;
intros _;
Refine SCList_elim ([_:SCList|FS](SCList|FS)->bool);
Refine SCList_elim ([_:(SCList|FS)]bool);
Refine true;
intros; Refine false;
intros s t1 t2 scl ih_sch;
Refine SCList_elim ([_:(SCList|FS)]bool);
Refine false;
intros s' t1' t2' scl' ih';
Refine bool_elim ([cond:bool](Eq (sort_eq s s') cond)->bool);
Refine +3 Eq_refl;
Next +1; intros; Refine false;
intros s_eq;
[s_eq1 = fst (Sort_Eq s s') s_eq];
Refine andalso;
Refine andalso (term_eq t1' (Eq_subst s_eq1 ? t1))
               (term_eq t2' (Eq_subst s_eq1 ? t2));
Refine ih_sch scl';
Save;

[rule_eq = [FS|FSig]
           [r1,r2:Rule|FS]
           andalso
            (list_eq (term_eq|FS|Formula) (Prems r1) (Prems r2))
            (andalso (sclist_eq (SideConds r1) (SideConds r2))
                     (term_eq (Conc r1) (Conc r2)))];

Goal RuleNameToNum :
     {FS|FSig}(Rule|FS)->(RuleSet|FS)->nat;
intros FS r RSet;
Refine NElist_elim ([_:RuleSet|FS]nat->nat);
Refine +2 RSet;
Refine +2 zero;
intros;
Refine suc H;
intros r rl ih n;
```

```
Refine bool_rec ? ? (rule_eq r r1);
Refine n;
Refine suc (ih n);
Save;

[ruleAp' =
        [FS|FSig]
        [RSet:RuleSet|FS]
        [r:Rule|FS]
        [i = RuleNameToNum r RSet]ruleAp RSet i];


(* The Theory's version of derive_cases_thm in HOL *)

[Jl_exists =
        [FS|FSig]
        [RSet:RuleSet|FS]
        [RSet':RuleSet|FS]
        [f:Term|FS Formula]
        [i:nat]
        sigma|(Subst|FS)
             |([sub:Subst|FS]
          prod (Jlist RSet
                     (TlistSubFn sub (Prems (RuleNum RSet' i))))
               (Eq f (TSubFn sub (Conc (RuleNum RSet' i)))))];

[JlistSum =
        [FS|FSig]
        [RSet:RuleSet|FS]
        [RSet':RuleSet|FS]
        [f:Term|FS Formula]
        NElist_elim ([_:RuleSet|FS]Type(0))
                    ([r:Rule|FS]
                     sigma|(Subst|FS)
                          |([Sub:Subst|FS]
                      prod (Jlist RSet (TlistSubFn Sub (Prems r)))
                           (Eq f (TSubFn Sub (Conc r)))))
                    ([r:Rule|FS]
                     [rl:NElist|(Rule|FS)]
                     [rl_ih:Type(0)]
                     sum (sigma|(Subst|FS)
                          |([Sub:Subst|FS]
                      prod (Jlist RSet (TlistSubFn Sub (Prems r)))
                           (Eq f (TSubFn Sub (Conc r)))))
                         rl_ih)
                    RSet'
];

(* Again we want to convert something referring to rules
   by numbers to something that is list recursive *)
Goal sngRSetEq :
        {FS|FSig}
        {r:Rule|FS}
        {i:nat}Eq (RuleNum (Nnil r) i) r;
intros FS r;
Refine nat_elim ([i:nat]Eq (RuleNum (Nnil r) i) r);
Refine Eq_refl;intros i ind_hyp;
Refine ind_hyp;
Save;

Goal ith_sum :
        {FS|FSig}
        {f:Term|FS Formula}
        {RSet:RuleSet|FS}
        {i:nat}
        (Jl_exists RSet RSet f i)->
          (JlistSum RSet RSet f);
intros FS f RSet;
Refine NElist_elim ([RSet':RuleSet|FS]
                    {i:nat}(Jl_exists RSet RSet' f i)->
                           JlistSum RSet RSet' f);
(*base case *)
intros r i; Expand Jl_exists; Qrepl sngRSetEq r i;
intros; Refine H;
```

```
(* ind step *)
intros r rl rl_ih;
Refine nat_elim ([i:nat](Jl_exists RSet (Ncons r rl) f i)->
                        JlistSum RSet (Ncons r rl) f);
intros; Refine in1; Refine H;
intros n n_ih;
Equiv (Jl_exists RSet rl f n)->(JlistSum RSet (Ncons r rl) f);
intros;
Refine in2;
Equiv JlistSum RSet rl f;
Refine rl_ih n H;
Save;

Goal JRSum :
     {FS|FSig}
     {RSet:RuleSet|FS}
     {f|(Term|FS Formula)}
     {j:Judgement RSet f}
     JlistSum RSet RSet f;
intros FS RSet;
Refine Judgement_elim RSet ([f|Term|FS Formula]
                            [_:Judgement RSet f]
                            JlistSum RSet RSet f)
                           ([fl|(list|(Term|FS Formula))]
                            [_:Jlist RSet fl]Jlist RSet fl);
Refine +1 jnil; Next +1;
intros f fl j jl j_ih jl_ih;
Refine jcons RSet j jl;
intros i Sub jl sch jl_ih;
Refine ith_sum (TSubFn Sub (Conc (RuleNum RSet i))) RSet i;
Refine dep_pair Sub;
Refine Pair jl;
Refine Eq_refl;
Save;
```