The Theory of LEGO

A Proof Checker for the Extended Calculus of Constructions

Robert Pollack

Doctor of Philosophy

University of Edinburgh

1994

Atomics is a very intricate theorem and can be worked out with algebra but you would want to take it by degrees because you might spend the whole night proving a bit of it with rulers and cosines and similar other instruments and then at the wind-up not believe what you had proved at all. If that happened you would have to go back over it till you got a place where you could believe your own facts and figures as delineated from Hall and Knight's Algebra and then go on again from that particular place till you had the whole thing properly believed and not have bits of it half-believed or a doubt in your head hurting you like when you lose the stud of your shirt in bed.

Flann O'Brien, *The Third Policeman*

Abstract

LEGO is a computer program for interactive typechecking in the Extended Calculus of Constructions and two of its subsystems. LEGO also supports the extension of these three systems with inductive types. These type systems can be viewed as logics, and as meta languages for expressing logics, and LEGO is intended to be used for interactively constructing proofs in mathematical theories presented in these logics. I have developed LEGO over six years, starting from an implementation of the Calculus of Constructions by Gérard Huet. LEGO has been used for problems at the limits of our abilities to do formal mathematics.

In this thesis I explain some aspects of the meta-theory of LEGO's type systems leading to a machine-checked proof that typechecking is decidable for all three type theories supported by LEGO, and to a verified algorithm for deciding their typing judgements, assuming only that they are normalizing. In order to do this, the theory of Pure Type Systems (PTS) is extended and formalized in LEGO. This extended example of a formally developed body of mathematics is described, both for its main theorems, and as a case study in formal mathematics. In many examples, I compare formal definitions and theorems with their informal counterparts, and with various alternative approaches, to study the meaning and use of mathematical language, and suggest clarifications in the informal usage.

Having outlined a formal development far too large to be surveyed in detail by a human reader, I close with some thoughts on how the human mathematician's state of understanding and belief might be affected by posessing such a thing.

Acknowledgments

Rather than the usual technical acknowledgements, I have written a history of LEGO (section 1.2) and of the project reported in this thesis (section 1.3). In those sections I try to say who and what were the important influences on this work. I thank all of them, as I have loved doing the work.

I also want to thank LFCS and the many friends I have here. Rod Burstall, my research supervisor, has endless enthusiasm for computer aided formal reasoning, and has supported and encouraged my work. I especially want to thank George Cleland, who, as administrator of LFCS, has always supported research over red tape. LFCS has always managed to find the resources I asked for.

Talking about resources, this thesis burned more computer cycles than most, and I don't just mean for LaTeX. The computing services available at LFCS are the best I have encountered. Thanks to Paul Anderson, George Cleland, Morna Findlay, and many other CO's.

Finally, I want to thank two people who have influenced my mathematical aesthetic. Gabe Stolzenberg taught me most of what I know about constructive mathematics, and encouraged looking deeply at the meaning of mathematical things. Clint McCrory is a college friend I haven't seen in many years. When I was an undergraduate at M.I.T. and he was a postgraduate at Brandeis, he somehow, through discussion of class problem sets from my second year analysis course, changed me from a mediocre student who enjoyed mathematics, to someone obsessed with the exact shape and presentation of proofs.

This work was supported by the ESPRIT Basic Research Actions on Logical Frameworks (LF) and Types for Proofs and Programs (TYPES), and by grants from the British Science and Engineering Research Council.

Declaration

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

Robert A. Pollack

Table of Contents

Chapter 1

Introduction

I am interested in completely formal mathematics: a proof is ultimately a derivation in some given formal system. I believe that in practice, this can only be achieved with machine assistance. LEGO [LP92,JP93,JP94] is an interactive proof development system (proof checker) for Luo's Extended Calculus of Constructions (ECC) [Luo90a,Luo94], and for two subsystems of ECC. To support the interactive construction of typing derivations in these three type theories, LEGO provides *refinement* proof, i.e. top-down, or goal-directed proof, and some non-trivial syntactic sugar. LEGO is a basic proof checker, robust and pragmatic.

The core of LEGO is a typechecker for ECC. In this thesis I develop the theory of a class of type systems, and based on this theory, give a (parameterized) typechecking algorithm and prove it correct for all three type systems supported by LEGO. This entire theory is developed in the formal language of ECC with inductive types [Luo94], and machine checked using LEGO. This is an important contribution because correctness of typechecking is central for the approach to proof checking based on the type theoretic representation of logic (section 1.1).

The theory developed to achieve this goal falls into two broad classes:

general type theory: A theory of some reduction, conversion, and typing relations over a language of explicitly typed (so called *Church style*) lambda terms. This theory is based on well known informal mathematics, and includes:

- Some basic lambda calculus, e.g. definitions of the language, substitution, reduction and conversion, and the basic theory of reduction and conversion. The major result here is the Church-Rosser theorem.
- The basic theory of Pure Type Systems (PTS). Here the typing judgement is defined, and results such as the substitution (cut) lemma, the correctness of types, and subject reduction (closure under reduction) are proved.

specific to type checking: Here we develop new mathematics (some of it previously published in [vBJMP94]) solving the difficulties encountered in extending previously understood techniques of typechecking to the type theories of LEGO.

Although the material on general type theory mentioned above is based on well known informal mathematics, our formal presentation is an important contribution of this thesis, as it is compared with informal presentations, clarifying the meaning of informal statements and their proofs. (For example, sections 3.2.5, 3.3.6, 4.1.2, 4.1.3.1 and 4.4.6.) Among the interesting points, our use of named variables, as opposed to de Bruijn nameless variables, throws some light on where alpha conversion is really necessary, and our formalization of the typing rules for PTS has suggested several modifications to the informal presentation, improving both intuitive understanding and technical properties of the system. Many of our proofs are more elegant than previously published informal proofs of the same results, having, for example, many fewer cases to check (see section 4.4.9). Viewing from another level, we have a realistic example of formal development of a non trivial body of mathematics. This work is not focussed on proving one big theorem, but is a body of formal knowledge that has developed in directions not originally anticapated, and that I hope will continue to be extended.

The material on typechecking is not beautiful mathematics; it is technology. Although the derived typechecker program is not a pragmatically acceptable implementation, it contains some of the detailed optimizations that are necessary to make such a typechecker efficient. What is of interest is that we are able to reason formally about such things in a straightfoward manner.

Finally, I suggest an approach to the problem of how to make sense of a big formal development that you cannot actually read or understand in its entirety. There are two questions here:

- How can you know the meaning of some statement appearing in a big formal development?
- How can you believe that a statement claimed to be proved in a big development is actually proved?

The first question is no different for formal mathematics than for informal: you must read the statement you want to understand, and all of the definitions (hereditarily) used in it. The second question is more interesting. When presented with several million ascii characters and the claim "here is a proof of **...**", one way to get evidence for believing the claim is to mechanically check the proof yourself. But human beings are not very good at such things; instead we can write a computer program to do it. My suggestion is that to believe a big formal development you write a simple checker for fully annotated proofs of the logic in question, and check the development with this simple checker.

1.1 Proofchecking and Typechecking

I said I am interested in machine checking mathematical proofs. How does an algorithm for checking typing judgements of various type theories contribute to machine checked mathematics? There are several methods of representing a formal system (loosely the *object system*) in another formal system (loosely the *meta system*) such that derivability of some meta system judgement implies derivability of some formally related object system judgement. This is the basic idea behind our reason for studying type theory and our methodology for formally doing so. Everything I will say about this is well known, but these ideas are not well classified in the folk wisdom, and there are not accepted names to make clear which method is being discussed, and what their relationships are.

1.1.1 The Curry–Howard–Martin-Löf–de Bruijn Isomorphism

This is undoubtedly the best known relationship between intuitionistic logic and type theory. Often called "propositions-as-types" this idea is based on the Heyting explanation [Hey71] of the intuitionistic connectives. For example, a proof of A and B is a pair of a proof of A and a proof of B, so the logical notion, *conjunction*, can be represented by the type theoretic notion of *cartesian product*. Similarly disjunction is represented as disjoint union (i.e. sum), implication as function space, universal quantifier as dependent function space (i.e. general product) and existential quantifier as dependent product (i.e. general sum). Much ink has been expended on detailing this idea; a modern and exact treatment is given in [Bar91]. In this sense we have a conservative embedding of intuitionistic higher order logic in ECC [Luo90b]. With this approach we develop the internal mathematics of a particular type theory, e.g. ECC, which is seen to be conservative over (or isomorphic to) some standard logical system.

1.1.2 Direct Inductive Representation of an Object Theory

Some formal systems represent inductively defined objects of some sort. Impredicative systems such as CC, ECC, or HOL [Gor88] can express "the smallest set (type, relation) containing $\ldots$". Some systems (e.g. Coq [PM93]), even predicative ones (e.g. the Martin-Löf framework [Dyb94] and Feferman's framework [Fef88]), have explicit forms of controlled inductive definition. Such formal systems can be used as a formal meta theory in which one can directly represent the inductive definition of derivations of an object system. If this is to be useful, the meta theory itself must have some internal logic, possibly quite weak; it is folk wisdom that Primitive Recursive Arithmetic (PRA) is adequate to formalize most of mathematics in this style. This internal logic of the meta theory may be based on the Curry–Howard isomorphism, as in ECC. With such a representation the object derivations are themselves objects of the meta theory, and one can prove admissible rules of the object theory, although a weak meta theory will prove fewer admissible rules than a strong one. This is the formal analogue of how mathematics is usually done informally, i.e. the formal meta theory stands in for our informal meta language, and the formal object theory stands in for the mathematical theory under study, e.g. first order logic with set theory. In this thesis I represent the type systems that are the objects of study in this style, using ECC with inductive definitions as a meta theory; inductive definitions are used to represent the object syntax (terms, contexts, ...) and the relations on that syntax (reduction, conversion, typing, ...).

1.1.3 Logical Frameworks

This idea appears in Automath [dB80], and is clarified and studied by Martin-Löf [NPS90] and by Plotkin *et. al.* [HHP92,AHMP92,Pym90,Gar92]. Here the type system used as a meta theory may be very weak (e.g. consistency of the Edinburgh Logical Framework (LF) can be proved in PRA), although stronger systems are also used as meta logics in this way, e.g. Isabelle [Pau93a], an impredicative higher order logic. An object system is represented as a signature of constants for its language, axioms and rules. The inductive structure of object theories is not represented by explicit inductive definitions in the meta theory, but is captured by the inductiveness of the definition of the meta theory itself (see examples in [HHP92, AHMP92,Gar92]). With this approach the theorems, proofs, and also derivable rules of the object theory may be represented, but the object system derivations themselves cannot be manipulated in the meta system; e.g. we can't talk about the length of a proof. Various forms of correctness of representation may be proved, but this depends on details of the object theory and of the representation. LF, designed to support this style of representation, is one of the type theories supported by LEGO, and its use depends on a typechecking algorithm.

1.2 A History of LEGO

I have always been interested in completely formal mathematics. Through a conventional undergratuate education in mathematics, and two years of postgraduate study of mathematical logic, I heard nothing of such a thing. In 1985, while studying program verification from Dick Kemmerer, I encountered Martin-Löf type theory and understood that it was, among other things, about formalization; nnfortunately, I understood nothing else about it. In spring of 1986, for Albert Meyer's Type Theory seminar, I read Christine Paulin-Mohring's paper on "Algorithm Development in the Calculus of Constructions" [Moh86]. In less than two pages she gives the language and typing judgement of the Pure Calculus of Constructions (CC), then gets down to some machine-checked examples. CC is beautiful, the most successful formulation to date of the logicistic program for founding mathematics (see [Ber90b] for that story). CC is also syntactically simple, so I thought I could write a computer program for checking judgements in this logic, use it to experiment with formal mathematics, and along the way learn a little about type theory.

It took less than a week to code a CC typechecker in Prolog, my first lambda calculus implementation, and I began working through the examples in [CH85]. In autumn 1986, at the University of Texas, Austin, my typechecker was received with enthusiastic interest from Bob Boyer, J Moore, and the logic seminar meeting at Boyer's house. In spring 1987, Gérard Huet came to Austin for a workshop, and gave me his prototype implementation of the Constructive Engine [Hue89] coded in CAML, which was really the start of my knowledge of typechecking algorithms (my Prolog typechecker was *very* slow). Through the recommendation of Boyer and Moore, and through Rod Burstall's desire to continue a theorem proving group at Edinbirgh, I came to LFCS, University of Edinburgh, in October 1987, and began seriously to develop Huet's prototype Constructive Engine as a proofchecker for CC and the Edinburgh Logical Framework[1]. The name LEGO was suggested by Paul Taylor (the Edinburgh Paul Taylor, who was the first LEGO user other than myself) to express the fun of formal constructive mathematics.

Between 1987 and the time I am writing this, April 1994, LEGO was frequently extended and improved. Major features include Definitions, Argument Synthesis, Universes, Typical Ambiguity, Sigma Types, Full Cumulativity, Inductive Types, and Modules. While I did most of the implementation work, including the many theoretical details necessary to make these features work[2], the main ideas are due to others, principally Coquand, Huet, Luo, Martin-Löf, and Paulin-Mohring.

Also during this time, students and researchers began to use LEGO in their work. (Rod Burstall, my research supervisor, is mostly responsible for this; he has lectured and written about how to start using LEGO for many conferences, workshops, summer schools, and university classes.) There are, to my knowledge, five University of Edinburgh M.Sc. projects done in LEGO [Mah90,Col90,Bra92,Wan92,Bai93]. There are at present two University of Edinburgh Ph.D. theses using LEGO proofs as significant case studies [McK92,Alt93a], and at least one other is in preparation [Mah94]. Papers or public talks based on large LEGO examples include [Ber90a,Jon93,Men92,Alt93b,MP93,Ciz93]. I know of three Ph.D. students outside of Edinburgh using LEGO in their research, two in Nijmegen and one in Munich.

---

[1] My first talk at Edinburgh was about how to view the Edinburgh Logical Framework as a subsystem of CC, and I implemented it that way from the beginning. For a modern view, see [Bar91,Bar92].

[2] However thanks to Kevin Mitchell who translated LEGO from CAML to SML, and *big* thanks to Claire Jones who implemented the syntax for inductive types, including checking that an inductive specification satisfies Luo's schema, automatically proving double elimination rules, and other useful tactics.

1.3 A History of the Project to Formalize PTS

While developing LEGO, many questions of theory arose. Could the Constructive Engine, the abstract machine at the core of LEGO, be described mathematically and proven correct? (This involves not just a type theory, but a *represenation* of a type theory; for example the Constructive Engine's translation into nameless representation [Pol94].) Could the Constructive Engine be extended to a type theory with universes [Hue87,HP91]? Type checking is only the start of the problem. For example, LEGO supports definitions, assigning a name to a (typed) term; do global definitions preserve normalizability of CC? Do local definitions also preserve normalizability [SP94]? LEGO also uses meta-variables to implement refinement proof. Are they handled correctly? Are LEGO's algorithms correct for testing conversion of types (with definitions) in a lazy manner, and for unification of types? Then there are the meta-operations on the state of the proof, such as natural deduction discharge of assumptions, weakening and strengthening. As a pragmatic system, LEGO has "syntactic sugar", what I've called "Implicit Syntax" [Pol90], such as LEGO's argument synthesis, typical ambiguity, and universe polymorphism; these features have to be explained too. I've still only begun to outline the the questions that actually come up in implementing a proofchecker.

In 1988, Bob Harper and I addressed the problem of typechecking cumulative universes [HP91]; we were far from formalizing our definitions and theorems, let alone their proofs, but we reasoned about complex algorithmic issues in terms of formal systems (i.e. inductively defined relations), and I began to see that it might be possible to formally reason about the kinds of questions raised above.

Based on what I learned from the work with Harper, I started working on operational semantics to describe LEGO features such as argument synthesis, typical ambiguity, and the refinement proof tactic, with something like the SML semantics [MTH90] in mind; that is, a semi-formal explanation of all, or most, of LEGO's operation. I was especially encouraged by Bob Harper and Furio Honsell. My occasional trips to Paris to visit Coquand and Huet opened up new ideas for me.

In 1989, I learned about Generalized Type Systems [Bar91,Ber90b,GN91,Bar92,vBJ93] from Barendregt, Berardi, and Geuvers. This is a framework in which many of the questions about pragmatic extensions to type systems could be expressed. It is also a very beautiful framework, and interesting in its own right. The extensions I suggested to Generalized Type Systems in 1989-90 (sigma types, definitions, cumulativity) were partially responsible for Barendregt changing the name of the core system to Pure Type Systems (PTS), the name I will use in this thesis. PTS is simple enough syntactically to dream of formalizing: five or six term constructors, six or seven inference rules, beta-reduction and beta-conversion are all that is required.

The direction of my work was changing from a semi-formal explanation of all of LEGO to a formal explanation of some core part. Many people have expressed disappointment that the operational semantics of LEGO never matured (and some still hound me with questions about my early writing on that topic [Pol88]), but my original purpose for LEGO was to *do* formal mathematics, and that is what I have started to do.

In 1990 Thierry Coquand suggested using distinct syntactic classes, *parameters* for free names and *variables* for bound names [Coq90,Coq91] as a way to formalize binding with explicit names. Being interested in "taking symbols seriously", I found this suggestion very appealing[3].

Also in spring 1990, Bert Jutting and I started discussing type checking for PTS. I thought the Constructive Engine would generalize easily to PTS, but I hadn't correctly worked out the details (Herman Geuvers pointed out my error). In autumn 1990, Henk Barendregt asked the Expansion Postponement (EP) question (see section 4.7), and by early 1991 Geuvers, Jutting and I pretty much understood the relationship between typechecking and EP, and were feverishly experimenting with obscure reduction relations in the proof of subject reduction in order to solve EP. (Now, typechecking is understood [vBJMP94] but EP is still open.) I was so bored hand-checking proofs of subject reduction for many closely related systems, and I made so many mistakes doing it, that I really felt the need for machine assistance.

By 1991 I had started to experiment with formalizing PTS, using the impredicatively defined types of CC to represent formal systems. With help from Christine Paulin-Mohring, and from reading Stefano Berardi's LEGO proof of normalization for System F [Ber90a], this worked well enough to experiment with both nameless representation and Coquand's named representation. By that time, however, inductive types had become a hot topic in implementation as well as in theory. I had seen Martin-Löf's "encoding" of inductive types implemented in the first version of ALF [ACN90] and, after discussions with Coquand and Luo, had a crude implementation of inductive types in LEGO by spring 1991. (I should have also discussed it with Paulin-Mohring, because I didn't understand about annotating the dependency of an inductive type on its constructors, which her implementation in Coq [DFH+93] had elegantly solved from the start.) With inductive types in LEGO, the formalized PTS project became much more feasible. By this time, my obsession to formalize PTS was driving the development of LEGO to support larger projects. With Claire Jones' implementation of user-level syntax for LEGO to mechanically generate elimination and computation rules from the constructors of an inductive type, it became quite convenient to define and use new inductive relations.

---

[3] de Bruijn nameless representation is also a notation that takes symbols seriously. Martin-Löf has always been concerned with the symbols themselves, but also, more subtly, with where the line is drawn, below which we may identify entities as being different instances of the same symbol, and above which we distinguish them. In his recent system, described in [Tas93], this line is at a very concrete level.

By spring 1992 I had formalized some basic theory of PTS in Coquand's named representation, but was really stuck on the Thinning Lemma (section 4.4.6). James McKinna made a suggestion to overcome the problem that was mathematically more subtle than the straightfoward inductions I had been trying. Together we proved the Thinning Lemma and packaged McKinna's idea as a theorem that could be used in instances of similar problems. A discussion with Coquand was also instrumental in generalizing the idea, which is discussed in sections 3.2.5.3 and 4.4.4. There is a lesson to be learned from this: the problems of formalization are *mathematical* problems, and should be addressed as such.

At this point (autumn 1992) all the pieces were present to formalize much of the theory of PTS. Jutting, McKinna, and I decided to formally check the paper we were writing on typechecking algorithms [vBJMP94]. While Jutting wrote the entire first draft, McKinna formalized the lambda calculus and I formalized the type theory; later, when I was travelling for two months, McKinna did both. By April 1993 it was checked.

1.4 An Overview of the Thesis

In this thesis I present a formal development, checked in LEGO, of an algorithm for type checking a class of type systems including the type systems of LEGO: $\boldsymbol{\lambda P}$, CC, and ECC. However, it is not the correctness of LEGO which I principally wish to communicate (no part of LEGO code is verified in any sense) or even the derivability of the theorems, but the flavor of formalizing a non-trivial body of concrete mathematics, and possibly some reasons for doing it.

Chapter 2 is an introduction to LEGO syntax, and to the basic types supported in the library distributed with LEGO. It is mainly intended to allow you to read the formalization in following chapters, although there are a few comments on induction of more general interest.

Chapter 3 develops the language of PTS, (a Church-style lambda calculus) and its elementary theory of reduction and conversion. I do not go into much detail about the basic language theory (substitution, etc.) or the proofs in this section (much of which will be reported in [McK94]), but concentrate on mathematical techniques of formalization that will recur in later sections, as well as definitions and theorems needed to understand the type theory to follow.

Chapter 4 has the most technical interest. It develops the basic theory of PTS through the subject reduction theorem, using a more general notion of *conversion* than beta-conversion, in order to later use the results for ECC, which is not a standard PTS. Many options and alternatives are examined, and some proofs are described in detail, and compared with published informal arguments. I close with everything I know about the notorious open problem, Expansion Postponement, which, however, is not very much.

Chapter 5 focuses on a subclass, the semi-full PTS. For these we give an inductive relation which characterizes the PTS judgements, but which is deterministic enough to be the basis for a typechecking algorithm. Such an algorithm is constructed from this deterministic relation, and shown to be correct.

Chapter 6 closes the thesis with some informal suggestions on how the reader's state of understanding and belief might be effected by a large formal development. The question is addressed in two layers: whether a claimed formal judgement is actually derivable in the given formal system (i.e. correctness of proof checkers), and what informal belief might be obtained from a formal judgement (i.e. the relationship between formal and informal presentations).

Acknowledgement The work reported in Chapters 3 and 4 is, to a large extent, joint work with James McKinna. While I had been working on formalizing PTS for some time, McKinna's ideas were instrumental in solving some problems I was stuck on. We continued the collaboration, and McKinna did much of the actual proofchecking. Formal proof is a lonely business; because of the detail involved it is difficult to discuss problems with people outside the project. The collaboration between McKinna and me payed off in results and in job satisfaction.

1.5 Production of this Thesis and its Verbatim Inclusions

This thesis is about a formal development, so it is essential I show the reader formal definitions and statements of the theorems. These are all displayed, in typewriter font, as they appear verbatim in the source files that LEGO checks. For example, here is a definition of the inductive type of booleans.

```
Inductive [BB:Prop] ElimOver Type(0) Constructors [tt,ff:BB];
```

The lemmas and theorems that I claim are machine checked will all start with the LEGO command `Goal`, as in this lemma that contraposition is a correct argument.

```
Goal contrapos: {A,B|Prop}(A->B)->(not B)->(not A);
```

This lemma is formally named `contrapos`, and may be referred to either by its formal name (in typewriter font) or informally as in "by contraposition". I occasionally use displayed typewriter font for something that is not checked in LEGO (as in "an alternative definition might be `...`"), where I believe no confusion will arise, but never for complete, syntactically correct, LEGO statements that are not checked. Proofs are written informally, using LaTeX mathematical notation and formal names. Occasionally, as in section 4.7, I state and prove some lemmas informally; those are not checked in LEGO, and no formal results depend on them.

To implement this verbatim display of formal statements I use very simple technology somewhat inspired by Knuth's WEB. Notations in the LEGO proof source files, treated by LEGO as comments, delimit and name sections of LEGO code. There is a program (generated by ml-lex) that scans a LEGO source file and extracts these marked sections into files that are given the name of the section; these files are `\input` into LaTeX and printed in `verbatim` mode. When I produce a version of the document, I check the whole proof development in LEGO to see that the current version is correct, and run the extraction program on all the LEGO source files (say, using Make); thus the document has an up to date and correct version of all the formal extracts.

It has worked well, but does it guarantee that I'm printing what LEGO actually checked? No; many things can go wrong, and several of them have, such as inadvertantly having the same name for different extracted sections, causing one to be overwritten by another. It might be better (but unwieldy for big proofs) to have LEGO do the extraction itself, and keep the names of extracted sections in its own context to prevent duplication, i.e. treat these extractions as formal objects. Would this guarantee that I'm printing what LEGO actually checked? No. A minor theme of this thesis is that there is no such thing as absolute certainty, and machine verification of various kinds does not alter that common truth about the world.

1.6 What to Look For

I don't think I've made it all look easy in the following chapters, but I have written them as ordinary mathematics, if more detailed than is usual. This is to emphasise that formalization is a mathematical problem; but it is a problem where the interaction between the subject matter and the underlying formal logic is much more explicit than in most mathematics. By working with inductively defined notions where intensional equality coincides with *book equality* [dB91] of the object theory (e.g. see the comment in section 3.1.1 on equality), I have restricted myself to subject matter that fits very well with the underlying logic, ECC, and for this reason the underlying logic doesn't appear much in the discussion. We are now pretty good at formalizing such subject matter, but it is a very different matter formalizing some extensional subject, such as algebra, in type theory.

My interest in formalism and formalization is based on a fascination with the details of how and why mathematics goes together. I hope you find in the following that we have examined the choices made and the reasons for those choices, considered some of the alternatives, and have been able to throw some light on the mathematics of PTS.

Chapter 2

Warming Up To LEGO

In this chapter I give a brief overview of LEGO, to assist the reader in understanding the formal definitions and statements of theorems in following chapters. I first present LEGO syntax (since I am not showing formal proofs, I omit the commands for constructing proofs in LEGO), and then present a small library of definitions of logical operators and basic types that will be used in later chapters.

2.1 LEGO

LEGO supports Luo's Extended Calculus of Constructions (ECC) [Luo90a,Luo94] further extended with inductive types. λP and CC [Bar91], viewed as subsystems of ECC, are also supported, and they may also be extended with inductive types. LEGO is freely available by ftp, along with a User's Manual [LP92], some documentation on recent changes [JP93,JP94], a library of definitions and theorems about some basic types [JM93], and some examples. For more details than contained in this chapter see [LP92,JP93,JP94]. The foundations of this thesis were laid before the LEGO library existed, so there are some differences between our basic definitions and [JM93].

In this thesis we use LEGO as a typechecker for ECC extended with inductive types, and for us ECC is used as a logic, in fact a constructive logic with full annotations for the computational content of theorems. However I will not write yet another introduction to the Curry-Howard isomorphism or propositions as types; see [NPS90].

2.1.1 LEGO Term Syntax

Using x as a metavariable for identifiers, and M as a metavariable for terms, the syntax of terms is given in table 2–1. In this thesis I don't use the sigma types or the universes of ECC,

M	::=	`Prop`	built-in type of propositions
	|	x	vars (ML identifiers)
	|	`[`x`:`M`]`M	lambda binding
	|	`[`x`|`M`]`M	implicit lambda binding
	|	`{`x`:`M`}`M	pi binding
	|	`{`x`|`M`}`M	implicit pi binding
	|	M`->`M	non-dependent pi
	|	M M	application
	|	M`|`M	explicit (forced) application
	|	`[`x`=`M`]`M	"let" (local definition)
	|	`(`M`:`M`)`	type cast

Table 2–1: The Basic Syntax of LEGO Terms

so won't describe their syntax. Some possibly mysterious notations in this table are explained below.

Binders The scope of binders goes as far to the right as possible. Implication, `->`, is shorthand for a non-dependent pi, and associates to the right. As usual, application associates to the left. Parentheses overide these conventions. Multiple bindings may share brackets and type labels, e.g. `{A,B:Prop}A->B->A`.

Typecasting in LEGO is similar to typecasting in functional languages such as SML [MTH90]. The term `(`a : A`)` has value a and type A, assuming a and A are both well-typed, and the type of a is convertible with A. (I am being somewhat inexact; see section 5.1.5, especially remark 5.7, for details)

2.1.2 LEGO Contexts

The *context* is a list of all the declarations (assumptions) and definitions currently in use. Using Γ as a meta-variable for contexts, the syntax of contexts is given by:

Γ	::=		empty context
	|	Γ`[`x`:`M`]`	declaration (assumption)
	|	Γ`[`x`=`M`]`	global definition

Contexts are constructed and checked incrementally; the system of locally valid contexts in section 4.4.10.2 makes this precise.

There is special syntax for typecasting at the top-level that is frequently used, for example

```
[x : A = a]
[y = a : A]
```

are shorthand for

```
[x = (a:A)]
[y = (a:A)]
```

This may not seem too interesting by itself, but it combines with another shorthand common in functional languages, so that

```
[x = ([y1:A1][y2:A2]b : {y1:A1}{y2:A2}B)]
```

can be written as

```
[x [y1:A1][y2:A2] = b:B]
```

which is a significant saving.

2.1.3 Argument Synthesis

In ML we can define a polymorphic identity function

```
- fun id x = x;
```

which can be called with values of different types

```
- id 3; id "a"; id id;
```

ECC has *explicit polymorphism*, and such an identity function needs to be passed a type as well as a value:

```
[id = [A:Prop][x:A]x];
```

Now `(id id)` is not well typed (because `id` is not a member of `Prop`); we must also give the "polymorphic instantiation" to instantiate `id` at the correct type:

```
id ({A:Prop}A->A) id;  id nat three;
```

You can see this might get tiresome. However, in ECC, given a term we can compute its type (if it has one), so given an argument to `id` a typechecker can compute its type, and use this type for the missing polymorphic instantiation. I call this *argument synthesis* or *implicit arguments* [Pol90], and LEGO has syntax, using `|`, to indicate positions for term synthesis:

```
[id = [A|Prop][x:A]x];
```

Now `(id id)` and `(id three)` have their expected types. Both ELF [Pfe89] and F-sub [Car91] have similar features, although with different syntax and slightly different semantics. There is also recent work on this idea [HT94].

With `id` defined as above, `(id nat)` is not well typed (because `nat` is a proposition, not an inhabitant of a proposition), so LEGO also has syntax to override implicit application: `(id|nat)` has type `nat->nat`.

2.1.4 The `Cut` Command

I describe this command here because it is quite new, and mentioned several times in following chapters. It implements an admissible rule often called the substitution lemma

$$\frac{G \vdash a : A \qquad G[x{:}A]H \vdash b : B}{G[x{=}a{:}A]H \vdash b : B}$$

The LEGO syntax is

```
Cut [x1=a1][x2=a2]...;
```

where the `xi` are identifiers and the `ai` are terms. The command `Cut [x=a]` changes the current context from

$$G[x{:}A]H$$

to

$$G[x{=}a{:}A]H$$

if `a` has type `A` in context `G`. In fact `Cut` is more liberal than this: it is only required that `a` has type `A` in context `G` *after expanding some definitions*. For example, if the current context is

```
[A,B:Prop][a:A][b:B][c=a]
```

then `Cut [B=A][b=c]` succeeds by first expanding `c`, returning the context

```
[A:Prop][B=A][a:A][b=a][c=a]
```

We will prove the substitution lemma constructively in section 4.4.7, but LEGO's `Cut` is implemented as an atomic rule, rather than by "running the constructive proof", as the latter would be very slow.

2.1.5 Inductive Definitions

LEGO supports inductively defined types and relations [Alt93a,Dyb94,Luo94,CPM90,PM93]. I especially recommend [DFH+93] for tutorial treatment of inductive types.

The inductive types in LEGO are "coded" in the style of Martin-Löf, and the implementation owes much to work of Coquand [ACN90]. For an example, consider the inductive type of natural numbers, corresponding to the ML datatype

```
datatype NN = Z of NN | S of NN->NN;
```

An inductive type is specified by giving the names and types of each of its constructors. (We will see in a moment that more information is required to specify inductive relations.) In LEGO, the datatype `NN` and its constructors are given by

```
[NN:Prop][Z:NN][S:NN->NN];
```

The constructors are the introduction rules of the type. To give such a type its "inductiveness", that is to say that the only objects of the type are those generated by the constructors, we must also assume an elimination rule

```
[NN_elim:{C:NN->Prop}(C Z)->({x:NN}(C x)->C (S x))->{z:NN}C z];
```

which is the usual second order induction principle. The idea that we can compute the elimination rule from the types of the constructors is originally due to Gentzen.

So far we have represented these things merely by assumptions in the LEGO context, which is what I mean when I call this a coding of inductive types. The elimination rule is seen to have the same type as the primitive recursion combinator for `NN`, but `NN_elim` is just a variable, without the computational meaning of primitive recursion. For example, *predecessor* is definable primitive recursively

```
[pred : NN->NN = NN_elim ([_:NN]NN) Z [x,_:NN]x];
```

but `(pred (S Z))` is a normal form, and does not compute to `Z`. To "animate" the elimination rule as the computation of primitive recursion we extend the underlying type theory by adding new compuation rules to its conversion relation. In this case we add two contractions

```
NN_elim C fZ fS Z  ==>  fZ
NN_elim C fZ fS (S x)  ==>  fS x (NN_elim C fZ fS x)
```

whenever `(C:NN->Prop)`, `(fZ:C Z)`, `(fS:{x:NN}(C x)->C (S x))` and `(x:NN)`. Again, the reductions to be added are computed from the shape of the elimination rule, which is computed from the shapes of the constructors. This is a synthesis of ideas of Gentzen and Gödel, clarified by Martin-Löf [Mar71a].

In LEGO, one can represent an inductive type by explicitly declaring the type, its constructors and elimination rule, and stating the reductions for its computation rule (see [LP92] for syntax of these specifications). However, the elimination rule and reductions are tedious to write, and are computable from the introduction rules, so should be constructed mechanically. Furthermore, it is certainly possible to create inconsistency with arbitrary sets of constructors, elimination rules and reductions, so some discipline is required to allow only *well founded* trees, the correct inductive definitions. For these reasons LEGO supports syntax for defining inductive types by their constructors that checks the constructors against a schema of Luo [Luo94] for correctness, and, if correct, mechanically generates the elimination rule and the reductions for the computation rule. For example, the type `NN` is defined by

```
Inductive [NN:Prop] Constructors [Z:NN][S:NN->NN];
```

I implemented the basic mechanism to support inductive types in LEGO, and Claire Jones implemented the user-level syntax, the check that it meets the correctness schema, and its translation into underlying primitives. There are examples of this translation in [JP93,JP94].

A parameterized type Consider now the type of lists, defined in LEGO by

```
Inductive [LL:Prop] Parameters [A|Prop] Constructors [NIL:LL][CONS:A->LL->LL];
```

The keyword `Parameters` shows this type to be parametric in an arbitrary type, `A`. After natural deduction discharge of the parametric assumption `A`, this generates

```
[LL : Prop->Prop]
[NIL : {A|Prop}LL|A]
[CONS : {A|Prop}A->(LL|A)->LL|A]
[LL_elim : {A|Prop}{C:(LL|A)->TYPE}
             (C (NIL|A))->
             ({x:A}{y:LL|A}(C y)->C (CONS x y))->
           {z:LL|A}C z];
```

with reductions

```
LL_elim C fNIL fCONS (NIL|A)  ==>  fNIL
LL_elim C fNIL fCONS (CONS x y)  ==> fCONS x y (LL_elim C fNIL fCONS y)
```

A family of types The (parameterized) family of types "lists of length n", or vectors, is defined by

```
Inductive [vect:NN->Prop] Parameters [A:Prop]
  Constructors [vnil:vect zero]
               [vcons:{a:A}{n|NN}{v:vect n}(vect (suc n))];
```

While the `NN` and `LL` themselves have types only showing arity (respectively `Prop` and `Prop->Prop`), we see that for families such as `vect`, the type of the family is significant and must be specified.

A relation Finally, for an example of a real relation, consider the intensional equality relation defined by

```
Inductive [Q:A->A->Prop] Parameters [A|Prop] Constructors [Q_refl:{a:A}Q a a];
```

Informally we can say that `(Q A)` is defined as the smallest reflexive relation over `A`, or the intersection of all reflexive relations over `A`. The type of `Q` shows that it is a relation.

2.2 Some Basic Types

We use LEGO's built-in library of impredicative definitions (see [LP92]) for the usual logical connectives `and`, `or`, `not`, `Ex`, and their properties, although inductive definitions would do just as well. For convenience we have defined multi-ary versions of these, so you will see `and3`, `and4`, `...`, `Ex3`, `Ex4`, `...` in the following chapters.

Decidability As we are working constructively, we often use the notion of decidable proposition:

```
[decidable [P:Prop] = or P (not P)];
```

Equality We use LEGO's library of basic inductive types. These include an inductive equality relation, `Q`, which is reflexive, and substitutive, hence also symmetric and transitive:

```
Inductive [Q:A->A->Prop] Parameters [A|Prop] Constructors [Q_refl:{a:A}Q a a];

Goal Q_subst: {a,b|A}(Q a b)->{P:A->Prop}(P a)->P b;
Goal Q_sym: {a,b|A}(Q a b)->(Q b a);
Goal Q_trans: {a,b,c|A}(Q a b)->(Q b c)->(Q a c);
```

Booleans There is a type of booleans, `BB` containing `tt` and `ff`

```
Inductive [BB:Prop] ElimOver Type(0) Constructors [tt,ff:BB];
```

Notice the optional keyword `ElimOver Type(0)` in the definition of `BB`. This generates a stronger elimination rule for `BB` than we have seen in previous examples:

```
{C_BB:BB->Type(0)}(C_BB tt)->(C_BB ff)->{z:BB}C_BB z
```

instead of

```
{C_BB:BB->Prop}(C_BB tt)->(C_BB ff)->{z:BB}C_BB z
```

This is the only instance of a so-called *large elimination rule* in the entire formalization, and the extra strength is used only to prove

```
Goal tt_not_ff: not (Q tt ff);
```

which is not provable in ECC without a large elimination rule [Smi88][1].

`BB` has the usual classical boolean operators, conjunction `andd`, disjunction `orr` and conditional `if`, together with the lifting functions `is_tt` and `is_ff`, which convert booleans to (decidable) propositions:

```
[is_tt [b:BB] = Q b tt];
[is_ff [b:BB] = Q b ff];

[if [a:BB][D|Prop][d,e:D] = BB_elim ([_:BB]D) d e a];
[andd [a,b:BB] = if a b ff];
[orr [a,b:BB] = if a a b];
[nott [a:BB] = if a ff tt];

[Brec: {C:BB->Type(0)}{d:C tt}{e:C ff}{b:BB}C b = BB_elim];
```

Notice the definition `Brec` in this extract. LEGO mechanically generates an elimination rule for `BB`, called `BB_elim`, which is correct according to Luo's schema for inductive definitions [Luo94], from the `Inductive` declaration of `BB` shown above. In `BB_elim`, the variables have machine generated names, so we define `Brec`, which is a new elimination rule for `BB`, with variable names we prefer; that is, we have proved a new `BB`-elimination rule from the machine generated one. The names of variables will appear in the proof states that users see and in the proof scripts they produce, and control of such intensional aspects of the system can be quite important in a big development.

[1] We could avoid this use of a large elimination rule by assuming `tt_not_ff` instead of proving it. Stefano Berardi points out that we would not lose any computational content because under normal order reduction we will never need a proof of `tt_not_ff` unless we are actually trying to prove false.

Natural numbers An inductive type of natural numbers, `NN`, is used to support induction on the length of terms and derivations:

```
Inductive [NN:Prop] Constructors [Z:NN][S:NN->NN];
```

```
[Nrec : {C:NN->Prop}{Nbase:C Z}{Nstep:{x:NN}{Nih:C x}C (S x)}{a:NN}C a
      = NN_elim];
[add [n,m:NN] : NN = Nrec ([_:NN]NN) m ([_,x:NN]S x) n];
[pred : NN->NN = Nrec ([_:NN]NN) Z [x,_:NN]x];
[sbt [n:NN] : NN->NN = Nrec ([_:NN]NN) n ([_,x:NN]pred x)];
```

It has a double induction principle

```
Goal double_induct:
     {C:NN->NN->Prop}
     (C Z Z)->
     ({y:NN}(C Z y)->(C Z (S y)))->
     ({x:NN}({y:NN}C x y)->C (S x) Z)->
     ({x:NN}({y:NN}C x y)->{y:NN}(C (S x) y)->C (S x) (S y))->
     {x,y:NN}C x y;
```

This formulation and proof of double induction, shown to me by Stefano Berardi, is the model for a LEGO tactic, written by Claire Jones, to mechanically prove double induction for ordinary inductive definitions of one type.) We will also use `NN` variously as a set of parameters, variables, or constants, so it is useful to know `NN` has a and decidable structural equality, defined by structural recursion

```
[nat_eq : NN->NN->BB
    = Nrec ([_:NN]NN->BB)
           (Nrec ([_:NN]BB) tt ([_:NN][_:BB]ff))
           ([_:NN][eqn:NN->BB]Nrec ([_:NN]BB) ff ([x:NN][_:BB]eqn x))];
```

which agrees with intensional equality

```
Goal nat_eq_character: {m,n:NN}iff (is_tt (nat_eq m n)) (Q m n);
```

The definition of `nat_eq` is essentially double recursion, although I have shown an explicit definition by nested single recursions, and `nat_eq_character` is proved by double induction. Furthermore, `NN` is infinite.

```
Goal NNinf: {l:nats}ex[n:NN] is_ff (member nat_eq n l);
```

Lists There is a type of polymorphic lists, `LL` with its induction principle `LLrec`, and many common operations such as `append` and `member`:

```
Inductive [LL:Prop] Parameters [A|Prop] Constructors [NIL:LL][CONS:A->LL->LL];

[unit [a:A] = CONS a NIL];
[hd [a:A][l:LL] = LL_elim ([_:LL]A) a ([x:A][_:LL][_:A]x) l];
[tl [l:LL] = LL_elim ([_:LL]LL) NIL ([_:A][k,_:LL]k) l];
[append [k,l:LL] = LL_elim ([_:LL]LL) l ([a:A][_,j:LL]CONS a j) k];
[length = LL_elim ([_:LL]NN) Z ([_:A][_:LL]S)];

[exist [P:A->BB] : LL->BB =
    LL_elim ([_:LL]BB) ff ([b:A][_:LL][rest:BB]orr (P b) rest)];
[member [eq:A->A->BB][a:A] : LL->BB = exist (eq a)];
```

Two fold operations are defined, which are respectively right and left associative

```
[B|Prop][g:A->B->B];
[foldright [l:LL][strt:B] = LL_elim ([_:LL]B) strt ([a:A][_:LL][b:B]g a b) l];
[foldleft = LL_elim ([_:LL]B->B) ([b:B]b) ([a:A][_:LL][f:B->B][b:B]f (g a b))];

Goal foldright_append_lem:
     {G,H:LL}{b:B}Q (foldright (append G H) b) (foldright G (foldright H b));
Goal foldleft_append_lem:
     {G,H:LL}{b:B}Q (foldleft (append G H) b) (foldleft H (foldleft G b));
```

Remark 2.1 *Many relations, such as `member` of lists can be defined in boolean-valued form by recursion (as above), and in `Prop`-valued form by induction, e.g.*

$$\texttt{member a (CONS b l)} \quad (\textit{if } \texttt{a = b}) \qquad\qquad \frac{\texttt{member a l}}{\texttt{member a (CONS b l)}}$$

Furthermore, using large elimination rules there may be a recursively defined `Prop`-valued form as well

$$\begin{aligned} \texttt{member a NIL} &= \textit{False} \\ \texttt{member a (CONS b l)} &= \texttt{a=b} \textit{ or } \texttt{member a l} \end{aligned}$$

These issues are explained well by Christine Paulin-Mohring in [DFH$^+$93].

Cartesian product There is a type of polymorphic cartesian products, `PROD`, with pairing `Pr`, and projections `Fst` and `Snd`:

```
Inductive [PROD:Prop] Parameters [A,B|Prop] Constructors [Pr:A->B->PROD];

[Fst = PROD_elim ([_:PROD]A) ([a:A][b:B]a)];
[Snd = PROD_elim ([_:PROD]B) ([a:A][b:B]b)];
```

For our purposes of encoding an object theory, this inductively defined type `PROD` is more satisfactory than than the sigma type of ECC because it has surjective pairing

```
Goal PROD_surj_pair: {b:PROD}Q b (Pr (Fst b) (Snd b));
```

which cannot be proved for the internal sigma type.

2.2.1 An Aside on Well Founded Induction Principles

In set theory well-founded induction, based on the well-foundedness of $\in$, is taken as primary and not only structural induction, but structure itself, is (laboriously) defined. (See [Pau93b] for a formalization of this approach.) In Type Theory we take *structural* induction as primary, and derive other induction principles from it. This approach follows the ideas of Per Martin-Löf, which go back through Prawitz [Pra73,Pra74] at least as far as Gentzen. The general idea is that types are characterized by their constructors, that is their introduction rules, and that structural induction is the elimination rule for a type. For discussion of the philosophical point, and technical schemas for generating an elimination rule from the introduction rules for a type, see [CPM90,Dyb94,Dyb91,PM93,Luo94]. I will mention some nuts and bolts of using well founded induction in Type Theory.

We have the type `NN` of naturals with its structural induction principle `Nrec` from section 2.2. Define the "less than" relation by:

```
[Lt [n,m:NN] = Ex[x:NN] Q (add n (S x)) m];
```

Now we prove complete induction:

```
Goal complete_induction:
     {P:NN->Prop}{ih:{n:NN}({x:NN}(Lt x n)->(P x))->P n}{m:NN}P m;
```

using two properties of `Lt`

```
not_Lt_n_Z = ... : {n:NN}not (Lt n Z)
LtnSm_character = ... : {n,m|NN}(Lt n (S m))->or (Lt n m) (Q n m)
```

Next well founded induction for any type, over ordertype ω,

```
Goal WF_induction:
     {T|Prop}{f|T->NN}{P:T->Prop}
     {wf_ih:{t:T}({x:T}(Lt (f x) (f t))->(P x))->P t}
     {k:T}P k;
```

is proved using complete induction with the predicate

```
[n:NN]{y:T}(Q n (f y))->P y.
```

This is what is usually meant by "induction over the length of `...`", where `f:T->NN` is the "length" function. Complete induction over `NN` $\times$ `NN`, i.e. "lexicographic induction"

```
Goal complete2_induction:
     {P:NN->NN->Prop}
     {wf_ih:{n,m:NN}
        {ih:{x,y:NN}(or (Lt x n) (and (Q x n) (Lt y m)))->P x y}
           P n m}
     {n,m:NN}P n m;
```

is just nested complete induction: first use predicate `[n:NN]{m:NN}P n m` and then predicate `[m1:NN](P n1 m1)`[2]. There is none of the coding of pairs as single numbers that is necessary in Primitive Recursive Arithmetic, where the first induction predicate above, with its explicit quantification `{m:NN}`, cannot be formulated[3]. Finally, well-founded induction on any type over ordertype $\omega \times \omega$ (i.e. "induction on first the length of ..., then the length of ...")

```
Goal WF2_induction:
     {A|Prop}{f,g:A->NN}{P:A->Prop}
     {wf_ih:{n:A}
        {ih:{x:A}(or (Lt (f x) (f n))
                     (and (Q (f x) (f n)) (Lt (g x) (g n))))->P x}
           P n}
     {n:A}P n;
```

is easily proved using `complete2_induction` with the induction predicate

```
[n,m:NN]{x:A}(Q n (f x))->(Q m (g x))->P x.
```

It is also possible to define higher ordertypes, as in

```
Inductive [ord:Prop]
Constructors [ozero:ord][osucc:ord->ord][olim:(NN->ord)->ord];
```

and derive well-founded induction over this datatype directly, but I have not needed such techniques.

[2] Thanks to Claire Jones for showing me this proof

[3] Thanks to Alex Simpson and Sean Matthews who pointed this out to me.

Chapter 3

Lambda Calculus: How to Handle Free Names

In this chapter I discuss a formalization, in LEGO, of the elementary theory of the language of PTS, including substitution, beta-reduction, and beta-conversion. The major formal results in this chapter are the Church-Rosser (CR) Theorems for reduction and conversion.

Acknowledgement This chapter and much of the next are joint work with James McKinna [MP93].

3.1 Pure Languages

A Pure Language (PL) is a triple $(\mathtt{PP}, \mathtt{VV}, \mathtt{SS})$ where

- $\mathtt{PP}$ is an infinite set of *parameters*, ranged over by $\boldsymbol{p}$, $\boldsymbol{q}$. Parameters are the global, or free, variables.
- $\mathtt{VV}$ is an infinite set of *variables*, ranged over by $\boldsymbol{x}$, $\boldsymbol{v}$, $\boldsymbol{u}$. Variables are the local, or bound, variables.
- $\mathtt{SS}$, a set of *sorts*, ranged over by $\boldsymbol{s}$, $\boldsymbol{t}$ $\boldsymbol{u}$. Sorts are the constants.

Informally, the terms of a PL are given by the grammar

atoms	α	$::=$	$\boldsymbol{v}$	*variable*
		$\mid$	$\boldsymbol{p}$	*parameter*
		$\mid$	$\boldsymbol{s}$	*sort*
terms	M	$::=$	α	*atoms*
		$\mid$	$[\boldsymbol{x}{:}M]M$	*lambda*
		$\mid$	$\{\boldsymbol{x}{:}M\}M$	*pi*
		$\mid$	$M\ M$	*application*

As usual, $[\boldsymbol{x}{:}\boldsymbol{A}]\boldsymbol{B}$ and $\{\boldsymbol{x}{:}\boldsymbol{A}\}\boldsymbol{B}$, bind $\boldsymbol{x}$ in $\boldsymbol{B}$ but not in $\boldsymbol{A}$. $\boldsymbol{M}$, $\boldsymbol{N}$, $\boldsymbol{A}$, $\boldsymbol{B}$, $\boldsymbol{C}$, $\boldsymbol{D}$, $\boldsymbol{E}$, $\boldsymbol{a}$, $\boldsymbol{b}$ range over informal terms. I will always use typewriter font (e.g. `[a:A]a` or *`[a:A]a`*) for LEGO syntax, and always use math font (e.g. $[\boldsymbol{a}{:}\boldsymbol{A}]\boldsymbol{a}$) for informal terms of a PL. Sometimes I will be so informal as to drop the distinction between parameters and variables in informal terms. We informally write $\boldsymbol{A} \to \boldsymbol{B}$ and $\boldsymbol{A} \twoheadrightarrow \boldsymbol{B}$ for one-step and many-step reduction relations to be defined, and $\boldsymbol{A} \simeq \boldsymbol{B}$ for beta-conversion.

3.1.1 Pure Language Formalized

Assume there is a type of parameters, `PP`, that is infinite and has a decidable equivalence relation. In fact we also assume that the decidable equivalence relation on `PP` is the same as intensional equality; this extra assumption may not be necessary but it vastly simplifies our formal development.

```
[PP:Prop];                                               (* Parameters *)
[PPeq:PP->PP->BB];
[PPeq_iff_Q:{p,q:PP}iff (is_tt (PPeq p q)) (Q p q)];
[PPs = LL|PP];                                  (* lists of parameters *)
[PPinf:{l:PPs}ex[p:PP] is_ff (member PPeq p l)];
```

`PPinf` is a "local gensym" operation. It says that for every list of parameters, `l`, there is a parameter, `p`, that is not a member of `l`. The general list operation `member` is defined with respect to some decidable equality; in this case we use `PPeq`.

These are not mathematical principles we are assuming, but part of the presentation of a PL. Having developed some theory of PL in LEGO, we may `Discharge` these assumptions, making the whole theory parametric in such a type of parameters (see [LP92]). The assumption that `PPeq` is equivalent to `Q` means that we may instantiate `PP` with, for example the type of natural numbers and its inductively definable decidable equality, or with the type of lists of characters, but not with the type of integers defined as a quotient over pairs of naturals, because in this latter type the intended equality, definable by induction, is not equivalent to `Q`.

Also, assume a type of variables, `VV`, with similar properties, `VVeq`, `VVeq_decide`, `VVinf`; and a type of sorts, `SS`, which has decidable equality, `SSeq`, `SSeq_decide`, but need not be infinite.

3.2 Terms

The type of terms is formalized as an inductive type.

```
Inductive [Trm:Prop]
Constructors [sort:SS->Trm]
             [var:VV->Trm]
             [par:PP->Trm]
             [pi:VV->Trm->Trm->Trm]
             [lda:VV->Trm->Trm->Trm]
             [app:Trm->Trm->Trm];
```

Every term is a finitely branching well-founded tree. In particular, the `lda` and `pi` constructors do *not* have type `Trm->(VV->Trm)->Trm`, which would create well-founded but infinitely branching terms. The intended binding structure is not determined by `Trm`, but by the definitions of substitution and occurrence below.

An induction principle for `Trm`, named `Trm_elim` is mechanically generated from the introduction rules (constructors) given in the definition of `Trm`. For historical reasons I coerce the type of `Trm_elim` to have names I have chosen.

```
[Trec = Trm_elim
  : {C:Trm->Prop}
    ({s:SS}C (sort s))->
    ({n:VV}C (var n))->
    ({n:PP}C (par n))->
    ({n:VV}{A,B:Trm}{ihA:C A}{ihB:C B}C (pi n A B))->
    ({n:VV}{A,B:Trm}{ihA:C A}{ihB:C B}C (lda n A B))->
    ({A,B:Trm}{ihA:C A}{ihB:C B}C (app A B))->
      {t:Trm}C t];
```

LEGO checks that this coercion is correct.

There is a boolean-valued structural equality function, `Trm_eq`, inductively definable on terms. Because `PPeq`, `VVeq`, and `SSeq` are equivalent to `Q`, `Trm_eq` is also provably equivalent to `Q`, hence is substitutive.

```
Goal Trm_eq_subst: {A,B|Trm}(is_tt (Trm_eq A B))->Q A B;
```

3.2.1 The Length of a Term

We define the *length* of a term as a measure for well-founded induction (section 2.2.1).

```
[lngth : Trm->NN =
  Trec ([_:Trm]NN)
       ([_:SS]one)
       ([_:VV]one)
       ([_:PP]one)
       ([_:VV][_,_:Trm][l,r:NN]S (add l r))
       ([_:VV][_,_:Trm][l,r:NN]S (add l r))
       ([_,_:Trm][l,r:NN]S (add l r))];
```

Two properties of this measure are important for its applications. First, if $\boldsymbol{A}$ is a proper subterm of $\boldsymbol{B}$ then `lngth` $\boldsymbol{A}$ is less than `lngth` $\boldsymbol{B}$; this is the property used in "induction on the length of terms" as in the proof in section 3.2.5.3. Second, every term has positive length.

```
Goal lngth_is_S: {A:Trm}Ex [n:NN] Q (lngth A) (S n);
```

This is used, for example, in induction on the "sum of the lengths of the terms in a context" as in section 5.1.6. Having made the choice to give atomic terms positive length, the "`S`" is not needed in the clauses for compound terms in the definition of `lngth`, but it is convenient; without it we would have to use `lngth_is_S` very frequently, and unpack its existential quantifier each time.

3.2.2 Deciding the Shape of a Term

It is decidable whether or not a term is constructed from a sort:

```
[IsSrt [A:Trm] = Ex [s:SS]is_tt (Trm_eq A (sort s))];

Goal decide_IsSrt: {A:Trm}decidable (IsSrt A);
```

Equivalently, there is a boolean-valued function, `isSrt:Trm->BB` definable by primitive recursion. The same applies to all the other term constructors.

3.2.3 Occurrences of Parameters and Sorts

The list of parameters occurring in a term is computed by primitive recursion over term structure, and the boolean judgement whether or not a given parameter occurs in a given term is computed by the `member` function on this list of parameters.

```
[params : Trm->PPs =
  Trec ([_:Trm]PPs)
       ([_:SS]NIL|PP)
       ([_:VV]NIL|PP)
       ([p:PP]unit p)
       ([_:VV][_,_:Trm][l,r:PPs]append l r)
       ([_:VV][_,_:Trm][l,r:PPs]append l r)
       ([_,_:Trm][l,r:PPs]append l r)];

[poccur [p:PP][A:Trm] : BB = member PPeq p (params A)];
```

Similarly `sorts` and `soccur` are defined.

```
[sorts : Trm->SSs =
  Trec ([_:Trm]SSs)
       ([s:SS]unit s)
       ([_:VV]NIL|SS)
       ([_:PP]NIL|SS)
       ([_:VV][_,_:Trm][l,r:SSs]append l r)
       ([_:VV][_,_:Trm][l,r:SSs]append l r)
       ([_,_:Trm][l,r:SSs]append l r)];

[soccur [s:SS][A:Trm] : BB = member SSeq s (sorts A)];
```

3.2.4 Substitution

For the machinery on terms, we need two kinds of substitution, both defined by primitive recursion over term structure using the induction principle `Trec`. Substitution of a term for a parameter, `psub`, is entirely textual, not preventing capture. Since parameters have no binding instances in terms (we may view them as being globally bound by the context in a judgement), there is no hiding of a parameter name by a binder.

```
[psub [M:Trm][n:PP] : Trm->Trm =
  Trec ([_:Trm]Trm)
       ([s:SS]sort s)
       ([v:VV]var v)
       ([p:PP]if (PPeq n p) M (par p))
       ([v:VV][_,_,l,r:Trm]pi v l r)
       ([v:VV][_,_,l,r:Trm]lda v l r)
       ([_,_,l,r:Trm]app l r)];
```

Informally write $[N/p]M$ for `(psub N p M)`.

Substitution of a term for a variable, `vsub`, does respect variable binders that hide their bound instances from substitution, but does not prevent capture.

```
[vsub [M:Trm][n:VV] : Trm->Trm =
  Trec ([_:Trm]Trm)
       ([s:SS]sort s)
       ([v:VV]if (VVeq n v) M (var v))
       ([p:PP]par p)
       ([v:VV][_,or,nl,nr:Trm]pi v nl (if (VVeq n v) or nr))
       ([v:VV][_,or,nl,nr:Trm]lda v nl (if (VVeq n v) or nr))
       ([_,_,l,r:Trm]app l r)];
```

Informally write $[N/v]M$ for `vsub N v M`.

Both of these will be used only in safe ways in the type theory and the theory of reduction and conversion, so as to prevent unintended capture of variables.

There are abbreviations for substituting a parameter for a variable, and a variable for a parameter.

```
[alpha [p:PP] = vsub (par p)];
[alpha' [v:VV] = psub (var v)];
```

These are not alpha conversion in the usual sense (see section 3.3.4).

An important lemma, used many times, can now be proved; informally:

$$[N/p][p/v]M = [N/v]M$$

where p is some fresh parameter not occurring in M. Formally:

```
Goal vsub_is_psub_alpha:
     {N:Trm}{v:VV}{p|PP}{M|Trm}(is_ff (poccur p M))->
     is_tt (Trm_eq (psub N p (alpha p v M)) (vsub N v M));
```

Many other properties of `psub` and `vsub` are proved in the formal development.

3.2.5 No Free Occurrences of Variables

What about the variables occurring free in a term? Intuitively parameters are the free names; variables are the *bound* names, and we are not interested in free variables at all! We will define inductively a notion `Vclosed` of *variable-closed term* such that only the `Vclosed` terms are considered to be well formed, in the same sense that only typable terms will be considered well formed for the type theory in later sections. (It will turn out that every typable term, and every type, is `Vclosed`). Thus `Vclosed` is used as an induction principle over well formed terms. As this relation is a simple case of ideas that recur many times in what follows, I will discuss it at some length.

Here is an informal definition of the relation `Vclosed`

VCL-SORT $\quad$ `Vclosed(`s`)`

VCL-PAR $\quad$ `Vclosed(`p`)`

VCL-PI $$\frac{\texttt{Vclosed(}A\texttt{)} \qquad \texttt{Vclosed(}[p/v]B\texttt{)}}{\texttt{Vclosed(}\{v{:}A\}B\texttt{)}}$$

VCL-LDA $$\frac{\texttt{Vclosed(}A\texttt{)} \qquad \texttt{Vclosed(}[p/v]B\texttt{)}}{\texttt{Vclosed(}[v{:}A]B\texttt{)}}$$

VCL-APP $$\frac{\texttt{Vclosed(}A\texttt{)} \qquad \texttt{Vclosed(}B\texttt{)}}{\texttt{Vclosed(}A\,B\texttt{)}}$$

In this definition we see a central idea of our formal handling of names: how to "go under binders". Of course all terms of form `(sort s)` and `(par p)` are `Vclosed` (rules VCL-SORT and VCL-PAR), and no terms of shape `(var v)` are `Vclosed` (there is no rule to introduce `Vclosed (var v)`), but how do we define `Vclosed` for the binders? For $\{v{:}A\}B$ to be `Vclosed`, we require `Vclosed(`A`)` and `Vclosed(`$[p/v]B$`)` for some parameter p. That is, *to go under a binder, first fill the hole with some parameter*. In cases less trivial than `Vclosed` we fill the hole with a *sufficiently fresh* parameter; see, for example, section 3.3.1.

3.2.5.1 Formalizing `Vclosed`.

In formalizing this relation (table 3–1), we view the rules as constructors of an inductive relation. It is clear that for rule `Vclosed_pi` to construct a proof of `Vclosed (pi v A B)` it must actually be given a proof of `Vclosed A` and a proof of `Vclosed (alpha p n B)`. But `p` doesn't appear in the conclusion of the rule; which parameter are we to use? Does it matter which one we use? We might require only that there exists a `p` with `Vclosed (alpha p n B)` derivable (which is weaker than actually providing such a `p`), or that `Vclosed (alpha p n B)` be derivable for all `p` (which is stronger than actually providing such a `p`).

Rule `Vclosed_pi` lacks the *subformula property*, i.e. there are subformulas of the premises that do not occur in the conclusion. A familiar rule that lacks the subformula property is Modus Ponens

MP $$\frac{P \rightarrow Q \qquad P}{Q}$$

where P occurs in the both premises but not in the conclusion. For MP it does matter which P we use; in some cases $P \rightarrow Q$ and P may be provable, and in others not. Thus in formalizing

```
Inductive [Vclosed:Trm->Prop]
Constructors
[Vclosed_sort:{s:SS}Vclosed (sort s)]

[Vclosed_par:{p:PP}Vclosed (par p)]

[Vclosed_pi:{n|VV}{A,B|Trm}{p|PP}
      {premA:Vclosed A}
      {premB:Vclosed (alpha p n B)}
   (***************************************)
       Vclosed (pi n A B)]

[Vclosed_lda:{n|VV}{A,B|Trm}{p|PP}
      {premA:Vclosed A}
      {premB:Vclosed (alpha p n B)}
   (***************************************)
       Vclosed (lda n A B)]

[Vclosed_app:{A,B|Trm}
      {premA:Vclosed A}
      {premB:Vclosed B}
   (***************************************)
       Vclosed (app A B)];
```

Table 3–1: The inductive property `Vclosed`

a rule lacking the subformula property we should require a particular derivation for each premise, hence particular choices for the subformulas not occurring in the conclusion; that is what we do in rules `Vclosed_pi` and `Vclosed_lda` of table 3–1, our "official" definition of `Vclosed`.

Remark 3.1 *`Vclosed` is equivalent to having no free variables. The only proofs I know use induction on the length of terms*[1]*. This observation may be of informal interest ("the definition of `Vclosed` is reasonable"; see section 6.2), but we do not use it formally because `Vclosed` allows us to avoid all talk of free variables.*

3.2.5.2 `Vclosed` Generation Lemmas

Suppose you have a proof of `Vclosed (pi v A B)`; without examining it you know it must be constructed by `Vclosed_pi` from proofs of `Vclosed A` and `Vclosed (alpha p v B)`

[1] Thanks to Thierry Coquand for the suggestion to use length induction.

because no other rule for `Vclosed` has a conclusion of shape `Vclosed (pi v A B)`. I have been using the word *rule* for constructors of inductively defined relations; e.g. `Vclosed_pi` is a rule. The very fact that a relation is inductively defined means that its judgements can only be derived by using its rules, and the shape of a judgement usually tells us something about which rules might be used to derive it, and hence something about the immediate subderivations of any derivation. We call the lemmas that express this property *generation lemmas* (after [Bar92]), while Christine Paulin-Mohring calls them *inversion properties* [DFH+93]. As they are very useful, the generation lemmas are among the first things we prove about an inductively defined relation.

The generation lemmas we expect from the definition of `Vclosed` are (in a context `[u|VV][A,B|Trm]`)

```
(Vclosed (var u))->absurd;
(Vclosed (pi u A B))->and (Vclosed A) (Ex[p:PP] Vclosed (alpha p u B));
(Vclosed (lda u A B))->and (Vclosed A) (Ex[p:PP] Vclosed (alpha p u B));
(Vclosed (app A B))->and (Vclosed A) (Vclosed B);
```

(Notice how existential quantifiers in the lemmas for pi and lambda express the failure of the subformula property in Vclosed.) However, having mentioned that a particular parameter must be supplied in the right premise of rules `Vclosed_pi` and `Vclosed_lda`, I will now turn around and observe that in the case of `Vclosed` it intuitively seems not to matter which parameter, `p`, we use. In practice, it would be very convenient to have stronger pi- and lambda-generation lemmas for `Vclosed` expressing this observation, for example

```
(Vclosed (pi u A B))->and (Vclosed A) ({p:PP}Vclosed (alpha p u B));
```

This is directly provable, but we have something much better in store.

3.2.5.3 A Better Induction Principle for `Vclosed`.

Table 3–2 defines a relation `aVclosed` (*alternative* `Vclosed`), differing from `Vclosed` only in the rules for lambda and pi, in which the right premise requires `aVclosed (alpha p n B)` for *every* `p` (i.e. a function of type `{p:PP}aVclosed (alpha p n B)`). We will show that `Vclosed` and `aVclosed` have the same judgements.

It is worth saying that `Vclosed` can be seen as a type of well-founded finitely branching trees; i.e. `Vclosed_sort` and `Vclosed_par` are the leaves, and `Vclosed_pi`, `Vclosed_lda`, and `Vclosed_app` are binary branching nodes. On the other hand, `aVclosed` should be thought of as containing infinitely branching (but well-founded) trees, where `aVclosed_pi` and `aVclosed_lda` create a branch for each parameter `p`. Notice also that for any term, `A`, there is at most one derivation of `(aVclosed A)`, while this is certainly not the case for `Vclosed`.

```
Inductive [aVclosed:Trm->Prop]
Constructors
[aVclosed_sort:{s:SS}aVclosed (sort s)]

[aVclosed_par:{p:PP}aVclosed (par p)]

[aVclosed_pi:{n|VV}{A,B|Trm}
       {premA:aVclosed A}
       {premB:{p:PP}aVclosed (alpha p n B)}
    (***************************************)
        aVclosed (pi n A B)]

[aVclosed_lda:{n|VV}{A,B|Trm}
       {premA:aVclosed A}
       {premB:{p:PP}aVclosed (alpha p n B)}
    (***************************************)
        aVclosed (lda n A B)]

[aVclosed_app:{A,B|Trm}
       {premA:aVclosed A}
       {premB:aVclosed B}
   (***************************************)
        aVclosed (app A B)]
NoReductions;
```

Table 3–2: The inductive property `aVclosed`

Generation Lemmas for `aVclosed` By `aVclosed`-structural induction we have:

```
Goal aVclosed_var_gen : {v|VV}not (aVclosed (var v));
Goal aVclosed_pi_gen : {A,B|Trm}{v|VV}(aVclosed (pi v A B))->
                       and (aVclosed A) ({p:PP}aVclosed (alpha p v B));
Goal aVclosed_lda_gen : {A,B|Trm}{v|VV}(aVclosed (lda v A B))->
                        and (aVclosed A) ({p:PP}aVclosed (alpha p v B));
Goal aVclosed_app_gen : {A,B|Trm}(aVclosed (app A B))->
                        and (aVclosed A) (aVclosed B);
```

Equivalence of `Vclosed` and `aVclosed`. It is trivial to prove

```
Goal aVclosed_Vclosed : {A|Trm}(aVclosed A)->Vclosed A;
```

by structural induction on the derivation of `(aVclosed A)`, as we may always choose fresh parameters.

For the converse, first prove a lemma.

```
Goal aVclosed_alpha:
     {B|Trm}{p|PP}{v|VV}(aVclosed (alpha p v B))->
     {q:PP}aVclosed (alpha q v B);
```

This proof is by induction on the *length* of B, for the usual reason that statements about change of names are proved by length induction rather than structural induction, because, for example, (alpha q v A) is not generally a subterm of (alpha p v (pi n A B)) but it is shorter than (alpha p v (pi n A B)). Still, there is something missing in this explanation: it is not that we replaced structural induction over terms by length induction over terms; rather we replaced structural induction over aVclosed by length induction over terms. This works because every term appearing in a premise of a rule of aVclosed is shorter than the term appearing in its conclusion; the typing relations to be considered later do not have this property, and more subtle proofs will be required (section 4.4.4.1).

Proof of aVclosed_alpha**.** By well-founded induction on $\mathtt{lngth}(B)$, we have the goal

$$\begin{array}{l} \forall A\ .\ (\forall X\ .\ \mathtt{lngth}(X) < \mathtt{lngth}(A) \Rightarrow \\ \qquad\qquad \forall p, v\ .\ \mathtt{aVclosed}([p/v]X) \Rightarrow \forall q\ .\ \mathtt{aVclosed}([q/v]X)) \Rightarrow \\ \qquad \forall p, v\ .\ \mathtt{aVclosed}([p/v]A) \Rightarrow \forall q\ .\ \mathtt{aVclosed}([q/v]A) \end{array}$$

Now using term structural induction on A, we have six cases, for sort, variable, parameter, pi, lambda and application (only case analysis is necessary here; we don't use the structural induction hypotheses). Consider the case for pi: we must show

$$\mathtt{aVclosed}([q/v]\{n{:}A\}B) \quad \text{i.e. } \mathtt{aVclosed}(\{n{:}[q/v]A\}(\mathtt{if}\ \ (v = n)\ \ B\ \ [q/v]B))$$

under the assumptions

$$\begin{array}{lll} \text{ih} & : & \forall X\ .\ \mathtt{lngth}(X) < \mathtt{lngth}(\{n{:}A\}B) \Rightarrow \\ & & \qquad \forall p, v\ .\ \mathtt{aVclosed}([p/v]X) \Rightarrow \forall q\ .\ \mathtt{aVclosed}([q/v]X) \\ \text{vclp} & : & \mathtt{aVclosed}([p/v]\{n{:}A\}B) \\ & & \qquad \text{i.e. } \mathtt{aVclosed}(\{n{:}[p/v]A\}(\mathtt{if}\ \ (v = n)\ \ B\ \ [p/v]B)) \end{array}$$

By aVclosed generation on assumption vclp we also know

$$\begin{array}{lll} \text{h1} & : & \mathtt{aVclosed}([p/v]A) \\ \text{h2} & : & \forall r\ .\ \mathtt{aVclosed}([r/n](\mathtt{if}\ \ (v = n)\ \ B\ \ ([p/v]B))) \end{array}$$

(Here we are using a aVclosed generation lemma, in place of aVclosed structural induction, to destruct a derivation which we already know has a certain form. This is typical of how generation lemmas are used.) By aVclosed_pi, it suffices to show

$$\mathtt{aVclosed}([q/v]A) \qquad \text{and} \qquad \forall r\ .\ \mathtt{aVclosed}([r/n](\mathtt{if}\ \ (v = n)\ \ B\ \ [q/v]B))$$

Noticing that alpha (i.e. $[p/v]B$) doesn't change length, the first of these holds by ih and h1. For the second, let r be an arbitrary parameter, and consider cases. If $v = n$ then we are done by ih and h2; i.e. $[q/v]B$ doesn't actually appear in the goal, and $[p/v]B$ doesn't actually appear in h2. Finally the interesting case: if $v \neq n$ we use a straightfoward lemma

$$\forall v, w \, . \, v \neq w \Rightarrow \forall r, q, A \, . \, [r/v][q/w]A = [q/w][r/v]A$$

to rewrite the goal to

$$\texttt{aVclosed}([q/v][r/n]B)$$

By ih it suffices to show

$$\texttt{aVclosed}([p/v][r/n]B)$$

which follows by h2 after again rewriting the order of substituting p and r. ∎

Using this lemma, it is easy to prove

```
Goal Vclosed_aVclosed : {M|Trm}(Vclosed M)->aVclosed M;
```

by structural induction on the derivation of `(Vclosed A)`.

What have we gained? By defining `aVclosed` and showing it to be extensionally equivalent to `Vclosed`, we can view `aVclosed_elim`, the elimination rule for `aVclosed`, as an induction principle for the extension of `Vclosed`. This is clearly stronger than `Vclosed_elim`. Notice that we could directly prove the analogue of `aVclosed_alpha` for `Vclosed` (the same proof outlined above works), but it is not just the stronger *premises* of `aVclosed` we are after (i.e. the generation lemmas), it is the stronger *induction hypotheses*.

3.2.6 A Technical Digression: Renamings

A *renaming* is, informally, a finite function from parameters to parameters. They are represented formally by their graphs as lists of ordered pairs.

```
[rp = PROD|PP|PP];
[Renaming = LL|rp];
```

We use `rho` and `sigma` (informally, ρ, σ) to range over renamings. Renamings are applied to parameters by `assoc`, and extended compositionally to `Trm`, `GB` and `Cxt`.

```
[renPar [rho:Renaming][p:PP] : PP = assoc (PPeq p) p rho];

[renTrm [rho:Renaming] : Trm->Trm =
  Trec ([_:Trm]Trm)
       ([s:SS]sort s)
       ([v:VV]var v)
       ([p:PP]par (renPar rho p))
       ([v:VV][_,_,l,r:Trm]pi v l r)
       ([v:VV][_,_,l,r:Trm]lda v l r)
       ([_,_,l,r:Trm]app l r)];

[renGB [rho:Renaming] : GB->GB =
  GBrec ([_:GB]GB) ([p:PP][t:Trm](Gb (renPar rho p) (renTrm rho t)))];

[renCxt [rho:Renaming] : Cxt->Cxt = map (renGB rho)];
```

This is a "tricky" representation. First, if there is no pair `(Pr p q)` in `rho`, `(assoc (PPeq p) p rho)` returns `p` (from the second occurrence of `p` in this expression), so `(renPar rho)` is always a total function with finite support. Also, while there is no assumption that renamings are the graphs of functional or injective relations, the *action* of a renaming (e.g. `renTrm`) is functional, because `assoc` only finds the first pair whose domain matches a given parameter. Conversely, consing a new pair to the front of a renaming will "shadow" any old pair with the same domain. Interestingly, we do not have to formalize these observations.

3.2.6.1 The action of a Renaming

Renaming is really iterated `psub`; informally,

$$((p,q){::}\rho)M \quad = \quad [q/r](\rho([r/p]M)) \qquad\qquad \text{if } r \text{ not in } \rho \text{ or } M$$

Formally:

```
Goal renTrm_is_conjugated_psub:
     {r|PP}{M|Trm}(is_ff (poccur r M))->
     {rho|Renaming}(is_ff (member PPeq r (lefts rho)))->
                  (is_ff (member PPeq r (rights rho)))->
     {p,q:PP}Q (renTrm (CONS (Pr p q) rho) M)
              (psub (par q) r (renTrm rho (psub (par r) p M)));
```

From this analysis it is easy to show that renaming respects any relation that `psub` respects; e.g.

```
Goal psub_resp_renTrm_resp:
     {P|Trm->Trm->Prop}
     {psub_resp_P:{N|Trm}(Vclosed N)->{A,B|Trm}(P A B)->
                                        {p:PP}P (psub N p A) (psub N p B)}
     {rho:Renaming}{A,B|Trm}(P A B)->P (renTrm rho A) (renTrm rho B);
```

(By the way, this lemma is stronger, not weaker, because of the apparently ugly occurrence of Vclosed in the hypotheses; we need this strength.) Similar results hold for n-ary relations P.

3.2.6.2 Injective and Surjective Renamings

It will be useful to have bijective renamings (e.g. in section 4.4.4.1). The definitions are standard.

```
[inj [rho:Renaming] =
 {p,q|PP}(is_tt (PPeq (renPar rho p) (renPar rho q)))->is_tt (PPeq p q)];
[sur [rho:Renaming] =
 {p:PP}Ex [q:PP]is_tt (PPeq (renPar rho q) p)];
```

It's a little difficult to construct bijective renamings in general because of the trickyness of the representation mentioned above. However it's clear that any renaming that only swaps parameters, e.g. $\{q \mapsto p,\ p \mapsto q\}$, is bijective.

```
[swap [p,q:PP] : Renaming = CONS (Pr p q) (unit (Pr q p))];
Goal swap_sur : {p,q:PP}sur (swap p q);
Goal swap_inj : {p,q:PP}inj (swap p q);
```

This is enough for our purposes.

3.2.7 Do Terms Really Exist?

PP, VV and SS, are not concretely given types, but types we have assumed to exist, and to have certain properties. Although we have formalized these three types in identical ways there are clearly two different meanings involved. For one thing, we have defined, not a particular language, but a class of languages. SS can be instantiated to suit different purposes. For example, SS might be instantiated with BB for the language of the lambda cube, with the interpretation $\star =$ tt and $\Box =$ ff; or with NN for the language of ECC, with the interpretation Prop $=$ Z and Type(n) $=$ Sn. Thus SS is universally (i.e. functionally) quantified: when we Discharge SS the type Trm should be functionally dependent on the type SS and its properties SSeq and SSeq_iff_Q.

On the other hand, PP, VV are merely part of the machinery. We don't intend to support theorems of the form "If PP has property ... then ... "[2]. There is, however, a reason for using abstract declaration of PP, VV and their properties, rather than a particular instance, such as lists of characters or natural numbers: the formalization will not use any particular properties of such an instance, but only the properties we have assumed. Thus it is apparent that PP and VV should be viewed as abstract datatypes, i.e. as existentially quantified. Our meta language doesn't really have syntax for this. In ECC, a declared parameter can only be discharged as Pi-bound (i.e. universal) or Sigma-bound (i.e. strong existential). Our universal assumption of PP and VV is certainly strong enough for what we intend to say, but we demand too much of the user of the type Trm, so that user, before taking delivery of Trm, should insist on seeing some concrete instance of these types. NN is the simplest such instance, and we can get LEGO to check this fact, and instantiate PP, VV and their properties, by

```
Cut [PP=NN][PPeq=nat_eq][PPeq_iff_Q=nat_eq_character][PPinf=NNinf]
    [VV=NN][VVeq=nat_eq][VVeq_iff_Q=nat_eq_character][VVinf=NNinf];
```

(The Cut command is described in section 2.1.4.) Since this changes the context destructively, we should now re-load from the definition of PP to regain abstractness.

---

[2] [GN91,Geu93] do put structure on VV, partitioning it into disjoint countably infinite sets, one for each sort. This can be formalized as VV $\times$ SS, rather than partitioning VV, so Trm depends on SS but VV does not.

3.3 Reduction and Conversion

In this section I outline the theory of reduction and conversion of Pure Languages. The main result is the Church-Rosser Theorem.

As in the definition of `Vclosed` (section 3.2.5), the interesting point in defining reduction is how the relation goes under binders: replace the free variable occurrences by a suitably fresh parameter, operate on the closed subterm, and undo the "closing up". For an example of how this works, consider informally one-step beta-reduction of untyped lambda calculus. In our style the β and ξ rules are:

$$\beta \qquad (\lambda x.M)\,N \to [N/x]M$$

$$\xi \qquad \frac{[q/x]M \to [q/y]N}{\lambda x.M \to \lambda y.N} \qquad q \notin M,\ q \notin N$$

where $[q/x]M$ is our `(alpha q x M)`. Here is an instance of ξ, contracting the underlined redex:

$$\frac{[q/x]\underline{(\lambda v.\lambda x.v)\ x} = \underline{(\lambda v.\lambda x.v)\ q} \quad \to \quad \lambda x.q = [q/y]\lambda x.y}{\lambda x.(\underline{(\lambda v.\lambda x.v)\ x}) \quad \to \quad \lambda y.\lambda x.y}$$

After removing the outer λx, replacing its bound instances by a fresh parameter, q, and contracting the closed weak-head redex thus obtained (there is no possibility of variable capture when contracting a closed weak-head redex), we must re-bind the hole now occupied by q. According to the rule ξ, we require a variable, y, and a term, N, such that $[q/y]N = \lambda x.q$. Such a pair is y, $\lambda x.y$ (the one we have used above), as is z, $\lambda x.z$ for any $z \neq x$. However x, $\lambda x.x$ will not do, for `vsub` will not substitute q for x under the binder λx.

3.3.1 One-Step Parallel Reduction

Table 3–3 informally shows the relation `par_red1`, the *one-step parallel reduction* used in the Tait–Martin-Löf proof of the Church-Rosser property for beta-reduction. The formal definition of `par_red1` is shown in table 3–4.

Note that, since `vsub` is not a correct substitution operation, there is something fishy about the use of `vsub` in the conclusion of the beta rule, where `A'` may not be `Vclosed`. See remark 3.2.

3.3.2 Many-step parallel reduction

The transitive closure of `par_red1`, named `par_redn`, is defined.

PAR-R1-REFL	$A \rightarrow A$	
PAR-R1-BETA	$\dfrac{A \rightarrow A' \qquad [p/u]B \rightarrow [p/v]B'}{([u{:}U]B)\,A \rightarrow [A'/v]B'}$	$p \notin B,\, p \notin B'$
PAR-R1-PI	$\dfrac{A \rightarrow A' \qquad [p/u]B \rightarrow [p/v]B'}{\{u{:}A\}B \rightarrow \{v{:}A'\}B'}$	$p \notin B,\, p \notin B'$
PAR-R1-LDA	$\dfrac{A \rightarrow A' \qquad [p/u]B \rightarrow [p/v]B'}{[u{:}A]B \rightarrow [v{:}A']B'}$	$p \notin B,\, p \notin B'$
PAR-R1-APP	$\dfrac{A \rightarrow A' \qquad B \rightarrow B'}{A\,B \rightarrow A'\,B'}$	

Table 3–3: 1-Step Parallel Reduction (Informal)

```
Inductive [par_redn: Trm->Trm->Prop] Constructors
  [par_redn_red1: {A,B|Trm}(par_red1 A B)->par_redn A B]
  [par_redn_trans: {t,u,v|Trm}(par_redn t u)->(par_redn u v)->par_redn t v];
```

Recall that `par_red1` is reflexive, so `par_redn` inherits this property.

3.3.2.1 A Church-Rosser Theorem

Using the argument of Tait and Martin-Löf, as beautifully modernized in [Tak], we prove the first CR theorem.

```
[CommonReduct [R,S:Trm->Trm->Prop]
    = [b,c:Trm]Ex [d:Trm]and (S b d) (R c d)];
[DiamondProperty [R:Trm->Trm->Prop]
    = {a,b,c|Trm}(Vclosed a)->(R a b)->(R a c)->CommonReduct R R b c];
[StripLemma [R,S:Trm->Trm->Prop]
    = {a,b,c|Trm}(Vclosed a)->(R a b)->(S a c)->CommonReduct R S b c];

Goal comp_dev_par_red1_DP : DiamondProperty par_red1;
Goal par_redn_red1_SL : StripLemma par_redn par_red1;
Goal par_redn_DP : DiamondProperty par_redn;
```

The proofs are by structural induction, and there is no need define or reason about the notion of α-conversion (but see section 3.3.4). See [McK94] for a detailed description of the formalization.

```
Inductive [par_red1:Trm->Trm->Prop] Constructors

[par_red1_refl: {t:Trm}par_red1 t t]

[par_red1_beta: {U:Trm}{A,A',B,B'|Trm}{u,v|VV}{p|PP}
      {premA:par_red1 A A'}                       {noccB:is_ff (poccur p B)}
                                                {noccB':is_ff (poccur p B')}
      {premB:par_red1 (alpha p u B) (alpha p v B')}
   (**************************************************)
      par_red1 (app (lda u U B) A) (vsub A' v B')]

[par_red1_pi: {A,A',B,B'|Trm}{u,v|VV}{p|PP}
      {premA:par_red1 A A'}                       {noccB:is_ff (poccur p B)}
                                                {noccB':is_ff (poccur p B')}
      {premB:par_red1 (alpha p u B) (alpha p v B')}
   (**************************************************)
      par_red1 (pi u A B) (pi v A' B')]

[par_red1_lda: {A,A',B,B'|Trm}{u,v|VV}{p|PP}
      {premA:par_red1 A A'}                       {noccB:is_ff (poccur p B)}
                                                {noccB':is_ff (poccur p B')}
      {premB:par_red1 (alpha p u B) (alpha p v B')}
   (**************************************************)
      par_red1 (lda u A B) (lda v A' B')]

[par_red1_app: {A,A',B,B'|Trm}
      {premA:par_red1 A A'}
      {premB:par_red1 B B'}
   (**************************************************)
      par_red1 (app A B) (app A' B')];
```

Table 3–4: 1-Step Parallel Reduction

3.3.3 Conversion

One must take care in defining conversion, `conv`, or the second Church-Rosser property will fail. We have only proved the diamond property for `Vclosed` terms, and while reduction preserves `Vclosed`, expansion does not. We define `conv` so that only `Vclosed` terms can participate in `conv` (because of `conv_redn`).

```
Inductive [conv : Trm->Trm->Prop] Constructors
  [conv_redn:{A,B|Trm}(Vclosed A)->(par_redn A B)->conv A B]
  [conv_sym:{t,u|Trm}(conv t u)->conv u t]
  [conv_trans:{A,D,B|Trm}(conv A D)->(conv D B)->conv A B];
```

This relation `conv` is a proper subrelation of the conversion defined in [MP93], being (extensionally) that relation intersected with `Vclosed` $\times$ `Vclosed`.

Remark 3.2 (`Vclosed` and reduction) *The reason so many `Vclosed` hypotheses are required in the CR theorem, the definition of `conv`, and other places, is the occurrence of `vsub` in the conclusion of the rule `par_red1_beta` (table 3–4), where, if `A'` is not `Vclosed`, variable capture may occur. In retrospect, we might better have defined `par_red1` differently, replacing `par_red1_refl` in table 3–4 with*

```
[par_red1_refl: {t|Trm}(Vclosed t)->par_red1 t t]
```

With this new definition, all instances of `vsub` in table 3–4 would substitute only `Vclosed` terms, in which case `vsub` is a correct substitution operation. It would be provable that

```
{A,B|Trm}(par_red1 A B)->Vclosed B;
```

but not *that*

```
{A,B|Trm}(par_red1 A B)->Vclosed A;
```

because the type label, U, is forgotten in the conclusion of `par_red1_beta`. This alternative packaging of the `Vclosed` requirements would better capture the informal meaning, and might remove the need for `Vclosed` side conditions in many places.

A more beautiful way to get the same effect is to replace `par_red1_refl` with "atomic" reflexivity rules[3]

```
[par_red1_refl_par: {p:PP}par_red1 (par p) (par p)]
[par_red1_refl_sort: {s:SS}par_red1 (sort s) (sort s)]
```

Compare this with the atomic weakening of section 4.1.2.

3.3.3.1 The Second Church-Rosser Theorem

It is now straightfoward to prove the CR theorem for conversion.

```
Goal convCR: {A,B|Trm}(conv A B)->CommonReduct par_redn par_redn A B;
```

---

[3] This observation appears in [Tak], and was pointed out to me by James McKinna.

```
Inductive [alpha_conv : Trm->Trm->Prop] Constructors

[alpha_conv_refl:{t:Trm}alpha_conv t t]

[alpha_conv_pi:{A,A',B,B'|Trm}{u,v|VV}{p|PP}
     {premA:alpha_conv A A'}                        {noccB:is_ff (poccur p B)}
                                                   {noccB':is_ff (poccur p B')}
     {premB:alpha_conv (alpha p u B) (alpha p v B')}
   (**************************************************)
     alpha_conv (pi u A B) (pi v A' B')]

[alpha_conv_lda:{A,A',B,B'|Trm}{u,v|VV}{p|PP}
     {premA:alpha_conv A A'}                        {noccB:is_ff (poccur p B)}
                                                   {noccB':is_ff (poccur p B')}
     {premB:alpha_conv (alpha p u B) (alpha p v B')}
   (**************************************************)
     alpha_conv (lda u A B) (lda v A' B')]

[alpha_conv_app:{A,A',B,B'|Trm}
     {premA:alpha_conv A A'}
     {premB:alpha_conv B B'}
   (**************************************************)
     alpha_conv (app A B) (app A' B')];
```

Table 3–5: Alpha-Conversion

3.3.4 Alpha-Conversion

Acknowledgement Special thanks to James McKinna for his work on this section.

As an implementor I'm pathologically interested in the names of variables. One motivation for the use of distinct classes of bound variables and free parameters was to avoid the need to reason about alpha-conversion or alpha-equivalence classes. I have described formal proofs of both Church-Rosser theorems without using the notion of alpha-conversion. In Chapters 4 and 5, and in [MP93], we apply this theory of reduction and conversion to PTS, proving significant type theory results without any mention of alpha-conversion. However, I will argue in the following two subsections, 3.3.5 and 3.3.6, that this point of view is misleading. It is only possible to pursue type theory without using alpha-conversion because our type theory uses reduction and conversion relations, `par_redn` and `conv`, that contain alpha-conversion. Further, decidability of conversion for normalizing terms depends on decidability of alpha-conversion.

Table 3–5 shows a definition of `alpha_conv` due to James McKinna. We informally write $\stackrel{\alpha}{\simeq}$

for `alpha_conv`. Note that `alpha_conv` is exactly `par_red1` without the rule `par_red1_beta`, so `par_red1` contains `alpha_conv`.

```
Goal alpha_conv_par_red1 : {A,B|Trm}(alpha_conv A B)->par_red1 A B;
```

Remark 3.3 *`par_red1` contains `alpha_conv`, but does not respect alpha-conversion classes. For example*

$$([x{:}q]x\;x)\,([w{:}q]w)\;\textit{par\_red1}\;([y{:}q]y)\,([y{:}q]y) \qquad \textit{for every } y,$$

but

$$\textit{not}\quad ([x{:}q]x\;x)\,([w{:}q]w)\;\textit{par\_red1}\;([y_1{:}q]y_1)\,([y_2{:}q]y_2) \qquad \textit{for } y_1 \neq y_2.$$

`par_redn`, being transitive, does respect alpha-conversion classes.

The properties of `alpha_conv` include:

```
Goal alpha_conv_refl_pocc:
    {X,Y|Trm}(alpha_conv X Y)->{p|PP}(is_tt (poccur p Y))->is_tt (poccur p X);
Goal Vclosed_resp_alpha_conv:
     {A,B|Trm}(alpha_conv A B)->(Vclosed A)->Vclosed B;
Goal alpha_conv_sym : sym alpha_conv;
Goal alpha_conv_trans : trans alpha_conv;
```

Informally, alpha-conversion is used for changing the names of variables. We do not have $\{x{:}A\}B \overset{\alpha}{\simeq} \{y{:}A\}([y/x]B)$ because $[y/x]B$ (i.e. `(vsub (var y) x B)`) is not necessarily correct as an informal substitution. However, we do have:

```
Goal true_alpha_conv_pi:
     {v|VV}{A,B|Trm}(Vclosed (pi v A B))->
     {u:VV} Ex [C:Trm] alpha_conv (pi v A B) (pi u A C);
Goal true_alpha_conv_lda:
     {v|VV}{A,B|Trm}(Vclosed (lda v A B))->
     {u:VV} Ex [C:Trm] alpha_conv (lda v A B) (lda u A C);
```

3.3.4.1 Deciding Alpha-Conversion

Alpha-conversion is decidable for `Vclosed` terms:

```
Goal decide_alpha_conv:
     {A,B|Trm}(Vclosed A)->(Vclosed B)->decidable (alpha_conv A B);
```

This is a straightfoward but messy proof, by double induction on `(aVclosed A)` and `(aVclosed B)`.

BN-SORT	$\mathsf{beta\_norm}\,(\boldsymbol{s})$	
BN-PAR	$\mathsf{beta\_norm}\,(\boldsymbol{p})$	
BN-PI	$\dfrac{\mathsf{beta\_norm}\,(\boldsymbol{A}) \qquad \forall \boldsymbol{p}\ .\ \mathsf{beta\_norm}\,([\boldsymbol{p}/\boldsymbol{u}]\boldsymbol{B})}{\mathsf{beta\_norm}\,(\{\boldsymbol{u}{:}\boldsymbol{A}\}\boldsymbol{B})}$	
BN-LDA	$\dfrac{\mathsf{beta\_norm}\,(\boldsymbol{A}) \qquad \forall \boldsymbol{p}\ .\ \mathsf{beta\_norm}\,([\boldsymbol{p}/\boldsymbol{u}]\boldsymbol{B})}{\mathsf{beta\_norm}\,([\boldsymbol{u}{:}\boldsymbol{A}]\boldsymbol{B})}$	
BN-APP	$\dfrac{\mathsf{beta\_norm}\,(\boldsymbol{A}) \qquad \mathsf{beta\_norm}\,(\boldsymbol{B})}{\mathsf{beta\_norm}\,(\boldsymbol{A}\ \boldsymbol{B})}$	$\boldsymbol{A}$ is not a lambda

Table 3–6: Inductive definition of beta-normal forms (informal).

```
Inductive [beta_norm:Trm->Prop]  NoReductions  Constructors
[bn_sort:{s:SS}beta_norm (sort s)]

[bn_par:{p:PP}beta_norm (par p)]

[bn_pi:{A,B|Trm}{v|VV}
      {lprem:beta_norm A}
      {rprem:{p:PP}beta_norm (alpha p v B)}
    (*************************************)
          beta_norm (pi v A B)]

[bn_lda:{A,B|Trm}{v|VV}
      {lprem:beta_norm A}
      {rprem:{p:PP}beta_norm (alpha p v B)}
    (*************************************)
          beta_norm (lda v A B)]

[bn_app:{A,B|Trm}{sc:is_ff (isLda A)}
      {lprem:beta_norm A}
      {rprem:beta_norm B}
    (*************************************)
          beta_norm (app A B)];
```

Table 3–7: Inductive definition of beta-normal forms (formal).

3.3.5 Normal Forms

A term is *beta normal*, `beta_norm`, if it has no beta redexes (tables 3–6 and 3–7). All `beta_norm` terms are `Vclosed`:

```
Goal beta_norm_Vclosed: {A|Trm}(beta_norm A)->Vclosed A;
```

A relation, *normal form*, and a property, *normalizing*, are defined:

```
[normal_form [N,A:Trm] = and (beta_norm N) (par_redn A N)];
[normalizing [A:Trm] = Ex [B:Trm] normal_form B A];
```

Our many-step reduction, `par_redn`, is reflexive, so there is reduction from a normal form, but every reduct of a normal form is a normal form.

```
Goal par_redn_bnorm_is_bnorm:
     {A,B|Trm}(par_redn A B)->(beta_norm A)->beta_norm B;
```

Any reduct of a normal form alpha-converts with that normal form.

```
Goal par_redn_bnorm_is_alpha_conv:
     {A,B|Trm}(par_redn A B)->(beta_norm A)->alpha_conv A B;
```

Hence, by Church-Rosser, normal forms of a term are unique up to alpha-conversion.

```
Goal nf_unique:
     {A|Trm}(Vclosed A)->
     {M,N|Trm}(normal_form M A)->(normal_form N A)->alpha_conv M N;
```

This lemma says that the class of normal forms of a (`Vclosed`) term is contained in an alpha-conversion equivalence class. Since alpha-conversion is contained in `par_red1`, if a term has a normal form it has an an entire alpha-conversion equivalence class of normal forms.

```
Goal nf_alpha_class:
     {A|Trm}(Vclosed A)->
     {M,N|Trm}(normal_form M A)->(alpha_conv M N)->normal_form N A;
```

Thus the class of normal forms of a (`Vclosed`) term is either empty or exactly an alpha-conversion equivalence class.

3.3.5.1 Deciding the Shape of Normal Forms

The normal forms of normalizing terms are computable, so it is decidable what their shape is (see section 3.2.2). For example, define what it means to *reduce to a sort* or *reduce to a pi*

```
[RedToSort [A:Trm] = Ex [s:SS] par_redn A (sort s)];
[RedToPi [A:Trm] = Ex3 [X,Y:Trm][v:VV] par_redn A (pi v X Y)];
```

and these properties are decidable.

```
Goal normalizing_decides_RedToSort:
     {A|Trm}(Vclosed A)->(normalizing A)->decidable (RedToSort A);
Goal normalizing_decides_RedToPi:
     {A|Trm}(Vclosed A)->(normalizing A)->decidable (RedToPi A);
```

These lemmas are used for typechecking in Chapter 5.

3.3.5.2 Deciding Conversion

Conversion is decidable for normalizing terms.

```
Goal normalizing_decides_conv:
     {A,B|Trm}(Vclosed A)->(normalizing A)->(Vclosed B)->(normalizing B)->
     decidable (conv A B);
```

This proof depends on Church-Rosser; since normal forms are unique only up to alpha-conversion, it also depends on decidability of alpha-conversion (section 3.3.4.1).

3.3.6 Ordinary Beta-Reduction: Church-Rosser Theorems Revisited

We have proved the first and second CR theorems for `par_redn` and for `conv` (which is defined using `par_redn`). The usual argument now is to show that `par_redn` is (extensionally) the same relation as ordinary many-step beta-reduction, hence the CR theorem holds for ordinary beta-reduction (e.g. see p. 62 in [Bar84]).

Ordinary one-step beta-reduction, `red1`, is defined informally in table 3–8 and formally in table 3–9; `Redn` is the reflexive-transitive closure of `red1`.

```
Inductive [Redn:Trm->Trm->Prop] Constructors
   [Redn_red1:{A,B|Trm}(red1 A B)->Redn A B]
   [Redn_refl:{t:Trm}Redn t t]
   [Redn_trans:{t,u,v|Trm}(Redn t u)->(Redn u v)->Redn t v];
```

We observed above that `alpha_conv` is contained in `par_red1`, hence `alpha_conv` is contained in `par_redn`. On the other hand, `red1` must actually contract a beta-redex, so surely does not contain `alpha_conv`. Thus `Redn` also does not contain `alpha_conv`, and

```
Goal not {M,N|Trm}(par_redn M N)->Redn M N;
```

is provable.

In fact `(DiamondProperty Redn)` is not provable. For example

$$[\boldsymbol{x}{:}\boldsymbol{q}]\,(\,[\boldsymbol{x}{:}\boldsymbol{q}]\boldsymbol{x}\,)\,\boldsymbol{x}\ \texttt{red1}\ [\boldsymbol{w}{:}\boldsymbol{q}]\boldsymbol{w} \qquad \text{for every } \boldsymbol{w}, \tag{$*$}$$

RED1-BETA	$([u{:}U]B)\,A \to [A/u]B$	
RED1-PI-L	$\dfrac{A \to A'}{\{u{:}A\}B \to \{v{:}A'\}B}$	
RED1-PI-R	$\dfrac{[p/u]B \to [p/v]B'}{\{u{:}A\}B \to \{v{:}A\}B'}$	$p \notin B,\, p \notin B'$
RED1-LDA-L	$\dfrac{A \to A'}{[u{:}A]B \to [v{:}A']B}$	
RED1-LDA-R	$\dfrac{[p/u]B \to [p/v]B'}{[u{:}A]B \to [v{:}A]B'}$	$p \notin B,\, p \notin B'$
RED1-APP-L	$\dfrac{A \to A'}{A\,B \to A'\,B}$	
RED1-APP-R	$\dfrac{B \to B'}{A\,B \to A\,B'}$	

Table 3–8: One-step beta-reduction (Informal)

and $[w{:}q]w$ has no `red1`-reducts at all. Choosing $w_1 \neq w_2$, we have

$$[x{:}q]([x{:}q]x)\,x\ \texttt{red1}\ [w_1{:}q]w_1 \quad \text{and} \quad [x{:}q]([x{:}q]x)\,x\ \texttt{red1}\ [w_2{:}q]w_2$$

where $[w_1{:}q]w_1$ and $[w_2{:}q]w_2$ have no common `Redn`-reduct. James McKinna claims that the correct CR theorem for `Redn` is

```
{A,Bl,Br|Trm}(Vclosed A)->(Redn A Bl)->(Redn A Br)->
    Ex2 [Cl,Cr:Trm] and3 (Redn Bl Cl) (Redn Br Cr) (alpha_conv Cl Cr);
```

Another possible solution is to change the definition of `red1` or `Redn` to contain `alpha_conv`. Then it would be provable that `par_redn` and `Redn` are the same relation, thus proving the CR theorem for ordinary beta-reduction. The choice between these two approaches is an informal question: does the informal notion of reduction contain alpha-conversion or not?

Remark 3.4 *The notion* `beta_norm` *of beta-normal form can be used to define*

```
Inductive [red1 : Trm->Trm->Prop] Constructors

[red1_beta:{u:VV}{U,A,B:Trm}red1 (app (lda u U B) A) (vsub A u B)]

[red1_pi_l:{A,A'|Trm}{premA:red1 A A'}
         (**********************************************)
           {v:VV}{B:Trm}red1 (pi v A B) (pi v A' B)]

[red1_pi_r:{u,v|VV}{B,B'|Trm}{p|PP}               {noccB:is_ff (poccur p B)}
                                                {noccB':is_ff (poccur p B')}
           {premB:red1 (alpha p u B) (alpha p v B')}
         (**********************************************)
           {A:Trm}red1 (pi u A B) (pi v A B')]

[red1_lda_l:{A,A'|Trm}{premA:red1 A A'}
         (**********************************************)
           {v:VV}{B:Trm}red1 (lda v A B) (lda v A' B)]

[red1_lda_r:{u,v|VV}{B,B'|Trm}{p|PP}               {noccB:is_ff (poccur p B)}
                                                 {noccB':is_ff (poccur p B')}
           {premB:red1 (alpha p u B) (alpha p v B')}
         (**********************************************)
           {A:Trm}red1 (lda u A B) (lda v A B')]

[red1_app_l:{A,A'|Trm}{premA:red1 A A'}
         (**********************************************)
          {B:Trm}red1 (app A B) (app A' B)]

[red1_app_r:{B,B'|Trm}{premB:red1 B B'}
         (**********************************************)
           {A:Trm}red1 (app A B) (app A B')];
```

Table 3–9: One-step beta-reduction

$$Redn\_normal\_form\ N\ A \quad = \quad (beta\_norm\ N)\ \text{and}\ (Redn\ A\ N)$$

Unlike the case for `par_redn`, there are no `Redn` steps starting from a normal form, so the class of `Redn_normal_forms` of a normalizing term may not be a complete `alpha_conversion` class. On the other hand such a class may not be a singleton (see equation $(*)$ above) because the rules `red1_pi_r` and `red1_lda_r` allow non-deterministic choice of the variable $\boldsymbol{v}$ that gets re-bound after the contraction. Thus deciding `Redn`-conversion for normalizing terms requires decidability of alpha-conversion.

Chapter 4

Pure Type Systems

A PTS is a PL ($\mathtt{PP}, \mathtt{VV}, \mathtt{SS}$) along with two relations

- `[ax:SS->SS->Prop]`, a set of *axioms*, written informally as $\mathtt{ax}(s_1{:}s_2)$
- `[rl:SS->SS->SS->Prop]`, a set of *rules*, written informally as $\mathtt{rl}(s_1, s_2, s_3)$

that parameterize an inductively defined typing judgement. We usually intend `ax` and `rl` to be decidable, but this assumption is not used in the basic theory of PTS. If we are interested in algorithms for typechecking (see Chapter 5), even stronger assumptions about decidability are needed.

4.1 What are Pure Type Systems

The typing judgement has the shape $\Gamma \vdash M : A$, meaning that in context Γ, term M has type A. Informally we call M (or (G, M)) the *subject* and A the *predicate* of the judgement. Also we informally write $\bullet$ for the empty context. In original papers such as [Ber90b,Bar91, GN91,vBJ93] the typing judgement of a PTS is given as the inductive closure of (something like) the rules in table 4–1. (In this section I am not distinguishing between parameters and variables; I will use x, $y, \ldots$, and use the word "variable".)

4.1.1 Pi-Formation

In 1990 I suggested to Henk Barendregt that the rule LDA of table 4–1 should be replaced with

$$\text{LDA} \qquad \frac{\Gamma[x{:}A] \vdash M : B \qquad \Gamma \vdash \{x{:}A\}B : s}{\Gamma \vdash [x{:}A]M : \{x{:}A\}B}$$

This formulation shows that formation of Pi-types is only allowed by the rule PI, not independently by the LDA rule; it is the presentation used in [Bar92], the "bible" of PTS. It's

AX	$\bullet \vdash s_1 : s_2$	$\mathtt{ax}(s_1{:}s_2)$
START	$\dfrac{\Gamma \vdash A : s}{\Gamma[x{:}A] \vdash x : A}$	$x \notin \Gamma$
WEAK	$\dfrac{\Gamma \vdash M : C \qquad \Gamma \vdash A : s}{\Gamma[x{:}A] \vdash M : C}$	$x \notin \Gamma$
PI	$\dfrac{\Gamma \vdash A : s_1 \qquad \Gamma[x{:}A] \vdash B : s_2}{\Gamma \vdash \{x{:}A\}B : s_3}$	$\mathtt{rl}(s_1, s_2, s_3)$
LDA	$\dfrac{\Gamma \vdash A : s_1 \qquad \Gamma[x{:}A] \vdash M : B \qquad \Gamma[x{:}A] \vdash B : s_2}{\Gamma \vdash [x{:}A]M : \{x{:}A\}B}$	$\mathtt{rl}(s_1, s_2, s_3)$
APP	$\dfrac{\Gamma \vdash M : \{x{:}A\}B \qquad \Gamma \vdash N : A}{\Gamma \vdash M\,N : [N/x]B}$	
tCONV	$\dfrac{\Gamma \vdash M : A \qquad \Gamma \vdash B : s}{\Gamma \vdash M : B}$	$A \simeq B$

Table 4–1: An old, informal, formulation of PTS

easy to see that the two formulations have the same judgements (they are extensionally the same relation), but don't have the same derivations of these judgements (they are intensionally different relations), so the elementary theory is slightly different in the two formulations.

4.1.2 Atomic Weakening

I recently suggested that the weakening rule, WEAK, should be restricted to atomic subjects, i.e. replaced by the two rules

vWEAK	$\dfrac{\Gamma \vdash y : C \qquad \Gamma \vdash A : s}{\Gamma[x{:}A] \vdash y : C}$	$x \notin \Gamma$
sWEAK	$\dfrac{\Gamma \vdash s : C \qquad \Gamma \vdash A : s}{\Gamma[x{:}A] \vdash s : C}$	$x \notin \Gamma$

Whereas WEAK can be used to derive any judgement with a non-empty context, these rules can only be used to derive a judgement with a variable or a sort, respectively, for its subject. Further, given a judgement, $\Gamma[y : A] \vdash x : B$, with a variable as its subject, there is no confusion whether it is derived by START or VWEAK: use START if $x = y$ and VWEAK otherwise. Similarly, given a judgement, $\Gamma \vdash s : B$, with a sort as its subject, there is no confusion whether it is derived by AX or SWEAK: use AX if Γ is empty and SWEAK otherwise. Thus, with atomic weakening, any judgement may only be derived by TCONV or by exactly one of the remaining rules.

4.1.2.1 Weakening and the Shape of Derivations

Given any derivation with atomic weakening, just replace every instance of VWEAK or SWEAK with an instance of WEAK to obtain a derivation of the same judgement in the system with WEAK. However, using WEAK there are more derivations of most of these judgements. For example, the following derivation works in both systems

$$\dfrac{\begin{array}{c}\vdots\\ \Gamma \vdash M : A \to B\\ \vdots\ \text{weakening } \Delta\\ \Gamma\Delta \vdash M : A \to B\end{array} \qquad \begin{array}{c}\vdots\\ \Gamma \vdash N : A\\ \vdots\ \text{weakening } \Delta\\ \Gamma\Delta \vdash N : A\end{array}}{\Gamma\Delta \vdash M\,N : B}\ \textsc{App}$$

while the next one only works in the system with WEAK.

$$\begin{array}{c}\dfrac{\begin{array}{c}\vdots\\ \Gamma \vdash M : A \to B\end{array} \qquad \begin{array}{c}\vdots\\ \Gamma \vdash N : A\end{array}}{\Gamma \vdash M\,N : B}\ \textsc{App}\\ \vdots\ \text{weakening } \Delta\\ \Gamma\Delta \vdash M\,N : B\end{array}$$

With atomic weakening all uses of weakening have been pushed as close as possible to the leaves. Consequently, the system with atomic weakening is closer in shape of derivations to the other standard presentation of type systems, where validity of contexts is defined by mutual induction with the typing judgement (the relations `vtyp` and `vcxt` in section 4.4.10.1). In fact, by restricting the derivations we get a more beautiful development of the basic theory, and this will be presented below.

From the observations above, the question whether the two systems derive the same judgements reduces to whether WEAK is an admissible rule in the system with atomic weakening. I show this to be the case in section 4.4.6.

AX	$\bullet \vdash s_1 : s_2$	$\mathtt{ax}(s_1{:}s_2)$
START	$\dfrac{\Gamma \vdash A : s}{\Gamma[p{:}A] \vdash p : A}$	$p \notin \Gamma$
vWEAK	$\dfrac{\Gamma \vdash q : C \qquad \Gamma \vdash A : s}{\Gamma[p{:}A] \vdash q : C}$	$p \notin \Gamma$
sWEAK	$\dfrac{\Gamma \vdash s : C \qquad \Gamma \vdash A : s}{\Gamma[p{:}A] \vdash s : C}$	$p \notin \Gamma$
PI	$\dfrac{\Gamma \vdash A : s_1 \qquad \Gamma[p{:}A] \vdash [p/x]B : s_2}{\Gamma \vdash \{x{:}A\}B : s_3}$	$p \notin B,\ \mathtt{rl}(s_1, s_2, s_3)$
LDA	$\dfrac{\Gamma[p{:}A] \vdash [p/x]M : [p/y]B \qquad \Gamma \vdash \{y{:}A\}B : s}{\Gamma \vdash [x{:}A]M : \{y{:}A\}B}$	$p \notin M,\ p \notin B$
APP	$\dfrac{\Gamma \vdash M : \{x{:}A\}B \qquad \Gamma \vdash N : A}{\Gamma \vdash M\,N : [N/x]B}$	
tCONV	$\dfrac{\Gamma \vdash M : A \qquad \Gamma \vdash B : s}{\Gamma \vdash M : B}$	$A \simeq B$

Table 4–2: The Informal Typing Rules of PTS

4.1.3 Parameters and Variables

Table 4–2 is a formulation of the system we have been discussing where the distinction between parameters and variables is made. First observe that the handling of parameters and variables in the PI and LDA rules is similar to that in the rules of table 3–4: to operate under a binder, locally extend the context replacing the newly freed variable by a new parameter, do the operation, then forget the parameter and bind the variable again. Huet's Constructive Engine [Hue89,Pol94], "weaves" back and forth between named global variables and local de Bruijn indices in a manner similar to that of table 4–2.

All instances of $[M/v]N$ in table 4–2 have M `Vclosed` (compare with remark 3.2).

4.1.3.1 A Strange Lambda Rule

The LDA rule allows "alpha converting" in the conclusion. Informal presentations [Bar92, Ber90b,GN91] suggest the rule

$$\text{LDA}' \qquad \frac{\Gamma[p{:}A] \vdash [p/x]M : [p/x]B \qquad \Gamma \vdash \{x{:}A\}B : s}{\Gamma \vdash [x{:}A]M : \{x{:}A\}B} \qquad p \notin M,\ p \notin B$$

To see why I use the rule LDA instead of LDA′, consider an example in the Pure Calculus of Constructions.

Example 4.1 *Let Γ be the context $[A{:}\star][P{:}A{\to}\star]$. Using* LDA *we have the natural derivation*

$$\cfrac{\cfrac{\begin{matrix}\vdots\\ \Gamma[p{:}A][q{:}Pp] \vdash q : Pp\end{matrix} \quad \vdots}{\Gamma[p{:}A] \vdash [x{:}Pp]x : \{y{:}Pp\}Pp}\,\text{LDA} \qquad \vdots}{\Gamma \vdash [x{:}A][x{:}Px]x : \{x{:}A\}\{y{:}Px\}Px}\,\text{LDA}$$

With LDA′ *we must use* TCONV *to alpha-convert in order to derive this judgement, for example*

$$\cfrac{\cfrac{\cfrac{\begin{matrix}\vdots\\ \Gamma[p{:}A][q{:}Pp] \vdash q : Pp\end{matrix} \quad \vdots}{\Gamma[p{:}A] \vdash [x{:}Pp]x : \{x{:}Pp\}Pp}\,\text{LDA}' \qquad \vdots}{\Gamma[p{:}A] \vdash [x{:}Pp]x : \{y{:}Pp\}Pp}\,\text{TCONV} \qquad \vdots}{\Gamma \vdash [x{:}A][x{:}Px]x : \{x{:}A\}\{y{:}Px\}Px}\,\text{LDA}'$$

The judgement $\Gamma \vdash [x{:}A][x{:}Px]x : \{x{:}A\}\{x{:}Px\}Px$ is not derivable with either rule, as it should not be.

The two systems derive the same judgements (see section 4.4.9.6), but I feel the system we informally have in mind does not use instances of TCONV solely for alpha-conversion. I will ask whether we have really eliminated the need to do this in section 4.4.9.7.

4.1.4 A Generalization: Abstract Conversion

The side condition of rule TCONV in table 4–2 mentions beta-conversion, the relation formally called `conv` (section 3.3.3) and informally written $A \simeq B$. In fact, very few properties of $\simeq$ are used in the theory of PTS until we come to prove the Subject Reduction Theorem (where we finally see that $\simeq$ must be related to contraction of redexes as constructed by the rule TLDA), and even that proof is far from determining what relation $\simeq$ must be.

There are several reasons to be interested in exactly what properties $\simeq$ must have. For one thing, the type theory ECC (see section 5.3), implemented in LEGO, is not actually a PTS because it uses a generalized notion of conversion called *cumulativity*. ECC is of special interest to us, so we want to discuss an extension of PTS which includes ECC. ECC's cumulativity relation elegantly solves the difficulties in typechecking cumulative hierarchies (discussed in [HP91]). (Typechecking for ECC will be discussed in chapter 5.)

Even for PTS, there is a notorious open problem, the *Expansion Postponement* problem [vBJMP94] (EP), which asks if the conversion relation in table 4–2 can be replaced by beta-reduction without changing the typability of any terms. (EP will be briefly discussed in section 4.7.)

In the formalization I use an arbitrary relation, `cnv` (informally written $A \leq B$), in place of `conv`, and keep track of the properties that $\leq$ must be assumed to have in order to prove the basic results about PTS, starting off with only three: reflexivity, transitivity, and invariance under `psub` (see section 4.3 for the exact properties).

Among the properties of $\simeq$ that do not generally hold for $\leq$ are symmetry (which is why I use an asymmetric symbol for $\leq$), $s_1 \simeq s_2 \ \Rightarrow\ s_1 = s_2$, $A \simeq s \ \Rightarrow\ A \twoheadrightarrow s$ and $A \twoheadrightarrow s \ \Rightarrow\ s \in A$.

4.2 Contexts Formalized

The type of global bindings, $[p{:}A]$, that occur in contexts, is a cartesian product of `PP` by `Trm`. We also give the two projections of global bindings, and a defined structural equality that is provably equivalent to `Q`.

```
[GB : Prop = PROD|PP|Trm];
[Gb : PP->Trm->GB = Pr|PP|Trm];
[namOf : GB->PP = Fst|PP|Trm];
[typOf : GB->Trm = Snd|PP|Trm];
[GBeq [b,c:GB] = andd (PPeq (namOf b) (namOf c))
                      (Trm_eq (typOf b) (typOf c))];
```

Contexts are lists of global bindings.

```
[Cxt = LL|GB];
[nilCxt = NIL|GB];
```

Notice that informally we are writing contexts as being extended ("consing") on the right, while formally they cons on the left.

In a context $\Gamma_1[p{:}A]\Gamma_2$ we informally say that the distinguished occurrence of p *binds* occurrences of p in Γ_2 (but not in Γ_1 or A).

4.2.1 Occurrences

Occurrence of a global binding in a context is defined using the `member` function and the defined structural equality `GBeq`.

```
[GBoccur [b:GB][G:Cxt] : BB = member GBeq b G];
```

Now we define the list of parameters bound by a context, and, using the `member` function as before, the boolean relation deciding whether or not a given parameter is bound by a given context.

```
[globalNames : Cxt->PPs = map namOf];
[Poccur [p:PP][G:Cxt] : BB = member PPeq p (globalNames G)];
```

4.2.2 Subcontexts

The subcontext relation is defined by containment of the respective occurrence predicates.

```
[subCxt [G,H:Cxt] = {b:GB}(is_tt (GBoccur b G))->is_tt (GBoccur b H)];
```

We say, informally, G is a *subcontext* of H, or H *extends* G. This is exactly the definition used informally in [Bar92,GN91,vBJ93]. It would require a very different, and more complicated definition to express a similar property in a representation using de Bruijn indices for global variables.

4.3 The Typing Judgement Formalized

We begin with arbitrary relations `ax` and `rl`

```
[ax:SS->SS->Prop]
[rl:SS->SS->SS->Prop];
```

and the abstract conversion relation discussed in section 4.1.4.

```
[cnv:Trm->Trm->Prop];
```

The three properties of `cnv`

```
[cnv_refl:{A|Trm}(Vclosed A)->cnv A A];
[cnv_trans:{A,D,B|Trm}(cnv A D)->(cnv D B)->cnv A B];
[psub_resp_cnv:{N|Trm}(Vclosed N)->{A,B|Trm}(cnv A B)->
       {p:PP}cnv (psub N p A) (psub N p B)];
```

are all that will be needed until we come to the Subject Reduction theorem in section 4.4.9. Recall from section 3.2.6.1 that the third of these assumptions allows us to prove

```
Goal renTrm_resp_cnv:
     {A,B|Trm}(cnv A B)->{rho:Renaming}cnv (renTrm rho A) (renTrm rho B);
```

which is used without further comment.

The typing rules, formalized as the constructors of an inductive relation, `gts`, are shown in table 4–3. The discussion in section 3.2.5 may be of use in understanding why we claim `gts` is the correct formalization of table 4–2.

Validity. The notion of `Valid` context is defined as participation in any derivable judgement.

```
[Valid [G:Cxt] = Ex2 [M,A:Trm] gts G M A];
```

The tail of a `Valid` context is `Valid`.

```
Goal {G|Cxt}(Valid G)->Valid (tl G);
```

No free variables. All well-typed terms are `Vclosed`.

```
Goal gts_Vclosed_lem:{G|Cxt}{M,A|Trm}(gts G M A)->and (Vclosed M) (Vclosed A);
```

4.4 Properties of Arbitrary PTS With Abstract Conversion

This section follows the general outline of the similarly titled section of [Bar92], although I include more material, and some details of the formalization. The idea is to present the elementary theory of PTS using the atomic weakening rules (section 4.1.2).

4.4.1 Parameter Lemmas

All parameter occurrences in a judgement are bound.

```
Goal free_params_lem1:
   {G|Cxt}{M,A|Trm}(gts G M A)->
   {p|PP}(is_ff (Poccur p G))->and (is_ff (poccur p M)) (is_ff (poccur p A));
Goal cxt_free_params_lem:
     {H|Cxt}{M,A,B|Trm}{p|PP}(gts (CONS (Gb p B) H) M A)->
     {q|PP}(is_ff (Poccur q H))->is_ff (poccur q B);
```

The binding instances in a `Valid` context are all distinct parameters, and terms in a `Valid` context can only mention parameters that are already bound.

```
Goal CxtCorrect0:
 {H|Cxt}{M,A,B|Trm}{p|PP}(gts (CONS (Gb p B) H) M A)->is_ff (Poccur p H);
Goal CxtCorrect1:
 {H|Cxt}{M,A,B|Trm}{p|PP}(gts (CONS (Gb p B) H) M A)->is_ff (poccur p B);
```

```
Inductive [gts:Cxt->Trm->Trm->Prop] Constructors

[Ax:{s1,s2|SS}                                          {sc:ax s1 s2}
        gts nilCxt (sort s1) (sort s2)]

[Start:{G|Cxt}{A|Trm}{s|SS}{p|PP}             {noccG:is_ff (Poccur p G)}
      {prem:gts G A (sort s)}
   (*****************************************)
        gts (CONS (Gb p A) G) (par p) A]

[vWeak:{G|Cxt}{D,A|Trm}{s|SS}{n,p|PP}          {noccG:is_ff (Poccur p G)}
      {l_prem:gts G (par n) D}
      {r_prem:gts G A (sort s)}
   (*****************************************)
        gts (CONS (Gb p A) G) (par n) D]

[sWeak:{G|Cxt}{D,A|Trm}{t,s|SS}{p|PP}           {noccG:is_ff (Poccur p G)}
      {l_prem:gts G (sort t) D}
      {r_prem:gts G A (sort s)}
   (*****************************************)
        gts (CONS (Gb p A) G) (sort t) D]

[Pi:{G|Cxt}{A,B|Trm}{s1,s2,s3|SS}{p|PP}{n|VV}             {sc:rl s1 s2 s3}
                                                {noccB:is_ff (poccur p B)}
      {l_prem:gts G A (sort s1)}
      {r_prem:gts (CONS (Gb p A) G) (vsub (par p) n B) (sort s2)}
   (*****************************************)
        gts G (pi n A B) (sort s3)]

[Lda:{G|Cxt}{A,M,B|Trm}{s|SS}{p|PP}{n,m|VV}       {noccM:is_ff (poccur p M)}
                                                {noccB:is_ff (poccur p B)}
      {l_prem:gts (CONS (Gb p A) G) (vsub (par p) n M) (vsub (par p) m B)}
      {r_prem:gts G (pi m A B) (sort s)}
   (*****************************************)
        gts G (lda n A M) (pi m A B)]

[App:{G|Cxt}{M,A,B,L|Trm}{n|VV}
      {l_prem:gts G M (pi n A B)}
      {r_prem:gts G L A}
   (*****************************************)
        gts G (app M L) (vsub L n B)]

[tCnv:{G|Cxt}{M,A,B|Trm}{s|SS}                             {sc:cnv A B}
      {l_prem:gts G M A}
      {r_prem:gts G B (sort s)}
   (*****************************************)
        gts G M B];
```

Table 4–3: The PTS Judgement as an Inductive Relation

4.4.2 Start Lemmas

Every axiom is derivable in every valid context.

```
Goal sStartLem:
     {G|Cxt}{M,A|Trm}(gts G M A)->
     {s1,s2|SS}(ax s1 s2)->(gts G (sort s1) (sort s2));
```

The global bindings of a valid context are all derivable.

```
Goal vStartLem:
     {G|Cxt}{M,A|Trm}(gts G M A)->
     {b|GB}(is_tt (GBoccur b G))->gts G (par (namOf b)) (typOf b);
```

4.4.3 Topsorts

We digress to consider a pretty notion due to Berardi [Ber90b].

Definition 4.2 *A* `typedsort` *is a sort that occurrs on the left of some axiom; a* `topsort` *is a sort that is not a* `typedsort`*.*

```
[typedsort [s:SS] = Ex[t:SS] ax s t];
[topsort [s:SS] = not (typedsort s)];
```

I use the informal notation $\mathtt{SS}_{\boldsymbol{T}}$ for the set of typedsorts.

A topsort cannot occur in the subject of a derivable judgement, and cannot occur as a proper subterm of the predicate of a derivable judgement. Since we are working constructively, these observations are stated positively.

```
Goal only_typedsort_in_left:
     {G|Cxt}{M,A|Trm}(gts G M A)->{s|SS}(is_tt (soccur s M))->typedsort s;

Goal topsort_maybe_in_right:
     {G|Cxt}{M,A|Trm}(gts G M A)->
     {s|SS}{ts:topsort s}(is_tt (soccur s A))->is_tt (Trm_eq A (sort s));
```

These are proved by structural induction on the derivation of `(gts G M A)`, and in the order given.

4.4.4 A Better Induction Principle for `gts`

As for the relations `Vclosed` and `par_red1` we define an alternative relation, called `apts` (table 4–4). `apts` differs from `gts` only in the right premise of the PI rule and the left premise of the LDA rule. In these premises we avoid choosing a particular parameter by requiring the

```
Inductive [apts:Cxt->Trm->Trm->Prop] Constructors
[aAx:{s1,s2|SS}{sc:ax s1 s2}apts nilCxt (sort s1) (sort s2)]

[aStart:{G|Cxt}{A|Trm}{s|SS}{p|PP}                {noccG:is_ff (Poccur p G)}
         {prem:apts G A (sort s)}
     (****************************************)
           apts (CONS (Gb p A) G) (par p) A]

[avWeak:{G|Cxt}{D,A|Trm}{s|SS}{n,p|PP}            {noccG:is_ff (Poccur p G)}
         {l_prem:apts G (par n) D}
         {r_prem:apts G A (sort s)}
     (****************************************)
           apts (CONS (Gb p A) G) (par n) D]

[asWeak:{G|Cxt}{D,A|Trm}{t,s|SS}{p|PP}            {noccG:is_ff (Poccur p G)}
         {l_prem:apts G (sort t) D}
         {r_prem:apts G A (sort s)}
     (****************************************)
           apts (CONS (Gb p A) G) (sort t) D]

[aPi:{G|Cxt}{A,B|Trm}{s1,s2,s3|SS}{n|VV}                    {sc:rl s1 s2 s3}
         {l_prem:apts G A (sort s1)}
         {r_prem:{p|PP}{noccG:is_ff (Poccur p G)}
             apts (CONS (Gb p A) G) (vsub (par p) n B) (sort s2)}
     (****************************************)
           apts G (pi n A B) (sort s3)]

[aLda:{G|Cxt}{A,M,B|Trm}{s|SS}{n,m|VV}
         {l_prem:{p|PP}{noccG:is_ff (Poccur p G)}
             apts (CONS (Gb p A) G) (vsub (par p) n M) (vsub (par p) m B)}
         {r_prem:apts G (pi m A B) (sort s)}
     (****************************************)
           apts G (lda n A M) (pi m A B)]

[aApp:{G|Cxt}{M,A,B,L|Trm}{n|VV}
         {l_prem:apts G M (pi n A B)}
         {r_prem:apts G L A}
     (****************************************)
           apts G (app M L) (vsub L n B)]

[atCnv:{G|Cxt}{M,A,B|Trm}{s|SS}                               {sc:cnv A B}
         {l_prem:apts G M A}
         {r_prem:apts G B (sort s)}
     (****************************************)
           apts G M B]
NoReductions;
```

Table 4–4: The inductive relation `apts` premise to hold for all parameters for which there is no reason it should not hold, that is, for all "sufficiently fresh" parameters. As before, we will show that gts and apts have the same judgements.

It is interesting to compare the side conditions of PI with those of API. In PI, as in the rules of table 3–4, we need (is_ff (poccur p B)) so that no unintended occurrences of p (i.e. those not arising from occurrences of the bound variable, n) are bound in the right premise; we do not need (is_ff (Poccur p G)), because the validity of (CONS (Gb p A) G) is obvious from the right premise. In aPi, we cannot require the right premise for all p, but only for those such that (CONS (Gb p A) G) remains valid, i.e. those not occurring in G. However the condition (is_ff (poccur p B)) is not required because of "genericity", that is, the right premise of aPi must hold for the infinitely many parameters not occurring in G, while only finitely many of these instances can involve unintended binding in B.

apts identifies all those derivations of a gts judgement that are inessentially different because of parameters occurring in the derivation but not in its conclusion.

4.4.4.1 apts is equivalent to gts

One direction is straightfoward by structural induction.

```
Goal apts_gts : {G|Cxt}{M,A|Trm}(apts G M A)->gts G M A;
```

To prove the interesting direction, first prove a lemma that bijective renamings respect apts.

```
Goal bij_ren_resp_apts :
     {rho|Renaming}(inj rho)->(sur rho)->
     {G|Cxt}{M,A|Trm}(apts G M A)->
     apts (renCxt rho G) (renTrm rho M) (renTrm rho A);
```

This is easy by apts-structural induction. Injectivity of a renaming is enough for it to preserve gts judgements, but surjectivity is needed as well to preserve apts, because of the negative occurrence of Poccur in the premises of aPi and aLda.

Recall from section 3.2.6.2 that for all p and q, (swap p q) is bijective. Now we can proceed.

```
Goal gts_apts : {G|Cxt}{M,A|Trm}(gts G M A)->apts G M A;
```

Proof. By structural induction on a derivation of $\Gamma \vdash M : A$. All cases are trivial except for the rules Pi and Lda. Consider the case for Pi: we must prove

$$G \vdash_{apts} \{n{:}A\}B : s_3$$

under the assumptions

$$\begin{array}{lcl}
\text{sc} & : & \mathtt{rl}(s_1, s_2, s_3) \\
\text{noccB} & : & p \notin B \\
\text{l_prem} & : & G \vdash A : s_1 \\
\text{r_prem} & : & G[p{:}A] \vdash [p/n]B : s_2 \\
\text{l_ih} & : & G \vdash_{apts} A : s_1 \\
\text{r_ih} & : & G[p{:}A] \vdash_{apts} [p/n]B : s_2
\end{array}$$

By rule aPi (using l_ih) it suffices to show

$$G[r{:}A] \vdash_{apts} [r/n]B : s_2$$

for arbitrary parameter $r \notin G$. Thus, using the free parameter lemmas of section 4.4.1, we know

$$\begin{array}{lcll}
\text{noccG} & : & r \notin G & \\
\text{norA} & : & r \notin A & \text{from l_prem and noccG} \\
\text{nopG} & : & p \notin G & \text{from r_prem} \\
\text{nopA} & : & p \notin A & \text{from r_prem}
\end{array}$$

Taking

$$\rho = (\mathtt{swap\ r\ p}) = \{r \mapsto p,\ p \mapsto r\},$$

we have $\rho(G[p{:}A]) \vdash_{apts} \rho([p/n]B) : \rho s_2$ is derivable using `bij_ren_resp_apts` to rename r_ih. Thus we are finished if we can show

$$\rho(G[p{:}A]) = G[r{:}A] \qquad \text{and} \qquad \rho([p/n]B) = [r/n]B.$$

It is clear that the first equation holds from nopG, noccG, norA and nopA. For the second equation, notice that if $r = p$ then ρ is the identity renaming, and we are done, so assume $r \neq p$, and hence $r \notin [p/n]B$ (from r_prem and noccG). Now we use a lemma easily proved by `Trm`-structural induction

$$\forall \rho, N, M, v \ .\ [\rho N/v]\rho M = \rho([N/v]M)$$

to reason

$$\begin{array}{rcll}
\rho([p/n]B) & = & \{p \mapsto r\}([p/n]B) & (r \notin [p/n]B) \\
 & = & [\{p \mapsto r\}p/n](\{p \mapsto r\}B)) & \text{(lemma above)} \\
 & = & [r/n]B & \text{(noccB)}
\end{array}$$

as required. ■

4.4.5 Generation Lemmas

In the presentation of PTS with the weakening rule WEAK, say [Bar92], uniqueness of generation holds only up to conversion *and weakening*. More precisely, going up the leftmost branch of a derivation tree from the root, there may be instances of both TCONV and WEAK before the syntax-directed rule (AX, START, PI, LDA, or APP) that constructed the subject of the root judgement. For this reason, the proof of the generation lemmas in [Bar92] uses the Thinning Lemma (section 4.4.6) to undo this weakening. In our presentation with atomic weakening, uniqueness of generation holds up to conversion (recall the discussion in section 4.1.2), so the generation lemmas are provable before the Thinning Lemma, and their proofs are straightfoward structural induction. I've given the strong versions of the pi and lambda cases, derived from the equivalence of `gts` and `apts` proved in section 4.4.4.1.

```
Goal gts_gen_sort:
     {G|Cxt}{s|SS}{C|Trm}(gts G (sort s) C)->
     Ex [s1:SS]and (ax s s1) (cnv (sort s1) C);

Goal gts_gen_par:
     {G|Cxt}{C|Trm}{p|PP}(gts G (par p) C)->
     Ex [B:Trm]and (is_tt (GBoccur (Gb p B) G)) (cnv B C);

Goal apts_gen_pi:
     {G|Cxt}{A,B,C|Trm}{v|VV}{d:gts G (pi v A B) C}
     Ex3 [s1,s2,s3:SS]
         and4 (rl s1 s2 s3)
              (gts G A (sort s1))
              ({p|PP}{noccG:is_ff (Poccur p G)}
                   gts (CONS (Gb p A) G) (vsub (par p) v B) (sort s2))
              (cnv (sort s3) C);

Goal apts_gen_lda:
     {G|Cxt}{A,N,C|Trm}{v|VV}{d:gts G (lda v A N) C}
     Ex3 [s:SS][B:Trm][u:VV]
         and3 (gts G (pi u A B) (sort s))
              ({p|PP}{noccG:is_ff (Poccur p G)}
                  gts (CONS (Gb p A) G) (vsub (par p) v N) (vsub (par p) u B))
              (cnv (pi u A B) C);

Goal gts_gen_app:
     {G|Cxt}{M,N,C|Trm}(gts G (app M N) C)->
     Ex3 [A,B:Trm][v:VV]and3 (gts G N A)
                             (gts G M (pi v A B))
                             (cnv (vsub N v B) C);
```

4.4.6 The Thinning Lemma

The Thinning Lemma is important to our formulation because it shows that full weakening is derivable in our system from atomic weakening (see section 4.1.2).

```
Goal thinning_lemma: {G|Cxt}{M,A|Trm}{j:gts G M A}
                     {G'|Cxt}{sub:subCxt G G'}{val:Valid G'}gts G' M A;
```

When we come to prove the Thinning Lemma, a serious difficulty arises from our use of parameters.

4.4.6.1 Naive attempt to prove the Thinning Lemma

By induction on the derivation of $G \vdash M : A$. Consider the case for the `Pi` rule: the derivation ends with

$$\textsc{Pi} \qquad \frac{G \vdash A : s_1 \qquad G[p{:}A] \vdash [p/x]B : s_2}{G \vdash \{x{:}A\}B : s_3} \qquad \begin{array}{r} p \notin B \\ \mathtt{rl}(s_1, s_2, s_3) \end{array}$$

left IH For any valid K extending G, $K \vdash A : s_1$.
right IH For any valid K extending $G[p{:}A]$, $K \vdash [p/x]B : s_2$.
to show For any valid G' extending G, $G' \vdash \{x{:}A\}B : s_3$.

So let G' be a valid extension of G. Using the `Pi` rule, we need to show $G' \vdash A : s_1$ and $G'[p{:}A] \vdash [p/x]B : s_2$. The first of these is proved by the left IH applied to G'. In order to use the right IH applied to $G'[p{:}A]$ to prove the second, we need to show $G'[p{:}A]$ is valid. However, this may be false, e.g. if p occurs in G'!

The left premise of the LDA rule presents a similar problem. All other cases are straightfoward.

Naive attempt to fix the problem Let us change the name of p in the right premise. We easily prove that any injective renaming preserves `gts` judgements (see section 4.4.4.1). However, this does not help us finish the naive proof of the Thinning Lemma, for the right IH still mentions p, not some fresh parameter q. That is, although we can turn the right subderivation $G[p{:}A] \vdash [p/x]B : s_2$ into a derivation of $G[q{:}A] \vdash [q/x]B : s_2$ for some fresh q, this is *not* a structural subderivation of the original proof of $G \vdash \{x{:}A\}B : s_3$, and is no help in structural induction.

4.4.6.2 Some correct proofs of the Thinning Lemma

We present three correct proofs of the Thinning Lemma, based on different analyses of what goes wrong in the naive proof. The two proofs in this section are, perhaps, possible to formalize. Section 4.4.4 gives the formalization we have chosen.

Changing the names of parameters. Once we say "by induction on the derivation ..." it is too late to change the names of any troublesome parameters, but one may fix the naive proof by changing the names of parameters in a derivation *before* the structural induction. First observe, by structural induction, that if d is a derivation of $G \vdash M : A$, G' is a Valid extension of G, and no parameter occurring in d is in globalNames(G')\globalNames(G), then $G' \vdash M : A$ by the naive structural induction above. Now for the Thinning Lemma, given G, G', and a derivation d of $G \vdash M : A$, change parameters in d to produce a derivation d' of $G \vdash M : A$ such that no parameter occurring in d' is in globalNames(G')\globalNames(G).

This proof takes the view that it is fortunate there are many derivations of each judgement, for we can find one suited to our purposes among them. To formalize it we need technology for renaming parameters in *derivations*, which is much heavier than the technology for renaming parameters in *judgements* that we have presented.

Induction on length. A different analysis of the failure of the naive proof shows that it is the use of *structural* induction that is at fault. Induction on the height of derivations appears to work, but we must show that renaming parameters in a derivation doesn't change its height; again reasoning about derivations rather than judgements. Induction on the length of the subject term, M, does not seem to work, as some rules have premises whose subject is not shorter than that of the conclusion (see the discussion in section 3.2.5.3).

4.4.6.3 A Better Solution

We take the view that there are too many derivations of each judgement, but instead of giving up on structural induction, we prove the Thinning lemma by structural induction on `apts`, which we have shown to have the same judgements as `gts`, while identifying derivations that differ only by parameter names not occurring in the root judgement.

Proof of the Thinning Lemma using `apts`. Using the equivalence of `gts` and `apts` it suffices to prove

```
Goal apts_thinning_lem:
     {G|Cxt}{M,A|Trm}{j:apts G M A}
     {G'|Cxt}{sub:subCxt G G'}{val:Valid G'}apts G' M A;
```

by structural induction on the derivation of $G \vdash_{apts} M : A$; all goes as in the naive proof except for the `aPi` and `aLda` rules. Consider the case for `aPi`.

left IH For any Valid K extending G, $K \vdash_{apts} A : s_1$.
right IH For any $p \notin G$, and any Valid K extending $G[p{:}A]$,
$K \vdash_{apts} [p/x]B : s_2$.

to show For any valid G' extending G, $G' \vdash_{apts} \{x{:}A\}B : s_3$.

By rule aPi and the left IH it suffices to show

$$\text{for any } p \notin G',\ G'[p{:}A] \vdash_{apts} [p/x]B : s_2$$

So choose $p \notin G'$. Using the right IH applied to p and $G'[p{:}A]$ we have only to show $G'[p{:}A]$ Valid, which follows by the aStart rule and the left IH.

The case for aLda is handled similarly. ■

4.4.6.4 The Weakening Rule

Now the full weakening rule, WEAK (section 4.1.2) is seen to be admissible.

```
Goal weakening: {G|Cxt}{M,B|Trm}(gts G M B)->
                {p|PP}(is_ff (Poccur p G))->
                {A|Trm}{s|SS}(gts G A (sort s))->
                gts (CONS (Gb p A) G) M B;
```

This is the only case of the thinning lemma that is used in all the type theory we have formalized.

4.4.7 The Substitution Lemma

This is a cut property, informally

$$\frac{\Gamma \vdash N : A \qquad \Gamma[p{:}A]\Delta \vdash M : B}{\Gamma([N/p]\Delta) \vdash [N/p]M : [N/p]B}$$

Formally:

```
Goal substitution_lemma:
    {Gamma|Cxt}{N,A|Trm}(gts Gamma N A)->
    {q:PP}[sub = psub N q][subGB = GBsub N q]
    {Delta:Cxt}{M,B|Trm}(gts (append Delta (CONS (Gb q A) Gamma)) M B)->
    gts (append (map subGB Delta) Gamma) (sub M) (sub B);
```

The proof is by induction on the derivation of $\Gamma[p : A]\Delta \vdash M : B$.

From substitution_lemma we trivially get the cut rule

```
Goal cut_rule: {G|Cxt}{N,A|Trm}(gts G N A)->
               {q:PP}{M,B|Trm}(gts (CONS (Gb q A) G) M B)->
               gts G (psub N q M) (psub N q B);
```

which is the commonly used case.

4.4.8 Correctness of Types

Among the principal correctness criteria for type systems is that every type is itself well formed in some sense. In PTS we define *correct type*

```
[correct_type [G:Cxt][A:Trm] = Ex [s:SS] or (is_tt (Trm_eq A (sort s)))
                                          (gts G A (sort s))];
```

and have the theorem:

```
Goal type_correctness: {G|Cxt}{M,A|Trm}(gts G M A)->correct_type G A;
```

The proof is a simple structural induction. The only non-trivial case is the APP rule, which uses the substitution lemma and `vsub_is_psub_alpha` (section 3.2.4).

4.4.8.1 Type Correctnes and Topsorts

The statement of `type_correctness` seems unsatisfactory; by `sStartLem` (section 4.4.2), the only way a sort can be untyped is to be a topsort, so why not make the disjunction more informative, as in

$$\Gamma \vdash M : A \;\Rightarrow\; A \text{ is a topsort or } \exists s\, .\, \Gamma \vdash A : s\, . \qquad (\ddagger)$$

This is too informative to be proved without some extra assumption:

Lemma 4.3 *For PTS where the relation cnv is instantiated with β-conversion, property ($\ddagger$) implies*

$$\forall s\, .\, s \in SS_T \text{ or } s \notin SS_T \qquad (\diamond)$$

Proof. Given a PTS $(\mathtt{SS}, \mathtt{ax}, \ldots)$ let $\star$ be a fresh symbol, not in SS, and consider the PTS $(\mathtt{SS} \cup \{\star\}, \mathtt{ax} \cup \{\, \star{:}s \mid s \in \mathtt{SS}\,\}, \ldots)$. In this expanded PTS we have $\bullet \vdash \star : s$ for every $s \in \mathtt{SS}$, hence by implication ($\ddagger$)

$$\forall s \exists t\, .\, s \notin \mathtt{SS}_T \text{ or } \bullet \vdash s : t$$

and by the generation lemma `gts_gen_sort` (section 4.4.5) property ($\diamond$) holds as claimed. (It is in the last step that we use the assumption that cnv is β-conversion; see section 4.4.11.2.) ■

Any PTS with SS infinite may fail to have property ($\diamond$), and thus fail to have property ($\ddagger$), and because of the existential quantifier in the definition of typedsort, this may happen even when ax is decidable. For example, let $\mathsf{T}(i, n)$ mean "the i^{th} Turing machine, when started with i on it's input tape, halts in exactly n steps", and consider a PTS whose set of sorts is the natural numbers and whose axiom relation is $\{\, (i{:}n) \mid \mathsf{T}(i, n)\,\}$. For any i,

$$i \in \mathtt{SS}_T \;\Leftrightarrow\; \exists n\, .\, \mathtt{ax}(i{:}n) \;\Leftrightarrow\; \text{the } i^{th} \text{ Turing machine halts on input } i$$

which we cannot decide in general. Thus, such a PTS fails to have property (‡) even though it has decidable axioms.

Conversely to lemma 4.3, property (◇) implies property (‡) for all PTS.

```
Goal decideTypedsort_type_correctness:
     {dTs:{s:SS}decidable (typedsort s)}
     {G|Cxt}{M,A|Trm}{d:gts G M A}
     Ex [s:SS]or (and (is_tt (Trm_eq A (sort s))) (topsort s))
                  (gts G A (sort s));
```

Proof. By type_correctness, $A \in$ SS or $\exists s\ .\ G \vdash A : s$. In the second case we are done. In the first case, by decideTypedsort, either $A \notin \mathrm{SS}_T$ (and we are done) or $A \in \mathrm{SS}_T$, in which case $G \vdash A : s$ for some s, by sStartLem. ∎

Remark 4.4 *By lemma 4.3, we cannot replace the assumption* decideTypedsort *in the statement of lemma* decideTypedsort_type_correctness *with the weaker assumption*

$$\forall s\ .\ s \notin SS_T \text{ or } (\neg s \notin SS_T).$$

4.4.9 Subject Reduction Theorem: Closure Under Reduction

An important property of type systems is that a term does not lose types under reduction, thus types are a classification of terms preserved by computation. In fact we will show entire gts-judgements are closed under reduction.

4.4.9.1 Properties of Conversion Needed for Subject Reduction

Now we need more properties of the abstract conversion relation cnv.

```
[cnv_red1:{A,B|Trm}(par_red1 A B)->(Vclosed A)->cnv A B];
[cnv_red1_sym:{A,B|Trm}(par_red1 A B)->(Vclosed A)->cnv B A];
[cnvCR_pi:{u,v|VV}{A,A',B,B'|Trm}{c:cnv (pi u A B) (pi v A' B')}
      and (cnv A' A) ({q:PP}cnv (alpha q u B) (alpha q v B'))];
```

Using cnv_red1 and cnv_red1_sym, transitivity of cnv, and Church-Rosser, we can prove that cnv contains conv.

```
Goal cnv_redn: {A,B|Trm}(par_redn A B)->(Vclosed A)->cnv A B;
Goal cnv_redn_sym: {A,B|Trm}(par_redn A B)->(Vclosed A)->cnv B A;
Goal cnv_conv: {A,B|Trm}(conv A B)->cnv A B;
```

Thus it would be equivalent to replace the two assumptions, cnv_red1 and cnv_red1_sym, by the single assumption cnv_conv. However I chose to keep the two directions of reduction seperate in hope of learning something about Expansion Postponement (section 4.7). $\leq$ is a transitive relation containing an equivalence relation, $\simeq$, so $\leq$ is well defined on $\simeq$-classes.

I think of the third assumption, `cnvCR_pi`, as an *internal* Church-Rosser property: if two terms of the same shape (in this case pi) are related by $\leq$, then their corresponding components are similarly related. (In section 4.4.11.3 we will consider *external* Church-Rosser properties required of $\leq$.) For beta-conversion, `cnvCR_pi`, is a corollary of the Church-Rosser property.

4.4.9.2 Non-Overlapping Reduction

In [GN91,Bar92] a lemma

$$G \vdash M : A \;\Rightarrow\; (M \to M' \;\Rightarrow\; G \vdash M' : A) \;\&\; (G \to G' \;\Rightarrow\; G' \vdash M : A)$$

is proved by induction on the structure of $G \vdash M : A$, where $\to$ is one-step-reduction, `red1`, of section 3.3.6. (The reason for this simultaneous proof is that in some of the typing rules, terms move from the subject to the context, e.g. PI.) This approach produces a large number of case distinctions, based on which subterm of the subject contains the one redex which is contracted, all of which are inessential except to isolate the one non-trivial case where the redex contracted is the application constructed by rule APP. Also (recall section 3.3.6), we have a strategy of using reduction relations that contain alpha-conversion; `red1` and its transitive-reflexive closure do not. Thus, in place of one-step-reduction, we use a new relation of *non-overlapping reduction*, `no_red1`, that differs from `par_red1` only in the β rule[1].

```
[no_red1_beta: {u:VV}{U,A,B:Trm}no_red1 (app (lda u U B) A) (vsub A u B)]
```

We informally write $\to$ for `no_red1`. Clearly `no_red1` is contained in `par_red1`

```
Goal no_par_red1: {A,B|Trm}(no_red1 A B)->par_red1 A B;
```

so the assumed properties `cnv_red1` and `cnv_red1_sym` extend to `no_red1`.

The distinction between a redex in the term and a redex in the context is also inessential in the proof of subject reduction, so we extend `no_red1` compositionally to contexts

[1] I discovered that non-overlapping reduction works well in the proof of subject reduction while "feverishly" trying to prove Expansion Postponement as described in section 1.3

```
Inductive [red1Cxt: Cxt->Cxt->Prop]  NoReductions  Constructors
 [red1CxtNIL: red1Cxt nilCxt nilCxt]
 [red1CxtCONS: {b,b'|GB}{G,G'|Cxt}{sc:and (Q (namOf b) (namOf b'))
                                        (no_red1 (typOf b) (typOf b'))}
               {premG:red1Cxt G G'}
               red1Cxt (CONS b G) (CONS b' G')];
```

and to pairs of a context and a term (suggesting again that the correct meaning of *subject* of a PTS judgement $G \vdash M : A$ is the pair $\langle G, M \rangle$).

```
Inductive [red1Subj: Cxt->Trm->Cxt->Trm->Prop] NoReductions  Constructors
 [red1SubjCT: {G,G'|Cxt}{premC:red1Cxt G G'}
              {A,A'|Trm}{premT:no_red1 A A'}
              red1Subj G A G' A'];
```

The important point is that we allow non-overlapping reduction throughout the subject (context and term), rather than restricting to a single redex, not that we package it in the form of `red1Subj`, which is only a funny conjunction after all.

We also define the transitive closure of `no_red1`

```
Inductive [no_redn: Trm->Trm->Prop]  NoReductions  Constructors
 [no_redn_red1: {A,B|Trm}(no_red1 A B)->no_redn A B]
 [no_redn_trans: {t,u,v|Trm}(no_redn t u)->(no_redn u v)->no_redn t v];
```

and show it is extensionally the same as `par_redn`.

```
Goal no_par_redn: {A,B|Trm}(no_redn A B)->par_redn A B;
Goal par_no_redn: {A,B|Trm}(par_redn A B)->no_redn A B;
```

4.4.9.3 The Main Lemma

```
Goal subject_reduction_lem: {G|Cxt}{M,A|Trm}(gts G M A)->
                            {G'|Cxt}{M'|Trm}(red1Subj G M G' M')->gts G' M' A;
```

Proof. By structural induction on $G \vdash M : A$. I show two cases

Start: Given

noccG	:	$p \notin G$
prem	:	$G \vdash A : s$
ih	:	$\forall k, r\,.\,(\langle G, A\rangle \to \langle k, r\rangle) \;\Rightarrow\; k \vdash r : s$
red_subj	:	$\langle G[p{:}A], p\rangle \to \langle K, R\rangle$

we must show $K \vdash R : A$. From red_subj we have $R = p$ and $K = G'[p{:}A']$, where $A \to A'$ and $G \to G'$, so we need to show $G'[p{:}A'] \vdash p : A$. By ih $G' \vdash A' : s$, and by Start, $G'[p{:}A'] \vdash p : A'$. We need to expand the predicate, A', back up to A: by

TCNV and `cnv_red1_sym` it suffices to show $G'[p{:}A'] \vdash A : s$. This follows from ih (hence $G' \vdash A : s$) and weakening.

App: Given

$$\begin{array}{lcl} \text{l\_prem} & : & G \vdash M : \{n{:}A\}B \\ \text{r\_prem} & : & G \vdash L : A \\ \text{l\_ih} & : & \forall k, r\,.\,(\langle G, M\rangle \to \langle k, r\rangle) \;\Rightarrow\; k \vdash r : \{n{:}A\}B \\ \text{r\_ih} & : & \forall k, r\,.\,(\langle G, L\rangle \to \langle k, r\rangle) \;\Rightarrow\; k \vdash r : A \\ \text{red\_subj} & : & \langle G, M\ L\rangle \to \langle K, R\rangle \end{array}$$

we must show $K \vdash R : [L/n]B$. By induction hypotheses

$$\begin{array}{lcl} \text{gtsKM} & : & K \vdash M : \{n{:}A\}B \\ \text{gtsKL} & : & K \vdash L : A \end{array}$$

By type correctness of gtsKM, for some s

$$\text{gtsKpi} : \; K \vdash \{n{:}A\}B : s$$

By the pi-generation lemma, for some $s1$, $s2$ and $p \notin B$

$$\text{gtsKB} : \; K[p{:}A] \vdash [p/n]B : s2$$

By the cut rule on gtsKL and gtsKB (we also use `vsub_is_psub_alpha` (section 3.2.4) here, and several more times in this case)

$$\text{gtsKBsub} : \; K \vdash [L/n]B : s2.$$

Now there are two subcases

$R = M'\ L'$ where $M \to M'$ and $L \to L'$. The goal is $K \vdash M'\ L' : [L/n]B$; by rule APP and the induction hypotheses we easily have $K \vdash M'\ L' : [L'/n]B$. Use rule TCNV and gtsKBsub to expand L' in the predicate back to L as required (this once again uses `cnv_red1_sym`).

$R = [L/v]b$ where $M = [v{:}A']b$. The goal is $K \vdash [L/v]b : [L/n]B$. By l_ih $K \vdash [v{:}A']b : \{n{:}A\}B$ and by the lambda-generation lemma, for some w, B' and s'

$$\begin{array}{lcl} \text{gtsKpi}' & : & K \vdash \{w{:}A'\}B' : s' \\ \text{gtsKb} & : & \forall p \notin K\,.\,K[p{:}A'] \vdash [p/v]b : [p/w]B' \\ \text{c}' & : & \{w{:}A'\}B' \leq \{n{:}A\}B \end{array}$$

By `(cnvCR_pi c')` (this is the only time it is used in this proof)

$$\begin{array}{lcl} \text{cnvA} & : & A \leq A' \\ \text{cnvB} & : & \forall q\,.\,[q/w]B' \leq [q/n]B \end{array}$$

By a generation lemma on gtsKpi′

gtsKA′ : $K \vdash A' : s1'$

By (`tCnv cnvA gtsKL gtsKA'`)

gtsKL′ : $K \vdash L : A'$

By TCNV, cnvB and gtsKBsub, it suffices to show $K \vdash [L/v]b : [L/n]B'$ which follows by cut on gtsKL′ and gtsKb.

The only other use of `cnv_red1_sym` in this proof is for the LDA rule, where the domain type, A, may get reduced, and its occurrence in the type must be expanded back, much as in the START case. ∎

4.4.9.4 Closure Under Reduction

It is now easy to show what is usually called the *Subject Reduction* theorem.

```
Goal gtsSR: {G|Cxt}{M,A|Trm}(gts G M A)->{M'|Trm}(par_redn M M')->gts G M' A;
```

From this we have a useful corollary.

```
Goal gtsPR: {G|Cxt}{M,A|Trm}(gts G M A)->{A'|Trm}(par_redn A A')->gts G M A';
```

Proof. By type correctness, either $A \in$ SS or $\exists s\ .\ G \vdash A : s$. In the first case $A' = A$ and we are done. In the second case, by `gtsSR`, we have $G \vdash A' : s$, so by rule TCNV we are done. ∎

Finally, extending `par_redn` compositionally to contexts, `gts` is closed under beta-reduction.

```
Goal gtsAllRed: {G|Cxt}{M,A|Trm}(gts G M A)->
                {G'|Cxt}(rednCxt G G')->
                {M'|Trm}(par_redn M M')->
                {A'|Trm}(par_redn A A')->gts G' M' A';
```

4.4.9.5 Closure Under Alpha-Conversion

From `gtsAllRed` and the fact that `alpha_conv` is contained in `par_red1`, it is trivial to show that `gts` judgements are preserved by `alpha_conv`. First extend `alpha_conv` to contexts:

```
Inductive [alphaCxt : Cxt->Cxt->Prop]  NoReductions  Constructors
 [alphaCxtNIL: alphaCxt nilCxt nilCxt]
 [alphaCxtCONS: {b,b'|GB}{G,G'|Cxt}{sc:and (Q (namOf b) (namOf b'))
                                           (alpha_conv (typOf b) (typOf b'))}
                {premG:alphaCxt G G'}
                alphaCxt (CONS b G) (CONS b' G')];
```

Then we have:

```
Goal gts_alpha_closed: {G|Cxt}{M,A|Trm}(gts G M A)->
                       {GG|Cxt}(alphaCxt G GG)->
                       {MM|Trm}(alpha_conv M MM)->
                       {AA|Trm}(alpha_conv A AA)->gts GG MM AA;
```

This is a good result for an implementor: it says an implementation may typecheck a judgement as stated by a user, rather than checking an alpha-equivalent judgement constructed for some intensional property such as no reuse of bound variable names. Still, from an intensional view this result is not completely satisfactory. It doesn't say that alpha-equivalent judgements have "alpha-equivalent derivations"; in fact, the proof of subject reduction explicitly uses rule `tCnv`, and the proof of `gts_alpha_closed` will insert instances of `tCnv` that do only alpha-conversion. We return to this point in section 4.4.9.7.

4.4.9.6 Correctness of the Lda Rule

Acknowledgement Special thanks to James McKinna who did the proofs of this section.

Recall the discussion about rules LDA and LDA′ in section 4.1.3.1. Although I gave "philosophical" motivations for our choice of LDA over LDA′, the real reason was to make the proof of subject reduction go smoothly. It is time to consider whether we have cheated or not.

Define an inductive relation `rlts` (restricted lambda type system) whose rules are the same as those of `gts` except for the lambda rule, which expresses our informal LDA′ rule.

```
[rlLda:{G|Cxt}{A,M,B|Trm}{s|SS}{p|PP}{n|VV}         {noccM:is_ff (poccur p M)}
                                                    {noccB:is_ff (poccur p B)}
      {l_prem:rlts (CONS (Gb p A) G) (vsub (par p) n M) (vsub (par p) n B)}
      {r_prem:rlts G (pi n A B) (sort s)}
   (*****************************************)
         rlts G (lda n A M) (pi n A B)]
```

Every `rlts`-derivation is a `gts`-derivation, so:

```
Goal rlts_gts: {G|Cxt}{M,A|Trm}(rlts G M A)->gts G M A;
```

It is not very easy to show the converse, even though we know `gts` is closed under alpha-conversion, because our present lemmas on changing names are not strong enough: in `true_alpha_conv_pi` (page 43) we don't know how C is related to B. A proof would go through if we knew that `rlts` was closed under alpha-conversion, but we are trying to avoid proving subject reduction for `rlts` directly. James McKinna suggested proving directly that `rlts` is *weakly* closed under alpha-conversion.

```
Goal rl_weak_alpha_lem: {G|Cxt}{M,A|Trm}(rlts G M A)->
                        {G'|Cxt}(alphaCxt G G')->
                        {M'|Trm}(alpha_conv M M')->
                        Ex [A':Trm]and (rlts G' M' A') (alpha_conv A A');
```

(*Weakness* means that the type is also only up to alpha-conversion.) The proof is similar in outline to that of subject reduction, but much simpler in detail. With this lemma McKinna showed that `rlts` and `gts` derive the same judgements.

```
Goal gts_rlts: {G|Cxt}{M,A|Trm}(gts G M A)->rlts G M A;
```

Of course they do not have the same derivations, as example 4.1 shows.

4.4.9.7 Alpha-Conversion and the Shape of Derivations

Informally consider a presentation of PTS using de Bruijn nameless variables for bound variables, and using our parameters for free variables (I call these *locally nameless* terms in [Pol94]). It is clear that every `gts` derivation can be translated into a correct locally nameless derivation just by replacing the variable names with appropriate binding depths throughout. Some derivations, such as the second one in example 4.1, when translated to the locally nameless system, will have redundant instances of the conversion rule where it was used for alpha-conversion in the named system. Translating in the other direction, from nameless derivations to named derivations, there are choices to make for the names of variables, and this leads to the question: can every nameless derivation be translated to some named derivation? Stated differently, is there always some translation of the root of a nameless derivation (the derived judgement) that can be extended to a translation of the full nameless derivation? (It is clear that not every translation of the root of a nameless derivation can be extended to a translation of the full nameless derivation.) I do not know the answer, and it surprises me that the difference between nameless terms and named terms is not generally taken more seriously.

4.4.10 Two Other Presentations of PTS

The rules of table 4–2 define a single relation, the typing judgement. Every leaf of every derivation has an empty context, each branch uses START and (atomic) weakening to build the context it needs, and the only way to access a context is by START. Validity of contexts is defined as participating in some derivable judgement. This presentation is convenient for meta theory because there is a single induction principle for the single relation. Intuitively, however, validity is often considered a different concept, mutually inductive with typing. Then every axiom holds in every valid context, and access to a valid context is by lookup, rather than by construction. Dependent type theories have often been presented in this way, e.g. [CH88, Luo90a]. In section 4.4.10.1 I give the usual system of valid contexts; in section 4.4.10.2 I give a system for implementation, with much smaller derivations of the same judgements as `gts`.

This entire section uses very little of the `gts` theory developed so far, and can be checked before section 4.4.3 on topsorts. However I deferred the presentation to this point because it uses arguments similar to that needed for the `gts` thinning lemma. It is of pragmatic interest for the purpose of efficient typechecking algorithms, and of technical interest because it shows the use of mutual inductive definition, and of dependent elimination of a relation (as opposed to a type, such as `Trm`), both for the first time in this thesis.

4.4.10.1 The System of Valid Contexts

Table 4–5 shows the system of valid contexts. It is a mutually inductive definition of two relations: `vcxt`, validity of contexts, and `vtyp`, the typing judgement. As this is the first mutually inductive definition in this thesis, here, edited for clarity, is the structure of the elimination rule generated for `vcxt`[2].

```
$vcxt_elim :
        (* predicates for each type defined *)
        {C_vcxt:{x1|Cxt}(vcxt x1)->TYPE}
        {C_vtyp:{x1|Cxt}{x2,x3|Trm}(vtyp x1 x2 x3)->TYPE}

  (* cases for each constructor of each type defined *)
  (C_vcxt vcNil)->
  ({G|Cxt}{p|PP}{A|Trm}{s|SS}{noccG:is_ff (Poccur p G)}
     {prem:vtyp G A (sort s)}       (* premises .............. *)
     (C_vtyp prem)->                (* induction hypotheses ... *)
       C_vcxt (vcCons noccG prem))->
  ({G|Cxt}{s1,s2|SS}{sc:ax s1 s2}
     {prem:vcxt G}
     (C_vcxt prem)->
       C_vtyp (vtSort sc prem))->

               (* ........  cases omitted  ....... *)

  (* the conclusion for vcxt only *)
  {x1|Cxt}{z:vcxt x1}C_vcxt z
```

We have chosen to have an elimination rule for each type defined, even though each elimination rule has all the same cases, so for `vtyp` we have

---

[2] Thanks to Claire Jones and Zhaohui Luo who specified and implemented the inductive types tactic.

```
Inductive [vcxt:Cxt->Prop][vtyp:Cxt->Trm->Trm->Prop] Constructors
                                          (** vcxt **)
[vcNil:vcxt nilCxt]

[vcCons:{G|Cxt}{p|PP}{A|Trm}{s|SS}               {noccG:is_ff (Poccur p G)}
       {prem:vtyp G A (sort s)}
     (****************************************)
       vcxt (CONS (Gb p A) G)]

                                          (** vtyp **)
[vtSort:{G|Cxt}{s1,s2|SS}                                      {sc:ax s1 s2}
       {prem:vcxt G}
     (****************************************)
       vtyp G (sort s1) (sort s2)]

[vtPar:{G|Cxt}{p|PP}{A|Trm}                 {sc:is_tt (GBoccur (Gb p A) G)}
       {prem:vcxt G}
     (****************************************)
       vtyp G (par p) A]

[vtPi:{G|Cxt}{A,B|Trm}{s1,s2,s3|SS}{p|PP}{n|VV}              {sc:rl s1 s2 s3}
                                                 {noccB:is_ff (poccur p B)}
       {l_prem:vtyp G A (sort s1)}
       {r_prem:vtyp (CONS (Gb p A) G) (vsub (par p) n B) (sort s2)}
     (****************************************)
       vtyp G (pi n A B) (sort s3)]

[vtLda:{G|Cxt}{A,M,B|Trm}{s|SS}{p|PP}{n,m|VV}      {noccM:is_ff (poccur p M)}
                                                 {noccB:is_ff (poccur p B)}
       {l_prem:vtyp (CONS (Gb p A) G) (vsub (par p) n M) (vsub (par p) m B)}
       {r_prem:vtyp G (pi m A B) (sort s)}
     (****************************************)
       vtyp G (lda n A M) (pi m A B)]

[vtApp:{G|Cxt}{M,A,B,L|Trm}{n|VV}
       {l_prem:vtyp G M (pi n A B)}
       {r_prem:vtyp G L A}
     (****************************************)
       vtyp G (app M L) (vsub L n B)]

[vttCnv:{G|Cxt}{M,A,B|Trm}{s|SS}                                {sc:cnv A B}
       {l_prem:vtyp G M A}
       {r_prem:vtyp G B (sort s)}
     (****************************************)
       vtyp G M B]
NoReductions;
```

Table 4–5: The System of Valid Contexts

```
$vtyp_elim :

                (* .......  same as above  ........ *)

  (* the conclusion for vtyp only *)
  {x1|Cxt}{x2,x3|Trm}{z:vtyp x1 x2 x3}C_vtyp z
```

`vtyp` **and** `gts` **are equivalent** It is easy to show that `vtyp` is sound for `gts` by structural induction on `vtyp`, where we specify induction predicates for both relations, `vcxt` and `vtyp`.

```
Goal vtyp_gts: {G|Cxt}{M,A|Trm}(vtyp G M A)->(gts G M A);
Refine vtyp_elim ([G|Cxt][_:vcxt G]or (Q G nilCxt) (Valid G))
                 ([G|Cxt][M,A|Trm][_:vtyp G M A]gts G M A);
```

There is a companion theorem, by eliminating `vcxt` with the same induction predicates.

```
Goal vcxt_Valid: {G|Cxt}(vcxt G)->or (Q G nilCxt) (Valid G);
```

By the way, `vcxt` is not the same as `Valid` because if `ax` is empty then there are no derivable `gts` judgements, hence no `Valid` contexts, but the empty context is still a `vcxt`.

After developing generation lemmas and weakening for `vtyp`, we prove that `vtyp` is complete for `gts` by `gts`-structural induction.

```
Goal gts_vtyp: {G|Cxt}{M,A|Trm}(gts G M A)->(vtyp G M A);
```

The proof of weakening for `vtyp` is no easier than for `gts`, and follows from a thinning lemma proved via an alternative version of `vtyp`, similar to sections 4.4.4 and 4.4.6.

4.4.10.2 The System of Locally Valid Contexts

`vtyp`-derivations are not very different in shape from `gts` derivations; all leaves are `vcNil` and all branches must build their own context with `vcCons`. This is very inefficient, as the rules for pi, lambda, application and conversion all mention the same context in both premises, so it must be checked in branches leading to both premises. This is apparently exponential in the length of the context. If an implementation actually checked in this way it wouldn't get very far. I now give a system with much smaller derivations, and show that it can be used to check `gts` judgements. The idea is originally due to Martin-Löf, but I learned it from Gérard Huet [Hue89].

The system of locally valid contexts has a typing judgement, `lvtyp` (table 4–6), similar to `vtyp` except it doesn't check validity of the context at all. More precisely a derivation of `lvtyp G M A` does not check the validity of `G`; however, each time a context in a derivation is extended, the extension is checked, assuming the original context is valid. For example, the rule `lvtPi` checks that `A` has a sort and `p` is fresh for `G`. There is also a validity judgement,

```
Inductive [lvtyp:Cxt->Trm->Trm->Prop]  Constructors
[lvtSort:{s1,s2|SS}                                        {sc:ax s1 s2}
       {G:Cxt}lvtyp G (sort s1) (sort s2)]

[lvtPar:{G|Cxt}{p|PP}{A|Trm}                 {sc:is_tt (GBoccur (Gb p A) G)}
       lvtyp G (par p) A]

[lvtPi:{G|Cxt}{A,B|Trm}{s1,s2,s3|SS}{p|PP}{n|VV}            {sc:rl s1 s2 s3}
                                                  {noccB:is_ff (poccur p B)}
                                                  {noccG:is_ff (Poccur p G)}
       {l_prem:lvtyp G A (sort s1)}
       {r_prem:lvtyp (CONS (Gb p A) G) (vsub (par p) n B) (sort s2)}
     (****************************************)
       lvtyp G (pi n A B) (sort s3)]

[lvtLda:{G|Cxt}{A,M,B|Trm}{s|SS}{p|PP}{n,m|VV}    {noccM:is_ff (poccur p M)}
                                                  {noccB:is_ff (poccur p B)}
                                                  {noccG:is_ff (Poccur p G)}
       {l_prem:lvtyp (CONS (Gb p A) G) (vsub (par p) n M) (vsub (par p) m B)}
       {r_prem:lvtyp G (pi m A B) (sort s)}
     (****************************************)
       lvtyp G (lda n A M) (pi m A B)]

[lvtApp:{G|Cxt}{M,A,B,L|Trm}{n|VV}
       {l_prem:lvtyp G M (pi n A B)}
       {r_prem:lvtyp G L A}
     (****************************************)
       lvtyp G (app M L) (vsub L n B)]

[lvtCnv:{G|Cxt}{M,A,B|Trm}{s|SS}                               {sc:cnv A B}
       {l_prem:lvtyp G M A}
       {r_prem:lvtyp G B (sort s)}
     (****************************************)
       lvtyp G M B];


Inductive [lvcxt:Cxt->Prop]  Constructors
[lvcNil:lvcxt nilCxt]

[lvcCons:{G|Cxt}{p|PP}{A|Trm}{s|SS}               {noccG:is_ff (Poccur p G)}
                                                     {sc:lvtyp G A (sort s)}
       {prem:lvcxt G}
     (****************************************)
       lvcxt (CONS (Gb p A) G)];
```

Table 4–6: The System of Locally Valid Contexts `lvcxt`, similar to `vcxt` except for using `lvtyp` in place of `lvtyp`. (Notice, however, that since `lvtyp` does not check validity, it doesn't depend on `lvcxt`, so the two are not mutually inductive.)

The locally valid system characterizes `gts`**.** We will show:

```
Goal iff_gts_lvcxt_lvtyp:
    {G:Cxt}{M,A:Trm}iff (gts G M A) (and (lvcxt G) (lvtyp G M A));
```

Direction $\Rightarrow$ is trivial, as

$$G \vdash M : A \;\Rightarrow\; G \vdash_{vtyp} M : A \;\Rightarrow\; \begin{cases} G \vdash_{lvtyp} M : A \\ G \vdash_{vcxt} \;\Rightarrow\; G \vdash_{lvcxt} \end{cases}$$

Direction $\Leftarrow$ can be proved directly by lexicographic induction, first on the sum of the lengths of the derivations of $G \vdash_{lvtyp} M : A$ and $G \vdash_{lvcxt}$, and second on the length of the derivation of $G \vdash_{lvtyp} M : A$. Here I will unwind this lexicographic dependency into two simple inductions using `vtyp` as an intermediate system. (Another reason to use an intermediate system is that `lvtyp`-typable terms and their types may not be `Vclosed` because `lvtyp` does not restrict its contexts.)

The first step is to show

```
Goal lvtyp_vcxt_vtyp: {G|Cxt}{M,A|Trm}(lvtyp G M A)->(vcxt G)->vtyp G M A;
```

by easy `lvtyp`-structural induction. This lemma says that an `lvtyp`-derivation starting with a context that is `vcxt` (i.e. "really valid"), can be transformed to a `vtyp`-derivation of the same judgement (which, we saw above, can be transformed to a `gts`-derivation). It is easy because except for accepting unvalidated contexts at its leaves (where the `vcxt` assumption is used), `lvtyp` is just as strict as `vtyp`.

The hard part, the main lemma of this section, is to show that this requirement on the context can be weakened from `vcxt` to `lvcxt`.

```
Goal lvtyp_lvcxt_vtyp: {G|Cxt}{M,A|Trm}{t:lvtyp G M A}{c:lvcxt G}vtyp G M A;
```

This is hard because `lvcxt`-derivations are much smaller than `vcxt`-derivations; we must give a terminating procedure for recomputing the annotations that are omitted in `lvcxt`-derivations[3]. The proof will be by induction on the sum of the lengths of `(lvtyp G M A)` and `(lvcxt G)`. Here is the general well-founded induction whose measure is an `NN`-valued function on pairs of judgements `(lvtyp G M A)`, `(lvcxt G)`.

---

[3] Thanks to Stefano Berardi for this point of view.

```
Goal wc_wt_WF_ind:
     {f:{G|Cxt}{M,A|Trm}(lvtyp G M A)->(lvcxt G)->NN}
     {C:{G|Cxt}{M,A|Trm}(lvtyp G M A)->(lvcxt G)->Prop}
     {wf_ih:{G|Cxt}{M,A|Trm}{t:lvtyp G M A}{c:lvcxt G}
            {ih:{g|Cxt}{m,a|Trm}{xt:lvtyp g m a}{xc:lvcxt g}
                (Lt (f xt xc) (f t c))->C xt xc}C t c}
     {G|Cxt}{M,A|Trm}{t:lvtyp G M A}{c:lvcxt G}C t c;
```

The proof follows that of `WF_induction` in section 2.2.1. The lengths of `lvtyp`- and `lvcxt`-derivations, `lvtyp_ln` and `lvcxt_ln`, are defined in the obvious way using structural recursion, `lvtyp_elim` and `lvcxt_elim` respectively. The sum of these lengths

```
[wc_wt_ln [G|Cxt][M,A|Trm][t:lvtyp G M A][c:lvcxt G] : NN
                                           = add (lvcxt_ln c) (lvtyp_ln t)];
```

is the right shape to use as the function, `f`, in the well-founded induction principle above.

Finally we can prove the main lemma.

Proof of `lvtyp_lvcxt_vtyp`**.** Here is an informal outline. The proof goes by well-founded induction on the sum of the lengths of the assumed derivations, t of $G \vdash_{lvtyp} M : A$ and c of $G \vdash_{lvcxt}$. Then, by structural induction (for case analysis) on t, there are cases for the six constructors of `lvtyp`. I will give two cases.

lvtSort t derives $G \vdash_{lvtyp} s_1 : s_2$ and c derives $G \vdash_{lvcxt}$. We must show $G \vdash_{vtyp} s_1 : s_2$ knowing $\mathtt{ax}(s_1{:}s_2)$. By structural induction (again for case analysis) on c, there are two cases: `lvcNil` and `lvcCons`. The former is immediate by `vtSort` and `vcNil`. In the latter, $G = K[p{:}B]$ for some K, p and B. By

$$\dfrac{\dfrac{K \vdash_{vtyp} B : s}{K[p{:}B] \vdash_{vcxt}}\,\mathtt{vcCons}}{K[p{:}B] \vdash_{vtyp} s_1 : s_2}\,\mathtt{vtSort}$$

it suffices to show $K \vdash_{vtyp} B : s$. By the well-founded induction hypothesis, we need derivations lt of $K \vdash_{lvtyp} B : s$ and lc of $K \vdash_{lvcxt}$ such that

$$\mathtt{lngth}(lt) + \mathtt{lngth}(lc) < \mathtt{lngth}(t) + \mathtt{lngth}(c)$$

But c, which derives $K[p{:}B] \vdash_{lvcxt}$, contains immediate subderivations of $K \vdash_{lvtyp} B : s$ and $K \vdash_{lvcxt}$, so this case is complete.

lvtPi t derives $G \vdash_{lvtyp} \{n{:}B\}C : s_3$, and c derives $G \vdash_{lvcxt}$. We must show $G \vdash_{vtyp} \{n{:}B\}C : s_3$ knowing

sc : $\mathtt{rl}(s_1, s_2, s_3)$
l_prem : $G \vdash_{lvtyp} B : s_1$
r_prem : $G[p{:}B] \vdash_{lvtyp} [p/n]C : s_2$

By `vtPi` it suffices to show $G \vdash_{vtyp} B : s_1$ and $G[p{:}B] \vdash_{vtyp} [p/n]C : s_2$. Assuming the first of these for the moment, the second follows from the lemma `lvtyp_vcxt_vtyp` by r_prem and `vcCons`. To prove the first by the well-founded induction hypothesis requires a derivation lt of $G \vdash_{lvtyp} B : s_1$ such that

$$\texttt{lngth}(lt) + \texttt{lngth}(c) < \texttt{lngth}(t) + \texttt{lngth}(c)$$

But t, contains an immediate subderivation of $G \vdash_{lvtyp} B : s_1$ (in fact, l_prem), so this case is finished.

Formalizing this proof is delicate. Our relations do not carry the heights of their derivations in the judgement syntax, so in the case `lvtPi` above, we can't read off that l_prem is shorter than t. Several approaches suggest themselves to solve this problem. In this case, needing only to know that *immediate subderivations* are shorter than their parents, I used dependent elimination when doing the structural case analyses, so that the elimination step itself instantiated the well-founded induction hypothesis. For example in the `lvtPi` case, t is instantiated with `(Pi sc noccB l_prem r_prem)` which is apparently longer than `l_prem`. ∎

4.4.11 Abstract Conversion Revisited

It is remarkable how little about `cnv` is needed for the theory we have developed. In this section we explore a few simple properties that beta-conversion PTS have but are not provable for abstract conversion PTS without extra assumptions. Much as I would like to present a few extra assumptions that yield all the PTS properties we use, and a satisfying justification of the assumptions I have chosen, I cannot do this. The axiomatization of `cnv` to this point is very weak, and it is not at all clear what other properties the "real" `cnv` should have. In this section I'm guided mainly by what is needed for typechecking ECC in Chapter 5.

4.4.11.1 `cnv` and `Vclosed`

Let me first assume:

```
[cnv_Vclosed: {A,B|Trm}(cnv A B)->and (Vclosed A) (Vclosed B)];
```

This is a hygienic assumption: for our purposes, any relation that doesn't have such a property is just a badly defined version of one that does (see remark 3.2). It hasn't been needed yet because the typing judgement already enforces `Vclosed`. It probably isn't needed in what follows either, but it is not worth the trouble to find out.

4.4.11.2 A Converse to `sStartLem`

For beta-conversion PTS we can prove

$$G \vdash s_1 : s_2 \;\Rightarrow\; \mathtt{ax}(s_1{:}s_2)$$

using a generation lemma and the fact $s_1 \simeq s_2 \;\Rightarrow\; s_1 = s_2$. We cannot prove it in general because of derivations like

$$\dfrac{\dfrac{\mathtt{ax}(s{:}t)}{\vdash s : t}\,\textsc{Ax} \qquad \dfrac{\mathtt{ax}(u{:}r)}{\vdash u : r}\,\textsc{Ax} \qquad t \le u}{\vdash s : u}\,\textsc{tCnv}$$

This gives a flavor of how the relations `cnv` and `ax` can interact. For example, there may be fundamentally different derivations of the judgement $\vdash s : u$, some using the rule TCNV in a non-trivial way. Notice however, by `gts_gen_sort`, that a sort has a type if an only if it is a typedsort.

To prove this lemma we need an extra hypothesis, `cnv_ax_full`.

```
[cnv_ax_full =
  {s1,s2,t|SS}(ax s1 s2)->(cnv (sort s2) (sort t))->(typedsort t)->ax s1 t];
Goal gts_reflects_ax:
  {caf:cnv_ax_full}{G|Cxt}{s1,s2|SS}(gts G (sort s1) (sort s2))->ax s1 s2;
```

This is cheating grossly, as `cnv_ax_full` merely throws in enough axioms to make the lemma true. `cnv_ax_full` is not only philosophically annoying, but counterproductive in practice: for typechecking algorithms we want functional PTS, while `cnv_ax_full` forces us to have extra axioms in a system that might otherwise be presented functionally. We do not actually make the assumption `cnv_ax_full` above, and will not refer to it again.

4.4.11.3 Predicate Conversion

For PTS with beta-conversion we have a trivial but useful lemma

Lemma 4.5 (Predicate conversion) *for PTS with beta-conversion*

$$(\Gamma \vdash M : A \text{ and } A \simeq s) \;\Rightarrow\; \Gamma \vdash M : s\,.$$

Proof. $A \simeq s$, so $A \twoheadrightarrow s$, so $\Gamma \vdash M : s$ by predicate reduction. ■

The point is that we don't ask whether s has a type or not. Of course it's clear that A must contain s, so if s is a topsort, $A = s$ by the topsort lemmas; otherwise the conversion rule applies.

Conversion and Weak Head Normal Forms. The question is how to generalize this argument to abstract conversion, where we do not have that $A \leq s$ implies $A \twoheadrightarrow s$ or that $A \leq s$ implies A contains s. Both of these implications fail for ECC, our motivating example of abstract conversion. We have already assumed `cnvCR_pi` (section 4.4.9) which I called an internal CR property. I now assume three *external* CR properties: if A is related by $\leq$ to some weak-head normal form, then A has a weak-head normal form of the same shape.

```
[cnv_sort_character_l:
     {s|SS}{A|Trm}{c:cnv A (sort s)}Ex [t:SS] par_redn A (sort t)];
[cnv_sort_character_r:
     {s|SS}{B|Trm}{c:cnv (sort s) B}Ex [t:SS] par_redn B (sort t)];
[cnv_pi_character_l:
     {A,Bl,Br|Trm}{v|VV}{c:cnv A (pi v Bl Br)}
     Ex3 [u:VV][Al,Ar:Trm] par_redn A (pi u Al Ar)];
```

I consider these three properties to be unproblematically part of what a conversion relation is, and make them as general assumptions whose use we will not keep track of.

`cnv_sort_character_l`, for example, does not tell us that $t \leq s$, but $\leq$ contains $\simeq$, so from `cnv_sort_character_l` (and `cnv_Vclosed`) we can prove

```
Goal cnv_sort_Character_l:
   {s|SS}{A|Trm}{c:cnv A (sort s)}
   Ex [t:SS]and (par_redn A (sort t)) (cnv (sort t) (sort s));
```

and similarly `cnv_sort_Character_r` and `cnv_pi_Character_l`. We might need `cnv_pi_character_r` someday, but it hasn't come up yet. As conversion is only useful for types, and no type has weak head normal form of shape $[v{:}A]M$, no such assumptions for lambda are required. If I was formalizing ECC's sigma types, we would need similar assumptions for sigma.

Now we can attempt to prove Predicate Conversion for abstract conversion.

Proof attempt. $A \leq s$ so for some t, $A \twoheadrightarrow t \leq s$, and $G \vdash M : t$ by predicate reduction. If $s \in \mathrm{SS}_T$ we are done by the conversion rule. Otherwise what to do?

Conversion and Typedsorts. It is now clear that topsorts are a difficulty with abstract conversion. Unfortunately our motivating example, ECC, is of little help, since ECC has no topsorts! Here are four possible anti-topsort properties arranged in roughly increasing strength:

```
[cnv_preserves_typedsort_dn =
   {s1,s2|SS}(cnv (sort s1) (sort s2))->(typedsort s2)->typedsort s1];
[cnv_preserves_typedsort_up =
   {s1,s2|SS}(cnv (sort s1) (sort s2))->(typedsort s1)->typedsort s2];
[cnv_range_typedsort =
   {s1,s2|SS}(cnv (sort s1) (sort s2))->
   or (is_tt (SSeq s1 s2)) (typedsort s2)];
[cnv_only_typedsort =
   {s1,s2|SS}(cnv (sort s1) (sort s2))->
   or (is_tt (SSeq s1 s2)) (and (typedsort s1) (typedsort s2))];
```

Clearly every beta-conversion PTS has all these properties. It is easy to see some relations between them.

```
Goal crt_cptd_cpt:
     {crt:cnv_range_typedsort}{cptd:cnv_preserves_typedsort_dn}
     cnv_only_typedsort;
Goal cpt_and3: {cpt:cnv_only_typedsort}and3 (cnv_range_typedsort)
                                            (cnv_preserves_typedsort_dn)
                                            (cnv_preserves_typedsort_up);
```

`cnv_only_typedsort` is the strongest, and we will need its whole strength to derive a syntax directed characterization of `gts` in section 5.1.3, but I prefer to think of it as the conjunction of `cnv_range_typedsort` and `cnv_preserves_typedsort_dn`.

Now we can prove:

```
Goal gtsPC:
     {crt:cnv_range_typedsort}
     {G|Cxt}{M,A|Trm}(gts G M A)->{s|SS}(cnv A (sort s))->gts G M (sort s);
```

Proof. $A \leq s$ so for some t, $A \twoheadrightarrow t \leq s$, and $G \vdash M : t$ by predicate reduction. By `cnv_range_typedsort` either $t = s$ (and we are done), or s is a typedsort, and we can use rule TCNV to finish. ∎

Remark 4.6 (on Topsorts) *Every PTS with `cnv_range_typedsort` can be extended to a new PTS that is, in a sense, conservative over the original one, and has property `cnv_only_typedsort`. The idea is to "top off" every topsort of the original PTS with a completely fresh sort symbol.*

Definition 4.7 *A sort is* isolated *if it is a topsort that participates in no rules, and converts only with itself. A sort, s, is a* pseudo topsort *if whenever $ax(s{:}T)$ then T is an isolated sort.*

There is a lemma that if $G \vdash M : A$ and t is a psuedo topsort then $t \in M \Rightarrow t = M$ and $t \in A \Rightarrow t = A$.

Now let $\vdash_t$ be a PTS with some distinguished topsort t, and $\vdash_T$ the same PTS, but with t made into a pseudo topsort by adding an isolated sort T and an axiom $ax(t{:}T)$. We can prove (using *cnv_range_typedsort)*

$$G \vdash_T M : A,\ t \notin G,\ t \notin M \quad \Leftrightarrow \quad G \vdash_t M : A$$

I haven't checked this in LEGO.

4.5 Properties of Arbitrary PTS With Beta-Conversion

Let us now use conv for the conversion relation of PTS, as the following would be messy to state and prove for our abstract conversion relation. So far we have the abstract conversion relation, cnv, and several of its properties, as assumptions in the LEGO context; i.e. as declared variables. We can tell LEGO to cut in the defined relation conv, and its proved properties, for these declared variables.

```
Cut [cnv=conv]
    [cnv_refl=conv_refl]
    [cnv_trans=conv_trans]
    [psub_resp_cnv=psub_resp_conv]
    [cnv_red1=conv_red1]
    [cnv_red1_sym=conv_red1_sym]
    [cnvCR_pi=cOnvCR_pi_lem]
    [cnv_Vclosed=conv_Vclosed_lem]
    [cnv_sort_character_l=conv_sort_character_l]
    [cnv_sort_character_r=conv_sort_character_r]
    [cnv_pi_character_l=conv_pi_character_l];
```

(The Cut command is described in section 2.1.4.) LEGO verifies that conv really has the properties assumed for cnv.

4.5.1 The Typing Lemma

I want to mention, because of its beauty, a lemma that explains the possible variations in the type of a given term. This lemma is originally due to Luo for ECC [Luo90a], where it explains cumulativity. A very similar result was found independently by Jutting [vBJ93] in his work to prove strengthening for PTS. I state it informally, and have not formalized the proof.

Lemma 4.8 (Typing Lemma) *If* $\Gamma \vdash a : A$ *and* $\Gamma \vdash a : B$ *then*

either $A \simeq B$

or $\exists s_A, s_B, n \geq 0, C_1, \ldots, C_n\ .$
$$A \twoheadrightarrow \{x_1{:}C_1\}\ldots\{x_n{:}C_n\}s_A \text{ and } B \twoheadrightarrow \{x_1{:}C_1\}\ldots\{x_n{:}C_n\}s_B$$

4.5.2 Strengthening

We have formalized the proof of strengthening in [vBJMP94].

```
Goal gts_strengthening:
     {Gamma|Cxt}{c,C,d,D|Trm}{Delta:Cxt}
     {q|PP}{noccd:is_ff (poccur q d)}
           {noccD:is_ff (poccur q D)}
           {noccDelta:is_ff (POCCUR q Delta)}
     {premD:gts (append Delta (CONS (Gb q C) Gamma)) d D}
      gts (append Delta Gamma) d D;
```

This project develops several systems similar to PTS, and by "cutting and pasting" existing proofs we have been able to effectively reuse some of the vast amount of work expended so far.

LEGO uses strengthening in its `Discharge` command.

4.6 Functional PTS

We are still using beta-conversion as the conversion relation of PTS.

Definition 4.9 *A PTS is* functional *if*

- $ax(s{:}t)$ *and* $ax(s{:}u)$ *implies* $t = u$*, and*
- $rl(s_1, s_2, t)$ *and* $rl(s_1, s_2, u)$ *implies* $t = u$*.*

Formally:

```
[Functional =
       and ({s,t,u|SS}(ax s t)->(ax s u)->is_tt (SSeq t u))
           ({s1,s2,t,u|SS}(rl s1 s2 t)->(rl s1 s2 u)->is_tt (SSeq t u))];
```

Functional PTS are well behaved and are, perhaps, the only ones that occur in practice. For a functional PTS, `ax` and `rl` are the graphs of partial functions, but we do not necessarily have procedures to compute these functions.

4.6.1 Uniqueness of Types

It is well known that functional PTS have uniqueness of types up to `conv`:

```
[conv_unique_types =
     {G|Cxt}{M,A|Trm}(gts G M A)->{B|Trm}(gts G M B)->conv A B];
Goal types_unicity: {f:Functional}conv_unique_types;
```

This is easily proved by structural induction. The proof uses symmetry of `conv`, and the CR property that if $s_1 \simeq s_2$ then $s_1 = s_2$.

4.6.2 Subject Expansion

Any PTS with uniqueness of types also has a *subject expansion* property:

```
Goal subject_expansion:
     {UT:conv_unique_types}
     {G|Cxt}{M,N,A,B|Trm}{r:par_redn M N}{j:gts G N A}{j':gts G M B}
     gts G M A;
```

Proof. From $G \vdash M : B$, $M \twoheadrightarrow N$ and SR, we have $G \vdash N : B$, so $A \simeq B$ by uniqueness of types. By type correctness, $G \vdash A : s$ or A is a sort. In the former case, we are done by the conversion rule; in the latter it must be that $B \twoheadrightarrow A$, and we are done by predicate reduction. ■

In general, non-functional PTS do not have subject expansion: a reduct of a term may have more types than the original term. There are examples of different ways this can happen in [vBJMP94].

4.7 Expansion Postponement

There is an open question about PTS called Expansion Postponement (EP) (see [vBJMP94]), proposed by Henk Barendregt. In this section I define the problem, and point out some of the difficulties that arise.

4.7.1 Two Different Expansion-Free Systems

4.7.1.1 The system $\vdash_{red}$

Informally I write $\vdash_{conv}$ for the system `gts` with the abstract conversion relation, `cnv` instantiated to beta-conversion, `conv` ($\vdash_{conv}$ is the PTS relation of [Bar92]); I write $\vdash_{red}$ for `gts` with the abstract conversion relation instantiated to reduction, `par_redn`. Thus the "conversion" rule of $\vdash_{red}$ is

$$\text{TRED} \qquad \frac{\Gamma \vdash_{red} M : A \qquad \Gamma \vdash_{red} B : s}{\Gamma \vdash_{red} M : B} \qquad A \twoheadrightarrow B$$

Since `par_redn` is contained in `conv`, $\vdash_{red}$ is contained in $\vdash_{conv}$. Since `conv` satisfies all the assumptions on `cnv` in sections 4.3 and 4.4.9, $\vdash_{conv}$ has every property we have proved for `gts`. On the other hand `par_redn` fails to have property `cnv_red1_sym` of section 4.4.9, so while $\vdash_{red}$ has the substitution lemma and type correctness, we have no reason to believe it has SR or PR.

Example 4.10 ($\vdash_{red}$ does not have SR for CC.) *Let $G = [A{:}\star][F{:}\{Y{:}\star\}Y]$ be a context in CC, and write I for $[X{:}\star]X$. $G \vdash_{red} F\,(I\,A) : (I\,A)$ is derivable, but, contracting the redex in the subject, $G \vdash_{red} F\,A : (I\,A)$ is not derivable; in fact $G \vdash_{red} F\,A : A$ is the only $\vdash_{red}$ judgement derivable for G and $F\,A$, because every type appearing in the "natural" derivation*

$$\dfrac{\begin{array}{c}\vdots\\ G \vdash_{red} F : \{Y{:}\star\}Y\end{array} \qquad \begin{array}{c}\vdots\\ G \vdash_{red} A : \star\end{array}}{G \vdash_{red} F\,A : A}\ \text{APP}$$

is already in normal form.

On the other hand, we can hope $\vdash_{red}$ has *weak* SR:

$$[G \vdash_{red} M : A \text{ and } M \twoheadrightarrow N] \Rightarrow \exists B\,[A \twoheadrightarrow B \text{ and } G \vdash_{red} N : B]$$

I do not know how to prove this.

4.7.1.2 The system $\vdash_{RED}$

The relation $\vdash_{conv}$ has as its conversion rule

$$\text{tCONV} \qquad \dfrac{\Gamma \vdash_{conv} M : A \qquad \Gamma \vdash_{conv} B : s}{\Gamma \vdash_{conv} M : B} \qquad A \simeq B$$

Let us split this rule for converting the predicate of a judgement into two, one for expanding the predicate and one for reducing the predicate. Knowing that $\vdash_{conv}$ has PR suggests it is safe to drop the type correctness premise in the rule for reducing the predicate, giving a relation, $\vdash_{er}$, having the same rules as $\vdash_{conv}$ except for tCONV, which is replaced by the rules

$$\text{tRED} \qquad \dfrac{\Gamma \vdash_{er} M : A}{\Gamma \vdash_{er} M : B} \qquad A \twoheadrightarrow B$$

$$\text{tEXP} \qquad \dfrac{\Gamma \vdash_{er} M : A \qquad \Gamma \vdash_{er} B : s}{\Gamma \vdash_{er} M : B} \qquad B \twoheadrightarrow A$$

In fact this suggestion is correct, and, using the Church-Rosser and Predicate Reduction properties of $\vdash_{conv}$, it is easy to show [vBJMP94] that $\vdash_{conv}$ and $\vdash_{er}$ have the same derivable judgements: for all PTS,

$$\forall \Gamma, M, A\ .\ \Gamma \vdash_{conv} M : A \quad \Leftrightarrow \quad \Gamma \vdash_{er} M : A\,.$$

Now consider a relation $\vdash_{RED}$ having the same rules as $\vdash_{er}$, but without the expansion rule tEXP. The difference between $\vdash_{RED}$ and $\vdash_{red}$ is that the the conversion rule of $\vdash_{red}$, TRED on page 87, has the type correctness premise that we dropped from rule TRED; otherwise $\vdash_{red}$ and $\vdash_{RED}$ are the same. However $\vdash_{RED}$ is not obviously an instance of the relation `gts`, and *a priori* we know nothing about it. In fact, by the same proof as for `gts`, it has the substitution lemma, but the proof of type correctness used for `gts` (which works for $\vdash_{red}$) fails for $\vdash_{RED}$ because it is precisely the type correctness condition that we dropped from the rule TRED. On the other hand $\vdash_{RED}$ clearly has PR, whereas $\vdash_{red}$ is not known to. (This suggests an interesting duality between type correctness and PR.) Finally, as in example 4.10 $\vdash_{RED}$ does not in general have SR; and I hope, but cannot prove, $\vdash_{RED}$ has weak SR.

Comparing the systems. We know that every $\vdash_{red}$-derivation is a $\vdash_{RED}$-derivation, every $\vdash_{RED}$-derivation is an $\vdash_{er}$-derivation, and to every $\vdash_{er}$-derivation there corresponds a $\vdash_{conv}$-derivation of the same judgement. Somewhat incorrectly I write

$$\vdash_{red} \subseteq \vdash_{RED} \subseteq \vdash_{er} = \vdash_{conv} .$$

4.7.2 Expansion Postponement defined

Definition 4.11 (Expansion Postponement) *Let $\vdash_R$ be, for each PTS (PL, ax, rl), a subrelation of $\vdash_{conv}$ (i.e. every $\vdash_R$ judgement is a $\vdash_{conv}$ judgement) for which the following rule is admissible*

$$\text{TR} \qquad \frac{\Gamma \vdash_R M : A \qquad \Gamma \vdash_R B : s \qquad A \twoheadrightarrow B}{\Gamma \vdash_R M : B}$$

that is, whenever $\Gamma \vdash_R M : A$, $\Gamma \vdash_R B : s$, and $A \twoheadrightarrow B$, then $\Gamma \vdash_R M : B$. A PTS has R-expansion postponement *(R-EP) iff*

$$\forall G, M, A \,.\, G \vdash_{conv} M : A \;\Rightarrow\; \exists B \,.\, A \twoheadrightarrow B \text{ and } G \vdash_R M : B \,.$$

Some easy observations

R-EP **implies $\vdash_R$ has weak Subject Reduction,** for if $G \vdash_R M : A$ and $M \twoheadrightarrow N$ then $G \vdash_{conv} N : A$, and by R-EP $G \vdash_R N : B$ for some B with $A \twoheadrightarrow B$.

R-EP **implies $\vdash_R$ type correctness,** for if $G \vdash_R M : A$ then $G \vdash_{conv} M : A$, and by $\vdash_{conv}$ type correctness, A is a sort or $G \vdash_{conv} A : s$ for some s. In the first case we are finished; in the second, by R-EP, $G \vdash_R A : s$.

R-EP **implies $\vdash_R$ has Predicate Reduction,** for if $G \vdash_R M : A$ and $A \twoheadrightarrow B$ then by $\vdash_R$ type correctness, either A is a sort (and we are done) or $G \vdash_R A : s$ for some s. In the latter case, $G \vdash_R B : s$ by $\vdash_R$ weak Subject Reduction, so $G \vdash_R M : B$ by rule TR.

4.7.3 Expansion Postponement for $\vdash_{red}$ and $\vdash_{RED}$

For $\vdash_{red}$ and $\vdash_{RED}$, weak SR is equivalent to EP.

Lemma 4.12 *If $\vdash_{red}$ (resp. $\vdash_{RED}$) has weak SR, then it has EP.*

Proof. Let $\vdash_R$ stand for $\vdash_{red}$ (resp. $\vdash_{RED}$). By induction on the structure of a derivation of $G \vdash_{conv} M : A$, show that for some B, $A \twoheadrightarrow B$ and $G \vdash_R M : B$. I do two cases.

Lda The $\vdash_{conv}$-derivation has shape

$$\dfrac{\begin{array}{c}\vdots\\ G[x{:}A] \vdash_{conv} b : B\end{array} \qquad \begin{array}{c}\vdots\\ G \vdash_{conv} \{x{:}A\}B : s\end{array}}{G \vdash_{conv} [x{:}A]b : \{x{:}A\}B}\ \textsc{Lda}$$

By induction hypothesis

> l_ih : $\exists C \,.\, B \twoheadrightarrow C$ and $G[x{:}A] \vdash_R b : C$
> r_ih : $G \vdash_R \{x{:}A\}B : s$

Now we claim $G \vdash_R [x{:}A]b : \{x{:}A\}C$; by the LDA rule for $\vdash_R$ and l_ih it only remains to show $G \vdash_R \{x{:}A\}C : s$, which follows from r_ih by weak SR, which we have assumed for $\vdash_R$.

This is the essential difficulty in attempts to prove EP; in the LDA rule of $\vdash_{conv}$, B appears on the right of the colon in one premise and on the left of the colon in the other premise. However, in expansion-free systems we can only hope for induction hypotheses "up to reduction" on the right of the colon, and weak SR is the only obvious way to bring the two parts together. The LDA rule is not the only problem however, for in the conversion rule B moves from the right of the colon in the conclusion to the left of the colon in the right premise.

tCnv The $\vdash_{conv}$-derivation has shape

$$\dfrac{\begin{array}{c}\vdots\\ G \vdash_{conv} M : A\end{array} \qquad \begin{array}{c}\vdots\\ G \vdash_{conv} B : s\end{array} \qquad A \simeq B}{G \vdash_{conv} M : B}\ \textsc{tCnv}$$

By induction hypothesis

> l_ih : $\exists C \,.\, A \twoheadrightarrow C$ and $G \vdash_R M : C$
> r_ih : $G \vdash_R B : s$

By CR let D be such that $C \twoheadrightarrow D$ and $B \twoheadrightarrow D$. By weak SR $G \vdash_R D : s$, so both systems, $\vdash_{red}$ and $\vdash_{RED}$, prove $G \vdash_R M : D$ where $B \twoheadrightarrow D$ as required. ∎

Lemma 4.13 *For functional* PTS*, if* $\vdash_{red}$ *(resp.* $\vdash_{RED}$*) has predicate reduction,* `decideTypedsort` *and* `decideTypedsort_type_correctness` *(section 4.4.8), then it has EP*[4]*.*

It is frustrating to observe that $\vdash_{red}$ does have `decideTypedsort_type_correctness`, and $\vdash_{RED}$ does have PR, but neither system obviously has both properties.

Proof. Let $\vdash_R$ stand for $\vdash_{red}$ (resp. $\vdash_{RED}$). Show, by induction on the length of a derivation of $G \vdash_{conv} M : A$, that for some B, $A \twoheadrightarrow B$ and $G \vdash_R M : B$. I do two cases.

Lda The $\vdash_{conv}$-derivation has shape

$$\dfrac{\begin{matrix}\vdots \\ G[x{:}A] \vdash_{conv} b : B\end{matrix} \qquad \begin{matrix}\vdots\ \mathrm{d} \\ G \vdash_{conv} \{x{:}A\}B : s\end{matrix}}{G \vdash_{conv} [x{:}A]b : \{x{:}A\}B}\ \textsc{Lda}$$

By induction hypothesis

l_ih : $\exists C \,.\, B \twoheadrightarrow C$ and $G[x{:}A] \vdash_R b : C$
r_ih : $G \vdash_R \{x{:}A\}B : s$

Now we claim $G \vdash_R [x{:}A]b : \{x{:}A\}C$; by the LDA rule for $\vdash_R$ and l_ih it only remains to show $G \vdash_R \{x{:}A\}C : t$ for some sort t. As in lemma 4.12 this is the essential difficulty. Here, not having weak SR, we push the problem back into $\vdash_{conv}$.

By type correctness of $\vdash_R$ applied to l_ih, we have

tc : $G[x{:}A] \vdash_R C : s_C$

(Notice, it cannot be that C is a topsort, for then C, being atomic, would occur in B, which is forbidden by the topsort theorem on the right premise.) By a generation lemma, derivation d contains *subderivations* of

l_pi : $G \vdash_{conv} A : s_1$
r_pi : $G[x{:}A] \vdash_{conv} B : s_2$

where $\mathtt{rl}(s_1, s_2, s)$. By induction hypothesis on l_pi

l_pi_ih : $G \vdash_R A : s_1$

If we can only show $s_C = s_2$ then we are done by the PI rule of $\vdash_R$ using l_pi_ih and tc. From tc we know $G[x{:}A] \vdash_{conv} C : s_C$. By subject expansion (using functionality and r_pi) $G[x{:}A] \vdash_{conv} B : s_C$. Finally, by uniqueness of types, and r_pi, $s_C = s_2$ as required.

---

[4] At a workshop in Båstad, Sweden (May 1992), I incorrectly claimed this argument proves "EP for functional PTS", because I failed to distinguish between the systems $\vdash_{red}$ and $\vdash_{RED}$. Thanks to Eric Poll for pointing out the error.

tCnv The $\vdash_{conv}$-derivation has shape

$$\dfrac{\begin{array}{c}\vdots\\ G \vdash_{conv} M : A\end{array} \qquad \begin{array}{c}\vdots\\ G \vdash_{conv} B : s\end{array} \qquad A \simeq B}{G \vdash_{conv} M : B}\ \text{tCNV}$$

By induction hypothesis

l_ih : $\exists C\,.\, A \twoheadrightarrow C$ and $G \vdash_R M : C$

By CR let D be such that $C \twoheadrightarrow D$ and $B \twoheadrightarrow D$. By PR of $\vdash_R$, $G \vdash_R M : D$ where $B \twoheadrightarrow D$ as required. ∎

Remark 4.14 *Reading this proof carefully, you will see we use more than structural induction (i.e. immediate subderivations have the induction property), and less than length induction (i.e. all smaller derivations have the induction property). In fact subderivation induction is perfect; all subderivations have the induction property. This observation may be useful in formalizing the argument above, because it uses a "strong" generation lemma, that requires* `apts` *to prove. Recall that* `apts`*, defined by infinitely branching trees, does not have an easily definable length or height function.*

Chapter 5

Semi-Full and Cumulative PTS: Typechecking ECC

What has come before is about beauty; what comes next is about technology. We want an algorithm for typechecking a class of PTS that includes the three type systems of LEGO, and we'll do grungy things to get it. We first show how to typecheck a class of PTS under many obscure assumptions about the abstract relation `cnv` ($\leq$) and about the axioms and rules of the PTS (section 5.1). Next (section 5.2) we give a parameterized class of conversion (`cnv`) relations, called cumulativity, that satisfy the restrictions on `cnv` of section 5.1. Finally, section 5.3 concretely presents the axioms and rules of a cumulative PTS that is very close to Luo's ECC and satisfies the remaining assumptions of section 5.1. We will thus have a verified typechecking algorithm for our variant of ECC depending only on the assumption that ECC is normalizing, which is not provable in ECC.

5.1 Typechecking Functional, Semi-Full PTS

5.1.1 Introduction to Type Checking

The Type Checking Problem (TCP) for a relation $\vdash_R$ is, given G, M and A, to decide whether or not $G \vdash_R M : A$. Since (G, M) may have different types, this is usually solved by computing some type, B, for G and M, and checking that $B \leq A$ and that A is well-formed in some way, e.g. that $G \vdash_R A : s$ for some sort s. This suggests the Type Synthesis Problem (TSP): given G and M, compute some type, B, for G and M, if possible, and fail if no such type exists.

Our strategy for solving TSP for some class of PTS is to find an inductive relation that is equivalent in some way to PTS, but deterministic in the sense that, given G and M, there will be no serious choice of which rule is the root of any derivation over G and M, or of what type is derived by that rule. Such a rule application will have premises which need to be satisfied, and the subjects of these subgoals should be determined by the subject, G and M, of the previous goal. Further, all the side conditions of the rules should be solvable. In [vBJMP94] we give definitions of *nearly syntax directed* (roughly, only one rule can be used over any G and M) and *syntax directed* (roughly, only one rule can be used over any G and M, and it can be used in only one way). These definitions are not really satisfactory to capture the intuitive idea, and anyway a completely deterministic system is not required (see section 5.1.4), so I prefer to remain informal on this point. Of course it is not clear that computing just any type will be good enough, so we want the deterministic relation to select a type which is *principal* in some sense. I have hinted at how such a computation for TSP might unfold, but there is one remaining issue: termination. We will have a formal induction whose measure proves termination of a computation for TSP.

There are difficulties in transforming `gts`[1] into a syntax directed presentation that can be used for solving TSP. Only the conversion rule is not syntax directed, so our plan is to eliminate it by permuting it through all the other rules. However the lambda rule prevents this program from being carried out. In section 5.1.1.1 I indormally clarify the difficulty (there is also discussion of this point in [vBJMP94]); in the rest of this chapter I show how to solve this difficulty for a class of PTS including all three of LEGO's logics. First we take advantage of a special property, semi-fullness (section 5.1.2.1), to change the troublesome right premise of the LDA rule (system `sfts` in section 5.1.2.2), then eliminate the conversion rule (system `sdsf` in section 5.1.3). We show that `sdsf` characterizes the judgements of `gts` (section 5.1.3.1), and introduce the notion of principal types in section 5.1.4. From this it follows (section 5.1.5) that TCP for `gts` is decidable assuming that TSP for `sdsf` is decidable and that some classes of side conditions are decidable. In section 5.1.6, we show that TSP for `sdsf` is decidable, assuming that some classes of side conditions are decidable. Putting these results together we get TS and TC for a class of PTS.

5.1.1.1 Expansion Postponment and Typechecking

This section is explanatory, and not referred to in the formal development that follows.

Our approach to typechecking is to use the shape of the subject of a judgement to determine which inference rule to use in deriving that judgement. Our system $\vdash$ of table 4–2, formalized

---

[1] For an efficient algorithm, we should start with the system of locally valid contexts, `lvtyp`, of section 4.4.10.2, but that would be too grungy even for this section, so we start with the system `gts`, which has many nice properties.

in table 4–3, is almost satisfactory for this, except for the conversion rule[2]. For example, a derivation of a judgement with shape $G \vdash \{x{:}A\}B : C$ must end with either the PI rule or the conversion rule; a derivation of a judgement with shape $\bullet \vdash s : C$ must end with either the AX rule or the conversion rule; a derivation of a judgement with shape $G[p{:}A] \vdash s : C$ must end with either the START rule or the conversion rule. Similarly, every shape of subject determines which rule must be used to derive it, up to possible use of the conversion rule. Since the conversion rule doesn't change the subject between its major premise and its conclusion, the shape of the subject gives no information on when to use this rule.

Our technique to remove this ambiguity is due originally to Martin-Löf [Mar71b], and made known to me by [Hue89]. Since the subject of a judgement determines its predicate, at best, only up to conversion, it is clear that the conversion rule must be available at the end of a derivation to "fix up the type" if necessary. We will try to permute the conversion rule down through every premise of every other rule in order to move all instances of TCONV to the end of derivations. We can't quite do this, as some rules require particular shapes in premises. For example the START rule

$$\text{START} \qquad \frac{\Gamma \vdash A : s}{\Gamma[p{:}A] \vdash p : A} \qquad p \notin \Gamma$$

requires the type of A in the premise to be a sort, and TCONV might be used to meet this requirement, as in

$$\cfrac{\cfrac{G \vdash A : C \qquad G \vdash s : t \qquad C \simeq s}{\Gamma \vdash A : s}\ \text{TCONV}}{\Gamma[p{:}A] \vdash p : A}\ \text{START}$$

However, we don't need arbitrary conversion before the START rule, only reduction to a sort, so permuting conversion past the START rule leaves a residue behind. The rule becomes:

$$\frac{\Gamma \vdash A : X \qquad X \twoheadrightarrow s}{\Gamma[p{:}A] \vdash p : A} \qquad p \notin \Gamma$$

Notation 5.1 *I will write* $\Gamma \vdash A : X \twoheadrightarrow s$ *instead of* $\Gamma \vdash A : X$ *and* $X \twoheadrightarrow s$ *.*

Similarly the right premises of both weakening rules and both premises of the PI rule are modified to allow reduction in the premise.

[2] For the present informal discussion I will use β-conversion, $\simeq$, rather than abstract conversion, $\leq$.

The APP rule is more interesting, as conversion might be used before both premises, as in (I omit some premises):

$$\dfrac{\dfrac{\Gamma \vdash M : X \quad \ldots \quad X \simeq \{x{:}A\}B}{\Gamma \vdash M : \{x{:}A\}B}\;\textsc{tConv} \qquad \dfrac{\Gamma \vdash N : Y \quad \ldots \quad Y \simeq A}{\Gamma \vdash N : A}\;\textsc{tConv}}{\Gamma \vdash M\,N : [N/x]B}\;\textsc{App}$$

There are some inessential choices to be made in permuting conversion through both premises. For this discussion, replace APP with:

$$\dfrac{\Gamma \vdash M : X \twoheadrightarrow \{x{:}A\}B \qquad \Gamma \vdash N : Y \simeq A}{\Gamma \vdash M\,N : [N/x]B}$$

Notice that $\{x{:}A\}B$ is a weak-head-normal form, so we could use weak-head reduction, or, non-deterministically, parallel reduction for the left premise. For the right premise, both Y and A are determined by other parts of the derivation.

Finally the LDA rule. It is easy to see that conversion is never needed before the right premise. Conversion might be used before the left premiss in the following way:

$$\dfrac{\dfrac{\Gamma[p{:}A] \vdash [p/x]M : X \quad \ldots \quad X \simeq [p/y]B}{\Gamma[p{:}A] \vdash [p/x]M : [p/y]B}\;\textsc{tConv} \qquad \Gamma \vdash \{y{:}A\}B : s}{\Gamma \vdash [x{:}A]M : \{y{:}A\}B}\;\textsc{Lda}$$

By analogy with the discussion above, we want to replace LDA with

$$\dfrac{\Gamma[p{:}A] \vdash [p/x]M : X \twoheadrightarrow [p/y]B \qquad \Gamma \vdash \{y{:}A\}B : s}{\Gamma \vdash [x{:}A]M : \{y{:}A\}B} \qquad \begin{array}{r} p \notin M \\ p \notin B \end{array}$$

Now the the conversion rule can be eliminated, since, by construction, it is not needed before any premiss of any rule. This gives the system I will call $\vdash_{bad}$ (table 5–1). We have the following lemma characterizing $\vdash_{bad}$ in terms of $\vdash_{RED}$ of section 4.7.1.2 (see section 1.3 in [vBJMP94]):

Lemma 5.2 ($\vdash_{bad}$ and $\vdash_{RED}$)

- $\Gamma \vdash_{bad} a : A \;\Rightarrow\; \Gamma \vdash_{RED} a : A$
- $\Gamma \vdash_{RED} a : A \;\Rightarrow\; \exists A_0\,[A_0 \twoheadrightarrow A \textit{ and } \Gamma \vdash_{bad} a : A_0\,]$

Recall that $\vdash_{RED}$ is sound for $\vdash$, but that its completeness for $\vdash$ is the Expansion Postponment problem for $\vdash_{RED}$. Thus $\vdash_{bad}$ is sound for $\vdash$, but we don't know how to prove its completeness, and the difficulty in the completeness proof (section 4.7.3) is caused by the right premise of the LDA rule.

AX	$\bullet \vdash_{bad} s_1 : s_2$	$\mathtt{ax}(s_1{:}s_2)$
START	$\dfrac{\Gamma \vdash_{bad} A : X \twoheadrightarrow s}{\Gamma[p{:}A] \vdash_{bad} p : A}$	$p \notin \Gamma$
vWEAK	$\dfrac{\Gamma \vdash_{bad} q : C \qquad \Gamma \vdash_{bad} A : X \twoheadrightarrow s}{\Gamma[p{:}A] \vdash_{bad} q : C}$	$p \notin \Gamma$
sWEAK	$\dfrac{\Gamma \vdash_{bad} s : C \qquad \Gamma \vdash_{bad} A : X \twoheadrightarrow s}{\Gamma[p{:}A] \vdash_{bad} s : C}$	$p \notin \Gamma$
PI	$\dfrac{\Gamma \vdash_{bad} A : X \twoheadrightarrow s_1 \qquad \Gamma[p{:}A] \vdash_{bad} [p/x]B : Y \twoheadrightarrow s_2}{\Gamma \vdash_{bad} \{x{:}A\}B : s_3}$	$p \notin B$, $\mathtt{rl}(s_1, s_2, s_3)$
LDA	$\dfrac{\Gamma[p{:}A] \vdash_{bad} [p/x]M : X \twoheadrightarrow [p/y]B \qquad \Gamma \vdash_{bad} \{y{:}A\}B : s}{\Gamma \vdash_{bad} [x{:}A]M : \{y{:}A\}B}$	$p \notin M$ $p \notin B$
APP	$\dfrac{\Gamma \vdash_{bad} M : X \twoheadrightarrow \{x{:}A\}B \qquad \Gamma \vdash_{bad} N : Y \simeq A}{\Gamma \vdash_{bad} M\,N : [N/x]B}$	

Table 5–1: Incomplete and non syntax-directed presentation of PTS

In any case, $\vdash_{bad}$ is not syntax directed, as the subject of the right premise of the LDA rule depends on the non deterministic reduction in the left premise. We can make a syntax directed system (call it $\vdash_{worse}$) by replacing the LDA rule of $\vdash_{bad}$ with:

$$\frac{\Gamma[p{:}A] \vdash_{worse} [p/x]M : [p/y]B \qquad \Gamma \vdash_{worse} \{y{:}A\}B : s}{\Gamma \vdash_{worse} [x{:}A]M : \{y{:}A\}B} \quad \begin{array}{r} p \notin M \\ p \notin B \end{array}$$

$\vdash_{worse}$ is obviously sound for $\vdash_{bad}$, hence for $\vdash$, and we can construct a sound typechecker for a large class of PTS based on it. In [Pol92] I show by example that $\vdash_{worse}$ is not in general complete for $\vdash_{bad}$, and even it were, we don't know how to show that $\vdash_{bad}$ is complete for $\vdash$. These two difficulties are both caused by the structural feature of PTS that the subject of the right premise of the LDA rule is not determined by the subject of its conclusion. Now we show how to fix this problem for a class of PTS.

5.1.2 Towards a Syntax Directed System: Fixing the Lambda Rule

5.1.2.1 Semi-Full PTS

A PTS is called *full* iff for all s_1, s_2 there exists s_3 with $\mathtt{rl}(s_1, s_2, s_3)$. In full PTS the troublesome second premise of the LDA-rule can be simplified. Focus on the LDA-rule:

$$\text{LDA} \qquad \frac{\Gamma[p{:}A] \vdash [p/x]M : [p/y]B \qquad \Gamma \vdash \{y{:}A\}B : s}{\Gamma \vdash [x{:}A]M : \{y{:}A\}B} \qquad p \notin M,\ p \notin B$$

The purpose of premise $\Gamma \vdash \{y{:}A\}B : s$ is to assure type correctness. But we know from the premise $\Gamma[p{:}A] \vdash [p/x]M : [p/y]B$ that $\Gamma \vdash A : s_A$ for some s_A. As long as $\Gamma[p{:}A] \vdash [p/y]B : s_B$ for some s_B (e.g. by type correctness), for full PTS we conclude there exists s with $\mathtt{rl}(s_A, s_B, s)$, so $\{x{:}A\}B$ is well typed. This suggests replacing the right premise of the LDA-rule by the requirement that B is not a topsort, or, making a positive statement, that $B \in \mathtt{SS}$ implies $\mathtt{ax}(B{:}s_B)$ for some s_B. We can generalize this idea somewhat beyond full PTS.

Definition 5.3 (Semi-Full) *A PTS is* semi-full *iff*

$$\forall s_1 \,.\, (\exists s_2, s_3 \,.\, rl(s_1, s_2, s_3)) \;\Rightarrow\; \forall s_2\, \exists s_3 \,.\, rl(s_1, s_2, s_3) \,.$$

Formally:[3]

```
[semiFull = {s1|SS}(Ex2 [s2,s3:SS]rl s1 s2 s3)->{s2:SS}Ex [s3:SS]rl s1 s2 s3];
```

While the Pure Calculus of Constructions, CC, and various extensions with type universes are full (ECC is full, as we will trivially show in section 5.3), the Edinburgh Logical Framework, λP, is semi-full. CC and λP are the only semi-full systems in the λ-cube.

To the best of my knowledge, this definition first appeared in [Pol92] where I used it to give a syntax directed presentation of a class of type theories including CC and λP. That paper, in improved form, is published as a section in [vBJMP94]. Here I present a similar development, extended to a class of type systems including ECC as well. The concept is also used, for different purposes, in [Geu93], where it is credited to [vBJMP94].

```
Inductive [sfts:Cxt->Trm->Trm->Prop]   NoReductions   Constructors
[sfAx:{s1,s2|SS}                                              {sc:ax s1 s2}
         sfts nilCxt (sort s1) (sort s2)]

[sfStart:{G|Cxt}{A|Trm}{s|SS}{p|PP}                 {noccG:is_ff (Poccur p G)}
      {prem:sfts G A (sort s)}
     (****************************************)
         sfts (CONS (Gb p A) G) (par p) A]

[sfvWeak:{G|Cxt}{D,A|Trm}{s|SS}{n,p|PP}             {noccG:is_ff (Poccur p G)}
      {l_prem:sfts G (par n) D}
      {r_prem:sfts G A (sort s)}
     (****************************************)
         sfts (CONS (Gb p A) G) (par n) D]

[sfsWeak:{G|Cxt}{D,A|Trm}{t,s|SS}{p|PP}             {noccG:is_ff (Poccur p G)}
      {l_prem:sfts G (sort t) D}
      {r_prem:sfts G A (sort s)}
     (****************************************)
         sfts (CONS (Gb p A) G) (sort t) D]

[sfPi:{G|Cxt}{A,B|Trm}{s1,s2,s3|SS}{p|PP}{n|VV}              {sc:rl s1 s2 s3}
                                                    {noccB:is_ff (poccur p B)}
      {l_prem:sfts G A (sort s1)}
      {r_prem:sfts (CONS (Gb p A) G) (vsub (par p) n B) (sort s2)}
     (****************************************)
         sfts G (pi n A B) (sort s3)]

[sfLda:{G|Cxt}{A,M,B|Trm}{s,s2,s3|SS}{p|PP}{n,m|VV}           {sc:rl s s2 s3}
                        {sc_ts:{t:SS}(is_tt (Trm_eq B (sort t)))->typedsort t}
                                                    {noccM:is_ff (poccur p M)}
                                                    {noccB:is_ff (poccur p B)}
      {l_prem:sfts (CONS (Gb p A) G) (vsub (par p) n M) (vsub (par p) m B)}
      {r_prem:sfts G A (sort s)}
     (****************************************)
         sfts G (lda n A M) (pi m A B)]

[sfApp:{G|Cxt}{M,A,B,L|Trm}{n|VV}
      {l_prem:sfts G M (pi n A B)}
      {r_prem:sfts G L A}
     (****************************************)
         sfts G (app M L) (vsub L n B)]

[sftCnv:{G|Cxt}{M,A,B|Trm}{s|SS}                                {sc:cnv A B}
      {l_prem:sfts G M A}
      {r_prem:sfts G B (sort s)}
     (****************************************)
         sfts G M B];
```

Table 5–2: The system for semi-full PTS.

5.1.2.2 Fixing the Lambda Rule

Define a correctness relation `sfts` (formally defined in table 5–2) by replacing rule LDA of `gts` with

$$\text{SFLDA} \qquad \frac{\Gamma[p{:}A] \vdash_{sf} [p/x]M : [p/y]B \qquad \Gamma \vdash_{sf} A : s}{\Gamma \vdash_{sf} [x{:}A]M : \{y{:}A\}B} \qquad \begin{array}{r} p \notin M,\ p \notin B, \\ \mathtt{rl}(s, s_2, s_3) \\ B \in \mathrm{SS} \Rightarrow B \in \mathrm{SS}_T \end{array}$$

(Recall $B \in \mathrm{SS}_T$ means B is a typedsort.) To show that for semi-full PTS, `sfts` has the same derivable judgements as `gts`, we treat the two directions separately.

```
Goal gts_sfts: {G|Cxt}{M,A|Trm}(gts G M A)->sfts G M A;
```

Proof. Induction on the derivation $\Gamma \vdash M : A$. Only the LDA rule is interesting: $\Gamma \vdash [x{:}A]M : \{x{:}A\}B$ as a consequence of $\Gamma[x{:}A] \vdash M : B$ and $\Gamma \vdash \{x{:}A\}B : s$. By induction hypothesis $\Gamma[x{:}A] \vdash_{sf} M : B$ and $\Gamma \vdash_{sf} \{x{:}A\}B : s$. By generation of the right ih, there are s, s_2, s_3 with $\Gamma \vdash_{sf} A : s_A$ and $\mathtt{rl}(s, s_2, s_3)$. By SFLDA it only remains to show $B \in \mathrm{SS} \Rightarrow B \in \mathrm{SS}_T$, so assume $B \in \mathrm{SS}$. By `only_typedsort_in_left` (section 4.4.3) on the right premise, $B \in \mathrm{SS}_T$ as required. ∎

```
Goal sfts_gts: {sf:semiFull}{G|Cxt}{M,A|Trm}(sfts G M A)->gts G M A;
```

Proof. Induction on the derivation of $\Gamma \vdash_{sf} M : A$. For the case SFLDA, have $\Gamma \vdash_{sf} [x{:}A]M : \{x{:}A\}B$ from $\Gamma[x{:}A] \vdash_{sf} M : B$ and $\Gamma \vdash_{sf} A : s$, with $\mathtt{rl}(s, s_2, s_3)$ and $B \in \mathrm{SS} \Rightarrow B \in \mathrm{SS}_T$. By ih $\Gamma[x{:}A] \vdash M : B$ and $\Gamma \vdash A : s$. By type correctness of the left ih, $B = t$ or $\Gamma[x{:}A] \vdash B : s_B$. In the first case, t must be a typedsort, so let s_B be such that $\mathtt{ax}(t{:}s_B)$. By semi-full there exists u with $\mathtt{rl}(s, s_B, u)$, and by LDA and PI it only remains to show $\Gamma[x{:}A] \vdash B : s_B$ by `sStartLem` (section 4.4.2). In the second case, again by semi-full, exists u with $\mathtt{rl}(s, s_B, u)$, and we are finished as before[4]. ∎

5.1.3 A Syntax Directed system: Eliminating the Conversion Rule.

The system `sfts` still has a conversion rule, SFTCNV, which is not syntax directed. For this system, with its tractable lambda rule, we can remove the conversion rule in favor of some

---

[3] `semiFull` is a definition, not an assumption in the global context, so each lemma using `semiFull` will assume it explicitly.

[4] My original proof used the assumption `decideTypedsort`. Thanks to Bert Jutting for this improved argument.

SDSFAX	$\bullet \vdash_{sdsf} s_1 : s_2$	$\mathtt{ax}(s_1{:}s_2)$
SDSFPAR	$\dfrac{\Gamma \vdash_{sdsf} A : X}{\Gamma[p{:}A] \vdash_{sdsf} p : A}$	$X \twoheadrightarrow s,\ p \notin \Gamma$
SDSFPWK	$\dfrac{\Gamma \vdash_{sdsf} q : C \qquad \Gamma \vdash_{sdsf} A : X}{\Gamma[p{:}A] \vdash_{sdsf} q : C}$	$X \twoheadrightarrow s,\ p \notin \Gamma$
SDSFSWK	$\dfrac{\Gamma \vdash_{sdsf} s : C \qquad \Gamma \vdash_{sdsf} A : X}{\Gamma[p{:}A] \vdash_{sdsf} s : C}$	$X \twoheadrightarrow s',\ p \notin \Gamma$
SDSFPI	$\dfrac{\Gamma \vdash_{sdsf} A : X \qquad \Gamma[p{:}A] \vdash_{sdsf} [p/x]B : Y}{\Gamma \vdash_{sdsf} \{x{:}A\}B : s_3}$	$\mathtt{rl}(s_1, s_2, s_3)$ $p \notin B$ $X \twoheadrightarrow s_1,\ Y \twoheadrightarrow s_2$
SDSFLDA	$\dfrac{\Gamma \vdash_{sdsf} A : X \qquad \Gamma[p{:}A] \vdash_{sdsf} [p/x]M : [p/y]B}{\Gamma \vdash_{sdsf} [x{:}A]M : \{y{:}A\}B}$	$\mathtt{rl}(s_1, s_2, s_3)$ $X \twoheadrightarrow s_1$ $p \notin M,\ p \notin B$ $B \in \mathsf{SS} \ \Rightarrow\ B \in \mathsf{SS}_T$
SDSFAPP	$\dfrac{\Gamma \vdash_{sdsf} M : X \qquad \Gamma \vdash_{sdsf} N : Y}{\Gamma \vdash_{sdsf} M\,N : [N/x]B}$	$X \twoheadrightarrow \{x{:}A\}B,\ Y \leq A$

Table 5–3: Syntax-directed semi-full PTS (Informal)

local reductions, obtaining a syntax directed system, `sdsf`, defined informally in table 5–3, and formally in table 5–4. I call this system "syntax directed", but I mean it is syntax directed for *functional* PTS, for which `ax` and `rl` are partial functions. In this case, given Γ and M, there is at most one derivation whose root has the shape $\Gamma \vdash_{sdsf} M : \_$. If we can compute the partial functions `ax` and `rl`, and decide the other side conditions, we can compute TSP for `sdsf` (section 5.1.6).

In order to use decidability of `sdsf` to decide `gts`, we need to prove a relationship between `sdsf` and `gts`, using the intermediate system `sfts`.

Lemma 5.4 (`sdsf` is sound for `sfts`)

```
Goal sdsf_sf: {sf:semiFull}{G|Cxt}{M,A|Trm}(sdsf G M A)->sfts G M A;
```

```
Inductive [sdsf:Cxt->Trm->Trm->Prop]  NoReductions  Constructors
[sdsfAx:{s1,s2|SS}                                          {sc:ax s1 s2}
         sdsf nilCxt (sort s1) (sort s2)]

[sdsfStart:{G|Cxt}{A,X|Trm}{s|SS}{p|PP}           {noccG:is_ff (Poccur p G)}
                                                  {redX:par_redn X (sort s)}
      {prem:sdsf G A X}
     (*****************************************)
         sdsf (CONS (Gb p A) G) (par p) A]

[sdsfvWeak:{G|Cxt}{D,A,X|Trm}{s|SS}{n,p|PP}       {noccG:is_ff (Poccur p G)}
                                                  {redX:par_redn X (sort s)}
      {l_prem:sdsf G (par n) D}
      {r_prem:sdsf G A X}
     (*****************************************)
         sdsf (CONS (Gb p A) G) (par n) D]

[sdsfsWeak:{G|Cxt}{D,A,X|Trm}{t,s|SS}{p|PP}       {noccG:is_ff (Poccur p G)}
                                                  {redX:par_redn X (sort s)}
      {l_prem:sdsf G (sort t) D}
      {r_prem:sdsf G A X}
     (*****************************************)
         sdsf (CONS (Gb p A) G) (sort t) D]

[sdsfPi:{G|Cxt}{A,B,X,Y|Trm}{t1,t2,t3|SS}{p|PP}{n|VV}       {rlt:rl t1 t2 t3}
                                                  {noccB:is_ff (poccur p B)}
                                                 {redX:par_redn X (sort t1)}
                                                 {redY:par_redn Y (sort t2)}
      {l_prem:sdsf G A X}
      {r_prem:sdsf (CONS (Gb p A) G) (vsub (par p) n B) Y}
     (*****************************************)
         sdsf G (pi n A B) (sort t3)]

[sdsfLda:{G|Cxt}{A,M,B,X|Trm}{s1,s2,s3|SS}{p|PP}{n,m|VV}     {rls:rl s1 s2 s3}
                        {sc_ts:{t:SS}(is_tt (Trm_eq B (sort t)))->typedsort t}
                                                  {noccM:is_ff (poccur p M)}
                                                  {noccB:is_ff (poccur p B)}
                                                 {redX:par_redn X (sort s1)}
      {l_prem:sdsf (CONS (Gb p A) G) (vsub (par p) n M) (vsub (par p) m B)}
      {r_prem:sdsf G A X}
     (*****************************************)
         sdsf G (lda n A M) (pi m A B)]

[sdsfApp:{G|Cxt}{M,A,Y,B,L,X|Trm}{n|VV}         {redX:par_redn X (pi n A B)}
                                                              {sc:cnv Y A}
         {l_prem:sdsf G M X}
         {r_prem:sdsf G L Y}
     (*****************************************)
           sdsf G (app M L) (vsub L n B)];
```

Table 5–4: The syntax directed system for semi-full PTS.

Proof. Easy induction on the derivation $G \vdash_{sdsf} M : A$ using predicate reduction for `sfts`, which we have by the equivalence of `gts` and `sfts`. ■

Lemma 5.5 (`sdsf` is complete for `sfts`)

```
Goal sfts_sdsf:
     {cf:cnv_full_below}{cpt:cnv_preserves_typedsort_dn}{cp:cnv_pi}
     {G|Cxt}{M,A|Trm}(sfts G M A)->Ex [E:Trm]and (sdsf G M E) (cnv E A);
```

The proof of this lemma will use the conversion character assumptions from section 4.4.11.3 without explicit mention (recall these are general assumptions in the context) as well as one of the conversion/typedsort assumptions from the same section, `cnv_preserves_typedsort_dn`, which is not a general assumption, and must be explicitly assumed when needed.

Two other properties are needed, which we do not make as general assumptions, so they will also be explicitly assumed when needed. First

```
[cnv_pi = {va,vb|VV}{Al,Ar,Bl,Br|Trm}{p|PP}
       {npoA:is_ff (poccur p Ar)}
       {npoB:is_ff (poccur p Br)}
       {l_prem:cnv Bl Al}
       {r_prem:cnv (vsub (par p) va Ar) (vsub (par p) vb Br)}
    (******************************************************)
       cnv (pi va Al Ar) (pi vb Bl Br)];
```

tells how to construct a `cnv` judgement for pi-types. This is clearly motivated by the notion of *cumulativity*. A version of this rule where the domains are not contravariantly related

$$\frac{A_l \simeq B_l \qquad A_r \leq B_r}{\{v_A{:}A_l\}A_r \leq \{v_B{:}B_l\}B_r}$$

is used in the definition of cumulativity for ECC as presented in [Luo94,Luo90a]. We will define cumulativity concretely in section 5.2, and that definition has a rule of the same shape as `cnv_pi`. Use of the assumption `cnv_pi` now shows that we are not really speaking of an abstract conversion relation anymore, but of an abstract cumulativity relation.

The next property seems ad hoc.

```
[cnv_full_below = {s1,s2,t1,t2,t3|SS}
                  {cv1:cnv (sort s1) (sort t1)}
                  {cv2:cnv (sort s2) (sort t2)}
                  {rlt:rl t1 t2 t3}
                  Ex [s3:SS] and (rl s1 s2 s3) (cnv (sort s3) (sort t3))];
```

It is a technical point for the mechanics of typechecking, saying that for any rule that can be used after some conversions (i.e. $\mathtt{rl}(t_1, t_2, t_3)$ can be used after $s_1 \leq t_1$ and $s_2 \leq t_2$), there's another rule that delays conversion (i.e. $\mathtt{rl}(s_1, s_2, s_3)$ with $s_3 \leq t_3$).

I needed these properties to prove an equivalence between `gts` and `sdsf`, but there are other combinations of properties that would also do. I am not satisfied with these somewhat arbitrary choices, but all of these assumptions will be discharged in section 5.3, where we show that ECC actually has these properties. Notice that every beta-conversion PTS (e.g. CC and $\boldsymbol{\lambda P}$) has these properties.

There is a delicate point in formalizing the proof, requiring a technical lemma

```
Goal shape_lemma:
     {p:PP}{M:Trm}Ex2 [v:VV][M':Trm] and (is_tt (Trm_eq M (alpha p v M')))
                                         (is_ff (poccur p M'));
```

saying that any term M can be expressed as $[p/v]M'$ for any p, where p does not occur in M'. Also, in order to prove `sfts_sdsf` as stated, without assuming the PTS is semi-full, we first prove a version of `typedsort_maybe_in_right` (section 4.4.3)

```
Goal sfts_typedsort_maybe_in_right:
     {G|Cxt}{M,A|Trm}(sfts G M A)->
     {s|SS}(is_tt (soccur s A))->(is_ff (Trm_eq (sort s) A))->typedsort s;
```

for `sfts` directly by induction, rather than translate back to `gts` using `sfts_gts`, which assumes semi-fullness.

Proof of `sfts_sdsf`**.** By induction on the derivation of `sfts G M A`. Consider the case for `sfLda`: we must show

$$\exists E \,.\, G \vdash_{sdsf} [n{:}A]M : E \text{ and } E \leq \{m{:}A\}B$$

from

sc	:	$\mathrm{rl}(s, s_2, s_3)$
sc_ts	:	$\forall t \,.\, B = t \Rightarrow t \in \mathrm{SS}_T$
noccM	:	$p \notin M$
noccB	:	$p \notin B$
l_prem	:	$G[p{:}A] \vdash_{sf} [p/n]M : [p/m]B$
r_prem	:	$G \vdash_{sf} A : s$

After unpacking the existential quantifier in the induction hypotheses, we also have

hX1	:	$G[p{:}A] \vdash_{sdsf} [p/n]M : X$
hX2	:	$X \leq [p/m]B$
hY1	:	$G \vdash_{sdsf} A : Y$
hY2	:	$Y \leq s$

Now one problem is clear: to use `sdsfLda` to solve the goal, we need to express X in the form $[p/\_]\_$, so we use the shape lemma to get

$$\begin{array}{lll} \text{eqX} & : & X = [p/v]X' \\ \text{noccX}' & : & p \notin X' \end{array}$$

There is now enough information to instantiate the existential quantifier of the goal: we claim

$$G \vdash_{sdsf} [n{:}A]M : \{v{:}A\}X' \quad \text{and} \quad \{v{:}A\}X' \leq \{m{:}A\}B$$

Addressing the second goal first, by `(cnvPi noccX' noccB)` it suffices to show $A \leq A$ (since A is `Vclosed` by r_prem) and $[p/v]X' \leq [p/m]B$ (by hX2 and eqX).

For the first goal, by `sdsfLda`, sc, noccM, noccX′, hY2, and hY1, it suffices to show

$$\forall t\,.\; X' = t \Rightarrow t \in \mathtt{SS}_T \quad \text{and} \quad G[p{:}A] \vdash_{sdsf} [p/n]M : [p/v]X'$$

The second of these is trivial by eqX and hX1.

Finally, the heart of the matter; assume X' is a sort, t, and show $t \in \mathtt{SS}_T$. Notice that X', being a sort, is `Vclosed`, so

$$t = X' = [p/v]X' = X \leq [p/m]B.$$

Thus, by `cnv_sort_character_r` there is a sort, u, with

$$\begin{array}{lll} \text{uh1} & : & [p/m]B \twoheadrightarrow u \\ \text{uh2} & : & t \leq u \end{array}$$

Notice u must occur in B by uh1. By `(cnv_preserves_typedsort_dn uh2)` it suffices to show $u \in \mathtt{SS}_T$. Either $B = u$ (and we are done by sc_ts), or $B \neq u$, and $u \in \mathtt{SS}_T$ by `sfts_typedsort_maybe_in_right` applied to l_prem. ■

In summary putting `sfts_sdsf` and `gts_sfts` together we have:

```
Goal gts_sdsf:
     {cf:cnv_full_below}{cpt:cnv_preserves_typedsort_dn}{cp:cnv_pi}
     {G|Cxt}{M,A|Trm}(gts G M A)->Ex [E:Trm]and (sdsf G M E) (cnv E A);
```

and putting `sdsf_sf` and `sfts_gts` together we have:

```
Goal sdsf_gts:{sf:semiFull}{G|Cxt}{M,A|Trm}(sdsf G M A)->gts G M A;
```

5.1.3.1 Characterizing `gts`

Lemmas `gts_sdsf` and `sdsf_gts` do not yet characterize `gts` in terms of `sdsf`. We have:

```
Goal sdsf_characterizes_gts:
     {sf:semiFull}
     {cf:cnv_full_below}{cpt:cnv_preserves_typedsort_dn}{cp:cnv_pi}
     {crt:cnv_range_typedsort}
     {G:Cxt}{M,A:Trm}
     iff (gts G M A)
         (Ex [E:Trm] and3 (sdsf G M E)
                          (cnv E A)
                          (Ex2 [D:Trm][s:SS]or (is_tt (Trm_eq A (sort s)))
                                               (and (sdsf G A D)
                                                    (par_redn D (sort s)))));
```

As well as the assumptions used in `gts_sdsf` and `sdsf_gts`, this lemma depends on `cnv_range_typed`, which authorizes the predicate conversion lemma from section 4.4.11.3.

Proof. of `sdsf_characterizes_gts`

$\Rightarrow$ By `gts_sdsf` we have E with $\Gamma \vdash_{sdsf} M : E \leq A$. Also by type correctness of `gts`, for some sort s, $A = s$ or $\Gamma \vdash A : s$. In the first case we are done. In the second case $\exists D \,.\, \Gamma \vdash_{sdsf} A : D \leq s$ by `gts_sdsf`, and D reduces to some sort.

$\Leftarrow$ By `sdsf_gts` we have $\Gamma \vdash M : E \leq A$. If $A = s$ we are done by predicate conversion of `gts`. Otherwise $\Gamma \vdash A : D \twoheadrightarrow s$ and we are done by predicate reduction and the conversion rule. ■

5.1.4 Principal Types

We have gone to a lot of trouble to make `sdsf` deterministic. In section 5.1.6 we will see that the effort has payed off, and type synthesis is computable for `sdsf` under some assumptions. The last lemma is very suggestive of an algorithm for typechecking `gts` given type synthesis for `sdsf`: to decide $G \vdash M : A$, compute an `sdsf`-type for (G, M), see if that type converts with A, and check if A itself is a correct `sdsf`-type. This lemma is still unsatisfactory for typechecking for two reasons. The first of these is that the quantifier $\exists E$ in `sdsf_characterizes_gts` not only allows M to fail to have an `sdsf` type in G, (in which case M fails to have a `gts` type in G) but also requires us to search through all `sdsf` types of M in G. Although `sdsf` stands for syntax-directed-semi-full, there are still four sources of non-determinism in `sdsf`.

1. If the PTS is not functional (section 4.6) the rules SDSFAX and SDSFPI can produce different types for the same subject. To handle non-functional systems requires the technique of

sort variables and *schematic terms* (see [HP91,vBJMP94]) which is not discussed in this thesis. I will assume the PTS is functional when necessary (it has not been used yet in this chapter), as this covers almost every case of practical interest including the three type systems of LEGO.

2. The parameter `p` in the rules SDSFPI and SDSFLDA is not determined by our intended syntax-directed computation because `p` does not occur in the conclusion of the rule (this issue is discussed in section 3.2.5.3 for the case of `Vclosed` and in section 4.4.4 for `gts`). This is a non-determinism of derivations, not of judgements, and for type synthesis it suffices to pick a fresh enough parameter[5].

3. The rule SDSFLDA allows a free choice of the variable `m` bound in the type `(pi m A B)`. For our present purposes it does no harm to accept that `sdsf` types are unique, at best, only up to alpha-conversion.

4. In the rule SDSFAPP the side condition `redX:par_redn X (pi n A B)` allows non-deterministic reduction as long as it stops at a `pi`. Many of the other rules have non-deterministic reduction to a sort, but a sort is a normal form, so this causes no multiplicity of types. A `pi` is not necessarily a normal form, but is a weak-head normal form, and we should, for moral purity, replace `par_redn` with weak-head reduction in `redX`. (James McKinna [McK94] has formalized a theory of weak-head reduction and weak-head normal forms in our setting that is adequate for this example. It is used to handle similar issues in formalizing [vBJMP94].) Instead, for our present purposes, we will accept that `sdsf` types are unique, at best, only up to beta-conversion. That is, we succeed in reasoning with less information about the reduction sequence used in this side condition, but are still free to compute this side condition using weak-head reduction when we construct an algorithm in section 5.1.6.

This motivates the lemma:

```
Goal sdsf_unique_types:
     {sf:semiFull}{f:Functional}
     {G|Cxt}{M,A|Trm}{j:sdsf G M A}{B|Trm}{k:sdsf G M B}conv A B;
```

This lemma is proved by structural induction on the derivation `j:sdsf G M A`. In each case of this induction the derivation `k:sdsf G M B` is destructed using an appropriate `sdsf` generation lemma, thereby avoiding the need for double induction.

---

[5] However, as for previously discussed relations, in order to have an adequate induction principle we need to prove `sdsf` has the same judgements as a relation without freely occurring parameters in the rules for PI and LDA. This proof is very much like that in section 4.4.4.

Remark 5.6 *I believe that assumption* `sf:semiFull` *of* `sdsf_unique_types` *is not required. I used it to save some work in the proof by translating* `sdsf` *judgements to* `gts` *(using* `sdsf_gts`*, which does require semi-fullness) to take advantage of already proved properties of* `gts`*.*

Principal Types The collection of `gts`-types of a subject may contain many `conv`-classes. However `sdsf_characterizes_gts` and `sdsf_unique_types` make it clear that all the `gts` types of a subject are characterized by one `conv`-class, namely the `sdsf`-type of the subject. To formalize this we have the definition from [Luo90a,Luo94]:

```
[principal_type [G:Cxt][M,A:Trm] =
    and (gts G M A)
        ({B:Trm}iff (gts G M B) (and (correct_type G B) (cnv A B)))];
```

(`correct_type` is defined in section 4.4.8.) Obviously principal types are unique up to `conv`. More interestingly, `sdsf` types are principal types.

```
Goal sdsf_type_is_principal:
     {f:Functional}{cp:cnv_pi}{sf:semiFull}{crt:cnv_range_typedsort}
     {cf:cnv_full_below}{cpt:cnv_preserves_typedsort_dn}
     {G|Cxt}{M,A|Trm}(sdsf G M A)->principal_type G M A;
```

Proof. Given $G \vdash_{sdsf} M : A$ (hence also $G \vdash M : A$), and any B, we must show

$$G \vdash M : B \quad \Leftrightarrow \quad (\texttt{correct\_type}\ G\ B) \text{ and } A \leq B$$

Consider the two directions.

$\Rightarrow$ $G \vdash M : B$ by hypothesis, so by lemma `type_correctness` (section 4.4.8) it suffices to show $A \leq B$. Also $G \vdash_{sdsf} M : E \leq B$ by `gts_sdsf`, so $A \simeq E$ by `sdsf_unique_types`, and we are done by transitivity $A \simeq E \leq B$.

$\Leftarrow$ We have $A \leq B$ and ($\exists s\ .\ B = s$ or $G \vdash B : s$). In case of the first disjunct, we are done by predicate conversion (section 4.4.11.3) because $A \leq B = s$. The second disjunct is even easier by rule TCNV. ■

5.1.5 Typechecking `gts`

I mentioned that lemma `sdsf_characterizes_gts` is suggestive of an algorithm for typechecking `gts` given a type synthesis algorithm for `sdsf`, but not adequate for two reasons. The first problem has been dealt with by the lemma `sdsf_type_is_principal`; now we address the second problem. From the algorithmic viewpoint suggested at the start of section 5.1.4, the statement of `sdsf_characterizes_gts` is misleading. Since the conversion relation, `cnv`, is only decidable for well-typed terms, an algorithmically more satisfactory statement (although logically equivalent, and uglier) delays checking conversion until after A is seen to be well-typed:

$$\begin{array}{lll} G \vdash M : A & \Leftrightarrow & \\ \quad \exists E \; . & G \vdash_{sdsf} M : E \text{ and} & \\ & \text{either} \quad (\exists s, t \; . \; A = s, \; E \twoheadrightarrow t \text{ and } t \leq s) & \\ & \text{or} \qquad (\exists D, s \; . \; G \vdash_{sdsf} A : D, \; E \leq A \text{ and } D \twoheadrightarrow s) & \end{array} \tag{†}$$

Notice that in the case $A = s$, `cnv_sort_character_l` has been used to further analyse the condition $E \leq A$.

Remark 5.7 *This characterization may be simplified in two cases of special interest. First, for PTS without topsorts, the first disjunct may be removed. (You may object that this is not an algorithmic improvement!) More interestingly, in some cases the test for $D \twoheadrightarrow s$ may be removed from the second disjunct. For example, for beta-conversion PTS (i.e. after cutting in `conv` and its properties for `cnv` and its properties, as in section 4.5) we can prove*

```
Goal beta_gts_levels :
     {f:Functional}
     {G|Cxt}{M,E|Trm}(gts G M E)->{A,D|Trm}(conv E A)->(gts G A D)->
     Ex [s:SS] par_redn D (sort s);
```

Proof. *By type correctness, for some t either $E = t$ or $G \vdash E : t$. In the first case $A \twoheadrightarrow t$, so $G \vdash t : D$ by subject reduction, and D reduces to some sort by the generation lemma. In the second case, let X be a common reduct of E and A. Then $G \vdash X : t$ and $G \vdash X : D$ by subject reduction, so D reduces to some sort by `types_unicity` of functional PTS (section 4.6).* ■

In fact all beta-conversion PTS have this property, using the typing lemma (section 4.5.1, not yet checked in LEGO) in place of `types_unicity` in this proof.

Guided by this informal statement, we are almost ready to use a program for the `sdsf`-TSP to decide `gts` judgements. First we need to consider decidability of some side conditions occurring in equivalence (†).

5.1.5.1 Decidability of Side Conditions

Define what it is to be a `gts_term` and a `gts_type`

```
[gts_term [M:Trm] = Ex2 [G:Cxt][A:Trm] gts G M A];
[gts_type [A:Trm] = Ex2 [G:Cxt][M:Trm] gts G M A];
```

and assume that `gts_terms` are normalizing:

```
[gts_term_Normalizing = {M|Trm}(gts_term M)->normalizing M];
```

By `type_correctness` `gts_types` are also normalizing:

```
Goal gts_type_normalizing:
     {gtn:gts_term_Normalizing}{M|Trm}(gts_type M)->normalizing M;
```

Recalling section 3.3.5.1 on decidability of the shape of normal forms, these lemmas are what we need to conclude that it is decidable whether a `gts_term` or `gts_type` reduces to a sort or to a pi.

Decidability of `cnv`. In section 3.3.5.2 we proved that `conv` ($\simeq$) is decidable for normalizing terms; however, this is not enough to prove that `cnv` ($\leq$) is decidable for normalizing terms. The reason for this is that `cnv` is an abstract relation which is only assumed to exist; without a closure property, e.g. an induction elimination rule, there is little hope to prove decidability. In section 5.2 we define a particular inductive relation, cumulativity, which is the example motivating our assumptions about `cnv`. For the moment, we explicitly keep track of uses of `normalizing_decides_cnv`:

```
[normalizing_decides_cnv =
     {A,B|Trm}(Vclosed A)->(normalizing A)->(Vclosed B)->(normalizing B)->
     decidable (cnv A B)];
```

As the condition $s \leq t$ occurs in equivalence $(\dagger)$, page 109, notice that from a proof of `normalizing_decides_cnv` it is trivial to prove that `cnv` is decidable on sorts:

```
Goal ndc_dcs:
     {ndc:normalizing_decides_cnv}{s,t:SS}decidable (cnv (sort s) (sort t));
```

Also, by the assumption that `gts_terms` are normalizing, `normalizing_decides_cnv` implies `cnv` is decidable for `gts_terms` and `gts_types`.

Decidability of type synthesis. We are also assuming type synthesis is computable for `sdsf` (this will be proved, under some assumptions, in section 5.1.6), so define "M is an `sdsf`-well-typed term in G"

```
[sdsfTS [G:Cxt][M:Trm] = Ex [A:Trm] sdsf G M A];
```

We will assume this is decidable.

5.1.5.2 A Typechecking Algorithm

With these assumptions, TCP for `gts` is decidable.

```
Goal decide_sdsfTS_decide_gts:
     {f:Functional}{sf:semiFull}
     {cf:cnv_full_below}{cptd:cnv_preserves_typedsort_dn}{cp:cnv_pi}
     {crt:cnv_range_typedsort}
     {ndc:normalizing_decides_cnv}
     {dec_sdsf:{G:Cxt}{M:Trm}decidable (sdsfTS G M)}
     {gtn:gts_term_Normalizing}
     {G:Cxt}{M,A:Trm}decidable (gts G M A);
```

Proof. With our assumptions, all the questions on the RHS of equivalence $(\dagger)$ (page 109) are decidable: if $G \vdash_{sdsf} M : E$ for some E, and either A is a sort and E reduces to a sort that converts with A, or $G \vdash_{sdsf} A : D$ for some D that reduces to a sort and $E \leq A$, then $G \vdash M : A$. This positive outcome is the easy part of the proof; more tedious is showing that if one of the conditions fails then $G \vdash M : A$ is *not* derivable.

(does M have a type?) By `(dec_sdsf G M)`

$$\text{either} \quad \exists E \,.\, G \vdash_{sdsf} M : E \quad \text{or} \quad \neg\exists E \,.\, G \vdash_{sdsf} M : E.$$

In the latter case, choosing the negative result, we want to prove

$$(\neg\exists E \,.\, G \vdash_{sdsf} M : E) \;\Rightarrow\; G \not\vdash M : A$$

which follows by contraposition from `gts_sdsf`. Thus we may assume $G \vdash_{sdsf} M : E$ for some E, and by `sdsf_type_is_principal`, have $G \vdash M : E$ and

$$\texttt{GME} \;:\; \forall B[G \vdash M : B \;\;\Leftrightarrow\;\; (\texttt{correct\_type}\; G\; B) \text{ and } E \leq B]$$

(is A a sort?) Either A is a sort or not (there is a boolean-valued function that decides this).

($A = s$; does E reduce to a sort?) E is a `gts_type`, so either E reduces to a sort or not. In the latter case, by `cnv_sort_character_1`, it must be that $E \not\leq s$, hence by $(\texttt{GME}\; s)$, we have proved the negative outcome $G \not\vdash M : s$. Thus we may assume $E \twoheadrightarrow t$.

($A = s$, $E \twoheadrightarrow t$; does $t \leq s$?) By `normalizing_decides_cnv` either $t \leq s$ or $t \not\leq s$. In the first case, conclude $E \leq s$, so $G \vdash M : s$ by $(\texttt{GME}\; s)$. In the second case $G \not\vdash M : s$ by contraposition and $(\texttt{GME}\; s)$. This finishes the cases for $A = s$, so we may assume A is not a sort.

(A not a sort; does A have a type?) By `(dec_sdsf G A)`

$$\text{either} \quad \exists D \,.\, G \vdash_{sdsf} A : D \quad \text{or} \quad \neg\exists D \,.\, G \vdash_{sdsf} A : D.$$

In the latter case, choosing the negative result we want to prove

$$(\neg\exists D \,.\, G \vdash_{sdsf} A : D) \;\Rightarrow\; G \not\vdash M : A$$

which follows by contraposition from `type_correctness` of `gts` and `gts_sdsf`. Thus we may assume $G \vdash_{sdsf} A : D$ for some D, so also, by `sdsf_type_is_principal`, $G \vdash A : D$ and

$$\texttt{GAD} \;:\; \forall B[G \vdash A : B \quad \Leftrightarrow \quad (\texttt{correct\_type}\; G\; B) \text{ and } D \leq B]$$

(A not a sort; $G \vdash_{sdsf} A : D$; does $E \leq A$?) A is a `gts_term` and E is a `gts_type`, so by `normalizing_decides_cnv` either $E \leq A$ or $E \not\leq A$ In the latter case, choosing the negative result, we want to prove

$$E \not\leq A \;\;\Rightarrow\;\; G \not\vdash M : A$$

which follows (by contraposition) from $(\texttt{GME}\; A)$. Thus we may assume $E \leq A$.

(A not a sort; $G \vdash_{sdsf} A : D$; $E \leq A$; does D reduce to a sort?) D is a `gts_type`, so

$$\text{either} \quad \exists s \,.\, D \twoheadrightarrow s \quad \text{or} \quad \neg\exists s \,.\, D \twoheadrightarrow s\,.$$

In the first case we have $G \vdash A : s$ by `gtsPR`, so the positive outcome, $G \vdash M : A$, holds by $(\texttt{GME}\; A)$. In the second case, choosing the negative result, by contraposition we want to prove

$$G \vdash M : A \;\;\Rightarrow\;\; (\exists s \,.\, D \twoheadrightarrow s)$$

Assuming $G \vdash M : A$, by `type_correctness` either A is a sort (contradicting the assumption that A is not a sort, so we are done) or $\exists t \,.\, G \vdash A : t$. In this last case, $D \leq t$ by $(\texttt{GAD}\; A)$, so $\exists s \,.\, D \twoheadrightarrow s$ by `cnv_sort_character_1`. ∎

This proof is not as hard to check in LEGO as it is to explain!

5.1.6 Type Synthesis

Finally we are going to formalize decidability of type synthesis for `sdsf`: given (G, M) compute an `sdsf`-type of (G, M) (and the typing derivation) if one exists, otherwise return a proof that (G, M) has no `sdsf`-type. This result easily gives Type Synthesis for `gts`.

5.1.6.1 More on Decidability of Side Conditions

Recall that by `sdsf_gts` (assuming semi-fullness) every `sdsf`-term (-type) is a `gts`-term (-type), so the lemmas and assumption in section 5.1.5.1 are relevant here as well.

Decidable properties of `ax` and `rl`. Three more restrictions on the PTS are needed. I have not assumed that the axiom and rule relations are decidable, but we will need something stronger than this to have a program that decides TSP. Consider running a type-synthesis algorithm on the input $(\bullet, s_1)$; it must return either a type for s_1, showing that s_1 is a `typedsort` (i.e. $\exists s_2.\mathtt{ax}(s_1{:}s_2)$, section 4.4.8), or a proof that s_1 has no type, showing that s_1 is not a `typedsort`. That is,

$$\exists s_2 \,.\, \mathtt{ax}(s_1{:}s_2) \quad \Leftrightarrow \quad \exists X \,.\, \bullet \vdash_{sdsf} s_1 : X$$

because the only possible derivation of $\bullet \vdash_{sdsf} s_1 : X$ has shape:

$$\frac{\mathtt{ax}(s_1{:}s_2)}{\bullet \vdash_{sdsf} s_1 : s_2}\,\text{SDSFAX}.$$

As `ax` is an arbitrarily given relation, we cannot hope to decide `typedsort` in general, even if `ax` is decidable; in order to prove that type synthesis is decidable for $\vdash_{sdsf}$ we must assume that `typedsort` is decidable. Similarly, if $\mathtt{ax}(t_1{:}s_1)$ and $\mathtt{ax}(t_2{:}s_2)$, we have

$$\exists u \,.\, \mathtt{rl}(s_1, s_2, u) \quad \Leftrightarrow \quad \exists X \,.\, \bullet \vdash_{sdsf} \{v{:}t_1\}t_2 : X$$

i.e. a type synthesis algorithm decides $\exists u.\mathtt{rl}(s_1, s_2, u)$, because the only possible derivation of $\bullet \vdash_{sdsf} \{v{:}t_1\}t_2 : X$ has shape

$$\frac{\bullet \vdash_{sdsf} t_1 : s_1 \qquad \begin{array}{c}\vdots\\ {[p{:}t_1] \vdash_{sdsf} t_2 : s_2}\end{array} \qquad \mathtt{rl}(s_1, s_2, u)}{\bullet \vdash_{sdsf} \{v{:}t_1\}t_2 : u}\,\text{SDSFPI}.$$

Also, with the same condition

$$\exists u_2, u_3 \,.\, \mathtt{rl}(s_1, u_2, u_3) \quad \Leftrightarrow \quad \exists X \,.\, \bullet \vdash_{sdsf} [x{:}t_1]t_2 : X$$

i.e. a type synthesis algorithm decides $\exists u_2, u_3.\mathtt{rl}(s_1, u_2, u_3)$, because the only possible derivation of $\bullet \vdash_{sdsf} [x{:}t_1]t_2 : X$ has shape

$$\frac{\bullet \vdash_{sdsf} t_1 : s_1 \qquad \begin{array}{c}\vdots\\ {[p{:}t_1] \vdash_{sdsf} t_2 : s_2}\end{array} \qquad \mathtt{rl}(s_1, u_2, u_3)}{\bullet \vdash_{sdsf} [x{:}t_1]t_2 : \{y{:}t_1\}s_2}\,\text{SDSFLDA}.$$

Thus we will assume the properties `ruledsort` and `ruledsorts`

```
[ruledsort [s1:SS] = Ex2 [s2,s3:SS] rl s1 s2 s3];
[ruledsorts [s1,s2:SS] = Ex [s3:SS] rl s1 s2 s3];
```

are decidable

Remark 5.8 *One might think that a solution to the Type Checking Problem requires only that* ax *be decidable, not that* `typedsort` *be decidable, as TCP differs from TSP in having no existential quantifier in the statement of the problem. However this is not the case, because the derivation skeleton*

$$\dfrac{\dfrac{ax(s{:}?)}{\bullet \vdash s : ?}\ \textsc{Axiom}}{[p{:}s] \vdash p : s}\ \textsc{Start}$$

shows (I omit some details) that

$$[p{:}s] \vdash p : s \quad \Leftrightarrow \quad \exists t\ .\ ax(s{:}t)\,.$$

For example, let $T(i, n)$ *mean "the* i^{th} *Turing machine, when started with* i *on it's input tape, halts in exactly* n *steps", and consider the* PTS *whose set of sorts is the natural numbers, whose axiom relation is* $\{\ (i{:}n)\ \mid\ T(i, n)\ \}$ *and whose rule relation is empty. For any* i*,*

$$[p{:}i] \vdash p : i \quad \Leftrightarrow \quad \exists n\ .\ ax(i{:}n) \quad \Leftrightarrow \quad \textit{the } i^{th} \textit{ Turing machine halts on input } i$$

This PTS *is functional, semi-full, strongly normalizing (there are no well typed redexes), and has decidable* ax *and* `rl` *relations, but a solution to its TCP also solves the halting problem! Similarly, one can see that decidability of TCP implies* `decideRuledsort` *and* `decideRuledsorts`*.*

5.1.6.2 A Type Synthesis Algorithm for `sdsf`

In view of the syntax directedness of `sdsf`, the most interesting remaining problem is termination of a type synthesis algorithm. We will use well founded induction on the sum of the lengths of the terms in the input (G, M).

```
[Cxt_ln : {G:Cxt}NN = LLrec ([_:LL|GB]NN) Z
                            ([b:GB][_:Cxt][ih:NN]add (lngth (typOf b)) ih)];
[Cxt_Trm_ln [G:Cxt][M:Trm] : NN = add (lngth M) (Cxt_ln G)];
```

We require a principle of well founded induction on objects of shape (G, M)

```
Goal Cxt_Trm_WF_ind:
     {f:{G:Cxt}{M:Trm}NN}
     {C:{G:Cxt}{M:Trm}Prop}
     {wf_ih:{G:Cxt}{M:Trm}{ih:{g:Cxt}{m:Trm}(Lt (f g m) (f G M))->C g m}C G M}
     {G:Cxt}{M:Trm}C G M;
```

which is trival to prove from the lemma `complete_induction` on natural numbers (section 2.2.1).

Now we can construct the type synthesis algorithm.

```
Goal sf_typSyn:
     {sf:semiFull}{f:Functional}
     {dt:{s:SS}decidable (typedsort s)}
     {dr:{s:SS}decidable (ruledsort s)}
     {drs:{s1,s2:SS}decidable (ruledsorts s1 s2)}
     {ndc:normalizing_decides_cnv}
     {gtn:gts_term_Normalizing}
     {G:Cxt}{M:Trm}decidable (sdsfTS G M);
```

Proof. By `Cxt_Trm_WF_ind` using `Cxt_Trm_ln` on (G, M), it suffices to show

```
Claim {M:Trm}{K:Cxt}
     {wf_ih:{g:Cxt}{m:Trm}(Lt (Cxt_Trm_ln g m) (Cxt_Trm_ln K M))->
                 or (Ex ([A:Trm]sdsf g m A)) (not (Ex ([A:Trm]sdsf g m A)))}
     or (Ex ([A:Trm]sdsf K M A)) (not (Ex ([A:Trm]sdsf K M A)));
```

which we do by cases on the shape of M (i.e. by structural induction on M, not using the induction hypotheses). For each shape of term M, use the appropriate rule(s) of `sdsf` to compute its type. We do two cases.

(M is a sort: $M = s$) SDSFAX and SDSFSWK are the only rules constructing an `sdsf`-type for a sort. If $\Gamma = \bullet$ then only SDSFAX can apply. By the assumption `typedsort` is decidable,

$$\exists t\ .\ \mathtt{ax}(s{:}t)\ \text{ or } \neg\exists t\ .\ \mathtt{ax}(s{:}t)\,.$$

In the first case use SDSFAX to return a proof of $\bullet \vdash_{sdsf} s : t$; in the second case, no rule can apply, so return a proof of $\neg\exists A\ .\ \bullet \vdash_{sdsf} s : A$.

Now we may assume $\Gamma \neq \bullet$, so $\Gamma = \Delta[q, A]$. Only rule SDSFSWK can apply. Fail (i.e. return a proof of $\neg\exists A\ .\ \Delta[q, A] \vdash_{sdsf} s : A$) if $q \in \Delta$ because SDSFSWK cannot apply, so assume $q \notin \Delta$. Addressing the left premise, by induction hypothesis on (Δ, s),

$$\exists X\ .\ \Delta \vdash_{sdsf} s : C\ \text{ or } \neg\exists X\ .\ \Delta \vdash_{sdsf} s : C\,.$$

Fail in case of the right disjunct, otherwise $\Delta \vdash_{sdsf} s : C$ for some C. Now address the right premiss; by induction hypothesis on (Δ, A)[6],

$$\exists X\ .\ \Delta \vdash_{sdsf} A : X\ \text{ or } \neg\exists X\ .\ \Delta \vdash_{sdsf} A : X\,.$$

Fail in case of the right disjunct, otherwise $\Delta \vdash_{sdsf} A : X$ for some X. X is a PTS-type, so it is decidable if X reduces to a sort; if not then fail, if so use SDSFSWK to return $\Gamma \vdash_{sdsf} s : C$.

[6] Here we use that `lngth`(s) is positive, so `Cxt_Trm_ln`(Δ, A) $<$ `Cxt_Trm_ln`$(\Delta[q, A], s)$.

(M is an application: $M = N\,L$) The only rule that can apply is SDSF-APP By induction hypothesis on (Γ, N),

$$\exists X\,.\;\Gamma \vdash_{sdsf} N : X \;\text{ or }\; \neg\exists X\,.\;\Gamma \vdash_{sdsf} N : X\,.$$

Fail if the right disjunct holds, otherwise $\Gamma \vdash_{sdsf} N : X$ for some X. Similarly for the right premise: fail if it is not derivable, or $\Gamma \vdash_{sdsf} L : Y$ for some Y. Fail if X does not reduce to some pi, otherwise have $X \twoheadrightarrow \{v{:}A\}B$ for some v, A and B. Now we have $\Gamma \vdash N : X$, so by predicate reduction $\Gamma \vdash N : \{v{:}A\}B$, by type correctness $\Gamma \vdash \{v{:}A\}B : Z$, and by the generation lemma for pi, A is a PTS-term. Thus we can decide if $Y \simeq A$; fail if not, and use SDSFAPP to return $\Gamma \vdash_{sdsf} N\,L : [L/v]B$ if so. ∎

5.1.6.3 Type Synthesis and Type Checking for `gts`.

From `sf_typSyn`, `sfts_gts` and `gts_sfts` (section 5.1.3) we have Type Synthesis for `gts`:

```
[gtsTS [G:Cxt][M:Trm] = Ex [A:Trm] gts G M A];

Goal gts_typSyn:
     {sf:semiFull}{f:Functional}
     {cf:cnv_full_below}{cpt:cnv_preserves_typedsort_dn}{cp:cnv_pi}
     {dt:{s:SS}decidable (typedsort s)}
     {dr:{s:SS}decidable (ruledsort s)}
     {drs:{s1,s2:SS}decidable (ruledsorts s1 s2)}
     {ndc:normalizing_decides_cnv}
     {gtn:gts_term_Normalizing}
     {G:Cxt}{M:Trm}decidable (gtsTS G M);
```

However, `sf_typSyn` is just as good for our purposes

Putting `decide_sdsfTS_decide_gts` and `sf_typSyn` together, we have Type Checking for `gts`:

```
Goal decide_gts:
     {sf:semiFull}{f:Functional}
     {cfb:cnv_full_below}{cpt:cnv_preserves_typedsort_dn}{cp:cnv_pi}
     {crt:cnv_range_typedsort}
     {dt:{s:SS}decidable (typedsort s)}
     {dr:{s:SS}decidable (ruledsort s)}
     {drs:{s1,s2:SS}decidable (ruledsorts s1 s2)}
     {ndc:normalizing_decides_cnv}
     {gtn:gts_term_Normalizing}
     {G:Cxt}{M,A:Trm}decidable (gts G M A);
```

This looks terrible, but we will now (section 5.2) define a natural class of PTS, including the three type systems supported by LEGO, that has many of the properties assumed in

CumConv	$A \preceq B$	$A \simeq B$
CumSort	$s_a \preceq s_b$	$\mathtt{CumBase}(s_a, s_b)$
CumPi	$\dfrac{B_l \preceq A_l \qquad [p/v_A]A_r \preceq [p/v_B]B_r}{\{v_A{:}A_l\}A_r \preceq \{v_B{:}B_l\}B_r}$	$p \notin A_r, p \notin B_r$
CumTrans	$\dfrac{A \preceq B \qquad B \preceq C}{A \preceq C}$	

Table 5–5: Informal definition of `Cum`

`decide_gts`, and then (section 5.3) specialize further to prove that ECC satisfies all the assumptions of `decide_gts` except normalization.

5.2 Cumulative PTS

We will define a particular conversion relation, `Cum`, and show that it satisfies the assumptions for an abstract conversion relation that have been used to prove `sdsf_characterizes_gts`. Luo's notion of cumulativity for ECC is our motivating example.

5.2.1 The Cumulativity Relation

As with the `ax` and `rl` parameters in the definition of PTS, the cumulativity relation is parameterized by a given relation, called `CumBase`:

```
[CumBase : SS->SS->Prop];
```

`Cum`, written informally $\preceq$, is defined informally by the rules of table 5–5, which are formalized in table 5–6. There is one technical point about this formalization which is different than previous inductive definitions in this thesis, and needs to be explained.

Free-Conclusioned Rules The rule CumSort could have been formalized as

```
[CumSort:{sa,sb|SS}{A,B|Trm}{base:CumBase sa sb}Cum sa sb]
```

which is similar to rule Ax (section 4.3) and the _Ax rules in all of our formal systems so far. Such a rule can only be applied in LEGO to solve a goal of form `(Cum s1 s2)`, so if the actual

```
Inductive [Cum:{A,B:Trm}Prop]  NoReductions  Constructors

[CumConv:{A,B|Trm}                                          {cnvAB:conv A B}
         Cum A B]

[CumSort:{sa,sb|SS}{A,B|Trm}                 {eqA:is_tt (Trm_eq A (sort sa))}
                                             {eqB:is_tt (Trm_eq B (sort sb))}
                                                        {base:CumBase sa sb}
         Cum A B]

[CumPi:{va,vb|VV}{Al,Ar,Bl,Br|Trm}{A,B|Trm}{p|PP}  {npoA:is_ff (poccur p Ar)}
                                         {eqA:is_tt (Trm_eq A (pi va Al Ar))}
                                                   {npoB:is_ff (poccur p Br)}
                                         {eqB:is_tt (Trm_eq B (pi vb Bl Br))}
       {l_prem:Cum Bl Al}
       {r_prem:Cum (vsub (par p) va Ar) (vsub (par p) vb Br)}
    (**********************************************************)
         Cum A B]

[CumTrans:{A,B,C|Trm}
       {l_prem:Cum A B}
       {r_prem:Cum B C}
    (**********************************************************)
         Cum A C];
```

Table 5–6: The Cumulativity Relation

goal is (Cum A B) where we may have $A = s1$ and $B = s2$ in the context of current assumptions, it is necessary to do the equality substitutions before using rule AX. Our present formulation of CUMSORT applies to any goal of shape (Cum A B), and returns some equality side conditions as subgoals. This does not make a very big difference (especially since LEGO's Claim command allows this same kind of deferring of goals), but I wanted to experiment with such *free-conclusioned* rules.

5.2.1.1 Cumulativity in ECC

This definition of Cum generalizes Luo's (definition 2.3 in [Luo94]) in two ways. As mentioned in section 5.1.3, our assumption cnv_pi, and the rule CUMPI intended to discharge that assumption, are contravariant in function domains, whereas Luo chose to require $\simeq$ on function domains. Also we use an arbitrary relation CumBase in place of Luo's particular base for ECC

$$\texttt{Prop} \preceq \texttt{Type}(\mathbf{0}) \preceq \texttt{Type}(\mathbf{1}) \ldots.$$

We do not require CumBase to be transitive.

Luo also requires $\preceq$ to be a partial order with respect to $\simeq$. While our `Cum` is reflexive with respect to $\simeq$ (i.e. contains $\simeq$) and transitive, it is not necessarily antisymmetric, as this property is not needed for the argument.

5.2.2 Properties of Cumulativity

We must check that `Cum` satisfies all the assumptions on `cnv` made so far. Most of this is straightfoward and tedious use of techniques described above. For example, we use a generalized inductive definition (similar to the systems `aVclosed` and `apts` of sections 3.2.5.3 and 4.4.4) to prove a better "structural induction principle" for `Cum`.

`The requirements of section 4.3.` `Cum` is reflexive because it contains the reflexive relation `conv`:

```
Goal Cum_refl: {A|Trm}(Vclosed A)->Cum A A;
```

`Cum` is transitive by rule CUMTRANS. Also:

```
Goal psub_resp_Cum:
     {N|Trm}{vclN:Vclosed N}
     {A,B|Trm}(Cum A B)->{p:PP}Cum (psub N p A) (psub N p B);
```

At this point it makes sense to define *cumulative PTS* (CPTS) as a PTS with its abstract conversion instantiated by `Cum`. Every CPTS has the PTS properties of section 4.4 through type correctness.

The requirements of section 4.4.9. We have

```
Goal Cum_red1:{A,B|Trm}(par_red1 A B)->(Vclosed A)->Cum A B;

Goal Cum_red1_sym:{A,B|Trm}(par_red1 A B)->(Vclosed A)->Cum B A;

Goal CumCR_pi:
     {u,v|VV}{A,A',B,B'|Trm}{c:Cum (pi u A B) (pi v A' B')}
     and (Cum A' A) ({q:PP}Cum (alpha q u B) (alpha q v B'));
```

Now we know that every CPTS has subject reduction.

The requirements of section 4.4.11. We have

```
Goal Cum_Vclosed: {A,B|Trm}(Cum A B)->and (Vclosed A) (Vclosed B);
```

```
Goal Cum_sort_character_l:
     {s|SS}{A|Trm}{c:Cum A (sort s)}Ex [t:SS] par_redn A (sort t);
Goal Cum_sort_character_r:
     {s|SS}{B|Trm}{c:Cum (sort s) B}Ex [t:SS] par_redn B (sort t);

Goal Cum_pi_character_l:
     {A,Bl,Br|Trm}{v|VV}{c:Cum A (pi v Bl Br)}
     Ex3 [u:VV][Al,Ar:Trm] par_redn A (pi u Al Ar);
```

In general we know nothing about `cnv_only_typedsort` for CPTS, but we do have that if `CumBase` has the property `cnv_only_typedsort` then so does `Cum`.

```
[CumBase_only_typedsort =
   {s1,s2|SS}(CumBase s1 s2)->
   or (is_tt (SSeq s1 s2)) (and (typedsort s1) (typedsort s2))];
Goal Cum_only_typedsort:
     {CBpt:CumBase_only_typedsort}
     {s1,s2|SS}(Cum (sort s1) (sort s2))->
     or (is_tt (SSeq s1 s2)) (and (typedsort s1) (typedsort s2));
```

The requirements of section 5.1.3. We have

```
Goal Cum_pi: {va,vb|VV}{Al,Ar,Bl,Br|Trm}{p|PP}
             {npoA:is_ff (poccur p Ar)}
             {npoB:is_ff (poccur p Br)}
             {l_prem:Cum Bl Al}
             {r_prem:Cum (vsub (par p) va Ar) (vsub (par p) vb Br)}
           (*******************************************************)
               Cum (pi va Al Ar) (pi vb Bl Br);
```

Again, we know nothing in general about `cnv_full_below`; in section 5.3.2 we will check that the particular CPTS, ECC, actually satisfies `cnv_only_typedsort` and `cnv_full_below`.

5.2.2.1 Decidability of Cumulativity

In sections 5.1.5.2 and 5.1.6.2 we use that `normalizing_decides_cnv` (page 110) is inhabited, although this could not be proved for the abstract relation `cnv`. `Cum` is still too abstract to have this property, since it depends on the abstract relation `CumBase`. Now we can prove only that if `Cum` is decidable on sorts then it is decidable on all normalizing terms:

```
[decideCumSorts = {s,t:SS}decidable (Cum (sort s) (sort t))];
```

```
Goal normalizing_decides_Cum:
     {dcs:decideCumSorts}
     {A,B|Trm}(Vclosed A)->(normalizing A)->(Vclosed B)->(normalizing B)->
     decidable (Cum A B);
```

It is clear that we want to prove this by some induction on A and B in order to handle the case where A and B reduce to pi-types, and $A \preceq B$ depends on cumulativity of some shorter terms, by the rule CUMPI. For definiteness, assume A and B are in normal form (weak-head normal form would do as well). Thus the main lemma required is

```
Goal beta_norm_decides_Cum:
     {dcs:decideCumSorts}
     {A,B|Trm}(beta_norm A)->(beta_norm B)->decidable (Cum A B);
```

Looking again at rule CUMPI, the contravariance of `Cum` on domains and covariance on ranges suggests that neither induction on A or on B will prove this lemma; we use induction on the sum of the lengths of A and B. With this idea the proof is straightfoward.

5.2.3 Type Synthesis and Type Checking for Cumulative PTS

We can cut in the properties proved about `Cum` in section 5.2.2

```
Cut [cnv=Cum]
    [cnv_refl=Cum_refl]
    [cnv_trans=CumTrans]
    [psub_resp_cnv=psub_resp_Cum]
    [cnv_red1=Cum_red1]
    [cnv_red1_sym=Cum_red1_sym]
    [cnvCR_pi=CumCR_pi]
    [cnv_Vclosed=Cum_Vclosed]
    [cnv_sort_character_l=Cum_sort_character_l]
    [cnv_sort_character_r=Cum_sort_character_r]
    [cnv_pi_character_l=Cum_pi_character_l];
```

The type synthesis algorithm, `sf_typSyn` of section 5.1.6.2 now has most of its global assumptions instantiated. For CPTS we have

```
Goal cpts_typSyn:
     {sf:semiFull}{f:Functional}
     {cf:cnv_full_below}{cpt:cnv_preserves_typedsort_dn}
     {dt:{s:SS}decidable (typedsort s)}
     {dr:{s:SS}decidable (ruledsort s)}
     {drs:{s1,s2:SS}decidable (ruledsorts s1 s2)}
     {dcs:decideCumSorts}
     {gtn:gts_term_Normalizing}
     {G:Cxt}{M:Trm}decidable (gtsTS G M);
```

Putting together `decide_gts` from section 5.1.6.2 with the results of the present section, we have a typechecking algorithm for CPTS

```
Goal decide_cpts:
     {sf:semiFull}{f:Functional}
     {cfb:cnv_full_below}{cpt:cnv_preserves_typedsort_dn}
     {crt:cnv_range_typedsort}
     {dt:{s:SS}decidable (typedsort s)}
     {dr:{s:SS}decidable (ruledsort s)}
     {drs:{s1,s2:SS}decidable (ruledsorts s1 s2)}
     {dcs:decideCumSorts}
     {gtn:gts_term_Normalizing}
     {G:Cxt}{M,A:Trm}decidable (gts G M A);
```

These depend on no unstated assumptions except the parameterization of CPTS on `PP`, `VV` and `SS`. In the next section we discharge all the stated assumptions except `gts_term_Normalizing`.

5.3 ECC

I will define ECC as a particular CPTS, and give type synthesis and typechecking algorithms for ECC which depend only on the stated assumption that typed terms are normalizing.

5.3.1 Definition of ECC

Taking the sorts of ECC to be NN with its decidable equality (section 2.2), we define the sorts, axioms and rules of ECC.

```
[ECCprop = Z];
[ECCtype [n:NN] = S n];
[ECCax [s1,s2:NN] = Q s2 (S s1)];
[ECCrl [s1,s2,s3:NN] =
       or (and (Q s2 Z) (Q s3 Z))                    (* impredicative rules *)
          (and (not (Q s2 Z)) (Q s3 (max s1 s2)))];  (*  predicative rules  *)
[ECCbase [s1,s2:NN] = Q s2 (S s1)];
```

We can cut in these definitions.

```
Cut [SS=NN][SSeq=nat_eq][SSeq_iff_Q=nat_eq_character]
    [CumBase=ECCbase]
    [ax=ECCax]
    [rl=ECCrl];
```

Although `ECCbase`, `ECCax`, and `ECCrl` occur in the context after `CumBase`, `ax`, and `rl`, these cuts can be made correct by expanding some definitions; this is what LEGO does.

Remark 5.9 (On intensionality of presentation.) *Is ECC a set of derivable judgements or a set of derivations? We could replace the definition of `ECCax` above with*

```
[ECCax [s1,s2:NN] = Lt s1 s2];
```

This new system has the same derivable judgements as the old one, but it has strictly more derivations of these judgements, and is not functional, *so our developed theory of typechecking for functional PTS does not directly apply. We have been reasoning about a set of derivations, using intensional knowledge for inductive proofs.*

5.3.2 Properties of ECC

This presentation of ECC is functional (section 4.6), with `ax` and `rl` given by total functions.

```
Goal ECC_functional: Functional;
```

ECC is trivially semifull, in fact full.

```
Goal ECC_semiFull: semiFull;
```

Since ECC has no topsorts, we easily meet our anti-topsort requirements

```
Goal ECC_no_topsorts: {s:SS}typedsort s;
Goal ECC_preserves_typedsort_dn:
     {s1,s2|SS}(cnv (sort s1) (sort s2))->(typedsort s2)->typedsort s1;
Goal ECC_range_typedsort: {s1,s2|SS}(cnv (sort s1) (sort s2))->
                          or (is_tt (SSeq s1 s2)) (typedsort s2);
```

and for the same reason, `decideTypedsort` is trivial:

```
Goal ECCdecideTypedsort: {s:SS}decidable (typedsort s);
```

For ECC, conversion of sorts is the same as $\leq$ on natural numbers.

```
Goal ECC_Cum_Le: {t1,t2|SS}(cnv (sort t1) (sort t2))->Le t1 t2;
Goal Le_ECC_Cum: {t1,t2|SS}(Le t1 t2)->cnv (sort t1) (sort t2);
```

Thus it is easy to show:

```
Goal ECC_full_below: {s1,s2,t1,t2,t3|SS}
                     {cv1:cnv (sort s1) (sort t1)}
                     {cv2:cnv (sort s2) (sort t2)}
                     {rlt:rl t1 t2 t3}
                     Ex [s3:SS] and (rl s1 s2 s3) (cnv (sort s3) (sort t3));
```

From `ECC_Cum_Le`, `Le_ECC_Cum`, and decidability of `Le` we easily have that `cnv` ($\leq$, which is now also $\preceq$) is decidable on sorts (i.e. $\leq$ on natural numbers):

```
Goal decideECCSorts: {s,t:SS}decidable (cnv (sort s) (sort t));
```

Combining this with `normalizing_decides_Cum` (section 5.2.2.1), we have finally proved that, for ECC, cumulativity is decidable for all normalizing terms.

Also every sort is a `ruledsort`, and the decidability of `ruledsorts` is trivial.

```
Goal ECC_all_ruledsorts: {s1,s2:SS}ruledsorts s1 s2;
Goal ECCdecideRuledsorts: {s1,s2:SS}decidable (ruledsorts s1 s2);
Goal ECCdecideRuledsort: {s:SS}decidable (ruledsort s);
```

5.3.3 Type Synthesis and Type Checking for ECC

Specializing `cpts_typSyn` we have a type synthesis algorithm

```
Goal ECC_typSyn: (gts_term_Normalizing)->{G:Cxt}{M:Trm}decidable (gtsTS G M);
```

Specializing `decide_cpts` we have

```
Goal decide_ECC: gts_term_Normalizing->{G:Cxt}{M,A:Trm}decidable (gts G M A);
```

These both depend on the parameterization of ECC on `PP` and `VV`. Instantiating `PP` and `VV` with `NN` as in section 3.2.7

```
Cut [PP=NN][PPeq=nat_eq][PPeq_iff_Q=nat_eq_character][PPinf=NNinf]
    [VV=NN][VVeq=nat_eq][VVeq_iff_Q=nat_eq_character][VVinf=NNinf];
```

`ECC_typSyn` and `decide_ECC` now depend only on the explicit assumption `gts_term_Normalizing`.

5.4 Further Work: Executable Typecheckers?

In section 5.1.6.3 we have a type synthesis algorithm, `gts_typSyn`, and a type checking algorithm, `decide_gts`, for a class of PTS. These two lemmas depend on many properties; while in many cases of interest these assumptions are satisfied, there are examples where we cannot prove that some assumption is satisfied (e.g. normalizability of well-typed terms in ECC), or where some assumption is actually not satisfied (e.g. decidability of `typedsort` in the PTS of remark 5.8). It is derirable to be able to program typechecking algorithms that are sound, even if incomplete[7]. As non-normalizability of well-typed terms is the most interesting example of this problem, I will consider partial correctness of TS and TC given a partially correct normalization program, and then, briefly, consider questions of efficiency.

5.4.1 Partial Correctness

λP meets the assumptions of `decide_gts`, and I expect to be able to formally prove that λP is normalizing, and produce a normalization algorithm for it. The Calculus of Constructions meets the assumptions of `decide_gts`, but it is a real challenge to formally prove normalization of CC in ECC. It is very probably not possible to prove that ECC is normalizing within ECC.

[7] We might want a semi-decision procedure, or just pragnatic incompleteness. I am not being too precise in this section, as it is partiality I want to discuss, not algorithmic content.

Further, we might be interested in typechecking a system like $\boldsymbol{\lambda}^*$ [Bar91] which gives types to non-normalizing terms, but otherwise meets all the assumptions of `decide_gts`.

It is possible to write a program in ECC that, for any n, computes n steps of reduction on any lambda term (I leave open how to count the steps), and thus to write a partially correct normalization function that returns either a normal form, or indication that (up to n steps) no normal form has been found. This is enough to write a partially correct type synthesis program and a partially correct typechecker for any PTS meeting all the assumptions of `decide_gts` except, possibly, normalization.

Representing partial functions. To represent partiality in ECC, I will use the notion of an *option type*. For example, in SML there is the type

```
datatype 'a option = SOME of 'a | NONE;
```

There are two ways to produce an element of `'a option`, one of them, `SOME`, requires evidence, i.e. an object of type `'a`, while the other, `NONE`, requires nothing. An object of type `'a option` can be destructed to see if it contains evidence or not. For our purpose of representing partiality, `NONE` is better named `MAYBE`, because it contains no information at all, not the information that some computation actually fails to terminate.

5.4.1.1 Partial Normalization Functions

Inductively define a predicate $\mathsf{optNormalizing}\,(M)$ with the constructors

$$\frac{\mathsf{normalizing}\,(M)}{\mathsf{optNormalizing}\,(M)}\ \text{ONSOME} \qquad \frac{}{\mathsf{optNormalizing}\,(M)}\ \text{ONMAYBE}.$$

Any proof of

$$\forall M\ .\ \ldots\ \Rightarrow\ \mathsf{optNormalizing}\,(M)$$

is a sound normalization program: given M (and possibly some other data) it returns either a normal form of M, or no information.

The particular normalization function required by `gts_typSyn` and `decide_gts` instantiates the specification `gts_term_Normalizing`,

$$\forall M\ .\ \mathsf{gts\_term}(M)\ \Rightarrow\ \mathsf{normalizing}(M). \tag{5.1}$$

It is worth considering the use of the premise $\mathsf{gts\_term}\,(M)$ in specification (5.1). For the supplier of a program meeting this specification, this premise provides a “recursor” necessary for normalizing M, as general recursion is not available in ECC. Since our plan for implementing partial normalization is just to compute n steps of some unspecified reduction strategy, we don't need this premise. However, for the user of the normalization program, i.e. a TS or TC program, this premise is hygienic: it prevents such a user from committing to normalization of a term until it is known that this term is really well typed, thus preventing possible incompleteness. If we hope to program a semi-decision procedure for ECC, we must keep this premise. Even for a non-normalizing PTS such as $\boldsymbol{\lambda}^*$, it is hard to find non-normalizing typable terms, so we might well decide to keep this premise in a specification of partial normalization, even though this premise is not used in the computation of partial normalization.

If we are satisfied to typecheck PTS that we believe are normalizing, then the reduction strategy used in a partial normalization program is not of (theoretical) importance. However, if we want to typecheck non-normalizing PTS, it would be wise to use a reduction strategy, $\longrightarrow$, that is *cofinal* in the sense that if $A \twoheadrightarrow B$ then $\exists C.\, A \longrightarrow\!\!\!\!\!\twoheadrightarrow C$ and $B \twoheadrightarrow C$. An example of such a strategy is *complete development*, that contracts all the redexes in a term at once. (In section 7 of [vBJMP94] we use this relation for reasoning about non-normalizing PTS.) This relation, which we use for our proof of Church-Rosser, is already formalized in LEGO.

5.4.1.2 Partial TS and TC Functions

Inductively define a relation $\mathsf{optTypChk}(\Gamma, M, A)$ with the constructors

$$\frac{\Gamma \vdash M : A}{\mathsf{optTypChk}(\Gamma, M, A)}\ \text{OTCSOME} \qquad \frac{}{\mathsf{optTypChk}(\Gamma, M, A)}\ \text{OTCMAYBE}.$$

Any proof of

$$\forall \Gamma, M, A\ .\ \ldots\ \Rightarrow\ \mathsf{optTypChk}(\Gamma, M, A) \tag{5.2}$$

is a sound typechecker: given Γ, M, A (and possibly some other data) it returns either a derivation of $\Gamma \vdash M : A$, or no information. Slightly modifying the specification of `decide_gts` we have:

$$\begin{array}{ll} \forall \Gamma, M, A\ . & \ldots \\ & (\forall M\ .\ \mathsf{gts\_term}(M)\ \Rightarrow\ \mathsf{optNormalizing}(M))\ \Rightarrow \\ & \mathsf{optTypChk}(\Gamma, M, A). \end{array} \tag{5.3}$$

A proof of specification (5.3) is at hand: the proof of `decide_gts` given above will do, except that we call the partial normalization program instead of a total normalization program, and fail if the normalization program fails. Also we can omit the reasoning justifying the negative disjuncts of that lemma, as we have now allowed ourselves to return "no information" without any evidence at all.

Remark 5.10 *We could throw away less information than I have suggested by having a third constructor for `optTypChk`*

$$\frac{\neg \Gamma \vdash M : A}{\mathit{optTypChk}\,(\Gamma, M, A)}\ \textsc{otc}\mathrm{NONE}.$$

With this definition, the proof of specification (5.3) can retain the information on failure that the proof of `decide_gts` already contains.

Remark 5.11 *For brevity I have shown the partial versions of only the normalizing and TypChk relations. In fact, partiality propagates throughout a proof of specification (5.3), and we need partial versions of several other relations I have mentioned, such as whether a term reduces to a sort, and whether two terms convert.*

Although we can see that the only cause of failure of our partial typechecker is failure of the given partial normalization program, specification (5.3) is very weak, as it allows returning OTCMAYBE without any justification. In order to be more precise, we can fix a reduction strategy, and index all our relations with the number of steps of reduction they may use. Then we can express that "if a judgement is derivable there is some n such that the typechecker succeeds on that judgement when allowed at least n steps of reduction". This can work even for non-normalizing PTS if we use a cofinal reduction strategy (see section 5.4.1.1). Some technical details of such an approach are worked out in section 7 of [vBJMP94].

5.4.2 Efficiency

A proof of `decide_gts` or specification (5.3) is a (partially) correct typechecking program for some type theories we are interested in. Can we expect to *actually* run such a type checker?

One reason why we cannot actually run our typechecker is that the number of rule applications in an sdsf-derivation tree is exponential in the size of the root (the conclusion). The reason for this is discussed in section 4.4.10.2, where a system, `lvtyp`, which avoids this blow-up is presented. `lvtyp` is more difficult to reason about than `gts`, and I have not yet carried its development through to a typechecking algorithm, but we cannot expect to execute infeasible algorithms, so must be able to handle such difficult arguments.

Another reason we cannot run our partially correct typechecker on the currently distributed LEGO is that LEGO is *very* slow at computing in its object languages. Sorting short lists has been known to take hours; an enterprising user [Bai93] actually burned 56 hours on a big workstation factoring a small polynomial. One reason for this is that LEGO, built to be an interactive proofchecker, does not use internal representation selected for fast computation, but for simplicity and a clear correspondence with the user's concrete representation. However, there is some recent work on the problem of "intensional representations" with efficient computation, e.g. [NW93], and there is no reason why a proofchecker cannot be much better than LEGO in this regard.

Finally, and most difficult in the long term, is the problem of efficiently executing the computational content of constructive proofs. There is now much literature about program extraction from constructive proofs, and some type theory implementations, such as Nuprl [Con86] and Coq [DFH+93] have an extraction mechanism, although LEGO does not. Whether such approaches can produce feasible programs from a proof of something like `decide_gts` remains to be seen.

Chapter 6

What Does It All Mean?

> At the moment you find an error, your brain may disappear because of the Heisenberg uncertainty principle, and be replaced by a new brain that thinks the proof is correct.
>
> Leonid A. Levin, Boston University, quoted in [Hor93]

This chapter is entirely informal, and somewhat tentative. It expresses my currently held opinion about some questions underlying the whole enterprise of mechanized mathematics.

In previous chapters I have outlined a large formal development that is checked in LEGO. Consider one theorem in the development, for example strengthening for arbitrary PTS (section 4.5.2).

```
Goal gts_strengthening:
     {Gamma|Cxt}{c,C,d,D|Trm}{Delta:Cxt}
     {q|PP}{noccd:is_ff (poccur q d)}
           {noccD:is_ff (poccur q D)}
           {noccDelta:is_ff (POCCUR q Delta)}
     {premD:gts (append Delta (CONS (Gb q C) Gamma)) d D}
      gts (append Delta Gamma) d D;
```

To check this theorem requires checking over 24,000 lines of LEGO source in over 60 files, which create over 2000 definitions and theorems in the LEGO state. LEGO itself is a program, written in SML, of around 7,000 lines of code. LEGO is usually compiled in New Jersey SML, itself a large programming system, and runs on computer networks whose complexity is incalculable. What are you to conclude from my claim that "we have formally checked a proof of strengthening for arbitrary PTS in LEGO"?

I consider this question in two parts, and examine some wider implications. In section 6.1 I ask "is it a theorem", e.g. is what I have called the strengthening lemma a derivable judgement in ECC? This is the question of correctness of LEGO, and of proof checkers in general, with respect to some given underlying formal system.

The second part of the question is about the meaning of a derivable judgement that may be too large to examine in detail yourself. What I have called the strengthening lemma, including all the definitions (hereditarily) used in its statement, is very long. The proof of what I have called the strengthening lemma is much longer still. Assuming that this proof derives a correct judgement in ECC, what does it tell you about the strengthening lemma? For example, how can you experience belief in strengthening for arbitrary PTS? These questions are addressed in section 6.2.

6.1 Is it a Theorem?

I bring you my 60 LEGO source files and say "Here is a derivation in ECC, you can check it in LEGO" (we are leaving aside the question of what it is a derivation of until section 6.2). How can you believe this thing is a correct derivation in ECC? One immediate question is how to parse the 60 files, or, more generally, how to read them; this is asked in section 6.1.2. Leaving that aside, I suggest that the way for you to believe that some abstract formal entity is a correct derivation is to check it yourself; but if you can't read it all with your own eyes, and interpret it all with your own brain, you reason indirectly, as we almost always do anyway. In this case, you write a proofchecker for the logic in question.

Any entity I call a theorem prover will check derivations in some formally specified object logic, and produce a fully annotated proof object that is *independently checkable* to witness each claimed theorem. Such a prover may use all kinds of tactics, search, unification, etc, but at the end must produce proof objects that are checkable in some elementary and formally specified way, such as syntactically matching with the rules of the object logic. This is not a completely new idea (for example [DB93]), but I mean to take it as part of a serious answer to the question of surveyability of formal proofs.

If you want to believe that some judgement constructed in such a prover is "really" derivable, you write a simple checker for the object logic, and check the proof object generated by the prover to witness that judgement. If your simple checker doesn't accept the proof object, you can go back to the proof creator (who developed the proof with some user-friendly theorem proving tool) and say "step 3946782 claims to use rule X, but premise 2 is not satisfied". Then you must work out the problem between you: did you understand rule X when you wrote your simple checker, is there an error in the complex theorem proving tool, or did some more complicated misunderstanding show up?

I'm addressing the psychological question of belief. To me this approach of writing your own proof checker is both mathematically and psychologically more satisfactory than using someone else's "completely verified" proof system; if you are given a completely verified proof system, all you really have is another big proof (its verification) to check. In my approach the creator of the proof is responsible for providing all the information necessary to understand

and believe the theorem, but the reader is responsible for actually understanding and believing it. You don't expect to directly read and understand large proofs that are completely formal and partially machine generated; but you understand the means of checking them (i.e. the mathematics of deciding derivability of judgements in the logic), you implement those means yourself, and if that implementation accepts a large proof object, you have evidence to believe it is a theorem.

If the responsibity mentioned in the previous paragraph is interpreted in terms of "safety critical" software, my suggestion is also organizationally more satisfactory. A mathematician using a full-featured proof tool proves that a specification is met. This can be checked independently, and there the mathematician's responsibility ends; s/he is not responsible for the characteristics of a physical product based on this specification.

6.1.1 A Few Details About Simple Proof Checkers

I have made it too simple above. For most logics of serious interest, a feasible checker cannot just syntactically match against the rules of the logic, but depends on its mathematical properties. Let me informally use the words *simple proof checker* (SPC) for one that only checks fully annotated proofs in some given object logic. An SPC needs no form of user programmed extensibility (but it may use tactics to keep the kernel small). It may be designed to check top down or bottom up. I am not suggesting that an SPC ever be used by a human being for formalizing known proofs, much less discovering proofs. An SPC should be used to check proofs discovered and constructed on more user-friendly proof checkers.

We have to be a little careful in detail; for example LEGO checks judgements incrementally, as in the system `lvtyp` (section 4.4.10.2), and writes out annotated objects incrementally. (This is how LEGO's seperate compilation of modules works; the `.o` files LEGO produces are the annotated objects.) When these annotated objects are later checked, LEGO again incrementally maintains a state that represents a correct judgement. This clearly needs some explanation, but `lvtyp` is most of that explanation[1]; we don't require an explanation of unification, refinement, rewriting, etc. None of this is necessary to check `.o` files.

Another point to be careful about is outstanding assumptions. In LEGO's incrementally constructed judgements there may be assumptions in the context that are not explicit in the statement of a particular theorem. Since I do not want to require the human mathematician to read the whole annotated proof, we should require that the proofchecker alerts you to all such assumptions. That is, the "official" judgements of the object logic are themselves too big to read, so we should define some derived, or specialized, judgement form, which flags things we are interested in. For example, we could define a judgement of shape $G \vdash^{names} M : A$,

---

[1] The main thing required, that I haven't discussed at all in this thesis, is *definitions*.

where $\boldsymbol{names}$ is a list of the names of undischarged assumptions in $\boldsymbol{G}$; alternatively we might require that for "official" checking, only assumption-free, fully discharged contexts are accepted.

An important issue for an SPC is computational feasibility of proofchecking. For example, it is completely infeasible to build an SPC for ECC based on the system `gts`. LEGO is based on combining the ideas of `lvtyp` and `sdsf`, and I don't know any other way to build an SPC for ECC that could check the strengthening theorem. This shows that the idea of this section is not as black and white as I may have made it appear. We cannot, in practice, take just any formal system and make it the specification of an SPC, if we hope to check non-trivial theorems. Mathematical work will be required to find a feasibly implementable specification and show its correctness. Beyond that it may be necessary to prove some rules to be admissible, such as weakening or strengthening, since actually executing the algorithm that shows they are admissible may also be infeasible.

6.1.2 Syntax Must be Explained

What I have called a proof of the strengthening lemma is actually a very long ascii string (ignoring the organization into files). In order to ask questions about correctness, we must know how to parse it, and this should be formally explained. Fortunately, parsing is one of the formally best understood areas of computer science, so this should be no problem. Extensible parsers are often used in proof systems, and this is a useful thing, but it is not always observed that proof system implementors and users must accept the limits of formal parsing.

6.2 Informal Understanding of a Formal Theorem

Having suggested in section 6.1 how one might come to believe that a large object is a derivation of a judgement in ECC, there is still a question: whether this formalization correctly represents our *informal* notions of term, substitution, reduction, ... This is an essentially informal question which cannot be settled by appeal to some formal system or machine; to answer it we must read (some part of) the formal development and *decide for ourselves*. In section 6.2.1 I bite the bullet and discuss how to experience understanding of a formal development. In section 6.2.2 I relent slightly to consider how choosing suitable representations can simplify the task of understanding a formal development. In light of section 6.1, I assume you believe some "thing" is a correct judgement, and want to know what informal understanding it represents.

6.2.1 How to Read a Formal Proof

You receive a document in the post from an unknown mathematician, claiming to be a proof of the euclidean algorithm. You read it, understand it, and believe it. It is not a completely formal proof, but you believe that "in principle" it could be formalized in some system that you and the author agree on. As part of reading and understanding the document, you have read the statement of the theorem and whatever definitions it depends on, and have decided that the statement agrees with your informal understanding of the euclidean algorithm (i.e. the meaning of its statement). This last involves, besides your informal understanding of the euclidean algorithm, some "in principle" understanding of how to associate informal understanding with statements in the implicitly agreed formal system.

In order to believe strengthening for PTS, at the least, you must read the statement of the theorem, and decide if that satisfies your informal understanding. Of course, the statement will use some defined notions, which you must also read and decide about for yourself, continuing this process until you have decided for yourself about all the definitions used (hereditarily) in the statement. It seems to me that this process is both tractable and unavoidable; that is how informal mathematics is done. Importantly, to know if the formal statement is the theorem we have in mind, it is not necessary to read any of the auxiliary definitions and lemmas used in the proof but not occurring in the (fully expanded) statement.

6.2.1.1 Declarations

Is deciding about the hereditarily occurring definitions enough? No, we must also examine *all* of the declarations, i.e. the variables that are assumed to inhabit types. In LEGO this is complicated by two issues:

1. use of declarations in our coding of inductive types (section 2.1.5),
2. use of declarations for some kinds of generality and abstraction (e.g. section 3.2.7).

Declarations Coding Inductive Types Declarations generated by LEGO's inductive types tactic are justified by Luo's schema [Luo94], and are part of the mechanics of ECC, not logical assumptions that we must understand and agree with. Although you may feel the need to read an inductive definition and decide that it satisfies your informal understanding, you should not feel the need to examine the various declarations that are mechanically generated for its implementation.

Declarations for Generality and Abstraction In section 3.2.7 I said that you should insist on concrete instances of `PP` and `VV`, and proofs that they have the properties assumed for these types, but that you should accept the abstraction `SS` as part of the meaning of a theorem about pure languages or `PTS`. I think that `ax` and `rl` are in the latter category, while `cnv` is in the former. If you are trying to believe some theorem because of a formal proof, it is up to you to decide which formal assumptions are "abstract datatypes", like `PP` and `VV`, which you want to see instantiated, and which are really part of the theorem itself.

My style of formalization is unsatisfactory in that it does not formally distinguish these two classes of assumption. I don't know whether it is my use of ECC that is deficient, or ECC itself. This is a serious area for further work.

6.2.2 Other Representations

You may find a formalization too obscure to understand at all. For example, our formalization of lambda calculus is very idiosyncratic, with its two classes of names and funny substitution operations. Given that binding and substitution are notoriously prone to incorrect definition, you might well feel that what we call the Church-Rosser theorem is not convincing for you. Then you must find a representation you can accept, and show it is equivalent in some sense with the given formalization. The idea is not to re-prove all the theorems, but only to translate an obscure development into another representation.

There are obviously advantages to having several related formal representations. One representation may be more natural for some readers, while other readers prefer a different representation; some theorems are easier to state or prove in one representation than in another. I believe one reason informal mathematics seems natural is that, implicitly, different representations are used for different purposes. For example, it is common in writing on lambda calculus to use informal named terms, with the caveat that if some question of name clash ever arises the "official" notation is nameless terms.

For the purposes of this section on how to read a formal proof, an important point is that one representation might require us to read significantly less of the formal development than another. For example we might define a type of nameless terms, give a translation between this type and our type `Trm`, and show that this translation commutes with reduction. Then a version of the Church-Rosser Theorem for nameless terms could be proved very easily, just by translating our theorem for `Trm`, but the list of definitions hereditarily used in the statement of this nameless CR theorem might be much shorter than for our statement of CR in section 3.3.2.1; for example, it won't contain anything related to `Vclosed`.

Bibliography

[ACN90] L. Augustsson, Th. Coquand, and B. Nordstrom. A short description of another logical framework. In *Informal Proceedings of First Workshop on Logical Frameworks, Antibes*, May 1990. Available by ftp.

[AHMP92] Arnon Avron, Furio Honsell, Ian Mason, and Robert Pollack. Using typed lambda calculus to implement formal systems on a machine. *Journal of Automated Reasoning*, 9:309–352, 1992.

[Alt93a] Thorsten Altenkirch. *Constructions, Inductive Types and Strong Normalization*. PhD thesis, University of Edinburgh, 1993.

[Alt93b] Thorsten Altenkirch. A formalization of the strong normalization proof for System F in LEGO. In *Proceedings of the International Conference on Typed Lambda Calculi and Applications, TLCA'93*. Springer-Verlag, LNCS 664, March 1993.

[Bai93] Anthony Bailey. Representing algebra in LEGO. Master's thesis, University of Edinburgh, 1993.

[Bar84] H. P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, Amsterdam, revised edition, 1984.

[Bar91] Henk Barendregt. Introduction to Generalised Type Sytems. *J. Functional Programming*, 1(2):125–154, April 1991.

[Bar92] Henk Barendregt. Lambda calculi with types. In Abramsky, Gabbai, and Maibaum, editors, *Handbook of Logic in Computer Science*, volume II. Oxford University Press, 1992.

[Ber90a] Stefano Berardi. Girard normalization proof in LEGO. In *Informal Proceedings of First Workshop on Logical Frameworks, Antibes*, May 1990. Available by ftp.

[Ber90b] Stefano Berardi. *Type Dependence and Constructive Mathematics*. PhD thesis, Dipartimento di Informatica, Torino, Italy, 1990.

[Bra92] Steven Bradley. A model of CCS in LEGO. Master's thesis, University of Edinburgh, 1992.

[Car91] Luca Cardelli. F-sub, the system. Technical report, DEC Systems Research Centre, 1991.

[CH85] Thierry Coquand and Gérard Huet. Constructions: A higher order proof system for mechanizing mathematics. In *Proceedings of the European Conference on Computer Algebra, EUROCAL'85*. Springer-Verlag, April 1985. LNCS 203.

[CH88] Thierry Coquand and Gérard Huet. The calculus of constructions. *Information and Computation*, 76(2/3):95–120, February/March 1988.

[Ciz93] Petr Cizek. A variation on Lagrange's theorem in LEGO: Characterizing the numbers that are the sum of two squares. Talk given at the Nijmegen Workshop on Types for Proofs and Programs, May '93, 1993.

[Col90] Andrew Coleman. Simulating VDM in LEGO. Master's thesis, University of Edinburgh, 1990.

[Con86] Robert L. Constable, *et. al. Implementing Mathematics with the Nuprl Proof Development System*. Prentice–Hall, Englewood Cliffs, NJ, 1986.

[Coq90] Thierry Coquand. Some comments on Pure Type Sytems. Unpublished lecture notes from the Summer School and Conference on Proof Theory, Leeds University, July 1990.

[Coq91] Thierry Coquand. An algorithm for testing conversion in type theory. In G. Huet and G. D. Plotkin, editors, *Logical Frameworks*. Cambridge University Press, 1991.

[CPM90] Th. Coquand and C. Paulin-Mohring. Inductively defined types. In P. Martin-Löf and G. Mints, editors, *Proceedings of Colog'88*, number 417 in LNCS. Springer-Verlag, 1990.

[dB80] Nicolas G. de Bruijn. A survey of the project AUTOMATH. In J. P. Seldin and J. R. Hindley, editors, *To H. B. Curry: Essays in Combinatory Logic, Lambda Calculus, and Formalism*, pages 589–606. Academic Press, 1980.

[dB91] N. de Bruijn. A plea for weaker frameworks. In G. Huet and G. D. Plotkin, editors, *Logical Frameworks*. Cambridge University Press, 1991.

[DB93] Gilles Dowek and Robert Boyer. Towards checking proof checkers. In Herman Geuvers, editor, *Informal Proceedings of the Nijmegen Workshop on Types for Proofs and Programs*, May 1993.

[DFH+93] Dowek, Felty, Herbelin, Huet, Murthy, Parent, Paulin-Mohring, and Werner. The Coq proof assistant user's guide, version 5.8. Technical report, INRIA-Rocquencourt, February 1993.

[Dyb91] Peter Dybjer. Inductive sets and families in Martin-Löf's type theory and their set-theoretic semantics. In G. Huet and G. Plotkin, editors, *Logical Frameworks*, pages 213–230. Cambridge University Press, 1991.

[Dyb94] Peter Dybjer. Inductive families. *Formal Aspects of Computing*, 6:1–26, 1994.

[Fef88] Solomon Feferman. Finitary inductively presented logics. In *Logic Colloquium '88, Padova*. August 1988.

[Gar92] Philippa Gardner. *Representing Logics in Type Theory*. PhD thesis, University of Edinburgh, July 1992.

[Geu93] Herman Geuvers. *Logics and Type Systems*. PhD thesis, Department of Mathematics and Computer Science, University of Nijmegen, 1993.

[GN91] Herman Geuvers and Mark-Jan Nederhof. A modular proof of strong normalization for the calculus of constructions. *Journal of Functional Programming*, 1(2):155–189, April 1991.

[Gor88] Michael Gordon. HOL: A proof generating system for higher-order logic. In Birtwistle and Subrahmanyam, editors, *VLSI Specification, Verification and Synthesis*, pages 73–128. Kluwer Academic Publishers, 1988.

[Hey71] A. Heyting. *Intuitionism, An Introduction*. Studies in Logic. North-Holland, 1971.

[HHP92] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the ACM*, 40(1):143–184, 1992. Preliminary version in LICS'87.

[Hor93] John Horgan. The death of proof. *Scientific American*, pages 74–82, October 1993.

[HP91] Robert Harper and Robert Pollack. Type checking with universes. *Theoretical Computer Science*, 89:107–136, 1991.

[HT94] Masami Hagiya and Yozo Toda. On implicit arguments. In N. Jones, M. Hagiya, and M. Sato, editors, *Logic, Language and Computation*, pages 10–30. Springer-Verlag, LNCS 792, 1994. Festschrift in honor of Satoru Takasu.

[Hue87] Gérard Huet. Extending the calculus of constructions with Type:Type. Unpublished manuscript, April 1987.

[Hue89] Gérard Huet. The constructive engine. In R. Narasimhan, editor, *A Perspective in Theoretical Computer Science*. World Scientific Publishing, 1989. Commemorative Volume for Gift Siromoney.

[JM93] Claire Jones and Savi Maharaj. The LEGO library. Available by ftp with LEGO distribution, 1993.

[Jon93] Claire Jones. Completing the rationals and metric spaces in LEGO. In G. Huet and G. Plotkin, editors, *Logical Environments*, pages 297–316. Cambridge University Press, 1993. Proceedings of the Second Workshop of the ESPRIT BRA on Logical Frameworks.

[JP93] Claire Jones and Randy Pollack. Incremental changes in LEGO: 1993. Available by anonymous ftp with LEGO distribution, May 1993.

[JP94] Claire Jones and Randy Pollack. Incremental changes in LEGO: 1994. Available by anonymous ftp with LEGO distribution, May 1994.

[LP92] Zhaohui Luo and Robert Pollack. LEGO proof development system: User's manual. Technical Report ECS-LFCS-92-211, LFCS, Computer Science Dept., University of Edinburgh, The King's Buildings, Edinburgh EH9 3JZ, May 1992. Updated version. Available by anonymous ftp with LEGO distribution.

[Luo90a] Zhaohui Luo. *An Extended Calculus of Constructions*. PhD thesis, Department of Computer Science, University of Edinburgh, June 1990.

[Luo90b] Zhaohui Luo. A problem of adequacy: Conservativity of calculus of constructions over higher-order logic. Technical Report ECS-LFCS-90-121, LFCS, Edinburgh University, 1990.

[Luo94] Z. Luo. *Computation and Reasoning: A Type Theory for Computer Science*. International Series of Monographs on Computer Science. Oxford University Press, 1994.

[Mah90] Savitri Maharaj. Implementing Z in LEGO. Master's thesis, University of Edinburgh, 1990.

[Mah94] Savitri Maharaj. Encoding Z schemas in type theory. In Henk Barendregt and Tobias Nipkow, editors, *Types for Proofs and Programs: International Workshop TYPES'93, Nijmegen, May 1993*, volume 806 of *LNCS*, pages 238–262. Springer-Verlag, 1994.

[Mar71a] Per Martin-Löf. Haupsatz for the intuitionistic theory of iterated inductive definitions. In J.E. Fenstad, editor, *Proceedings of the Second Scandinavian Logic Symposium*, pages 179–216. North Holland, 1971.

[Mar71b] Per Martin-Löf. A theory of types. Technical Report 71-3, University of Stockholm, 1971.

[McK92] James McKinna. *Deliverables: a Categorical Approach to Program Development in Type Theory*. PhD thesis, University of Edinburgh, 1992.

[McK94] James McKinna. Typed $\boldsymbol{\lambda}$-calculus formalized: Church-Rosser and standardisation theorems. In preparation, 1994.

[Men92] Nax Mendler. A model of constructive set theory in LEGO. Talk given by Peter Aczel at the Båstad Workshop on Logical Frameworks, May '92, 1992.

[Moh86] Christine Mohring. Algorithm development in the calculus of constructions. In *Proceedings of the Symposium of Logic in Computer Science*, June 1986. Cambridge Mass.

[MP93] James McKinna and Robert Pollack. Pure Type Sytems formalized. In M.Bezem and J.F.Groote, editors, *Proceedings of the International Conference on Typed Lambda Calculi and Applications, TLCA'93*, pages 289–305. Springer-Verlag, LNCS 664, March 1993.

[MTH90] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT Press, 1990.

[NPS90] Bengt Nordström, Kent Petersson, and Jan Smith. *Programming in Martin-Löf's Type Theory. An Introduction*. Oxford University Press, 1990.

[NW93] Gopalan Nadathur and Debra Sue Wilson. A notation for lambda terms I: A generalization of environments. Technical Report Technical Report CS-1993-22, Duke University, 1993.

[Pau93a] Lawrence C. Paulson. The Isabelle reference manual. Technical Report 283, The University of Cambridge Computer Laboratory, 1993. Available by ftp, along with the implementation.

[Pau93b] Lawrence C. Paulson. Set theory for verification: II. induction and recursion. Technical Report 312, The University of Cambridge Computer Laboratory, 1993. Available by ftp.

[Pfe89] Frank Pfenning. Elf: A language for logic definition and verified metaprogramming. In *Proceedings of the Fourth Annual Symposium on Logic in Computer Science, Asilomar, California*, June 1989.

[PM93] Christine Paulin-Mohring. Inductive definitions in the system coq; rules and properties. In M.Bezem and J.F.Groote, editors, *Proceedings of the International Conference* *on Typed Lambda Calculi and Applications, TLCA'93*, pages 328–345. Springer-Verlag, LNCS 664, March 1993.

[Pol88] Robert Pollack. The theory of LEGO. Thesis proposal, October 1988.

[Pol90] Robert Pollack. Implicit syntax. Informal Proceedings of First Workshop on Logical Frameworks, Antibes, May 1990.

[Pol92] R. Pollack. Typechecking in Pure Type Sytems. In *Informal Proceedings of the 1992 Workshop on Types for Proofs and Programs, Båstad, Sweden*, pages 271–288, June 1992. Available by ftp.

[Pol94] Robert Pollack. Closure under alpha-conversion. In Henk Barendregt and Tobias Nipkow, editors, *Types for Proofs and Programs: International Workshop TYPES'93, Nijmegen, May 1993*, volume 806 of *LNCS*, pages 313–332. Springer-Verlag, 1994.

[Pra73] D. Prawitz. Towards a foundation of a general proof theory. In P. Suppes *et. al.*, editor, *Logic, Methodology, and Philosophy of Science IV*, 1973.

[Pra74] D. Prawitz. On the idea of a general proof theory. *Synthese*, 27, 1974.

[Pym90] David Pym. *Proofs, Search and Computations in General Logic*. PhD thesis, Department of Computer Science, Edinburgh University, 1990.

[Smi88] Jan Smith. The independence of Peano's fourth axiom from Martin-Löf's type theory without universes. *Journal of Symbolic Logic*, 53(3), 1988.

[SP94] Paula Severi and Erik Poll. Pure type systems with definitions. In *LFCS'94*, number 813 in LNCS, pages 316–328. Springer-Verlag, 1994.

[Tak] Masako Takahashi. Parallel reductions in λ-calculus (revised version). *Information and Computation*. To appear. Previous version in *Journal of Symbolic Computation* (7) 113-123 (1989).

[Tas93] A. Tasistro. Formulation of Martin-Löf's theory of types with explicit substitutions. Master's thesis, Chalmers Tekniska Högskola and Göteborgs Universitet, May 1993.

[vBJ93] L.S. van Benthem Jutting. Typing in Pure Type Sytems. *Information and Computation*, 105(1):30–41, July 1993.

[vBJMP94] L.S. van Benthem Jutting, James McKinna, and Robert Pollack. Checking algorithms for Pure Type Systems. In Henk Barendregt and Tobias Nipkow, editors, *Types for Proofs and Programs: International Workshop TYPES'93, Nijmegen, May 1993*, volume 806 of *LNCS*, pages 19–61. Springer-Verlag, 1994.

[Wan92] Paul Wand. Functional programming and verification with LEGO. Master's thesis, University of Edinburgh, 1992.