PROOF-THEORETIC INVESTIGATIONS INTO INTEGRATED LOGICAL AND FUNCTIONAL PROGRAMMING

Luis Filipe Ribeiro Pinto

Proof-Theoretic Investigations into Integrated Logical and Functional Programming

Luis Filipe Ribeiro Pinto

June 1996

TL C82

Dedication

To my parents.

Abstract

This thesis is a proof-theoretic investigation of logic programming based on hereditary Harrop logic (as in λProlog). After studying various proof systems for the first-order hereditary Harrop logic, we define the proof-theoretic semantics of a logic LFPL, intended as the basis of logic programming with functions, which extends higher-order hereditary Harrop logic by providing definition mechanisms for functions in such a way that the logical specification of the function rather than the function may be used in proof search.

In Chap. 3, we define, for the first-order hereditary Harrop fragment of LJ, the class of *uniform linear focused* (ULF) proofs (suitable for goal-directed search with backchaining and unification) and show that the ULF-proofs are in 1-1 correspondence with the expanded normal deductions, in Prawitz's sense. We give a system of *proof-term* annotations for LJ-proofs (where proof-terms uniquely represent proofs). We define a rewriting system on proof-terms (where rules represent a subset of Kleene's permutations in LJ) and show that: its irreducible proof-terms are those representing ULF-proofs; it is weakly normalising. We also show that the composition of Prawitz's mappings between LJ and NJ, restricted to ULF-proofs, is the identity.

We take the view of logic programming where: a program P is a set of formulae; a goal G is a formula; and the different means of achieving G w.r.t. P correspond to the expanded normal deductions of G from the assumptions in P (rather than the traditional view, whereby the different means of goal-achievement correspond to the different answer substitutions).

LFPL is defined in Chap. 4, by means of a sequent calculus. As in LeFun, it extends logic programming with functions and provides mechanisms for defining names for functions, maintaining proof search as the computation mechanism (contrary to languages such as ALF, Babel, Curry and Escher, based on equational logic, where the computation mechanism is some form of rewriting). LFPL also allows definitions for declaring logical properties of functions, called *definitions of dependent type.* Such definitions are of the form: $(f, x) =_{def} (\Lambda, w) : \Sigma_{x:\tau} F$, where f is a name for Λ and x is a name for w, a proof-term witnessing that the formula $[\Lambda/x]F$ holds (*i.e.* Λ meets the specification $\Sigma_{x:\tau}F$). When searching for proofs, it may suffice to use the formula $[\Lambda/x]F$ rather than Λ itself.

We present an interpretation of LFPL into $NN^{\lambda norm}$, a natural deduction system for hereditary Harrop logic with λ-terms. The means of goal-achievement in LFPL are interpreted in $NN^{\lambda norm}$ essentially by cut-elimination, followed by an interpretation of cut-free sequent calculus proofs as normal deductions.

We show that the use of definitions of dependent type may speed up proof search because the equivalent proofs using no such definitions may be much longer and because normalisation may be done lazily, since not all parts of the proof need to be exhibited. We sketch two methods for implementing LFPL, based on goal-directed proof search, differing in the mechanism for selecting definitions of dependent type on which to backchain. We discuss techniques for handling the redundancy arising from the equivalence of each proof using such a definition to one using no such definitions.

Declarations

I, Luis Filipe Ribeiro Pinto, hereby certify that this thesis, which is approximately 45,000 words in length, has been written by me, that it is the record of work carried out by me, and that it has not been submitted in any previous application for a higher degree.

date 5 June 1996 *signature of candidate* ___

I was admitted as a research student in October 1991 and as a candidate for the degree of doctor of philosophy in October 1992; the higher study for which this is a record was carried out in the University of St Andrews between 1991 and 1996.

date 5 June 1996 *signature of candidate* ___

I hereby certify that the candidate has fulfilled the conditions of the Resolution and Regulations appropriate for the degree of doctor of philosophy in the University of St Andrews and that the candidate is qualified to submit this thesis in application for that degree.

date 5 June 1996 *signature of supervisor* ___

In submitting this thesis to the University of St Andrews I understand that I am giving permission for it to be made available for use in accordance with the regulations of the University Library for the time being in force, subject to any copyright vested in the work not being affected thereby. I also understand that the title and abstract will be published, and that a copy of the work may be made and supplied to any bona fide library or research worker.

date 5 June 1996 *signature of candidate* ___

Acknowledgments

My sincere thanks:

- to Roy Dyckhoff for his expertise and patience in guiding my research and for his friendly support in my difficulties;
- to Paula and Michael Smith for their constant help and encouragement;
- to my long-suffering office-mate Nick, who always had a pertinent saying to my problems;
- to João, Charlie, Tó-zé and Shaun for putting up with such an inquisitive flat-mate;
- to Cristina, Julia, Adalberto, Jose Luis and Thanasis for their friendship;
- to the Computer Science Division of St Andrews University, very specially to the Theory Group, and to the Department of Mathematics of Minho University, for making research possible;
- to JNICT for sponsoring this research (grant BD/1423/91-IA) and to ESPRIT for funding my participation in various research meetings (grant BRA 7232 "GENTZEN");
- to Helena for her love;
- and to my family for the happiness of living.

Contents

Chapter 1

Introduction

1.1 Logical Foundations

The λ-calculus is a model of computation introduced by Church [Chu40]. It resulted from an attempt to provide a foundation for mathematics [Chu32, Chu33], which proved to be inconsistent [KR36]. (For books on λ-calculus, see *e.g.* [Bar81, HS86].) λ-calculus is often regarded as the precursor of current practice in functional programming [Jon87, Tho91]. Section 1.2 describes an abstract model of functional programming, based on typed λ-calculus, followed in this thesis.

Proof theory also has its roots in foundational studies, mainly those of Hilbert and his followers, in the first decades of the century. Unsatisfied with *axiomatic systems*, Gentzen devoted part of his work to formalisations of logic, reflecting more closely the logical principles of reasoning used by mathematicians. He introduced the *natural deduction* and the *sequent calculus* systems for first-order *classical* and *intuitionistic* logic in [Gen35]. For this thesis, LJ means the cut-free fragment of Gentzen's sequent calculus LJ; LJ^{cut} means the unrestricted calculus, using the cut rule. In [Gen35], Gentzen described how to transform each LJ^{cut}-derivation into a LJ-derivation, *i.e.* how to perform *cut elimination*. He also described a mapping from deductions in NJ (Gentzen's natural deduction system for first-order intuitionistic logic) to LJ^{cut}-derivations.

Prawitz in [Pra65] revived the interest in Gentzen's systems. He described a *normalisation* procedure for NJ, that was later realised, through the Curry-Howard correspondence, to be a counterpart of normalisation in functional systems [Tai67]. He defined a mapping from LJ-derivations to normal deductions in NJ and a mapping from normal deductions in NJ to LJ-derivations.

Miller *et al* [NM88, MNPS91] and Beeson [Bee89] are amongst the proponents of using LJ-based systems to give proof-theoretic characterisations of logic programming, a view that we follow in this thesis. Under this view, the concepts of *program*, *goal* and *achievement* of a goal w.r.t. a program are defined by means of sequent calculi systems for intuitionistic logic.

Typed theories were introduced in the beginning of the century, motivated by foundational studies [RW10]. Type theory has seen a resurgence of interest in the last 30 years or so, mainly for its applications to computer science, see *e.g.* [ML82, Hue90, NPS90, Tho91].

The view of Curry [CF58] and Howard [How69] of propositions as types motivated a succession of works following this idea, the prominent works being de Bruijn's Automath project [dB80] and MartinLöf's predicative type systems [ML84, NPS90]. As in other works [Pot77, TvD88, Min94], we follow the view of propositions as types for assigning *proof-terms* to derivations in formal systems. Section 2.3.2 defines the calculi LJ^{pt} and NJ^{pt} which are, respectively, natural deduction and sequent calculus systems for first-order intuitionistic logic with proof-term annotations.

Parallel investigations into type theory, based on Church's simple theory of types, led Girard to his impredicative systems F and $F\omega$ [Gir72, GLT89]; Reynolds discovered independently a system equivalent to system F [Rey74]. In [CH88], Coquand and Huet presented the *Calculus of Constructions*, which combines Girard's impredicative systems with de Bruijn's and Martin-Löf's proposal for *dependent types*. Luo extended the Calculus of Constructions with Martin-Löf's predicative *type universes* into the *Extended Calculus of Constructions* [Luo94]. Church's simple theory of types, systems F and $F\omega$, the Calculus of Constructions and some of de Bruijn's systems have equivalent systems in Barendregt's λ-*cube* [Bar93].

An application of type theory relevant for this thesis is the use of type theory as an integrated framework for developing specifications and programs. In particular, in our proposal for integrating logic and functional programming, we use ideas similar to the theory of *deliverables* [MB93], based on the Extended Calculus of Constructions, for attaching logical properties (specifications) to the functions defined in programs.

1.2 Functional Programming

This section describes an abstract model of functional programming, based on simply typed λ-calculus. This simplistic view of functional programming may be described as follows. A *program* is a list of definitions of the form $x =_{def} \Lambda : \tau$, where: x is a variable, the *definiendum*, that may be thought of as a name for a function; Λ is a λ-term, that may be thought of as the *definiens* of the function; and τ (the type of the definition) is a simple type. (See Sec. 2.2.1 for a definition of the syntactic categories of λ-terms and *simple types*.)

A program is required to verify some properties to be a *well-formed program*. Roughly: a definiendum may not occur in the program before it has been declared; a definiendum may not have different definientia; a definiendum may not occur in its definiens, *i.e.* no recursive definitions are allowed; the definiens of the definition must be of the type of the definition. (The rules of Sec. 2.2.1 may be used to define the well-formed λ-terms of a type, where a signature is used for gathering the types of the definienda occurring before in the program.)

Given a program P and a λ-term Λ of type τ, the *evaluation* of Λ w.r.t. P consists of the replacement of the occurrences of the definienda of P in Λ by their definientia and *normalisation*. The strategy followed for combining these two operations is irrelevant for the purposes of this thesis. A fundamental property of this abstract model is that each valid term of a type has a unique *normal form*. The result of an evaluation is the normal form of the original term.

This abstract view of functional programming may be rephrased proof-theoretically. Through the Curry-Howard correspondence between formulae and types, λ-terms may be regarded as proofs in a natural deduction formulation of intuitionistic implicational logic, as in [Coq90]. Then, a program is a list of definitions of the form $x =_{def} e : F$, where x is a name for the proof e of formula F. The evaluation mechanism may be described as the replacement of defined names for proofs by their definientia together with *proof-normalisation*, as described in Prawitz's [Pra65].

There are some other important features, usually present in functional programming languages, *e.g.* [Pau91, Tur86, HW90], not considered in the abstract model described above. Usually functional languages allow richer type theories. *Polymorphism* is often allowed, by introducing type variables. (See [Rey74, Gir72], for extensions of λ-calculus with polymorphism.) Another form of types usually provided in functional programmming is the *datatype*. Roughly, datatypes combine primitive types, *sum types* and well-founded recursion for building new types. Datatypes induce the use of *patterns* as a means for performing case analysis in a definition involving terms whose type is a datatype. Another ubiquitous feature in functional programming is the use of *recursive* function definitions. However, the uncontrolled use of recursive definitions is a source of non-terminating computations. We intend in future work, by adding some of the features mentioned above to our abstract model of functional programming, to extend our proposal for integrating logic and functional programming.

1.3 Logic Programming

Like the functional programming paradigm, the logic programming paradigm provides more readable and expressive languages for programming, as compared to imperative languages; in addition, logic programming provides search mechanisms for solutions to queries. The origins of logic programming and the first developments of the leading exponent in the paradigm, Prolog [SS86], are described in [Kow88, Coh88]. For the purposes of this thesis, we concentrate on pure logic programming, *i.e.* non-logical features, such as control strategies, are left aside, even though they have an important role in the semantics of a concrete implementation [And92b].

Most logic programming languages have their logical foundations in the first-order classical logic theory of *Horn clauses* [vEK76, Hod93]. A Horn clause may be written as: $A \leftarrow A_1, ..., A_n$, where $A, A_1, ..., A_n$ are *atoms*, *i.e.* atomic formulae built up by applying *predicate symbols* to *first-order terms.* As usual, first-order terms are built up from a set of *variables* and a set of *function symbols* and a first-order term is either a variable or a function symbol applied to first-order terms. A *program* is a set of Horn clauses and a *goal* is an atom. (See *e.g.* [Hod93] for a proof that the problem of whether or not a formula is a theorem of first-order classical logic may be encoded as the problem of whether or not a set of Horn clauses is inconsistent.)

A Horn clause $A \leftarrow A_1, ..., A_n$ is interpreted as the formula $\forall_{x_1} ... \forall_{x_m}((A_1 \wedge ... \wedge A_n) \supset A)$, where $x_1, ..., x_m$ is the set of variables occurring in the atomic formulae $A, A_1, ..., A_n$. The logical interpretation of a goal A is the formula $\exists_{x_1} ... \exists_{x_m} A$, where $x_1, ..., x_m$ is the set of variables occurring in A.

An atom A_1 is an *instance* of an atom A_2 iff there exists a substitution θ s.t. $\theta(A_2) = A_1$, where: a substitution θ is a mapping from variables to terms equal to the identity except for a finite set of variables x, for which x has no occurrences in $\theta(x)$; and the notation $\theta(A_2)$ represents the atom obtained from A_2 by replacing each occurrence of x by $\theta(x)$, for all variables. The *ground instances* of an atom A are all the instances of A that contain no variable. An atom is *ground* if it contains no variables. A set $\mathcal{S}$ of ground atoms is a *model* for a logic program if for each ground instance of a clause $A \leftarrow A_1, ..., A_n$ in the program it is the case that A is in $\mathcal{S}$ if $A_1, ..., A_n$ are in $\mathcal{S}$. The *least model* of a program is the intersection of all models of the program.

The tradition in logic programming semantics [vEK76, Llo84] has been to give an *operational semantics*, usually based on some form of *resolution* [Rob65], and a *declarative semantics* based on *model-theory*. (Sometimes [Wol93] a *fixpoint semantics* is used for bridging the gap between operational and declarative semantics.) Usually in declarative semantics [Llo84], the denotation of a *program* is the *least model* of the program. A goal is *achievable* w.r.t. the program iff some ground instance of it is in the least model.

More recently have appeared some studies giving *proof-theoretic* characterisations of logic programming, amongst others [NM88, Bee89, HSH90, Pym90, MNPS91, Pfe92, Ker92, And92b, Har94, NL95]. The proof-theoretic approach uses *formal systems* (*calculi*) for describing the semantics of logic programming, where computation is regarded as proof-search. As argued in [MNPS91, Bee89], proof-theoretic semantics presents a clear logical account of semantics which is closer to operational semantics than model-theoretic semantics.

We take the following proof-theoretic view of logic programming. A *program* is a set of closed formulae. A *goal* is a closed formula. *Achieving* a goal w.r.t. a program consists of a search for a proof of the goal w.r.t. the program in a formalisation of the logic underlying the language. Following Miller *et al* [MNPS91]'s view, we consider: the calculus to be a cut-free sequent calculus system for first-order intuitionistic logic; program formulae to be *hereditary Harrop formulae* and goals to be *hereditary Harrop goals.* (Hereditary Harrop formulae are obtained from Harrop formulae [Har60] by allowing no disjunctions and no existential quantifiers in positive subformulae. See Sec. 3.2 for a definition of the two classes of formulae.) The theory of hereditary Harrop formulae is a conservative extension of the theory of Horn clauses, in the sense that any Horn clause is an hereditary Harrop formula and, for a Horn program (set of Horn formulae), hereditary Harrop logic does not allow any new form of deriving formulae, as compared to Horn logic. Languages based on hereditary Harrop logic provide important abstraction mechanisms, such as modules and abstract datatypes. However, for these languages resolution is no longer an adequate implementation method, essentially because of implicational goals that augment the program; instead, goal-directed proof-search is used.

Traditional model-theoretic semantics capture the ideas about the number of solutions to a query (the different means of goal-achievement) only in a restricted manner. In model-theoretic semantics the different means of goal-achievement correspond to the ground instances of the goal in the least model. In proof-theoretic semantics the different means of goal-achievement may be captured more conveniently, since proofs are themselves the results of computations. For defining what are the different means of goal-achievement, it suffices to define an equality relation on proofs and regard the different means of goal-achievement as the different proofs under such equality relation. In the language λProlog [NM88] this issue is addressed by fixing the means of goal-achievement as the proofs which are *uniform* and use *backchaining* for dealing with atomic goals.

Section 3.6 presents a proof-theoretic semantics of a logic programming language called FOPLP, based on the theory of first-order hereditary Harrop formulae. This language follows closely Miller *et. al*'s [Mil90, MNPS91] proof-theoretic view of logic programming. (We give a detailed presentation of the proof-theoretic semantics of FOPLP not for its novelty, but as a foundation to our proposal for integrating logic and functional programming.) For describing hereditary Harrop logic, we use a sequent calculus with proof-term assignment. We assign distinct variables to formulae in the antecedent of a sequent and assign a proof-term to the succedent formula of a derivable sequent, so that the proof-term determines uniquely the derivation. So, the result of a computation in FOPLP may be described as a proof-term, from which the instantiations for the existentially quantified variables of the goal may be obtained. Pfenning's encoding of the *Logical Framework* [HHP93], Elf [Pfe89], may be regarded as a logic programming language where the programmer has access to the proof-terms and the result of a computation is a proof-term.

There are many works extending logic programming, based on Horn clauses, in various directions; below a few are mentioned. Several works, *e.g.* [Cla78, GL90], address extensions of Prolog that allow some form of negation. Some languages, such as Gödel [HL94], have considered multi-sorted extensions of Horn logic. Some works propose extensions of logic programming with mechanisms for defining functions. (We analyse some of these proposals in Sec. 1.4.) In [GR84] Horn clauses are extended with hypothetical reasoning. Miller *et al.* considered hereditary Harrop logic and studied various higher-order versions of it, proposing the languages λProlog [NM88] and L_λ[Mil91], which provide various abstraction mechanisms.

Loveland and his colleagues have been studying *disjunctive logic programming* [Lov91, NL95]. In [HSH90], Horn clauses are considered as rules of a formal system and the inference schema of *definitional reflection* is allowed. Logic programming languages based on Girard's *linear logic* [Gir87] are also flourishing [AP91, HM94, Mil94, HW95], see the survey [Mil95]. These languages provide a means for writing directly in the language some operations typically described in other languages by means of extra-logical predicates, such as resource management and concurrency.

1.4 Integrated Logical and Functional Programming

Logic programming and functional programming are two distinct approaches to declarative programming. From a type-theoretic perspective, these two styles of programming may be described as follows. In functional programming is given a term t of a type and the result of a computation is the canonical form of t. In logic programming is given a type T and the result of a computation is a term of type T; as opposed to functional programming, there may exist several terms of type T that may be computed as result.

There have been many proposals for integrating logic programming and functional programming, see [DL86, BL86, Han94] for surveys. We call the languages obtained from such integration *integrated logical and functional* languages. Often, when encoding a problem in a programming language we realise that parts of the problem are typically functional whereas other parts of the problem are relational in nature. Compared to functional programming, integrated logical and functional programming provides a more direct means of encoding a problem, since arbitrary relations need not be forcefully encoded as functions. So, integrated logical and functional programming may provide a clearer form of programming.

In integrated logical and functional programming there is no need for abstracting away from the functional specificity of a relation or need for imposing a functional behaviour on a relation by using extra-logical arguments. For example, if there is a relation in a program that is solely used in a functional form such relation could be described as a function. Then, during search there would be no need for attempting alternative forms of using such relation for achieving a goal. In [Han92] are shown examples of logic programs that when interpreted into an integrated logical and functional programming language acquire a better operational behaviour, becoming more efficient.

In our quest for a language supporting arbitrary relations as well as functions, we consider extensions of logic programming where functions are allowed for building predicates and a mechanism for defining names for functions is provided. So, according to our views of functional programming and logic programming, expressed in Sec. 1.2 and 1.3, we take the following view, as a starting point, of integrated logical and functional programming. A program is a pair $\langle\Delta, \Gamma\rangle$, where: Δ is a list of definitions of the form $x =_{def} \Lambda : \tau$, with x ranging over variables, Λ ranging over λ-terms and τ ranging over simple types; Γ consists of a set of logical formulae.

(The definienda of Δ may be used in building up the logical formulae in Γ, just as any other variables.) A goal G is achievable w.r.t. a program $\langle\Delta,\Gamma\rangle$ iff $[\Delta]G$ is achievable w.r.t. $[\Delta]\Gamma$ in the underlying logic programming language, where $[\Delta]G$ stands for the formula obtained from G by replacing the definienda of Δ by their definientia and subsequent normalisation ($[\Delta]\Gamma$ has a similar interpretation). Sec. 4.2 presents a proof-theoretic semantics for a language following these ideas of integrating logic and functional programming. The programming language LeFun [AKN89] essentially implements the ideas described above.

This thesis proposes an integration of logic and functional programming beyond the ideas described above. Our proposal provides a new form of definitions, *definitions of dependent type*. By using such definitions, when defining a function we may declare more properties about functions than merely the type of their arguments. Then, during proof-search, such properties may be used as lemmas for goal-achievement.

Section 4.6 describes a proof-theoretic semantics of the language LFPL that implements the ideas mentioned above. The language is described by means of a sequent calculus system with *cuts* for higher-order hereditary Harrop logic. Cut elimination is then the interpretation of LFPL into pure logic programming.

1.5 Overview of the Thesis and Related Work

This thesis presents the language LFPL as a novel approach for integrating logic and functional programming. This language extends ideas first presented in [Pin94, PD94]. There are approaches for integrating logic and functional programming, such as ALF [Han90], Babel [MNRA92], Curry [Han95], Escher [Llo94], which are based on equational logic, where predicates and logical operations are seen as boolean functions. These languages use narrowing (ALF, Babel, Curry) or some other form of rewriting (Escher) as the basic computation mechanism. The language LFPL follows a different form of integration. As in the integrated logical and functional language LeFun [AKN89], LFPL maintains functions and predicates at distinct levels and extends logic programming by allowing a mechanism for defining names for functions and by allowing function names for building terms and formulae. LFPL takes this extension even further, by allowing a mechanism for declaring specifications of functions, which may be used as lemmas in goal-achievement.

Chapter 2 lays down the logical foundations required for this thesis. Section 2.2 presents a simply typed λ-calculus, called λ^{ST}. We recall some properties of λ-calculus and review some properties of unification of λ-terms, following the works [Hue75, NM94]. Section 2.3 presents two calculi, based upon Gentzen's NJ and LJ, with proof-term assignment, called respectively NJ^{pt} and LJ^{pt}. The calculus NJ^{pt} essentially results by extending the correspondence between λ-terms and natural deductions in intuitionistic implicational logic [CF58, How69, Coq90] to the other connectives; a similar calculus is presented in [Gal93]. (See [Laf89, Gal93] for other forms of assigning terms to LJ, motivated by applications to functional programming.) LJ^{pt} allows no explicit structural rules, but these rules are admissible in the calculus. In Sec. 2.3.3 is given an encoding of Kleene's permutations for LJ [Kle52] by means of transformations on proof-terms of LJ^{pt} and is shown that the image of proof-terms under the mapping ϕ (an encoding of Prawitz's interpretation of sequent calculus proofs as natural deductions) is invariant under such transformations.

Chapter 3 presents two logic programming languages FOPLP and HOPLP. The semantics of these languages are defined by means of the cut-free sequent calculi hH and HH, which are systems of first-order and higher-order[1] hereditary Harrop logic, respectively. This view of first-order and higher-order logic programming has its roots in the works of Miller *et al* [NM88, MNPS91]. A departure point from Miller's work is the use of proof-terms for encoding derivations. The use of proof-terms permits to regard them as the results of computations, an idea followed in type-theoretic accounts of logic programming, such as [Pfe92].

We build on Miller *et al*'s idea of *uniform proofs* [MNPS91], Pfenning's idea of *immediate entailment* [Pfe91, Pfe94] and Andreoli's idea of *focusing proofs*, in the context of linear logic, arriving at the notion of *uniform linear focused* (ULF) derivations for first-order hereditary Harrop logic, in Sec. 3.3. It is shown that each derivation is permutable (in the sense of Sec. 2.3.3) to a ULF derivation. The calculus hH^{ULF} is introduced as a calculus that captures exactly ULF derivations. Section 3.5 shows a 1-1 correspondence between ULF derivations and *expanded normal deductions* [Pra65] for first-order hereditary Harrop logic. (This work was in collaboration with Dyckhoff, see [DP94, DP96b]. In [DP96b] the 1-1 correspondence is extended to full first-order intuitionistic logic.) Other proofs of essentially the same result may be found in [Pfe94, Min94]. The semantics of a logic programming language needs to define what are the different means of goal-achievement. Traditionally [Llo84], the different means of goal-achievement correspond to the ground instances of the goal in the least model. In FOPLP the different means of goal-achievement correspond to the expanded normal deductions of the goal. So, FOPLP has a clear interpretation by means of Gentzen's NJ. The language HOPLP, defined by means of the system HH, is obtained by extending the ideas above to higher-order hereditary Harrop logic. The natural deduction system $NN^{\lambda norm}$, for higher-order hereditary Harrop logic, is used for interpreting HOPLP.

Chapter 4 presents the integrated logical and functional language LFPL. This language is an extension of the language HOPLPD, defined in Sec. 4.2, which in turn is an extension of HOPLP with a mechanism for defining names for λ-terms. (The language HOPLPD essentially corresponds to LeFun.) The language LFPL extends HOPLPD by allowing definitions of the form $x =_{def} e : \Sigma_{y:\tau} F$, where x is a name for the proof-term e, which is essentially a *deliverable*

[1]The calculus HH allows no quantification over predicates; however, it is an higher-order logic in the sense it allows quantification over λ-terms. A similar calculus, hh^{ω}, except for the absence of disjunctions and existential quantifiers, is used in [Fel91] for encoding LF-specifications [HHP93].

[MB93], *i.e.* e is a pair (Λ, e_1) where Λ is a λ-term of type τ, usually a function, and e_1 is a witness for $[\Lambda/y]F$, *i.e.* a proof that Λ satisfies the specification $\Sigma_{y:\tau}F$. The semantics of LFPL is defined by means of HH^{def}, a sequent calculus system with proof-term annotations for higher-order hereditary Harrop logic that allows definition mechanisms. Section 4.5 presents an interpretation of HH^{def} into HH, so there is an interpretation of LFPL into HOPLP. The mapping from HH^{def} into HH is essentially cut elimination. (See Appendix A for a summary of the relations amongst various calculi used throughout this thesis.)

Chapter 5 studies methods of implementing LFPL. The semantics of LFPL is redefined by means of the calculus $HH^{def'}$, a calculus where proof-search becomes more efficient. Section 5.3 describes a class of $HH^{def'}$-derivations complete for LFPL, *i.e.* a class of derivations where all means of goal-achievement may be found. The concept of *extended uniform linear focused* (EULF) derivations is an extension of the concept of ULF derivations to $HH^{def'}$, by regarding specifications of functions just as any other program formulae. The class of *sensible derivations* only allows the use of the specification attached to a function in case the name of the function occurs in the goal.

The semantics of LFPL is defined in such a way that given a goal G and a program P, the means of achieving G w.r.t. P in LFPL are in a 1-1 correspondence with the means of achieving G' (the interpretation of G in HOPLP) w.r.t. P' (the interpretation of P in HOPLP) in HOPLP. The class of EULF derivations which are sensible is excessive for LFPL, *i.e.* there are derivations which are EULF and sensible that are regarded as the same means of goal-achievement in LFPL. So, an implementation needs to get rid of this redundancy. Our proposal for implementing LFPL simply collects the various means of goal-achievement and compares them with other means already obtained, discarding those which have the same interpretation in HOPLP as another found before, as described in Sec. 5.4.

Following [Pfe92], where a type-theoretic account of logic programming is given, in LFPL, we may think of: a program as a type assignment (context); a goal as a type; and achieving a goal G w.r.t. a program P as a search for a term of type G under type assignment P. Thus, the formal system underlying LFPL, HH^{def}, may be seen as a type system and an implementation of LFPL may be seen as a method to search for inhabitants of types.

The fragment of HH^{def} with no definition mechanisms (HH) and with no existential quantifiers and no disjunctions in goal formulae may be seen as a sequent calculus for a fragment of the $\lambda\Pi$-calculus [How69, dB80, HHP87]. The implementation suggested for LFPL, when restricted to such fragment, follows ideas similar to those in [Pfe91, PW91, Dow93] for proof-search in the $\lambda\Pi$-calculus, *i.e.* for non-atomic types search is determined by the structure of the type and for atomic types resolution is used. (It is noteworthy that the works mentioned above study proof-search in full $\lambda\Pi$-calculus; in fact, in [Dow93] is described a method that is applicable to all the type systems of Barendregt's cube.) The work [PW91] has another similarity with this thesis, in that proof-search is studied by means of sequent calculi and the terms that may be found in the calculus inducing the smallest search space are long $\beta\eta$-normal forms.

In [TS96] is presented an approach to proof-search in a type system allowing definitions. There, the problem of inhabitedness in fragments of Martin-Löf's type system [NPS90] is encoded as a first-order Horn logic theory, whereby: terms in the type system are translated as first-order terms; implicit definitions are translated by using first-order equality; and explicit definitions are not translated, instead they are used for expanding definienda, followed by normalisation. Under this translation, the rules for application and substitution in the type system are encoded by hyperresolution. The implementation we present for LFPL uses simple definitions for expanding definientia, as in the method above, but uses definitions of dependent type only for suggesting their types as lemmas in proof-search.

In Chapter 6, we present conclusions of this thesis and state unresolved problems for future investigations.

Chapter 2

Logical Preliminaries

2.1 Introduction

In this chapter are introduced various definitions and results used throughout this thesis.

Each formal system (calculus) presented in this thesis is defined by following LF's methodology [HHP93] for encoding logics, which is based on ideas pioneered by de Bruijin [dB80] and Martin-Löf [ML85] for describing logics. For each calculus there is a definition of: (i) the classes of objects used by the calculus; (ii) the notion of equality for each class of objects; (iii) the forms of judgement of the calculus; (iv) the derivable judgements of the calculus. The symbol $=$ is used for equality between objects.

Section 2.2 defines the typed λ-calculus λ^{ST} and recalls some of its properties. Some aspects of higher-order unification, following [Hue75, NM94], are also recalled.

Section 2.3 introduces the calculi NJ^{pt} and LJ^{pt}, which are formalisations of first-order intuitionistic[1] typed logic, where typing of first-order terms is according to λ^{ST}. The calculus NJ^{pt} is a sequent-style formalisation of first-order intuitionistic typed logic, based on Gentzen's NJ [Gen35]. As in NJ, NJ^{pt} has introduction and elimination rules for each connective. NJ^{pt} uses proof-terms to annotate logical formulae. The proof-terms used in NJ^{pt} are essentially forms of encoding deductions in NJ; the proof-term annotation follows closely [TvD88]. The calculus NJ^{pt} may also be seen as a type system, where formulae are seen as types and proof-terms are the inhabitants of the types.

The calculus LJ^{pt} is a sequent calculus also formalising first-order intuitionistic typed logic, based on Gentzen's sequent calculus LJ [Gen35]. The calculus LJ^{pt} uses proof-terms for annotating logical formulae. The calculus LJ^{pt} follows closely the formalisation of first-order intuitionistic typed logic in [Mil90], except for the use of proof-term annotations. In [Mil90] is shown that this formalisation of typed logic only coincides with traditional formalisations if all types are inhabited. The proof-terms used in LJ^{pt} essentially constitute a means of encoding

[1]Only a fragment of first-order intuitionistic logic is formalised, since *absurdity* and *negation* are not included in these formalisations. In fact, the logic formalised is closer to *minimal logic*.

derivations in LJ. In LJ^{pt} for each logical connective there are rules to introduce the logical connective in the antecedent of a sequent and rules to introduce the logical connective in the succedent of a sequent.

Following Kleene's [Kle52], a study of permutability in LJ and LK[2] , we present a list of transformations on proof-terms, encoding permutations, used for showing completeness of some classes of derivations.

Another concept used in this thesis is Prawitz's mapping ϕ [Pra65], from LJ-derivations to NJ-deductions. This mapping is defined in Subsec. 2.3.4 by means of a transformation on proof-terms.

2.2 Simply Typed λ-calculus

2.2.1 The Calculus λ^{ST}

There are two classes of objects in the calculus λ^{ST}: the class τ of *(simple) types* and the class Λ of *λ-terms*. Types are used to classify terms. For defining the class τ of simple types, a fixed set $\mathcal{S}$ of *primitive types* is assumed. The grammar defining simple types τ is as follows:

$$\tau ::= s \mid (\tau \to \tau),$$

where $s \in \mathcal{S}$. Below, τ, possibly indexed, is used as a meta-variable ranging over simple types. In a type of the form $(\tau_1 \to \tau_2)$, parentheses are usually omitted, in which case association is to the right. So, any type may be written in the form $\tau_1 \to \ldots \to \tau_n \to \tau$, where each τ_i, for $1 \leq i \leq n$, is an arbitrary type and τ is a primitive type.

We assume a denumerable fixed set $\mathcal{X}$ of variables and use x, y, z, w, possibly indexed, as meta-variables ranging over $\mathcal{X}$. The grammar defining the set of λ-terms Λ is as follows:

$$\Lambda ::= x \mid \lambda x : \tau.\Lambda \mid (\Lambda\Lambda).$$

As usual, terms of the form $\lambda x : \tau.\Lambda$ are called *abstractions* and terms of the form $(\Lambda_1\Lambda_2)$ are called *applications*. In an application usually parentheses are omitted, in which case association is to the left.

The class t of *first-order terms* is the subclass of λ-terms of primitive type containing no abstractions. Sometimes, λ-terms are also called *higher-order terms*. t and Λ, possibly indexed, are used as meta-variables ranging over first-order and higher-order terms, respectively.

The concepts of *free* and *bound* occurrences of variables and capture-avoiding *substitution*, notation $[\Lambda_1/x]\Lambda_2$, are defined as usual, see *e.g.* [Bar93]. We use the notation $x \notin \Lambda$ meaning that the variable x has no free occurrences in the λ-term Λ. Two λ-terms are called *α-convertible* iff they are the same up to *renaming of bound variables*, or, equivalently, iff the

[2]LK is a sequent calculus formalisation of classical logic due to Gentzen [Gen35].

λ-terms have the same representation using de Bruijn's indices [dB72]. As usual, we consider α-convertible terms to be equal.

Signatures Σ are sets of pairs $\langle x, \tau\rangle$, usually written $x : \tau$, where x is a variable and τ a simple type. We use the notation $\langle\rangle$ for the empty signature and the notation $\Sigma, x : \tau$ for the signature $\Sigma \cup \{x : \tau\}$.

The forms of judgement of the calculus λ^{ST} are shown in Fig. 2.1.

(i)	$\vdash \Sigma$ *signature*	(signature)
(ii)	$\Sigma \vdash \Lambda : \tau$	(term of a type)
(iii)	$\Sigma \vdash \Lambda \rhd_\tau \Lambda$	(one step reduction)
(iv)	$\Sigma \vdash \Lambda \rhd_\tau^+ \Lambda$	(one or more steps reduction)
(v)	$\Sigma \vdash \Lambda \rhd_\tau^* \Lambda$	(zero or more steps reduction)
(vi)	$\Sigma \vdash \Lambda \equiv_\tau \Lambda$	(conversion)

Figure 2.1: Forms of judgement of λ^{ST}.

A *derivation* of a judgement S is a tree of judgements, constructed by using instances of inference rules verifying the side conditions, whose root is the judgement S and whose leaves are axiom judgements, *i.e.* instances of rules with no premises. A judgement is said to be *derivable* iff there is a derivation of that judgement. (These notions of a derivation of a judgement and derivable judgements are common to all the other calculi used throughout this thesis.)

The rules defining derivable signatures are shown in Fig. 2.2. The notation $x \notin \Sigma$ means that there is no type τ s.t. $x : \tau$ is a member of Σ. Roughly, a signature is derivable if different types have not been assigned to the same variable.

$$\frac{}{\vdash \langle\rangle\ \textit{signature}} \qquad \frac{\vdash \Sigma\ \textit{signature} \quad x \notin \Sigma}{\vdash \Sigma, x : \tau\ \textit{signature}}$$

Figure 2.2: Derivable signatures.

The rules defining derivable judgements of the form $\Sigma \vdash \Lambda : \tau$ are shown in Fig. 2.3; they depend upon derivable signatures. Briefly, a derivable judgement of this form signifies that the term Λ is of type τ under the assignment of types to variables Σ. If $\Sigma \vdash \Lambda : \tau$ is derivable, Λ is said to be *well-formed* (of type τ) w.r.t. Σ.

Observe that a judgement of the form $\Sigma \vdash \lambda x : \tau.\Lambda : \tau \to \tau_1$ is not directly derivable in λ^{ST}, case $x \in \Sigma$. However, since $\lambda x : \tau.\Lambda$ is equal (α-convertible) to $\lambda x_1 : \tau.[x_1/x]\Lambda$, when x_1 has no free occurrences in Λ, the original judgement may be derivable in λ^{ST} .

The rules defining derivable judgements of the forms (iii), (iv) and (v) of Fig. 2.1 are shown in Fig. 2.4. Derivable judgements of form (iii) capture the usual notion of the one step reduction

$$\frac{\vdash \Sigma, x:\tau \ \mathit{signature}}{\Sigma, x:\tau \vdash x:\tau}$$

$$\frac{\Sigma, x:\tau \vdash \Lambda:\tau_1}{\Sigma \vdash \lambda x:\tau.\Lambda : \tau \to \tau_1}\ x \notin \Sigma \qquad \frac{\Sigma \vdash \Lambda:\tau_1 \to \tau \quad \Sigma \vdash \Lambda_1:\tau_1}{\Sigma \vdash \Lambda\Lambda_1 : \tau}$$

Figure 2.3: Derivable λ-terms of a type.

relation on λ-terms; they depend upon derivable terms of a type. In λ^{ST} there is a notion of reduction for each type. Derivable judgements of form (iv) capture the transitive closure of the one step reduction relation and derivable judgements of form (v) capture the reflexive and transitive closure of the one step reduction relation.

$$\frac{\Sigma, x:\tau \vdash \Lambda:\tau_1 \quad \Sigma \vdash \Lambda_1:\tau}{\Sigma \vdash (\lambda x:\tau.\Lambda)\Lambda_1 \rhd_{\tau_1} [\Lambda_1/x]\Lambda}\ x \notin \Sigma$$

$$\frac{\Sigma, x:\tau \vdash \Lambda \rhd_{\tau_1} \Lambda_1}{\Sigma \vdash \lambda x:\tau.\Lambda \rhd_{\tau\to\tau_1} \lambda x:\tau.\Lambda_1}\ x \notin \Sigma$$

$$\frac{\Sigma \vdash \Lambda \rhd_{\tau\to\tau_1} \Lambda_1 \quad \Sigma \vdash \Lambda_2:\tau}{\Sigma \vdash \Lambda\Lambda_2 \rhd_{\tau_1} \Lambda_1\Lambda_2} \qquad \frac{\Sigma \vdash \Lambda \rhd_{\tau} \Lambda_1 \quad \Sigma \vdash \Lambda_2:\tau\to\tau_1}{\Sigma \vdash \Lambda_2\Lambda \rhd_{\tau_1} \Lambda_2\Lambda_1}$$

$$\frac{\Sigma \vdash \Lambda \rhd_{\tau} \Lambda_1}{\Sigma \vdash \Lambda \rhd_{\tau}^{+} \Lambda_1} \qquad \frac{\Sigma \vdash \Lambda \rhd_{\tau}^{+} \Lambda_1 \quad \Sigma \vdash \Lambda_1 \rhd_{\tau}^{+} \Lambda_2}{\Sigma \vdash \Lambda \rhd_{\tau}^{+} \Lambda_2}$$

$$\frac{\Sigma \vdash \Lambda \rhd_{\tau} \Lambda_1}{\Sigma \vdash \Lambda \rhd_{\tau}^{*} \Lambda_1}$$

$$\frac{\Sigma \vdash \Lambda:\tau}{\Sigma \vdash \Lambda \rhd_{\tau}^{*} \Lambda} \qquad \frac{\Sigma \vdash \Lambda \rhd_{\tau}^{*} \Lambda_1 \quad \Sigma \vdash \Lambda_1 \rhd_{\tau}^{*} \Lambda_2}{\Sigma \vdash \Lambda \rhd_{\tau}^{*} \Lambda_2}$$

Figure 2.4: Derivable reduction judgements.

It may be easily proved that: if a judgement $\Sigma \vdash \Lambda_1 \rhd_\tau \Lambda_2$ is derivable in λ^{ST}, then both $\Sigma \vdash \Lambda_1 : \tau$ and $\Sigma \vdash \Lambda_2 : \tau$ are derivable in λ^{ST}.

Judgements of the form $\Sigma \vdash \Lambda \equiv_\tau \Lambda$ are called *convertibility judgements.* The rules defining derivable convertibility judgements are shown in Fig. 2.5. Convertibility judgements capture the usual notion of β-convertible λ-terms. The notion of convertibility corresponds to the transitive and symmetric closure of reduction in zero or more steps. We say that two λ-terms Λ_1 and Λ_2 are *convertible* if the judgement $\Sigma \vdash \Lambda_1 \equiv_\tau \Lambda_2$ is derivable, for some type τ.

$$\frac{\Sigma \vdash \Lambda \rhd_\tau^* \Lambda_1}{\Sigma \vdash \Lambda \equiv_\tau \Lambda_1}$$

$$\frac{\Sigma \vdash \Lambda_1 \equiv_\tau \Lambda}{\Sigma \vdash \Lambda \equiv_\tau \Lambda_1} \qquad \frac{\Sigma \vdash \Lambda \equiv_\tau \Lambda_2 \quad \Sigma \vdash \Lambda_2 \equiv_\tau \Lambda_1}{\Sigma \vdash \Lambda \equiv_\tau \Lambda_1}$$

Figure 2.5: Derivable convertibility judgements.

2.2.2 Properties of λ^{ST}

In this section are reviewed some definitions and properties of the λ-calculus needed in subsequent material of this thesis.

Definition 2.1 (normal forms) *A λ-term is called a* normal form *if it contains no subterms of the form $(\lambda x : \tau.\Lambda)\Lambda_1$, called β-*redexes.

Normal forms can be syntactically characterised as the λ-terms of the form:

$$\lambda x_1 : \tau_1 ... \lambda x_n : \tau_n . x\Lambda_1 ... \Lambda_m,$$

where $n, m \geq 0$, x is a variable that may or may not be one of the x_i, $n \geq i \geq 1$, and $\Lambda_1, ..., \Lambda_m$ are themselves normal forms, see *e.g.* [CHS72] for a proof of this result.

Theorem 2.1 (Strong Normalisation) *Let $\Sigma \vdash \Lambda_1 : \tau$ be derivable in λ^{ST}. Then, every sequence of terms $\Lambda_1, \Lambda_2, \Lambda_3, ..., \Lambda_n, ...$ s.t., for every $n \geq 1$, $\Sigma \vdash \Lambda_n \rhd_\tau \Lambda_{n+1}$ is derivable in λ^{ST}, is finite.*

A proof of this result may be obtained by adapting the methods, for example, in [Tai67] or in [GLT89] to λ^{ST} .

Theorem 2.2 (Church-Rosser) *Let the judgements $\Sigma \vdash \Lambda \rhd_\tau^* \Lambda_1$ and $\Sigma \vdash \Lambda \rhd_\tau^* \Lambda_2$ be derivable in λ^{ST}. Then, there exists Λ_3 s.t. the judgements $\Sigma \vdash \Lambda_1 \rhd_\tau^* \Lambda_3$ and $\Sigma \vdash \Lambda_2 \rhd_\tau^* \Lambda_3$ are derivable in λ^{ST}.*

For proving this result, by using Newman's lemma and Theorem 2.1, it suffices to show that $\rhd_\tau$ is weakly Church-Rosser, see *e.g.* [Bar81, Bar93].

From Theorems 2.2 and 2.1, it may be shown that if $\Sigma \vdash \Lambda_1 : \tau$ is derivable then there exists a unique normal form Λ_2 s.t. $\Sigma \vdash \Lambda_1 \rhd_\tau \Lambda_2$ is derivable; Λ_2 is called the *normal form* of Λ_1.

A term $\lambda x_1 : \tau_1 ... x_n : \tau_n . x\Lambda_1 ... \Lambda_m$, well-formed w.r.t. a signature Σ, is an *expanded normal form* under Σ if: $x : \tau_1' \to ... \to \tau_m' \to \tau$, where τ is a primitive type; $x \in \{x_1, ..., x_n\}$ or $x \in \Sigma$, and Λ_i, for $1 \leq i \leq m$, are expanded normal forms under $\Sigma \cup \{x_1 : \tau_1, ..., x_n : \tau_n\}$.

Definition 2.2 (**η-convertibility**) *Two λ-terms are* η-convertible *if they are in the congruence closure of the relation: $\lambda x.\Lambda x \equiv_\eta \Lambda$, if x has no free occurrences in Λ.*

Every normal form is η-convertible to an expanded normal form, which is unique up to renaming of bound variables, see [CHS72, Bar81] for proofs of this result. We write $\lambda norm(\Lambda)$ for the expanded normal form of Λ. We consider normal forms to be *equal* if they have the same expanded normal form. It is decidable whether or not a judgement of the form $\Sigma \vdash \Lambda_1 \equiv_\tau \Lambda_2$ is derivable in λ^{ST}. It suffices to calculate $\lambda norm(\Lambda_1)$ and $\lambda norm(\Lambda_2)$ and check whether or not they are the same.

One property used several times below is the *substitution property*, *i.e.* if $x \neq x_1$ and $x_1 \notin \Lambda$ then

$$[\Lambda/x]([\Lambda_1/x_1]\Lambda_2) = [[\Lambda/x]\Lambda_1/x_1]([\Lambda/x]\Lambda_2).$$

See [Bar81] for a proof of this result.

We now review some aspects of unification of λ-terms. As opposed to unification of first-order terms [Her67, Rob65, BS94], unification of λ-terms is only semi-decidable and for unifiable λ-terms there is a recursively enumerable set of unifiers which are "most general", but, in contrast to the first-order case, such set may have more than one element, see [Hue75]. In [Hue75] is presented a semi-decision procedure for the existence of unifiers of λ-terms. The procedure enumerates some[3] unifiers, when λ-terms are unifiable modulo $\alpha\beta\eta$-convertibility, but may fail to terminate if there is no unifier. Further, this enumeration is non-redundant, *i.e.* no unifier in the enumeration may be obtained from another unifier in the enumeration.

Below we use a formula

$$unify(S, \Theta_{in}, \Theta_{out}, V_{in}, V_{out}{}^{4}, \Sigma)$$

meaning that:

- Σ, V_{in} and V_{out} are signatures s.t. $V_{in} \subseteq V_{out}$ and no variable is simultaneously in Σ and in V_{out};
- S is a set of pairs $\langle \Lambda_1, \Lambda_2 \rangle$ s.t. the judgements $\Sigma, V_{in} \vdash \Lambda_1 : \tau$ and $\Sigma, V_{in} \vdash \Lambda_2 : \tau$ are derivable, for some τ;
- Θ_{in} and Θ_{out} are *substitutions*, *i.e.* mappings from variables to λ-terms s.t., there exists Θ s.t. $\Theta_{out} = \Theta \circ \Theta_{in}$, where if $x : \tau \notin V_{out}$ then $\Theta_{out}(x) = x$, otherwise $\Sigma, V_{out} \vdash \Theta_{out}(x) : \tau$ is derivable;
- for each pair $\langle \Lambda_1, \Lambda_2 \rangle$ of S, $\lambda norm(\Theta_{out}(\Lambda_1)) = \lambda norm(\Theta_{out}(\Lambda_2))$.

[3]Only some unifiers are enumerated; in the problem of unifying *flexible-flexible* pairs is acknowledged the existence of unifiers but there is no search for them.

[4]The procedure described in [Hue75], for unification of λ-terms, introduces new free variables at the *imitation* and *projection* steps.

Thus, for checking whether or not two λ-terms Λ_1 and Λ_2, for which the judgements $\Sigma, V_{in} \vdash \Lambda_1 : \tau$ and $\Sigma, V_{in} \vdash \Lambda_2 : \tau$ are derivable, are unifiable (may be made equal by replacing free occurrences of variables in V_{in}) it suffices to check whether or not there exists Θ and V_{out} s.t. the formula

$$unify(\langle \Lambda_1, \Lambda_2 \rangle, id, \Theta, V_{in}, V_{out}, \Sigma)$$

holds.

A procedure for finding unifiers for λ-terms satisfying the predicate *unify* may be obtained by following the works [Hue75, NM94]. The predicate *unify* is used in Secs. 3.7, 3.8.2 and 5.4 for describing means of implementing first-order and higher-order logic programming languages.

2.3 The calculi NJ^{pt} and LJ^{pt}

2.3.1 The calculus NJ^{pt}

We assume the set $\mathcal{S}$ of primitive types to have a special type *prop*, called the *type of formulae*. We assume a fixed set $\mathcal{P}$ of pairs $\langle p, \tau \rangle$, usually written as $p : \tau$, where: τ is a type of the form $\tau_1 \to ... \to \tau_n \to prop$, where $n \geq 0$ and, for $1 \leq i \leq n$, τ_i is a type with no occurrences of *prop*; p is a symbol, called a *predicate symbol.* The set $\mathcal{P}$ is called *the set of predicate symbols.*

Atomic formulae are of the form $pt_1...t_n$, where p is a predicate symbol and $t_1, ..., t_n$ are first-order terms. An atomic formula $pt_1...t_n$ is *well-formed* w.r.t. a signature Σ if $p : \tau_1 \to ... \to \tau_n \to prop \in \mathcal{P}$, and for $1 \leq i \leq n$, $\Sigma \vdash t_i : \tau_i$ is derivable in λ^{ST}. $A, A_1, A_2, ...$ are used as meta-variables ranging over atomic formulae.

The set F of *(logical) formulae* is defined by the grammar:

$$F ::= A \mid F \wedge F \mid F \vee F \mid F \supset F \mid \exists_{x:\tau} F \mid \forall_{x:\tau} F.$$

As usual, in a formula of one the forms $\exists_{x:\tau}F, \forall_{x:\tau}F$, x is called a *bound variable.* Two formulae are *equal* if they are the same up to renaming of bound variables.

The set of *well-formed* formulae w.r.t. a signature Σ is inductively defined as follows.

(i) The well-formed atomic formulae w.r.t. Σ are well-formed w.r.t. Σ.

(ii) $F_1 \wedge F_2$, $F_1 \vee F_2$ and $F_1 \supset F_2$, where F_1 and F_2 are well-formed w.r.t. Σ, are well-formed w.r.t. Σ;

(iii) $\exists_{x:\tau}F$ and $\forall_{x:\tau}F$, where $\Sigma \cup \{x : \tau\}$ is a derivable signature and F is well-formed w.r.t. $\Sigma \cup \{x : \tau\}$, are well-formed w.r.t. Σ.

The notation $[t/x]F$ stands for the result of replacing free occurrences of x by t in F.

A *context* is a set Δ of pairs $\langle x, F \rangle$, usually written $x : F$, where x is a variable and F is a formula, s.t. if $x : F_1$ and $x : F_2$ are elements of Δ then F_1 is equal to F_2; in other words, different formulae are annotated with different variables. Δ, possibly indexed, is used as a meta-variable ranging over contexts. The notation $x \in \Delta$ is used when there is a formula F in Δ whose annotation is the variable x, *i.e.* $x : F$ is an element of Δ. The notation $\Delta, x : F$ is used for the context $\Delta \cup \{x : F\}$.

The set of *d-proof-terms*, also called *NJ^{pt}-proof-terms*, is defined by the grammar:

$$\begin{array}{rcl} d & ::= & (d,d) \mid i(d) \mid j(d) \mid \lambda x.d \mid (t,d) \mid \lambda_q x.d \\ & \mid & x \mid fst(d) \mid snd(d) \mid wn(d, x.d, x.d) \mid app(d,d) \mid exists(d, x.x.d) \mid app_q(d,t), \end{array}$$

where x ranges over the set $\mathcal{X}$ of variables. For the purposes of this thesis, a distinction could have been made between simply typed variables, *i.e.* variables used for building terms of simple type and variables that occur at the underlined position in proof-terms of the form $\lambda_q \underline{x}.d$ and $exists(d, \underline{x}.x_1.d)$, and variables of formula type, *i.e.* variables used for annotating formulae in derivations. However, we have chosen to share the set $\mathcal{X}$ of variables for variables of both kinds.

The d-proof-term constructors *fst*, *snd*, *wn*, *app*, *exists*, *app*$_q$ are called *left constructors.* The other d-proof-term constructors are called *right constructors.* (In a functional programming setting left constructors are usually called *destructors.*)

In proof-terms of one of the forms $\lambda x.d$, $wn(d_1, x.d, x_1.d_2)$, $wn(d_1, x_1.d_2, x.d)$, $\lambda_q x.d$, $exists(d_1, x_1.x.d)$, x is called a *binder* whose *scope* is d and an occurrence of x in d is called *bound.* Also, in proof-terms of the form $exists(d_1, x.x_1.d)$, x is called a *binder* whose *scope* is $x_1.d$ and an occurrence of x in d is called *bound.* A non-bound occurrence of a variable x is called *free.* The notation $x \notin d$ means that the variable x has no free occurrences in the proof-term d.

Two proof-terms d_1, d_2 are *equal* if they are the same up to renaming of bound variables.

A *sequent* in NJ^{pt} is a quadruple $\langle \Sigma, \Delta, d, F \rangle$, written as $\Sigma; \Delta \vdash d : F$, where Σ is a signature, as defined in λ^{ST}, Δ is a context, d is a d-proof-term and F, the *succedent formula*, is a logical formula. A sequent $\Sigma; \Delta \vdash d : F$ is *well-formed* if $\vdash \Sigma$ *signature* is derivable and all formulae in Δ and the formula F are well-formed w.r.t. Σ. The only judgement form of NJ^{pt} is that of being a derivable (well-formed) sequent. The rules defining derivable sequents of NJ^{pt} are presented in Fig. 2.6.

In Fig. 2.6, rules of the form $C - Intr$ are called *introduction rules* and rules of the form $C - Elim$ are called *elimination rules.* The leftmost premiss of an elimination rule is called its *main premiss.*

In order to distinguish between derivations in NJ^{pt} and derivations in LJ^{pt}, usually derivations in NJ^{pt} are called *deductions.*

Let π be a deduction of $\Sigma; \Delta \vdash d : F$. π is called a deduction for *deducing* F from $\Sigma; \Delta$ and d is called the *proof-term* of π. It is noteworthy that in a NJ^{pt}-deduction of a well-formed sequent all occurrences of a sequent are well-formed. The traditional eigenvariable conditions on $\exists$-$Elim$ and $\forall$-$Intr$ are satisfied in a deduction, for all sequents in a deduction are well-formed.

$$\frac{}{\Sigma; \Delta, x : F \vdash x : F}\ axiom \qquad \begin{array}{r}\vdash \Sigma \ \textit{signature} \text{ derivable,}\\ (\Delta, x : F) \text{ well-formed}\\ \text{w.r.t. } \Sigma\end{array}$$

$$\frac{\Sigma; \Delta \vdash d_1 : F_1 \quad \Sigma; \Delta \vdash d_2 : F_2}{\Sigma; \Delta \vdash (d_1, d_2) : F_1 \wedge F_2}\ \wedge - Intr$$

$$\frac{\Sigma; \Delta \vdash d : F_1 \wedge F_2}{\Sigma; \Delta \vdash fst(d) : F_1}\ \wedge_l - Elim \qquad \frac{\Sigma; \Delta \vdash d : F_1 \wedge F_2}{\Sigma; \Delta \vdash snd(d) : F_2}\ \wedge_r - Elim$$

$$\frac{\Sigma; \Delta \vdash d : F_1}{\Sigma; \Delta \vdash i(d) : F_1 \vee F_2}\ \vee_l - Intr \qquad F_2 \text{ well-formed w.r.t. } \Sigma$$

$$\frac{\Sigma; \Delta \vdash d : F_2}{\Sigma; \Delta \vdash j(d) : F_1 \vee F_2}\ \vee_r - Intr \qquad F_1 \text{ well-formed w.r.t. } \Sigma$$

$$\frac{\Sigma; \Delta \vdash d : F_1 \vee F_2 \quad \Sigma; \Delta, x_1 : F_1 \vdash d_1 : F \quad \Sigma; \Delta, x_2 : F_2 \vdash d_2 : F}{\Sigma; \Delta \vdash wn(d, x_1.d_1, x_2.d_2) : F}\ \vee - Elim \qquad x_1 \notin \Delta, x_2 \notin \Delta$$

$$\frac{\Sigma; \Delta, x : F_1 \vdash d : F_2}{\Sigma; \Delta \vdash \lambda x.d : F_1 \supset F_2}\ \supset -Intr \qquad x \notin \Delta$$

$$\frac{\Sigma; \Delta \vdash d_1 : F_1 \supset F_2 \quad \Sigma; \Delta \vdash d_2 : F_1}{\Sigma; \Delta \vdash app(d_1, d_2) : F_2}\ \supset -Elim$$

$$\frac{\Sigma; \Delta \vdash d : [t/x]F}{\Sigma; \Delta \vdash (t, d) : \exists_{x:\tau} F}\ \exists - Intr \qquad \Sigma \vdash t : \tau \text{ derivable}$$

$$\frac{\Sigma; \Delta \vdash d : \exists_{x:\tau} F_1 \quad \Sigma, x : \tau; \Delta, x_1 : F_1 \vdash d_1 : F}{\Sigma; \Delta \vdash exists(d, x.x_1.d_1) : F}\ \exists - Elim \qquad x_1 \notin \Delta, x \notin \Sigma$$

$$\frac{\Sigma, x : \tau; \Delta \vdash d : F}{\Sigma; \Delta \vdash \lambda_q x.d : \forall_{x:\tau} F}\ \forall - Intr \qquad x \notin \Sigma$$

$$\frac{\Sigma; \Delta \vdash d : \forall_{x:\tau} F}{\Sigma; \Delta \vdash app_q(d, t) : [t/x]F}\ \forall - Elim \qquad \Sigma \vdash t : \tau \text{ derivable}$$

Figure 2.6: Rules for derivable sequents of NJ^{pt} .

The proof-term d and the context of a deduction's endsequent determine uniquely the deduction up to the naming of bound variables of d, as shown below.

Theorem 2.3 *Any NJ^{pt}-deductions of the sequent $\Sigma; \Delta \vdash d : F$ differ at the most up to the names of the bound variables of d.*

Proof: By induction on the structure of d.

Case d is a variable. Then, if $\Sigma; \Delta \vdash d : F$ is derivable, by inspection of the rules allowed for deriving sequents, $d : F \in \Delta$ and the only possible deduction of $\Sigma; \Delta \vdash d : F$ is:

$$\frac{}{\Sigma; \Delta \vdash d : F} \; axiom.$$

Case $d = \lambda x.d_1$. (Recall that $d = \lambda x_1.[x_1/x]d_1$, for every $x_1 \notin d_1$.) Then, if $\Sigma; \Delta \vdash d : F$ is derivable, by inspection of the rules allowed for deriving sequents, F is of the form $F_1 \supset F_2$ and any deduction of $\Sigma; \Delta \vdash d : F$ must be of the form:

$$\frac{\Sigma; \Delta, x_1 : F_1 \vdash [x_1/x]d_1 : F_2}{\Sigma; \Delta \vdash \lambda x_1.[x_1/x]d_1 : F_1 \supset F_2} \supset -Intr,$$

for some variable x_1, possibly x, s.t. $x_1 \notin \Delta$. So, by I.H., any deductions of the premiss differ at the most up to the names of the bound variables of $[x_1/x]d_1$. Thus, since x is bound in d, any deductions of $\Sigma; \Delta \vdash d : F$ differ at the most up to the names of the bound variables of d.

The other cases follow by similar arguments. □

Proof-terms differing only up to renaming of bound variables are *equal*. The theorem above justifies our notion of equality for deductions. Given a formula F and a pair $\Sigma; \Delta$, we consider NJ^{pt}-deductions for deducing F from $\Sigma; \Delta$ to be *equal* if their proof-terms are equal. So, any two deductions of a sequent in NJ^{pt} are equal. It is enough to concentrate on the proof-term of a deduction rather than having to deal with deductions themselves.

The subclasses N and a of d-proof-terms, whose members are respectively called *normal proof-terms* and *atomic normal proof-terms*, are defined as follows:

$$\begin{array}{lcl} N & ::= & (N, N) \mid i(N) \mid j(N) \mid \lambda x.N \mid (t, N) \mid \lambda_q x.N \mid a \\ a & ::= & x \mid fst(a) \mid snd(a) \mid wn(a, x.N, x.N) \mid app(a, N) \mid exists(a, x.x.N) \mid app_q(a, t). \end{array}$$

A deduction whose proof-term is normal is called a *normal deduction*. It may be shown that normal deductions are deductions with no *maximal segment*, *i.e.* a branch $S_1, ..., S_n$, with one or more sequents, where: all the sequents S_i $(1 \leq i \leq n)$ have the same succedent formula, S_1 is the conclusion of an introduction rule and S_n is the main premiss of an elimination rule[5] .

Theorem 2.4 (normalisation) *If $\Sigma; \Delta \vdash d : F$ is derivable in NJ^{pt} then there exists a normal proof-term N s.t. $\Sigma; \Delta \vdash N : F$ is derivable in NJ^{pt}.*

[5]The succedent formula of a maximal segment containing only one sequent is usually called a *maximal formula*

This result may be proved by adapting to NJ^{pt} the method for proving the *normalisation theorem* in [Pra65, Pra70], or, by regarding propositions as types, by adapting the methods for proving normalisation theorems in [Tai67, GLT89].

The normal deductions mentioned above correspond to β-normal forms on λ-terms. There are other kinds of normal forms on λ-terms, most notably $\beta\eta$-normal forms and *long* $\beta\eta$-normal forms [Bar81, HS86]. In [Pra70], the deductions corresponding to *long* $\beta\eta$-normal forms are called *expanded normal deductions*, a terminology that we adopt in this thesis. Deductions with no $\vee - Elim$ nor $\exists - Elim$ rules in expanded normal form are those normal deductions s.t. the succedent formula of each sequent which is simultaneously the conclusion of an elimination rule and a premiss of an introduction rule is atomic. As for λ-terms, from the expanded normal form of a deduction D one may easily compute all the β-normal forms $\beta\eta$-equivalent to D and D's $\beta\eta$-normal form.

2.3.2 The calculus LJ^{pt}

The set of *e-proof-terms*, also called *LJ^{pt}-proof-terms*, is defined by the grammar:

$$\begin{array}{rcl} e & ::= & pair(e,e) \mid inl(e) \mid inr(e) \mid lambda(x.e) \mid pair_q(t,e) \mid lambda_q(x.e) \\ & \mid & x \mid splitl(x,x.e) \mid splitr(x,x.e) \mid when(x,x.e,x.e) \mid apply(x,e,x.e) \\ & \mid & exists(x,x.x.e) \mid apply_q(x,t,x.e). \end{array}$$

The e-proof-term constructors $split_l$, $split_r$, $when$, $apply$, $exists$, $apply_q$ are called *left constructors.* The e-proof-term constructors $pair$, inl, inr, $lambda$, $pair_q$, $lambda_q$ are called *right constructors.* e, possibly indexed, is used as a meta-variable ranging over e-proof-terms.

In proof-terms of one of the following forms: $lambda(x.e)$, $lambda_q(x.e)$, $splitl(x_1, x.e)$, $splitr(x_1, x.e)$, $when(x_1, x.e, x_2.e_1)$, $when(x_1, x_2.e_1, x.e)$, $apply(x_1, e_1, x.e)$, $apply_q(x_1, t, x.e)$, $exists(x_1, x_2.x.e)$, the variable x is called a *binder* of *scope* e; an occurrence of x in e is called *bound.* Also, in proof-terms of the form $exists(x_1, x.x_2.e)$, x is called a *binder* whose *scope* is $x_2.e$; an occurrence of x in e is called *bound.* A non-bound occurrence of a variable is called *free.* The notation $x \notin e$ means that the variable x has no free occurrences in the proof-term e. Two e-proof-terms are *equal* if they are the same up to renaming of bound variables.

In proof-terms of one of the following forms: $splitl(x, x_1.e)$, $splitr(x, x_1.e)$, $when(x, x_1.e_1, x_2.e_2)$, $apply(x, e, x_1.e_1)$, $exists(x, x_1.x_2.e)$, $apply_q(x, t, x_1.e)$, the variable x is called the *head variable* of the proof-term.

A *sequent* in LJ^{pt} is a quadruple $\langle \Sigma, \Delta, e, F\rangle$, written as $\Sigma; \Delta \Rightarrow e : F$, where Σ is a signature, as defined in λ^{ST}, Δ is a context, as defined for NJ^{pt}, e is an e-proof-term and F is a logical formula, as defined for NJ^{pt}. In a sequent $\Sigma; \Delta \Rightarrow e : F$, Δ is called the *antecedent (context)*, F is called the *succedent (formula)* and e the *proof-term* of the sequent.

A sequent $\Sigma; \Delta \Rightarrow e : F$ is *well-formed* if $\vdash \Sigma$ *signature* is derivable and all formulae in Δ and the formula F are well-formed w.r.t. Σ. The only judgement form of LJ^{pt} is that of being a derivable (well-formed) sequent. The rules defining derivable sequents of LJ^{pt} are presented in Fig. 2.7.

As usual, rules of the form $\Rightarrow C$ are called *right rules* and rules of the form $C \Rightarrow$ are called *left rules.* Observe that the *outermost constructor* of the proof-term of the conclusion of a right (left) rule is a right (left) constructor. In the rules for deriving sequents in Fig. 2.7: each of the formulae F, $F_1 \wedge F_2$, $F_1 \wedge F_2$, $F_1 \vee F_2$, $F_1 \supset F_2$, $\exists_{x:\tau} F_1$ and $\forall_{x:\tau} F_1$ is, respectively, the *main formula* of *axiom*, $\wedge_l \Rightarrow$, $\wedge_r \Rightarrow$, $\vee \Rightarrow$, $\supset\Rightarrow$, $\exists \Rightarrow$ and $\forall \Rightarrow$; F_1 is the *side formula* of $\wedge_l \Rightarrow$, F_2 is the *side formula* of $\wedge_r \Rightarrow$, F_1 and F_2 are the *side formulae* of $\vee \Rightarrow$, F_2 is the *side formula* of $\supset\Rightarrow$, F_1 is the *side formula* of $\exists \Rightarrow$ and $[t/x]F_1$ is the *side formula* of $\forall \Rightarrow$.

Let π be a derivation of the sequent $\Sigma; \Delta \Rightarrow e : F$. π is called a derivation for *deriving* F from $\Sigma; \Delta$ and e is called the *proof-term* of π and is also called a *proof-term for deriving* F w.r.t. $\Sigma; \Delta$. As for NJ^{pt}, the proof-term e and the context of the endsequent of a derivation π determine uniquely, up to renaming of bound variables of e, the derivation π. So, we consider LJ^{pt}-derivations for deriving a formula F from a pair $\Sigma; \Delta$ to be *equal* if their proof-terms are equal.

Theorem 2.5 (weakening admissibility) *Let* $\Sigma; \Delta \Rightarrow e : F$ *be derivable in* LJ^{pt}, $x \notin \Delta$ *and let* F_1 *be a well-formed formula w.r.t.* Σ. *Then,* $\Sigma; \Delta, x : F_1 \Rightarrow e : F$ *is also derivable in* LJ^{pt}.

Proof: Follows easily by induction on the derivation of $\Sigma; \Delta \Rightarrow e : F$. Observe that $(\Delta, x : F_1)$ is a well-formed context w.r.t. Σ, since Δ is well-formed w.r.t. Σ, $x \notin \Delta$ and F_1 is well-formed w.r.t. Σ. □

2.3.3 Permutations in LJ^{pt}

Kleene introduced in [Kle52] a notion of *permutation on derivations*, for Gentzen's LK and LJ. Roughly, permutations are transformations on derivations that reverse the order in which inference rules occur in a derivation. As shown in [Kle52, Sha92], there are cases where reversing the order in which inference rules occur in an LJ-derivation is not possible. In the calculus LJ^{pt}, since the proof-term of a derivation determines uniquely the derivation up to renaming of bound variables, permutations on derivations may be captured at the level of proof-terms, as is shown below.

The rules on proof-terms shown in Figs. 2.8, 2.9, 2.10 and 2.11 are called *permutations.* Permutations in Fig. 2.8, called *right permutations*, encode a reversing of a right rule below a left rule. Permutations in Fig. 2.9, called *left permutations*, encode a reversing of left rules. Permutations in Fig. 2.10, called *reductive permutations*, are not permutations in the sense they encode a reversing of rules; essentially, they eliminate redundant left rules. Permutations in Fig. 2.11, called *linearising permutations*, are also not encoding a reversing of rules; essentially, they encode a form of reducing the number of uses of side formulae in a derivation. Reversing

$$\frac{}{\Sigma; \Delta, x : F \Rightarrow x : F}\ axiom \qquad \vdash \Sigma\ \textit{signature}\ \text{derivable}, (\Delta, x : F)\ \text{well-formed w.r.t. } \Sigma$$

$$\frac{\Sigma; \Delta \Rightarrow e_1 : F_1 \quad \Sigma; \Delta \Rightarrow e_2 : F_2}{\Sigma; \Delta \Rightarrow pair(e_1, e_2) : F_1 \wedge F_2} \Rightarrow \wedge$$

$$\frac{\Sigma; \Delta, x : F_1 \wedge F_2, x_1 : F_1 \Rightarrow e : F}{\Sigma; \Delta, x : F_1 \wedge F_2 \Rightarrow splitl(x, x_1.e) : F} \wedge_l \Rightarrow \qquad x_1 \notin \Delta$$

$$\frac{\Sigma; \Delta, x : F_1 \wedge F_2, x_1 : F_2 \Rightarrow e : F}{\Sigma; \Delta, x : F_1 \wedge F_2 \Rightarrow splitr(x, x_1.e) : F} \wedge_r \Rightarrow \qquad x_1 \notin \Delta$$

$$\frac{\Sigma; \Delta \Rightarrow e : F_1}{\Sigma; \Delta \Rightarrow inl(e) : F_1 \vee F_2} \Rightarrow \vee_l \qquad F_2\ \text{well-formed w.r.t. } \Sigma$$

$$\frac{\Sigma; \Delta \Rightarrow e : F_2}{\Sigma; \Delta \Rightarrow inr(e) : F_1 \vee F_2} \Rightarrow \vee_r \qquad F_1\ \text{well-formed w.r.t. } \Sigma$$

$$\frac{\Sigma; \Delta, x : F_1 \vee F_2, x_1 : F_1 \Rightarrow e_1 : F \quad \Sigma; \Delta, x : F_1 \vee F_2, x_2 : F_2 \Rightarrow e_2 : F}{\Sigma; \Delta, x : F_1 \vee F_2 \Rightarrow when(x, x_1.e_1, x_2.e_2) : F} \vee \Rightarrow \qquad x_1 \notin \Delta,\ x_2 \notin \Delta$$

$$\frac{\Sigma; \Delta, x : F_1 \Rightarrow e : F_2}{\Sigma; \Delta \Rightarrow lambda(x.e) : F_1 \supset F_2} \Rightarrow \supset \qquad x \notin \Delta$$

$$\frac{\Sigma; \Delta, x : F_1 \supset F_2 \Rightarrow e : F_1 \quad \Sigma; \Delta, x : F_1 \supset F_2, x_1 : F_2 \Rightarrow e_1 : F}{\Sigma; \Delta, x : F_1 \supset F_2 \Rightarrow apply(x, e, x_1.e_1) : F} \supset \Rightarrow \qquad x_1 \notin \Delta$$

$$\frac{\Sigma; \Delta \Rightarrow e : [t/x]F}{\Sigma; \Delta \Rightarrow pair_q(t, e) : \exists_{x:\tau} F} \Rightarrow \exists \qquad \Sigma \vdash t : \tau\ \text{derivable}$$

$$\frac{\Sigma, x : \tau; \Delta, x_1 : \exists_{x:\tau} F_1, x_2 : F_1 \Rightarrow e : F}{\Sigma; \Delta, x_1 : \exists_{x:\tau} F_1 \Rightarrow exists(x_1, x.x_2.e) : F} \exists \Rightarrow \qquad x_2 \notin \Delta,\ x \notin \Sigma$$

$$\frac{\Sigma, x : \tau; \Delta \Rightarrow e : F}{\Sigma; \Delta \Rightarrow lambda_q(x.e) : \forall_{x:\tau} F} \Rightarrow \forall \qquad x \notin \Sigma$$

$$\frac{\Sigma; \Delta, x_1 : \forall_{x:\tau} F_1, x_2 : [t/x]F_1 \Rightarrow e : F}{\Sigma; \Delta, x_1 : \forall_{x:\tau} F_1 \Rightarrow apply_q(x_1, t, x_2.e) : F} \forall \Rightarrow \qquad x_2 \notin \Delta,\ \Sigma \vdash t : \tau\ \text{derivable}$$

Figure 2.7: Rules for derivable sequents of LJ^{pt}.

left rules below right rules is not relevant for the purpose of this thesis, although such reversing is possible in many cases, as shown in [Kle52, Sha92]. Note that most of the side conditions imposed on permutations are satisfied simply by renaming of bound variables.

(1) $splitl(x, x_1.pair(e_1, e_2)) \triangleright pair(splitl(x, x_1.e_1), splitl(x, x_1.e_2))$

(2) $splitr(x, x_1.pair(e_1, e_2)) \triangleright pair(splitr(x, x_1.e_1), splitr(x, x_1.e_2))$

(3) $apply(x, e, x_1.pair(e_1, e_2)) \triangleright pair(apply(x, e, x_1.e_1), apply(x, e, x_1.e_2))$

(4) $apply_q(x, t, x_1.pair(e_1, e_2)) \triangleright pair(apply_q(x, t, x_1.e_1), apply_q(x, t, x_1.e_2))$

(5) $splitl(x, x_1.lambda(x_2.e)) \triangleright lambda(x_2.splitl(x, x_1.e)),\ x_2 \neq x,\ x_1 \neq x_2$

(6) $splitr(x, x_1.lambda(x_2.e)) \triangleright lambda(x_2.splitr(x, x_1.e)),\ x_2 \neq x,\ x_1 \neq x_2$

(7) $apply(x, e, x_1.lambda(x_2.e_1)) \triangleright lambda(x_2.apply(x, e, x_1.e_1)),\ x_2 \neq x,\ x_1 \neq x_2$

(8) $apply_q(x, t, x_1.lambda(x_2.e)) \triangleright lambda(x_2.apply_q(x, t, x_1.e)),\ x_2 \neq x,\ x_1 \neq x_2$

(9) $splitl(x, x_1.inl(e)) \triangleright inl(splitl(x, x_1.e))$

(10) $splitr(x, x_1.inl(e)) \triangleright inl(splitr(x, x_1.e))$

(11) $apply(x, e, x_1.inl(e_1)) \triangleright inl(apply(x, e, x_1.e_1))$

(12) $apply_q(x, t, x_1.inl(e_1)) \triangleright inl(apply_q(x, t, x_1.e_1))$

(13) $splitl(x, x_1.inr(e)) \triangleright inr(splitl(x, x_1.e))$

(14) $splitr(x, x_1.inr(e)) \triangleright inr(splitr(x, x_1.e))$

(15) $apply(x, e, x_1.inr(e_1)) \triangleright inr(apply(x, e, x_1.e_1))$

(16) $apply_q(x, t, x_1.inr(e_1)) \triangleright inr(apply_q(x, t, x_1.e_1))$

(17) $splitl(x, x_1.lambda_q(x_2.e)) \triangleright lambda_q(x_2.splitl(x, x_1.e)),\ x_2 \neq x,\ x_1 \neq x_2$

(18) $splitr(x, x_1.lambda_q(x_2.e)) \triangleright lambda_q(x_2.splitr(x, x_1.e)),\ x_2 \neq x,\ x_1 \neq x_2$

(19) $apply(x, e, x_1.lambda_q(x_2.e_1)) \triangleright lambda_q(x_2.apply(x, e, x_1.e_1)),\ x_2 \neq x,\ x_1 \neq x_2$

(20) $apply_q(x, t, x_1.lambda_q(x_2.e)) \triangleright lambda_q(x_2.apply_q(x, t, x_1.e)),\ x_2 \neq x,\ x_1 \neq x_2$

(21) $splitl(x, x_1.pair_q(t, e)) \triangleright pair_q(t, splitl(x, x_1.e))$

(22) $splitr(x, x_1.pair_q(t, e)) \triangleright pair_q(t, splitr(x, x_1.e))$

(23) $apply(x, e, x_1.pair_q(t, e_1)) \triangleright pair_q(t, apply(x, e, x_1.e_1))$

(24) $apply_q(x, t, x_1.pair_q(t_1, e)) \triangleright pair_q(t_1, apply_q(x, t, x_1.e))$

Figure 2.8: Right permutations.

Theorem 2.6 *For every rule* $e_1 \triangleright e_2$ *of Figs. 2.8, 2.9, 2.10 and 2.11, provided the side conditions are satisfied, if* $\Sigma; \Delta \Rightarrow e_1 : F$ *is derivable in* LJ^{pt} *then* $\Sigma; \Delta \Rightarrow e_2 : F$ *is derivable in* LJ^{pt}.

Proof:
Case rule (7). A derivation of $\Sigma; \Delta \Rightarrow apply(x, e, x_1.lambda(x_2.e_1)) : F$ must be of the form:

$$\dfrac{\overset{\pi_1}{\Sigma; \Delta_1, x : F_1 \supset F_2 \Rightarrow e : F_1} \quad \dfrac{\overset{\pi_2}{\Sigma; \Delta_1, x : F_1 \supset F_2, x_1 : F_2, x_2 : F_3 \Rightarrow e_1 : F_4}}{\Sigma; \Delta_1, x : F_1 \supset F_2, x_1 : F_2 \Rightarrow lambda(x_2.e_1) : F_3 \supset F_4} \Rightarrow\supset}{\Sigma; \Delta_1, x : F_1 \supset F_2 \Rightarrow apply(x, e, x_1.lambda(x_2.e_1)) : F_3 \supset F_4} \supset\Rightarrow$$

(25) $splitl(x, x_1.splitl(x_2, x_3.e)) \rhd splitl(x_2, x_3.splitl(x, x_1.e))$

(26) $splitl(x, x_1.splitr(x_2, x_3.e)) \rhd splitr(x_2, x_3.splitl(x, x_1.e))$

(27) $splitl(x, x_1.apply(x_2, e, x_3.e_1)) \rhd apply(x_2, splitl(x, x_1.e), x_3.splitl(x, x_1.e_1))$

(28) $splitl(x, x_1.apply_q(x_2, t, x_3.e)) \rhd apply_q(x_2, t, x_3.splitl(x, x_1.e))$

(29) $splitr(x, x_1.splitl(x_2, x_3.e)) \rhd splitl(x_2, x_3.splitr(x, x_1.e))$

(30) $splitr(x, x_1.splitr(x_2, x_3.e)) \rhd splitr(x_2, x_3.splitr(x, x_1.e))$

(31) $splitr(x, x_1.apply(x_2, e, x_3.e_1)) \rhd apply(x_2, splitr(x, x_1.e), x_3.splitr(x, x_1.e_1))$

(32) $splitr(x, x_1.apply_q(x_2, t, x_3.e)) \rhd apply_q(x_2, t, x_3.splitr(x, x_1.e))$

(33) $apply(x, e, x_1.splitl(x_2, x_3.e_1)) \rhd splitl(x_2, x_3.apply(x, e, x_1.e_1))$

(34) $apply(x, e, x_1.splitr(x_2, x_3.e_1)) \rhd splitr(x_2, x_3.apply(x, e, x_1.e_1))$

(35) $apply(x, e, x_1.apply(x_2, e_1, x_3.e_2)) \rhd apply(x_2, apply(x, e, x_1.e_1), x_3.apply(x, e, x_1.e_2))$

(36) $apply(x, e, x_1.apply_q(x_2, t, x_3.e_1)) \rhd apply_q(x_2, t, x_3.apply(x, e, x_1.e_1))$

(37) $apply_q(x, t, x_1.splitl(x_2, x_3.e)) \rhd splitl(x_2, x_3.apply_q(x, t, x_1.e))$

(38) $apply_q(x, t, x_1.splitr(x_2, x_3.e)) \rhd splitr(x_2, x_3.apply_q(x, t, x_1.e))$

(39) $apply_q(x, t, x_1.apply(x_2, e, x_3.e_1)) \rhd apply(x_2, apply_q(x, t, x_1.e), x_3.apply_q(x, t, x_1.e_1))$

(40) $apply_q(x, t, x_1.apply_q(x_2, t_1, x_3.e)) \rhd apply_q(x_2, t_1, x_3.apply_q(x, t, x_1.e))$

All permutations are subject to the conditions: $x_1 \neq x_2$, $x \neq x_3$ and $x_1 \neq x_3$.

Figure 2.9: Left permutations.

where $\Delta = (\Delta_1, x : F_1 \supset F_2)$ and $F = F_3 \supset F_4$. Thus, the following derivation may be formed:

$$\cfrac{\cfrac{\overset{\pi_3}{\Sigma; \Delta_1, x : F_1 \supset F_2, x_2 : F_3 \Rightarrow e : F_1} \quad \overset{\pi_2}{\Sigma; \Delta_1, x : F_1 \supset F_2, x_1 : F_2, x_2 : F_3 \Rightarrow e_1 : F_4}}{\Sigma; \Delta_1, x : F_1 \supset F_2, x_2 : F_3 \Rightarrow apply(x, e, x_1.e_1) : F_4}\supset\Rightarrow}{\Sigma; \Delta_1, x : F_1 \supset F_2 \Rightarrow lambda(x_2.apply(x, e, x_1.e_1)) : F_3 \supset F_4}\Rightarrow\supset,$$

where π_3 may be obtained from π_1 by weakening.

The cases corresponding to the other rules of Fig. 2.8 are similar. These rules also correspond to a movement of a left rule above a right rule.

Case rule (35). The last step of a derivation of the sequent

$$\Sigma; \Delta \Rightarrow apply(x, e, x_1.apply(x_2, e_1, x_3.e_2)) : F$$

must be a rule $\supset\Rightarrow$ s.t.: its left premiss is of the form

$$\Sigma; \Delta \Rightarrow e : F_1,$$

derivable by a derivation π_1; its right premiss has a derivation of the form

$$\cfrac{\overset{\pi_2}{\Sigma; \Delta, x_1 : F_2 \Rightarrow e_1 : F_3} \quad \overset{\pi_3}{\Sigma; \Delta, x_1 : F_2, x_3 : F_4 \Rightarrow e_2 : F}}{\Sigma; \Delta, x_1 : F_2 \Rightarrow apply(x_2, e_1, x_3.e_2) : F}\supset\Rightarrow$$

(41) $splitl(x, x_1.e) \triangleright e,\ x_1 \notin e$

(42) $splitr(x, x_1.e) \triangleright e,\ x_1 \notin e$

(43) $apply(x, e, x_1.e_1) \triangleright e_1,\ x_1 \notin e_1$

(44) $apply_q(x, t, x_1.e) \triangleright e,\ x_1 \notin e$

Figure 2.10: Reductive permutations.

(45) $splitl(x, x_1.e) \triangleright splitl(x, x_1.splitl(x, x_2.e_1))$

(46) $splitr(x, x_1.e) \triangleright splitr(x, x_1.splitr(x, x_2.e_1))$

(47) $apply(x, e_2, x_1.e) \triangleright apply(x, e_2, x_1.apply(x, e_2, x_2.e_1))$

(48) $apply_q(x, t, x_1.e) \triangleright apply_q(x, t, x_1.apply_q(x, t, x_2.e_1))$

All permutations are subject to the conditions:

$x_2 \notin e$, x_1 occurs more than once in e

and e_1 is obtained from e by replacing one of the occurrences of x_1.

Figure 2.11: Linearising permutations.

where $x_3 \notin (\Delta, x_1 : F_2)$; $\Delta = (\Delta_1, x : F_1 \supset F_2, x_2 : F_3 \supset F_4)$ and $x_1 \notin \Delta$. Thus, the derivation below may be formed, where π_4 may be obtained by weakening from π_1, since $x_3 \notin \Delta$.

$$\dfrac{\dfrac{\overset{\pi_1}{\Sigma; \Delta \Rightarrow e : F_1} \quad \overset{\pi_2}{\Sigma; \Delta, x_1 : F_2 \Rightarrow e_1 : F_3}}{\Sigma; \Delta \Rightarrow apply(x, e, x_1.e_1) : F_3} \supset\Rightarrow \quad \dfrac{\overset{\pi_4}{\Sigma; \Delta, x_3 : F_4 \Rightarrow e : F_1} \quad \overset{\pi_3}{\Sigma; \Delta, x_1 : F_2, x_3 : F_4 \Rightarrow e_2 : F}}{\Sigma; \Delta, x_3 : F_4 \Rightarrow apply(x, e, x_1.e_2) : F} \supset\Rightarrow}{\Sigma; \Delta \Rightarrow apply(x_2, apply(x, e, x_1.e_1), x_3.apply(x, e, x_1.e_2)) : F} \supset\Rightarrow$$

Note that $x_1 \notin (\Delta, x_3 : F_4)$ and $x_3 \notin \Delta$.

The cases corresponding to the other rules of Fig. 2.9 are similar, they also correspond to a movement of a left rule above a left rule.

Case rule (43). If a sequent $\Sigma; \Delta, x : F \Rightarrow e : F_1$ is derivable in LJ^{pt} and $x \notin e$, then it may be easily proved by induction on the structure of e that the sequent $\Sigma; \Delta \Rightarrow e : F_1$ is derivable in LJ^{pt}. Thus, from a derivation of the form:

$$\dfrac{\overset{\pi_1}{\Sigma; \Delta_1, x : F_2 \supset F_3 \Rightarrow e_1 : F_2} \quad \overset{\pi_2}{\Sigma; \Delta_1, x : F_2 \supset F_3, x_1 : F_3 \Rightarrow e_2 : F_1}}{\Sigma; \Delta_1, x : F_2 \supset F_3 \Rightarrow apply(x, e_1, x_1.e_2) : F_1} \supset\Rightarrow,$$

where $x_1 \notin e_2$, it follows that $\Sigma; \Delta_1, x : F_2 \supset F_3 \Rightarrow e_2 : F_1$ is derivable.

The cases corresponding to the other rules of Fig. 2.10 are similar. These rules also correspond to eliminating a left rule from a derivation if its side formulae are not used in the derivation.

Case rule (47). First is proved the lemma: if a sequent $\Sigma; \Delta, x : F \Rightarrow e : F_1$ is derivable in LJ^{pt}, then, for every $x_1 \notin (\Delta, x : F)$ and for every e_1 obtained by replacing zero or more occurrences of x in e by x_1, the sequent $\Sigma; \Delta, x : F, x_1 : F \Rightarrow e_1 : F_1$ is derivable in LJ^{pt}. (This transformation on derivations is essentially inverse to *contraction*.) The proof of the lemma follows by induction on the structure of e.

Case e is the variable x. Then, a derivation of $\Sigma; \Delta, x : F \Rightarrow e : F_1$ must be of the form:

$$\frac{}{\Sigma; \Delta, x : F \Rightarrow x : F_1}\ axiom,$$

so F is the same as F_1. Thus the following derivation may be formed:

$$\frac{}{\Sigma; \Delta, x : F, x_1 : F \Rightarrow x_1 : F}\ axiom,$$

since $(\Delta, x : F, x_1 : F)$ is well-formed w.r.t. Σ, for: $(\Delta, x : F)$ is well-formed w.r.t. Σ, which implies that F is well-formed w.r.t. Σ, and $x_1 \notin (\Delta, x : F)$. The case where e is a variable x_2 different of x follows by forming an axiom whose main formula is the formula annotated by x_2.

Case $e = apply(x_2, e_2, x_3.e_3)$. Then, case $x_2 = x$, a derivation of $\Sigma; \Delta, x : F \Rightarrow e : F_1$ must be of the form:

$$\frac{\overset{\pi_1}{\Sigma; \Delta, x : F_2 \supset F_3 \Rightarrow e_2 : F_2} \quad \overset{\pi_2}{\Sigma; \Delta, x : F_2 \supset F_3, x_3 : F_3 \Rightarrow e_3 : F_1}}{\Sigma; \Delta, x : F_2 \supset F_3 \Rightarrow apply(x, e_2, x_3.e_3) : F_1}\ \Rightarrow\supset$$

where $F = F_2 \supset F_3$. By the I.H., for every e_4, e_5 obtained from e_2, e_3, respectively, by replacing some occurrences of x by x_1, there exist derivations π_3 and π_4 of the sequents:

$$\Sigma; \Delta, x : F_2 \supset F_3, x_1 : F_2 \supset F_3 \Rightarrow e_4 : F_2;$$
$$\Sigma; \Delta, x : F_2 \supset F_3, x_3 : F_3, x_1 : F_2 \supset F_3 \Rightarrow e_5 : F_1.$$

Now the following two cases must be considered.
(a) Case $e_1 = apply(x_1, e_2, x_3.e_3)$, *i.e.* the occurrence of x as head variable of e has been replaced by x_1, the following derivation may be formed:

$$\frac{\overset{\pi_3}{\Sigma; \Delta, x : F_2 \supset F_3, x_1 : F_2 \supset F_3 \Rightarrow e_4 : F_2} \quad \overset{\pi_4}{\Sigma; \Delta, x : F_2 \supset F_3, x_1 : F_2 \supset F_3, x_3 : F_3 \Rightarrow e_5 : F_1}}{\Sigma; \Delta, x : F_2 \supset F_3, x_1 : F_2 \supset F_3 \Rightarrow apply(x_1, e_4, x_3.e_5) : F_1}\ \supset\Rightarrow.$$

(b) Case $e_1 = apply(x, e_2, x_3.e_3)$, *i.e.* the occurrence of x as head variable of e has not been replaced by x_1, the following derivation may be formed:

$$\frac{\overset{\pi_3}{\Sigma; \Delta, x : F_2 \supset F_3, x_1 : F_2 \supset F_3 \Rightarrow e_4 : F_2} \quad \overset{\pi_4}{\Sigma; \Delta, x : F_2 \supset F_3, x_3 : F_3, x_1 : F_2 \supset F_3 \Rightarrow e_5 : F_1}}{\Sigma; \Delta, x : F_2 \supset F_3, x_1 : F_2 \supset F_3 \Rightarrow apply(x, e_4, x_3.e_5) : F_1}\ \supset\Rightarrow.$$

Case $x_2 \neq x$ the proof follows easily by the I.H..

Proofs for the other possible forms of e follow by similar arguments, concluding the proof of the lemma.

Now, consider a derivation of the form:

$$\frac{\overset{\pi_1}{\Sigma; \Delta, x : F_2 \supset F_3 \Rightarrow e : F_2} \quad \overset{\pi_2}{\Sigma; \Delta, x : F_2 \supset F_3, x_1 : F_3 \Rightarrow e_1 : F_1}}{\Sigma; \Delta, x : F_2 \supset F_3 \Rightarrow apply(x, e, x_1.e_1) : F_1}\ \supset\Rightarrow.$$

By the lemma, for every $x_2 \notin (\Delta, x : F_2 \supset F_3, x_1 : F_3)$ and for every e_2 obtained by replacing zero or more occurrences of x_1 in e_1 by x_2, there is a derivation π_3 of

$$\Sigma; \Delta, x : F_2 \supset F_3, x_1 : F_3, x_2 : F_3 \Rightarrow e_2 : F_1.$$

So, the following derivation may be formed:

$$\dfrac{\overset{\pi_1}{\Sigma; \Delta, x : F_2 \supset F_3 \Rightarrow e : F_2} \quad \dfrac{\overset{\pi_4}{\Sigma; \Delta, x : F_2 \supset F_3, x_1 : F_3 \Rightarrow e : F_2} \quad \overset{\pi_3}{\Sigma; \Delta, x : F_2 \supset F_3, x_1 : F_3, x_2 : F_3 \Rightarrow e_2 : F_1}}{\Sigma; \Delta, x : F_2 \supset F_3, x_1 : F_3 \Rightarrow apply(x, e, x_2.e_2) : F_1} \supset\Rightarrow,}{\Sigma; \Delta, x : F_2 \supset F_3 \Rightarrow apply(x, e, x_1.apply(x, e, x_2.e_2)) : F_1} \supset\Rightarrow$$

where π_4 may be obtained from π_1 by weakening.

The cases corresponding to the other rules of Fig. 2.11 are similar. □

2.3.4 Relating LJ^{pt} and NJ^{pt}

The mapping ϕ, from the set of LJ^{pt}-proof-terms to the set of NJ^{pt}-proof-terms, and the *substitution operation* of a variable x by a NJ^{pt}-proof-term d_1 in a NJ^{pt}-proof-term d_2, notation $[d_1/x]d_2$, are defined in Fig. 2.12. Essentially, the mapping ϕ is an encoding of Prawitz's mapping ϕ from LJ-derivations to normal NJ-deductions, presented in [Pra65].

Theorems 2.7 and 2.8 state, respectively, that LJ^{pt} is sound and complete for normal deductions w.r.t. NJ^{pt}.

Theorem 2.7 *If $\Sigma; \Delta \Rightarrow e : F$ is derivable in LJ^{pt} then $\Sigma; \Delta \vdash \phi(e) : F$ is derivable in NJ^{pt}. Further, $\phi(e)$ is a normal proof-term.*

Proof: The first part of the result may be proved following [Pra65]. The second part of the result may be easily proved by induction on the structure of e. □

Theorem 2.8 *Let $\Sigma; \Delta \vdash d : F$ be derivable in NJ^{pt}, where d is a normal proof-term. Then, there exists e s.t. $\Sigma; \Delta \Rightarrow e : F$ is derivable in LJ^{pt} and $\phi(e) = d$.*

Proof: This result may be proved by defining a mapping ρ, from normal proof-terms in NJ^{pt} to proof-terms in LJ^{pt} , following Prawitz's construction of LJ-derivations from normal NJ-deductions, in pp. 92-93 of [Pra65], s.t. $\phi o \rho$ is the identity on normal NJ-deductions. (Section 3.5 presents mappings δ, ψ s.t.: $\psi o \delta$ is a mapping, from a subset of LJ^{pt}-proof-terms to a subset[6] of normal NJ^{pt}-proof-terms, essentially, encoding Prawitz's mapping ρ.) □

The binary relation $\cong_e$ on e-proof-terms is the *reflexive*, *symmetric*, *transitive* and *compatible closure* of the permutations in Figs. 2.8, 2.9, 2.10 and 2.11, *i.e.* of the relation consisting of all

[6]The mappings δ and ψ may be easily extended to the full set of LJ^{pt}-proof-terms, in such a way that $\psi o \delta$ is still a right inverse of ϕ.

$[d/x](d_1, d_2) =_{def} ([d/x]d_1, [d/x]d_2)$
$[d/x]i(d_1) =_{def} i([d/x]d_1)$
$[d/x]j(d_1) =_{def} j([d/x]d_1)$
$[d/x]\lambda x_1.d_1 =_{def} \lambda x_1.[d/x]d_1,\ x \neq x_1, x_1 \notin d$
$[d/x](t, d_1) =_{def} (t, [d/x]d_1)$
$[d/x]\lambda_q x_1.d_1 =_{def} \lambda_q x_1.[d/x]d_1,\ x \neq x_1, x_1 \notin d$
$[d/x]x =_{def} d$
$[d/x]x_1 =_{def} x_1,\ x \neq x_1$
$[d/x]fst(d_1) =_{def} fst([d/x]d_1)$
$[d/x]snd(d_1) =_{def} snd([d/x]d_1)$
$[d/x]wn(d_1, x_1.d_2, x_2.d_3) =_{def} wn([d/x]d_1, x_1.[d/x]d_2, x_2.[d/x]d_3),\ x \neq x_1, x \neq x_2, x_1 \notin d, x_2 \notin d$
$[d/x]app(d_1, d_2) =_{def} app([d/x]d_1, [d/x]d_2)$
$[d/x]exists(d_1, x_1.x_2.d_2) =_{def} exists([d/x]d_1, x_1.x_2.[d/x]d_2),\ x \neq x_1, x \neq x_2, x_1 \notin d, x_2 \notin d$
$[d/x]app_q(d_1, t) =_{def} app_q([d/x]d_1, t)$

$\phi(pair(e_1, e_2)) =_{def} (\phi(e_1), \phi(e_2))$
$\phi(inl(e)) =_{def} i(\phi(e))$
$\phi(inr(e)) =_{def} j(\phi(e))$
$\phi(lambda(x.e)) =_{def} \lambda x.\phi(e)$
$\phi(pair_q(t, e)) =_{def} (t, \phi(e))$
$\phi(lambda_q(x.e)) =_{def} \lambda_q x.\phi(e)$
$\phi(x) =_{def} x$
$\phi(splitl(x, x_1.e)) =_{def} [fst(x)/x_1]\phi(e)$
$\phi(splitr(x, x_1.e)) =_{def} [snd(x)/x_1]\phi(e)$
$\phi(when(x, x_1.e_1, x_2.e_2)) =_{def} wn(x, x_1.\phi(e_1), x_2.\phi(e_2))$
$\phi(apply(x, e, x_1.e_1)) =_{def} [app(x, \phi(e))/x_1]\phi(e_1)$
$\phi(exists(x, x_1.x_2.e)) =_{def} exists(x, x_1.x_2.\phi(e))$
$\phi(apply_q(x, t, x_1.e)) =_{def} [app_q(x, t)/x_1]\phi(e)$

Figure 2.12: The substitution operation and the mapping ϕ.

pairs (e_1, e_2) s.t. $e_1 \rhd e_2$ is a rule in one of the Figs. 2.8, 2.9, 2.10 and 2.11. Proof-terms which are $\cong_e$-related are called *permutable* proof-terms. Permutable LJ^{pt}-proof-terms map under ϕ to the same normal NJ^{pt}-proof-terms, as stated below.

Theorem 2.9 *If $e_1 \cong_e e_2$, then $\phi(e_1) = \phi(e_2)$.*

Proof: It suffices to show that, for each of the rules in Figs. 2.8, 2.9, 2.10 and 2.11, the left and right sides both map under ϕ to the same NJ^{pt} proof-term. □

We conjecture that: if $\Sigma; \Delta \Rightarrow e_1 : F$ and $\Sigma; \Delta \Rightarrow e_2 : F$ are derivable in LJ^{pt} and $\phi(e_1) = \phi(e_2)$, then $e_1 \cong_e e_2$. If this conjecture holds then ϕ is a means of deciding whether or not proof-terms of derivations are $\cong_e$-related. (This conjecture has been shown [DP96a] to hold for the fragment of implicational logic. The proof of this result uses permutations (7), (35), (43) and (47). We believe the same arguments carry over to the general case.) The conjecture is of a result similar to (and we believe essentially the same as) the result proved by Mints, in [Min94], using different techniques and with different inference rules.

The mapping ϕ is not injective, in other words, LJ^{pt}-derivations are not in 1-1 correspondence to normal NJ^{pt}-deductions; but ϕ is onto. In Sec. 3.5 is shown a class of LJ^{pt}-derivations that is in 1-1 correspondence to the class of expanded normal deductions, for the fragment of hereditary Harrop logic. (The result may be carried over to full first-order intuitionistic logic [DP96b].)

Chapter 3

Proof Theory and Pure Logic Programming

3.1 Introduction

This chapter presents a proof-theoretic approach to semantics of logic programming languages, that is used in Chapter 4 as a foundation for integrating logic and functional programming. In order to define a semantics for a logic programming language it is necessary to define: (i) what is a program in the language; (ii) what is a goal in the language; and (iii) when is a goal achievable w.r.t. a program and how is a goal achievable w.r.t. a program, i.e. what are the different means of goal-achievement. A proof-theoretic semantics for a logic programming language defines these concepts by means of the proof theory of a formal system.

In this chapter two logic programming languages based on hereditary Harrop logic are defined. One of the languages, FOPLP, is based upon the calculus hH (*h*ereditary *H*arrop), which is a sequent calculus formalisation of first-order intuitionistic hereditary Harrop logic. The calculus hH is essentially a restriction of LJ^{pt} for hereditary Harrop logic, where the notions of well-formedness are encoded by means of derivable judgements. The other language, HOPLP, is based upon the calculus HH (*h*igher-order *h*ereditary *H*arrop), which is a sequent calculus formalisation of a higher-order hereditary Harrop logic (the logic obtained from first-order hereditary Harrop logic by replacing first-order terms by λ-terms).

In order to fix the means of goal-achievement in FOPLP, the class of uniform linear focused derivations of hH is introduced in 3.3. Roughly, in FOPLP: a program is a *basis* of hH; a goal is a formula of hH; the different means of achieving a goal G w.r.t. a program P are the proof-terms, encoding uniform linear focused derivations, for deriving G w.r.t. P in hH. The calculus hH^{ULF} is introduced in Sec. 3.4 to capture exactly the class of uniform linear focused derivations of hH. Section 3.5 shows that hH^{ULF}-derivations are in a 1-1 correspondence to expanded normal deductions. So, there is a simple interpretation of the semantics of FOPLP by means of natural deduction systems for first-order hereditary Harrop logic. In Sec. 3.8, these ideas are extended to the higher-order language HOPLP. There is given an interpretation of HOPLP by means of $NN^{\lambda norm}$, a natural deduction system for higher-order hereditary Harrop logic.

3.2 The Calculus hH for First-Order Hereditary Harrop Logic

The classes of objects of the calculus hH are defined as follows: the class τ of simple types, the class t of first-order terms, the class Σ of signatures and the class A of atomic formulae are defined as in LJ^{pt}; the classes of H and *G-formulae*—both subclasses of LJ^{pt} logical formulae—the class Δ of *hH-contexts*—a subclass of LJ^{pt}-contexts— and the class e of *hH-proof-terms*—a subclass of LJ^{pt} -proof-terms— are defined in Fig. 3.1. H-formulae are often called *hereditary*

$$\begin{array}{lcll}
H & ::= & A \mid H \wedge H \mid G \supset H \mid \forall_{x:\tau} H & (H\text{-formulae}) \\
G & ::= & A \mid G \wedge G \mid G \vee G \mid H \supset G \mid \exists_{x:\tau} G \mid \forall_{x:\tau} G & (G\text{-formulae}) \\
\Delta & ::= & \langle\rangle \mid \Delta, x : H & (\text{contexts}) \\
e & ::= & pair(e,e) \mid inl(e) \mid inr(e) \mid lambda(x.e) \mid pair_q(t,e) & \\
 & \mid & lambda_q(x.e) \mid x \mid splitl(x,x.e) \mid splitr(x,x.e) & \\
 & \mid & apply(x,e,x.e) \mid apply_q(x,t,x.e) & (\text{proof-terms})
\end{array}$$

x ranges over the set $\mathcal{X}$ of variables and τ ranges over simple types.

Figure 3.1: Classes of objects of hH .

Harrop formulae or *program formulae* and G-formulae are called *goal formulae.* As for LJ^{pt}, hH-contexts are sets and the notation $(\Delta, x : H)$ stands for $\Delta \cup \{x : H\}$; $x \notin \Delta$ means that there is no H-formula H s.t. $x : H$ is an element of Δ. The symbol identifying a class of objects, possibly indexed, is used as a meta-variable ranging over such a class, e.g. $G, G_1, G_2, \ldots$ are used as meta-variables ranging over G-formulae.

The forms of judgement of the calculus hH are presented in Fig. 3.2. Judgements of the form $\Sigma; \Delta \Rightarrow e : G$ are called *(hH)-sequents.* Sequents are the *main judgements* of hH; any other form of judgement is called an *auxiliary judgement* of hH. The derivable auxiliary judgements of forms (i) and (ii) are the same as those of the calculus λ^{ST}, defined by the rules in Figs. 2.2 and 2.3. The derivable auxiliary judgements of forms (iii)-(vi) are defined by the rules in Fig. 3.3. The derivable sequents are defined by the rules in Fig. 3.4; essentially, the rules defining derivable sequents are obtained by constraining the rules of LJ^{pt} to hH-sequents, except for *axioms* that enforce a further constraint, i.e. hH only allows *axioms* whose main formula is atomic. Note that, in rules $\Rightarrow \forall$ the eigenvariable condition is captured by the side conditions, since if $\vdash \Sigma; \Delta$ *basis* is derivable and $x \notin \Sigma$ then x has no free occurrences in

(i)	$\vdash \Sigma$ *signature*	(signatures)
(ii)	$\Sigma \vdash t : \tau$	(terms of simple type)
(iii)	$\vdash \Sigma; \Delta$ *basis*	(bases)
(iv)	$\Sigma \vdash A\ af$	(atomic formulae)
(v)	$\Sigma \vdash H\ hf$	(hereditary Harrop formulae)
(vi)	$\Sigma \vdash G\ gf$	(goal formulae)
(vii)	$\Sigma; \Delta \Rightarrow e : G$	(proof-terms of a goal formula)

Figure 3.2: Judgement forms of hH .

formulae of Δ. The notions of *left* and *right rules*, and the notions of *main* and *side formulae* of a left rule are as for LJ^{pt}, see Sec. 2.3.2. In left rules, the rightmost sequent premiss is called the *main premiss.*

The *principal part* of a sequent derivation π is the tree obtained from π by deleting each subtree whose root is not a sequent. In a sequent $\Sigma; \Delta \Rightarrow e : G$, e is called *the proof-term of the sequent.* If $\Sigma; \Delta \Rightarrow e : G$ has a derivation π then e is called *the proof-term of* π and e is called a *proof-term for deriving* G w.r.t. $\Sigma; \Delta$. As for LJ^{pt}, it may be easily shown that the proof-term and the context of a derivation's endsequent determine uniquely, up to renaming of bound variables, the principal part of such derivation.

The calculus hH is used in Sec. 3.6 to define a semantics for the first-order pure logic programming language FOPLP. In such a language a logic program is a basis of hH; a goal is a G-formula of hH; achieving a goal G w.r.t. a program $\Sigma; \Delta$ is a search for a proof-term e s.t. the sequent $\Sigma; \Delta \Rightarrow e : G$ is derivable in hH; any such proof-term e is called a *witness* for the achievement of G w.r.t. $\Sigma; \Delta$.

In order fully to determine a semantics for FOPLP, it remains to define what counts as different means of goal-achievement. Given a basis $\Sigma; \Delta$ and a goal G there may be several proof-terms e s.t. the sequent $\Sigma; \Delta \Rightarrow e : G$ is derivable in hH. For example, let Δ be a context of the form $(x : A_1 \supset (A_2 \supset A_3), x_1 : A_1)$, where A_1, A_2 and A_3 are atomic formulae, and let Σ be a signature s.t. the judgement $\vdash \Sigma; \Delta$ *basis* is derivable in hH. Let G be the formula $A_1 \supset (((A_2 \supset A_3) \supset A_2) \supset A_3)$. The proof-terms in Fig. 3.5 are five possible witnesses for the achievement of G w.r.t. $\Sigma; \Delta$. (See Appendix B for the hH-derivation corresponding to witness (i).)

Should the five witnesses, for the achievement of G w.r.t. $\Sigma; \Delta$, shown in Fig 3.5, be considered as different means of achieving G w.r.t. $\Sigma; \Delta$ in FOPLP? Or, should some of these witnesses be regarded as essentially the same means of goal-achievement?

Under traditional declarative semantics for logic programming, based on minimal models, as referred to in Sec. 1.3, all the five witnesses above for the achievement of G w.r.t. $\Sigma; \Delta$ are

$$\frac{\vdash \Sigma \; signature}{\vdash \Sigma; \langle\rangle \; basis} \qquad \frac{\vdash \Sigma; \Delta \; basis \quad \Sigma \vdash H \; hf}{\vdash \Sigma; \Delta, x : H \; basis} \; x \notin \Delta$$

Rules for well-formed bases.

$$\frac{\Sigma \vdash t_1 : \tau_1 \cdots \Sigma \vdash t_n : \tau_n}{\Sigma \vdash pt_1 ... t_n \; af} \; p : \tau_1 \to ... \to \tau_n \to prop \in \mathcal{P}$$

Rules for well-formed atomic formulae.

$$\frac{\Sigma \vdash A \; af}{\Sigma \vdash A \; hf} \qquad \frac{\Sigma \vdash H_1 \; hf \quad \Sigma \vdash H_2 \; hf}{\Sigma \vdash H_1 \wedge H_2 \; hf}$$

$$\frac{\Sigma \vdash H \; hf \quad \Sigma \vdash G \; gf}{\Sigma \vdash G \supset H \; hf} \qquad \frac{\Sigma, x : \tau \vdash H \; hf}{\Sigma \vdash \forall_{x:\tau} H \; hf} \; x \notin \Sigma$$

Rules for well-formed program formulae.

$$\frac{\Sigma \vdash A \; af}{\Sigma \vdash A \; gf} \qquad \frac{\Sigma \vdash G_1 \; gf \quad \Sigma \vdash G_2 \; gf}{\Sigma \vdash G_1 \wedge G_2 \; gf}$$

$$\frac{\Sigma \vdash G_1 \; gf \quad \Sigma \vdash G_2 \; gf}{\Sigma \vdash G_1 \vee G_2 \; gf} \qquad \frac{\Sigma \vdash G \; gf \quad \Sigma \vdash H \; hf}{\Sigma \vdash H \supset G \; gf}$$

$$\frac{\Sigma, x : \tau \vdash G \; gf}{\Sigma \vdash \exists_{x:\tau} G \; gf} \; x \notin \Sigma \qquad \frac{\Sigma, x : \tau \vdash G \; gf}{\Sigma \vdash \forall_{x:\tau} G \; gf} \; x \notin \Sigma$$

Rules for well-formed goal formulae.

Figure 3.3: Rules for derivable auxiliary judgments.

$$\frac{\vdash \Sigma;\Delta, x : A \; basis}{\Sigma;\Delta, x : A \Rightarrow x : A}\; axiom$$

$$\frac{\Sigma;\Delta \Rightarrow e_1 : G_1 \quad \Sigma;\Delta \Rightarrow e_2 : G_2}{\Sigma;\Delta \Rightarrow pair(e_1, e_2) : G_1 \wedge G_2} \Rightarrow \wedge$$

$$\frac{\Sigma;\Delta, x : H_1 \wedge H_2, x_1 : H_1 \Rightarrow e : G}{\Sigma;\Delta, x : H_1 \wedge H_2 \Rightarrow splitl(x, x_1.e) : G} \wedge_l \Rightarrow,\; x_1 \notin \Delta$$

$$\frac{\Sigma;\Delta, x : H_1 \wedge H_2, x_1 : H_2 \Rightarrow e : G}{\Sigma;\Delta, x : H_1 \wedge H_2 \Rightarrow splitr(x, x_1.e) : G} \wedge_r \Rightarrow,\; x_1 \notin \Delta$$

$$\frac{\Sigma;\Delta \Rightarrow e : G_1 \quad \Sigma \vdash G_2\, gf}{\Sigma;\Delta \Rightarrow inl(e) : G_1 \vee G_2} \Rightarrow \vee_l \qquad \frac{\Sigma;\Delta \Rightarrow e : G_2 \quad \Sigma \vdash G_1\, gf}{\Sigma;\Delta \Rightarrow inr(e) : G_1 \vee G_2} \Rightarrow \vee_r$$

$$\frac{\Sigma;\Delta, x : H \Rightarrow e : G}{\Sigma;\Delta \Rightarrow lambda(x.e) : H \supset G} \Rightarrow\supset,\; x \notin \Delta$$

$$\frac{\Sigma;\Delta, x : G_1 \supset H_1 \Rightarrow e : G_1 \quad \Sigma;\Delta, x : G_1 \supset H_1, x_1 : H_1 \Rightarrow e_1 : G}{\Sigma;\Delta, x : G_1 \supset H_1 \Rightarrow apply(x, e, x_1.e_1) : G} \supset\Rightarrow,\; x_1 \notin \Delta$$

$$\frac{\Sigma;\Delta \Rightarrow e : [t/x]G \quad \Sigma \vdash t : \tau}{\Sigma;\Delta \Rightarrow pair_q(t, e) : \exists_{x:\tau} G} \Rightarrow \exists$$

$$\frac{\Sigma, x : \tau;\Delta \Rightarrow e : G \quad \vdash \Sigma;\Delta\; basis}{\Sigma;\Delta \Rightarrow lambda_q(x.e) : \forall_{x:\tau} G} \Rightarrow \forall \;\; x \notin \Sigma$$

$$\frac{\Sigma;\Delta, x_1 : \forall_{x:\tau} H, x_2 : [t/x]H \Rightarrow e : G \quad \Sigma \vdash t : \tau}{\Sigma;\Delta, x_1 : \forall_{x:\tau} H \Rightarrow apply_q(x_1, t, x_2.e) : G} \forall \Rightarrow,\; x_2 \notin \Delta$$

Figure 3.4: Rules for derivable sequents of hH.

regarded as the same means of achieving G w.r.t. $\Sigma;\Delta$. (Note that the goal has no existentially quantified variables.) The language λProlog is defined by means of a sequent calculus formalisation of a higher-order hereditary Harrop logic *cf* [NM88]. Such sequent calculus, when restricted to first-order logic, essentially corresponds to hH, without proof-term annotations. There, the different means of achieving a goal w.r.t. a program correspond to the different instantiations that may be given to the existentially quantified variables in the goal. So, the five witnesses above are regarded as the same means of achieving G. However, if the means of goal-achievement are considered to be the derivations which are *uniform* and use the admissible rule of *backchaining*[1] for deriving atomic goals, then the witnesses (i)-(iv) are regarded as the

[1]See Sec. 4.5 for the admissibility of a rule similar to *backchaining* in a calculus that extends hH.

(i) $apply(x, x_1, x_4.$
$lambda(x_2.lambda(x_3.apply(x_4, apply(x_3, lambda(x_7.apply(x_4, x_7, x_8.x_8)), x_6.x_6), x_5.x_5))))$

(ii) $lambda(x_2.lambda(x_3.apply(x_3, lambda(x_7.apply(x, x_1, x_9.apply(x_9, x_7, x_8.x_8))), x_6.$
$apply(x, x_1, x_4.apply(x_4, x_6, x_5.x_5)))))$

(iii) $lambda(x_2.lambda(x_3.apply(x, x_1, x_4.$
$apply(x_4, apply(x_3, lambda(x_7.apply(x_4, x_7, x_8.x_8)), x_6.x_6), x_5.x_5))))$

(iv) $lambda(x_2.lambda(x_3.apply(x, x_1, x_4.$
$apply(x_4, apply(x_3, lambda(x_7.apply(x, x_1, x_9.apply(x_9, x_7, x_8.x_8))), x_6.x_6), x_5.x_5))))$

(v) $lambda(x_2.lambda(x_3.apply(x, x_2, x_4.$
$apply(x_4, apply(x_3, lambda(x_7.apply(x, x_1, x_9.apply(x_9, x_7, x_8.x_8))), x_6.x_6), x_5.x_5))))$

Figure 3.5: Witnesses for the achievement of G w.r.t. $\Sigma; \Delta$.

same means of achieving G, but (v) constitutes a different means of achieving G. (Note that the proof-terms (i)-(iv) map under ϕ to the same normal NJ^{pt}-proof-term N and N is different from the image of the proof-term (v) under ϕ.)

In functional programming the computation mechanism consists of evaluation of an expression to some kind of normal form, *e.g.* expressions of ground type ("printable values") are usually evaluated to canonical forms of the type whereas expressions of non-ground type are only evaluated to some kind of weak normal form.

In logic programming, we take the view that the means of goal-achievement should correspond to a class of derivations satisfying some normality constraint. The result of a computation, a witness for the achievement of a goal w.r.t. a program, does not need to satisfy such normality constraint, but a normal form should be easily computable from it, if desired. Natural deductions are usually seen as the archetypal forms of reasoning for intuitionistic logic. Given a formula and a set of assumptions there may be several deductions of the formula from the assumptions. Often, deductions having the same normal form are identified. We choose the different means of goal-achievement in FOPLP to be in 1-1 correspondence with expanded normal deductions of hereditary Harrop logic. Recall that from the expanded normal form of a deduction D one may easily compute all the β-normal forms $\beta\eta$-equivalent to D and D's $\beta\eta$-normal form.

Section 3.5 shows that there is a class of hH-derivations, *uniform linear focused derivations* that is in 1-1 correspondence to expanded normal deductions. Uniform linear focused derivations may be shown to correspond precisely to Miller's uniform derivations with backchaining for deriving atomic formulae.

3.3 Uniform, Uniform Focused and Uniform Linear Focused Derivations

This section studies three classes of derivations in hH: (i) the class of *uniform* (U) derivations; (ii) the class of *uniform focused* (UF) derivations; (iii) the class of *uniform linear focused* (ULF) derivations. These three classes form a hierarchy, where ULF is a subclass of UF, which in turn is a subclass of U. It is shown in Section 3.5 that the ULF-derivations of a G-formula w.r.t. a basis $\Sigma; \Delta$ are in a 1-1 correspondence to the expanded normal deductions of G w.r.t. $\Sigma; \Delta$ in the natural deduction system NN, which is a restriction of NJ^{pt} to first-order hereditary Harrop logic allowing only normal deductions.

For each of the three classes of derivations U, UF and ULF is described a rewriting system on proof-terms. The rules of these rewriting systems are taken from the permutations on LJ^{pt}-proof-terms presented in Sec. 2.3.3. Each of the three classes U, UF and ULF is a *complete class of derivations* for hH, *i.e.* for each class, if a sequent $\Sigma; \Delta \Rightarrow e : G$ is derivable in hH then there exists e_1 s.t. $\Sigma; \Delta \Rightarrow e_1 : G$ is derivable in that class. This section describes, for each of the three classes, a procedure to obtain such e_1 given e, using only permutations from the associated rewriting system.

The first class of derivations being studied is the class of *uniform derivations.* The notion of uniform derivations was introduced in [MNPS91]. Briefly, a uniform derivation can be described as a derivation where every occurrence of a sequent whose succedent is non-atomic is the conclusion of a right rule.

Definition 3.1 (Uniform Proof-Terms) *The following grammar defines the sets of* uniform proof-terms e_u *and* atomic[2] uniform proof-terms a_u*:*

$$\begin{array}{lcl} e_u & ::= & pair(e_u, e_u) \mid inl(e_u) \mid inr(e_u) \mid lambda(x.e_u) \mid pair_q(t, e_u) \mid lambda_q(x.e_u) \mid a_u; \\ a_u & ::= & x \mid splitl(x, x.a_u) \mid splitr(x, x.a_u) \mid apply(x, e_u, x.a_u) \mid apply_q(x, t, x.a_u). \end{array}$$

Definition 3.2 (Uniform Derivations) *A derivation of a sequent is* uniform *if its proof-term is uniform.*

The proof-term (i) of Fig. 3.5 constitutes an example of a non-uniform proof-term. So, its corresponding derivation, shown in Appendix B, is an example of a non-uniform derivation. As shown in Lemma 3.1, the succedent formula of a sequent having a derivation whose proof-term is atomic uniform is an atomic formula.

[2]The main constructor of an atomic uniform proof-term is a left constructor. The terminology *atomic uniform proof-term* originates from the observation, in Lemma 3.1, that a derivation whose proof-term is atomic uniform has an atomic formula as its endsequent's succedent formula.

Lemma 3.1 *Let $\Sigma; \Delta \Rightarrow a_u : G$ be derivable. Then, G is an atomic formula.*

Proof: The proof follows by induction on the structure of a_u.

Case a_u is a variable then the last step of a derivation of $\Sigma; \Delta \Rightarrow a_u : G$ must be an axiom, so G must be atomic. (Recall that the main formula of an axiom in hH is required to be atomic.)

Case a_u is of the form $apply(x, e, x_1.a_{u_1})$. Then, the last step of a derivation of $\Sigma; \Delta \Rightarrow a_u : G$ is of the form:

$$\frac{\Sigma; \Delta_1, x : G_1 \supset H_1 \Rightarrow e : G_1 \quad \Sigma; \Delta_1, x : G_1 \supset H_1, x_1 : H_1 \Rightarrow a_{u_1} : G}{\Sigma; \Delta_1, x : G_1 \supset H_1 \Rightarrow apply(x, e, x_1.a_{u_1}) : G} \supset\Rightarrow$$

where $\Delta = (\Delta_1, x : G_1 \supset H_1)$. By the I.H., since $\Sigma; \Delta_1, x : G_1 \supset H_1, x_1 : H_1 \Rightarrow a_{u_1} : G$ is derivable, G is an atomic formula.

The other cases follow, as the latter case, easily from the I.H.. □

Recall that the transformations on LJ^{pt}-proof-terms presented in Fig. 2.8 encode the reversing of right rules below left rules in LJ^{pt}-derivations. The rewriting system associated to uniform derivations is called RS_u and is defined by means of the permutations for moving right rules below left rules.

Definition 3.3 (RS_u) *RS_u is the rewriting system consisting of the rules in Fig. 2.8, where proof-terms are restricted to hH-proof-terms. The rewrite relation induced*[3] *by RS_u is called $\rhd_u$. A proof-term e_1 is* reducible *(*rewrites*) by RS_u to a proof-term e_2 if the pair (e_1, e_2) is in the* transitive closure *of $\rhd_u$.*

Lemma 3.2 *For every rule $e_1 \rhd e_2$ of RS_u, if $\Sigma; \Delta \Rightarrow e_1 : G$ is derivable then $\Sigma; \Delta \Rightarrow e_2 : G$ is derivable.*

Proof: See the case corresponding to rule (7) in the proof of Theorem 2.6. Other cases are similar. □

Below is shown that the class of uniform derivations is complete for hH. It is also shown that the uniform proof-terms are the proof-terms to which no rule of RS_u applies. Theorem 3.1 shows that every non-uniform proof-term is reducible by RS_u to a uniform proof-term. The techniques used for proving this result are essentially the same as those used in [Mil89]. There is proved the slightly weaker result: if there is a derivation of a sequent there is a uniform derivation of that sequent, a result proved for a sequent calculus formalisation of hereditary Harrop logic, essentially corresponding to hH with no proof-terms.

Theorem 3.1 *Every non-uniform proof-term is reducible by RS_u to a uniform proof-term.*

[3]The rewrite relation induced by a list of rules R is the *compatible* and *substitutive* closure of the binary relation $\{(r, s) : r \rhd s \textit{ is a rule of } R\}$, see [Pla93].

Proof: It has to be shown that for every non-uniform proof-term e there is a uniform proof-term e_u s.t. e rewrites to e_u by using rules from RS_u. The proof follows by induction on the structure of e.

(i) If e is of the form $lambda(x.e_1)$, then either e_1 is uniform (and so is e) or e_1 is non-uniform. In the latter case, by the I.H., e_1 is reducible by RS_u to a uniform proof-term e_{u_1} and so $lambda(x.e_{u_1})$ is a uniform proof-term to which e reduces by RS_u.

(ii) The other cases where the outermost constructor of e is a right constructor follow as case (i) easily from the I.H..

(iii) The proof-term e cannot be a variable, otherwise e is uniform.

(iv) Case e is of the form $apply(x, e_1, x_1.e_2)$. We consider the case where e_1 and e_2 are non-uniform. (The other cases follow by similar arguments.) By the I.H., e_1 and e_2 are reducible by RS_u to uniform proof-terms e_{u_1} and e_{u_2}. Now, the proof follows by induction on the structure of e_{u_2}. Case e_{u_2} is an atomic uniform proof-term, then $apply(x, e_{u_1}, x_1.e_{u_2})$ is a uniform proof-term, to which $apply(x, e_1, x_1.e_2)$ reduces by RS_u. Case the outermost constructor of e_{u_2} is a right constructor, permutations from RS_u may be used to rewrite $apply(x, e_{u_1}, x_1.e_{u_2})$ to a uniform proof-term. For example, if e_{u_2} is of the form $lambda(x_2.e_{u_3})$ then, by permutation (7), $apply(x, e_{u_1}, x_1.lambda(x_2.e_{u_3}))$ is reducible to $lambda(x_2.apply(x, e_{u_1}, x_1.e_{u_3}))$. By the latter I.H., $apply(x, e_{u_1}, x_1.e_{u_3})$ is reducible by RS_u to a uniform proof-term e_{u_4}. Thus, $lambda(x_2.e_{u_4})$ is a uniform proof-term to which $apply(x, e_{u_1}, x_1.lambda(x_2.e_{u_3}))$ reduces by RS_u. The cases where the outermost constructor of e_{u_2} is either $pair$, inl, inr, $lambda_q$, $pair_q$ follow by similar arguments, using permutations (3), (11), (15), (19), (23), respectively.

(v) The other cases where the outermost constructor of e is a left constructor may be proved by using ideas similar to those used in case (iv).

□

Corollary 3.1 *The class of uniform derivations is complete for hH.*

Proof: It needs to be shown that: if $\Sigma; \Delta \Rightarrow e : G$ is derivable in hH then there exists e_1 s.t. $\Sigma; \Delta \Rightarrow e_1 : G$ has a uniform derivation. If e is uniform then a derivation of $\Sigma; \Delta \Rightarrow e : G$ is uniform. Otherwise, by Theorem 3.1, e is reducible by RS_u to a uniform proof-term e_u. By applying repeatedly Lemma 3.2, at each step of the rewriting of e into e_u, we may conclude that the sequent $\Sigma; \Delta \Rightarrow e_u : G$ is derivable in hH, *i.e.* there exists e_1 s.t. $\Sigma; \Delta \Rightarrow e_1 : G$ has a uniform derivation. □

Proposition 3.1 below shows that the proof-terms irreducible under RS_u are precisely the uniform proof-terms.

Proposition 3.1 *The set of proof-terms to which no rule of RS_u applies is the set of uniform proof-terms.*

Proof:

It needs to be shown that no rule in Fig. 2.8 is applicable to a uniform proof-term. The proof follows by induction on the structure of uniform proof-terms; e.g. no rule of RS_u applies to $apply(x, e_u, x_1.a_u)$ since, by the induction hypothesis, no rule applies either to e_u or to a_u and the outermost constructor of a_u cannot be a right constructor.

Also, it needs to be shown that any proof-term to which no rule of RS_u is applicable is a uniform proof-term, which follows immediately from Theorem 3.1. □

Theorem 3.1 shows that RS_u is weakly normalising, i.e. there is a strategy to rewrite every proof-term into a uniform proof-term. From a rewriting systems viewpoint a pertinent question to ask is whether or not RS_u is Church-Rosser and strongly normalising. We conjecture that the answer to both questions is positive.

Although the class of uniform derivations is a proper subclass of hH-derivations, there are still different uniform proof-terms mapping under ϕ to the same normal natural deduction proof-term. The proof-terms (ii), (iii), (iv) and (v) of Fig. 3.5 are all uniform. The proof-term (v) maps under ϕ to the NJ^{pt}-proof-term:

$$\lambda x_2.\lambda x_3.app(app(x, x_2), app(x_3, \lambda x_4.app(app(x, x_1), x_4))).$$

The proof-terms (ii), (iii) and (iv) map under ϕ to the NJ^{pt}-proof-term:

$$\lambda x_2.\lambda x_3.app(app(x, x_1), app(x_3, \lambda x_4.app(app(x, x_1), x_4))).$$

So, since ϕ is onto the set of normal natural deduction proof-terms, a class of derivations in 1-1 correspondence to normal natural deductions needs to be more restrictive than the class of uniform derivations.

Below is studied the class of *uniform focused* derivations that is a subclass of uniform derivations yet complete for hH. Derivations in this class have the properties of being uniform and of being focused. Briefly, a derivation is focused if the side formula in the main premise S of a left rule is the main formula of the inference rule whose conclusion is S.

In [Mil90] is described the rule of *backchaining*, which is shown to be admissible in the sequent calculus formalisation of hereditary Harrop logic there presented. This rule essentially captures the notion of focusing derivations. A more direct account of the notion of focusing derivations is described in [Pfe94] by means of the concept of *immediate implication.* We borrowed the name of *focused derivations* from Andreoli's work [And92a], in the more general context of linear logic.

Definition 3.4 (Uniform Focused Proof-Terms) *The following context sensitive grammar defines the sets of* uniform focused proof-terms e_{uf} *and* atomic uniform focused proof-terms $a_{uf}^{x_i}$ *of* head variable x_i*:*

$$\begin{array}{lll} e_{uf} & ::= & pair(e_{uf}, e_{uf}) \mid inl(e_{uf}) \mid inr(e_{uf}) \mid lambda(x.e_{uf}) \\ & \mid & pair_q(t, e_{uf}) \mid lambda_q(x.e_{uf}) \mid a^{x_i}_{uf}; \\ a^{x_i}_{uf} & ::= & x_i \mid splitl(x_i, x_j.a^{x_j}_{uf}) \mid splitr(x_i, x_j.a^{x_j}_{uf}) \\ & \mid & apply(x_i, e_{uf}, x_j.a^{x_j}_{uf}) \mid apply_q(x_i, t, x_j.a^{x_j}_{uf}). \end{array}$$

In the previous definition, the superscript notation $a^{x_i}_{uf}$ is used to represent contextual information. The last rule defining e_{uf} is an abbreviation for an infinite list of rules, one for each x_i in $\mathcal{X}$. The first rule defining $a^{x_i}_{uf}$ is an abbreviation for an infinite list of rules, one for each x_i in $\mathcal{X}$. All the other rules defining $a^{x_i}_{uf}$ are abbreviations for infinite lists of rules, one for each combination of x_i and x_j, elements of $\mathcal{X}$. Sometimes, the head variable of an atomic uniform focused proof-term is omitted.

Definition 3.5 (Uniform-Focused Derivations) *A derivation is* uniform focused *if its proof-term is uniform focused.*

The proof-term (ii) of Fig. 3.5 is an example of a uniform proof-term which is not uniform focused, for it contains a proof-term of the form:

$$apply(x_3, ..., x_6.apply(x, x_1, x_4.apply(x_4, x_6, x_5.x_5))).$$

The derivation encoded by the proof-term (ii) is uniform but not uniform focused.

Figure 2.9 presents a list of transformations on proof-terms that represent the reversing of the order in which left rules occur in a LJ^{pt}-derivation. Figure 2.10 presents a list of proof-term transformations that encode the elimination of redundant left rules. These two lists of transformations are used below in transforming non-focused derivations into focused derivations. The rewriting system associated to uniform focused derivations is called RS_{uf}; it is defined as follows.

Definition 3.6 (RS_{uf}) *RS_{uf} is the rewriting system consisting of the rules in Figs. 2.8, 2.9 and 2.10, where the rules are restricted to hH-proof-terms. The rewrite relation induced by RS_{uf} is called $\triangleright_{uf}$. A proof-term e_1* is reducible *(*rewrites*) by RS_{uf} to a proof-term e_2 if the pair (e_1, e_2) is in the* transitive closure *of $\triangleright_{uf}$.*

Lemma 3.3 *For every rule $e_1 \triangleright e_2$ of RS_{uf}, if $\Sigma; \Delta \Rightarrow e_1 : G$ is derivable in hH then $\Sigma; \Delta \Rightarrow e_2 : G$ is derivable in hH.*

Proof: See the cases corresponding to rules (7), (35) and (43) of Theorem 2.6. □

Proposition 3.2 *The rewriting system RS_{uf} is not strongly normalising.*

Proof: An infinite reduction sequence may be easily constructed from a proof-term of the form

$$apply(x, e, x_1.apply(x_2, e_1, x_3.e_2)),$$

where $x_1 \neq x_2$ and $x_1 \notin e_1$, by using rules (35) and (43). □

Although RS_{uf} is not strongly normalising, we conjecture the existence of a rewriting system RS'_{uf}, obtained by restricting the rules of RS_{uf}, s.t.: (i) RS'_{uf} is strongly normalising; and (ii) every proof-term which is not uniform focused is reducible by RS'_{uf} to a uniform focused proof-term.

The following lemma is used in proving Theorem 3.2, which provides a means for showing completeness of uniform focused derivations for hH.

Lemma 3.4 *Let e_{uf_1}, e_{uf_2} be uniform focused proof-terms. Then, every proof-term of the form*

$$apply(x, e_{uf_1}, x_1.e_{uf_2})$$

which is not uniform focused is reducible by RS_{uf} to a uniform focused proof-term e_{uf}. Further, if e_{uf_2} is atomic uniform focused of head variable $x_2 \neq x_1$, then e_{uf} is atomic uniform focused of head variable x_2.

Proof: The proof is by induction on the structure of e_{uf_2}.

(i) If e_{uf_2} is of the form $lambda(x_3.e_{uf_3})$, then permutation (7) may be applied to

$$apply(x, e_{uf_1}, x_1.e_{uf_2})$$

obtaining the proof-term

$$lambda(x_3.apply(x, e_{uf_1}, x_1.e_{uf_3})).$$

The proof-term $apply(x, e_{uf_1}, x_1.e_{uf_3})$ is either uniform focused or, by I.H., is reducible by RS_{uf} to a uniform focused proof-term. Thus, in both cases $apply(x, e_{uf_1}, x_1.e_{uf_2})$ is reducible by RS_{uf} to a uniform focused proof-term.

(ii) Proofs of the other cases where the outermost constructor of e_{uf_2} is a right constructor may be obtained by similar arguments to those used in the case above.

(iii) If e_{uf_2} is atomic uniform focused of head variable x_2, there are the following cases.

(a) Case e_{uf_2} is the variable x_2. Then, $x_2 \neq x_1$, otherwise $apply(x, e_{uf_1}, x_1.x_2)$ is uniform focused. So, the proof-term

$$apply(x, e_{uf_1}, x_1.x_2)$$

is reducible by permutation (43) to x_2 and x_2 is atomic uniform focused of head variable x_2.

(b) Case e_{uf_2} is of the form $apply(x_2, e_{uf_3}, x_3.a^{x_3}_{uf_1})$. Then, $x_2 \neq x_1$, otherwise

$$apply(x, e_{uf_1}, x_1.apply(x_2, e_{uf_3}, x_3.a^{x_3}_{uf_1}))$$

is uniform focused. So, $apply(x, e_{uf_1}, x_1.e_{uf_2})$ is reducible by permutation (35) to the proof-term

$$apply(x_2, apply(x, e_{uf_1}, x_1.e_{uf_3}), x_3.apply(x, e_{uf_1}, x_1.a^{x_3}_{uf_1})).$$

(Note that $x_1 \neq x_2$ and the other side conditions for applying permutation (35) are satisfied by renaming of bound variables.) The proof-term $apply(x, e_{uf_1}, x_1.e_{uf_3})$ is either uniform focused or, by the I.H., reducible to a uniform focused proof-term. Also by the I.H., $apply(x, e_{uf_1}, x_1.a^{x_3}_{uf_1})$ is reducible to an atomic uniform focused proof-term of head variable x_3, say $a^{x_3}_{uf_2}$. The proof of this case is concluded by observing that a proof-term of the form $apply(x_2, e_{uf_4}, x_3.a^{x_3}_{uf_2})$, where e_{uf_4} is uniform focused, is atomic uniform focused of head variable x_2.

(c) Similar reasoning may be used for showing the cases corresponding to the other forms of atomic uniform focused proof-terms.

□

Theorem 3.2 *Every proof-term which is not uniform focused is reducible by RS_{uf} to a uniform focused proof-term.*

Proof: Let e be a proof-term which is not uniform focused. The proof follows by induction on the structure of e.

Case e is of the form $lambda(x.e_1)$. Then, e_1 cannot be uniform focused, otherwise e is uniform focused. So, by the I.H., e_1 is reducible by RS_{uf} to a uniform focused proof-term and, thus, e is reducible by RS_{uf} to a uniform focused proof-term.

The other cases where the outermost constructor of e is a right constructor follow by similar arguments.

The proof-term e cannot be a variable, otherwise e is uniform focused.

Case e is of the form $apply(x, e_1, x_1.e_2)$. Assume, without loss of generality, that both e_1 and e_2 are not uniform focused proof-terms. By the I.H., each of the proof-terms e_1 and e_2 is reducible by RS_{uf} to a uniform focused proof-term, say e_{uf_1} and e_{uf_2}, respectively. So, by using Lemma 3.4, e is reducible by RS_{uf} to a uniform focused proof-term.

The cases where the outermost constructor of e is a left constructor, different from *apply*, may be shown by proving lemmas similar to Lemma 3.4. □

Corollary 3.2 *The class of uniform focused derivations is complete for hH.*

Proof: By using Theorem 3.2 together with Lemma 3.3. □

Although the class of uniform focused derivations is more restrictive than the class of uniform derivations, there are yet different uniform focused proof-terms that map under ϕ to the same normal natural deduction proof-term. The proof-terms (iii) and (iv) of Fig. 3.5 are both uniform focused and, as mentioned above, map into the same NJ^{pt}-proof-term under ϕ. So, the class of uniform focused derivations is yet too wide to be in 1-1 correspondence to the class of normal natural deductions. Below is defined the class of *uniform linear focused* derivations that is more restrictive than the class of uniform focused derivations. Roughly, a derivation is uniform linear focused if it is uniform focused and the side formula of each left rule is used exactly once as the main formula of a rule in the derivation.

Definition 3.7 (Uniform Linear Focused Proof-Terms) *A proof-term is* uniform linear focused *if it is uniform focused and each variable x bound by a left constructor occurs exactly once in the scope of x.*

Definition 3.8 (Uniform Linear Focused Derivations) *A derivation of a sequent is* uniform linear focused *if its proof-term is uniform linear focused.*

The proof-term (iii) of Fig. 3.5 is an example of a uniform focused proof-term which is not uniform linear focused, for the variable binder x_4 has two free occurrences of x_4 in its scope. The proof-term (iv) of Fig. 3.5 is an example of a uniform linear focused proof-term. So, the derivation encoded by (iii) is an example of a uniform focused derivation, which is not uniform linear focused, and the derivation encoded by (iv) is an example of a uniform linear focused derivation.

Before proving that every derivation in hH can be transformed to a uniform linear focused derivation, some definitions and results are introduced.

Definition 3.9 (Affine Proof-Terms) *A proof-term is* affine *if each variable x bound by a left constructor occurs freely at most once in the scope of x.*

Notice that a variable bound by a left constructor in an affine proof-term is allowed to have no occurrences.

Lemma 3.5 *If e is affine and uniform focused then e is uniform linear focused.*

Proof: In a uniform focused proof-term a variable bound by a left constructor occurs at least once in its scope. For example, a proof-term of the form $apply(x, e, x_1.e_1)$ is uniform focused only if e_1 is atomic uniform focused of head variable x_1, so x_1 occurs at least once in e_1. Since e is affine then, no variable bound by a left constructor occurs more than once in e. Then, e is a uniform linear focused proof-term. □

Figure 2.11 presents a set of proof-term transformations used for linearising a proof-term, i.e. the permutations that allow transforming the corresponding derivation to a derivation where all side formulae of left rules are used at most once.

Definition 3.10 (RS_a) *RS_a is the rewriting system defined by the rules in Fig. 2.11, where proof-terms are restricted to hH-proof-terms. The rewrite relation induced by RS_a is called $\triangleright_a$. A proof-term e_1* is reducible *(*rewrites*) by RS_a to a proof-term e_2 if the pair (e_1, e_2) is in the* transitive closure *of $\triangleright_a$.*

Lemma 3.6 *For every rule $e_1 \triangleright e_2$ of RS_a, if $\Sigma; \Delta \Rightarrow e_1 : G$ is derivable then $\Sigma; \Delta \Rightarrow e_2 : G$ is derivable.*

Proof: See the case corresponding to rule (47) in the proof of Theorem 2.6. Other cases are similar. □

The theorem below may be thought of as a transformation of the natural deduction graph corresponding to a sequent calculus derivation into a natural deduction tree. We hope to make precise this connection in future work.

Theorem 3.3 *Every non-affine proof-term is reducible by RS_a to an affine proof-term.*

Proof: Let e be a non-affine proof-term. The proof is by induction on the structure of e.

(i) If e is of the form $lambda(x.e_1)$, then e_1 is non-affine, otherwise e is affine. So, by the I.H., e_1 is reducible by RS_a to an affine proof-term e_2. Thus, e is reducible by RS_a to an affine proof-term. Analogous arguments may be used for proving the other cases where the outermost constructor of e is a right constructor.

(ii) The proof-term e cannot be a variable, otherwise it is affine.

(iii) Case e is of the form $apply(x, e_1, x_1.e_2)$. We assume, without loss of generality, that e_1 and e_2 are non-affine. By the I.H., e_1 and e_2 are reducible by RS_a to affine proof-terms e_3, e_4, respectively. The only case in which $apply(x, e_3, x_1.e_4)$ is non-affine is the case where x_1 has more than one occurrence in e_4. The proof follows by induction on the number of occurrences of x_1 in e_4.

- **(a)** If x_1 occurs at most once in e_4, then $apply(x, e_3, x_1.e_4)$ is affine.
- **(b)** If x_1 occurs more than once in e_4, then the proof-term $apply(x, e_3, x_1.e_4)$ is reducible by permutation (47) to $apply(x, e_3, x_1.apply(x, e_3, x_2.e_5))$, where e_5 results from e_4 by replacing one of the occurrences of x_1 in e_4 by x_2. The proof-term $apply(x, e_3, x_2.e_5)$ is affine and has fewer occurrences of x_1 than e_4. So, by the I.H., $apply(x, e_3, x_1.apply(x, e_3, x_2.e_5))$ is reducible by RS_a to an affine proof-term.

(iv) The other cases where the outermost constructor of e is a left constructor may be proved by using similar arguments.

□

Lemma 3.7 *If e is an affine proof-term, then every proof-term e_1 s.t. e is reducible to e_1 by RS_{uf} is also an affine proof-term.*

Proof: It suffices to observe that for each rule of RS_{uf} if its left side is affine then its right side is also affine. □

Definition 3.11 (RS_{ulf}) *The relation RS_{ulf} is the rewrite relation whose set of rules is the union of the sets of rules of RS_{uf} and RS_a. The rewrite relation induced by RS_a is called $\triangleright_{ulf}$. A proof-term e_1* is reducible *(*rewrites*) by RS_{ulf} to a proof-term e_2 if the pair (e_1, e_2) is in the* transitive closure *of $\triangleright_{ulf}$.*

Theorem 3.4 *Every proof-term which is not uniform linear focused is reducible by RS_{ulf} to a uniform linear focused proof-term.*

Proof: Let e be a proof-term which is not uniform linear focused. Then, by Lemma 3.5, e cannot be simultaneously affine and uniform focused.

If e is affine, by Theorem 3.2, e is reducible by RS_{ulf} to a uniform focused proof-term e_1 and, since e is affine, by Lemma 3.7, e_1 is also affine. Then, by Lemma 3.5, e_1 is uniform linear focused.

If e is non-affine, by Theorem 3.3, e is reducible by RS_{ulf} to an affine proof-term; an argument similar to that used for the previous case completes the proof. □

Corollary 3.3 *The class of uniform linear focused derivations is complete for hH.*

Proof: By using Theorem 3.4 and Lemmas 3.3 and 3.6. □

Proposition 3.3 below shows that the proof-terms irreducible under RS_{ulf} are the uniform linear focused proof-terms.

Proposition 3.3 *The set of proof-terms to which no rule of RS_{ulf} applies is the set of uniform linear focused proof-terms.*

Proof:

Analogously to Proposition 3.1, it suffices to note that no rule of RS_{ulf} is applicable to a uniform linear focused proof-term and that, by Theorem 3.4, any proof-term to which no rule of RS_{ulf} is applicable is a uniform linear focused proof-term. □

In the next section we introduce the calculus hH^{ULF} that allows exactly the uniform linear focused derivations of hH. This calculus is used in Sec. 3.5 for showing a 1-1 correspondence between uniform linear focused derivations and expanded normal deductions.

3.4 The Calculus hH^{ULF}

This section describes the calculus hH^{ULF}, which may be regarded as a calculus to obtain exactly the uniform linear focused derivations of hH. The classes of objects of hH^{ULF} are the same as those of hH. The forms of judgement of hH^{ULF} are the same as those of hH with the exception of sequents. In hH^{ULF} sequents are replaced by the following two forms of judgement:

$$\begin{array}{ll}\text{(i)} & \Sigma; \Delta \longrightarrow e : G; \\ \text{(ii)} & \Sigma; \Delta \xrightarrow{x:H} e : A.\end{array}$$

We also call *sequents* these two forms of judgement. The rules defining derivable sequents are shown in Fig. 3.6. These two forms of sequent describe the uniform linear focused derivations of hH. Sequents of form (i) describe uniform derivations of compound goals. Sequents of form (ii) describe linear focused derivations of atomic goals. The rules defining these two forms of sequent are mutually recursive— see the rules $\xrightarrow{\supset}$ and *choice*. See also [DP96b]. Calculi formalised in the same fashion as hH^{ULF} are used in [Pfe94, Mil94].

The following lemmas are used in proving the main result of this section, Theorem 3.7, which relates derivations in hH^{ULF} with uniform linear focused derivations in hH.

Lemma 3.8 (weakening) *If the judgements $\Sigma; \Delta \Rightarrow e : G$ and $\Sigma \vdash H\ hf$ are derivable in hH then, for every $x \notin \Delta$, the sequent $\Sigma; \Delta, x : H \Rightarrow e : G$ is derivable in hH.*

Proof: This result may be proved similarly to Theorem 2.5, admissibility of weakening for LJ^{pt}. □

Lemma 3.9

(1) *If $\Sigma; \Delta, x : H \longrightarrow e : G$ is derivable and $x \notin e$ then $\Sigma; \Delta \longrightarrow e : G$ is derivable.*

(2) *If $\Sigma; \Delta, x : H \xrightarrow{x_1:H_1} a^{x_1}_{ulf} : A$ is derivable and $x \notin a^{x_1}_{ulf}$ then $\Sigma; \Delta \xrightarrow{x_1:H_1} a^{x_1}_{ulf} : A$ is derivable.*

Proof: See the case corresponding to rule (43) in the proof of Theorem 2.6 for a similar result. □

Lemma 3.10 *If sequent $\Sigma; \Delta, x : H \xrightarrow{x:H} a^{x}_{ulf} : A$ is derivable and x occurs only once in a^{x}_{ulf} then $\Sigma; \Delta \xrightarrow{x:H} a^{x}_{ulf} : A$ is derivable.*

$$\frac{\Sigma;\Delta \longrightarrow e_1 : G_1 \quad \Sigma;\Delta \longrightarrow e_2 : G_2}{\Sigma;\Delta \longrightarrow pair(e_1, e_2) : G_1 \wedge G_2} \longrightarrow\wedge$$

$$\frac{\Sigma;\Delta \longrightarrow e : G_1 \quad \Sigma \vdash G_2\ gf}{\Sigma;\Delta \longrightarrow inl(e) : G_1 \vee G_2} \longrightarrow\vee_l$$

$$\frac{\Sigma;\Delta \longrightarrow e : G_2 \quad \Sigma \vdash G_1\ gf}{\Sigma;\Delta \longrightarrow inr(e) : G_1 \vee G_2} \longrightarrow\vee_r$$

$$\frac{\Sigma;\Delta, x : H \longrightarrow e : G}{\Sigma;\Delta \longrightarrow lambda(x.e) : H \supset G} \longrightarrow\supset \qquad x \notin \Delta$$

$$\frac{\Sigma;\Delta \longrightarrow e : [t/x]G \quad \Sigma \vdash t : \tau}{\Sigma;\Delta \longrightarrow pair_q(t, e) : \exists_{x:\tau}G} \longrightarrow\exists$$

$$\frac{\Sigma, x : \tau;\Delta \longrightarrow e : G \quad \vdash \Sigma;\Delta\ basis}{\Sigma;\Delta \longrightarrow lambda_q(x.e) : \forall_{x:\tau}G} \longrightarrow\forall \qquad x \notin \Sigma$$

$$\frac{\Sigma;\Delta \xrightarrow{x:H} e : A}{\Sigma;\Delta \longrightarrow e : A}\ choice \qquad x : H \in \Delta$$

$$\frac{\vdash \Sigma;\Delta\ basis \quad \Sigma \vdash A\ af}{\Sigma;\Delta \xrightarrow{x:A} x : A}\ axiom \qquad x \notin \Delta \text{ or } x : A \in \Delta$$

$$\frac{\Sigma;\Delta \xrightarrow{x_1:H_1} e : A \quad \Sigma \vdash H_2\ hf}{\Sigma;\Delta \xrightarrow{x:H_1\wedge H_2} splitl(x, x_1.e) : A} \xrightarrow{\wedge_l} \qquad \begin{array}{r} x_1 \notin \Delta \text{ and either} \\ x \notin \Delta \text{ or } x : H_1 \wedge H_2 \in \Delta \end{array}$$

$$\frac{\Sigma;\Delta \xrightarrow{x_1:H_2} e : A \quad \Sigma \vdash H_1\ hf}{\Sigma;\Delta \xrightarrow{x:H_1\wedge H_2} splitr(x, x_1.e) : A} \xrightarrow{\wedge_r} \qquad \begin{array}{r} x_1 \notin \Delta \text{ and either} \\ x \notin \Delta \text{ or } x : H_1 \wedge H_2 \in \Delta \end{array}$$

$$\frac{\Sigma;\Delta \longrightarrow e_1 : G \quad \Sigma;\Delta \xrightarrow{x_1:H} e : A}{\Sigma;\Delta \xrightarrow{x:G\supset H} apply(x, e_1, x_1.e) : A} \xrightarrow{\supset} \qquad \begin{array}{r} x_1 \notin \Delta \text{ and either} \\ x \notin \Delta \text{ or } x : G \supset H \in \Delta \end{array}$$

$$\frac{\Sigma;\Delta \xrightarrow{x_2:[t/x]H} e : A \quad \Sigma \vdash t : \tau}{\Sigma;\Delta \xrightarrow{x_1:\forall_{x:\tau}H} apply_q(x_1, t, x_1.e) : A} \xrightarrow{\forall} \qquad \begin{array}{r} x_2 \notin \Delta \text{ and either} \\ x_1 \notin \Delta \text{ or } x_1 : \forall_{x:\tau}H \in \Delta \end{array}$$

Figure 3.6: Rules for derivable sequents of hH^{ULF} .

Proof: The proof-term a^{x}_{ulf} must either be x or of one of the forms:

$$\begin{array}{l} splitl(x, x_1.a^{x_1}_{ulf_1}); \\ splitr(x, x_1.a^{x_1}_{ulf_1}); \\ apply(x, e_{ulf}, x_1.a^{x_1}_{ulf_1}); \\ apply_q(x, t, x_1.a^{x_1}_{ulf_1}). \end{array}$$

In no case may x occur in $a^{x_1}_{ulf}$ or in e_{ulf}, for x occurs only once in a^{x}_{ulf}. So, by using Lemma 3.9, we may easily conclude this proof. □

The following theorem shows how to go from uniform linear focused derivations in hH to derivations in hH^{ULF}.

Theorem 3.5 *Let $\Sigma; \Delta \Rightarrow e_{ulf} : G$ be derivable in hH. Then, $\Sigma; \Delta \longrightarrow e_{ulf} : G$ is derivable in hH^{ULF}. Further, if e_{ulf} is atomic uniform linear focused of head variable x then G is atomic and there exists an H-formula H s.t. $x : H \in \Delta$ and $\Sigma; \Delta \xrightarrow{x:H} e_{ulf} : G$ is derivable in hH^{ULF}.*

Proof: Let π be a hH-derivation of $\Sigma; \Delta \Rightarrow e_{ulf} : G$. The proof follows by induction on the structure of π. Consider the following cases.

- Case last step of π is of the form

$$\frac{\overset{\pi_1}{\Sigma; \Delta, x : H \Rightarrow e_{ulf_1} : G_1}}{\Sigma; \Delta \Rightarrow lambda(x.e_{ulf_1}) : H \supset G_1} \Rightarrow\supset,$$

  where $x \notin \Delta$ and $G = H \supset G_1$. Then, by the I.H., $\Sigma; \Delta, x : H \longrightarrow e_{ulf_1} : G_1$ has a derivation σ. Thus, the following hH^{ULF}-derivation may be formed:

$$\frac{\overset{\sigma}{\Sigma; \Delta, x : H \longrightarrow e_{ulf_1} : G_1}}{\Sigma; \Delta \longrightarrow lambda(x.e_{ulf_1}) : H \supset G_1} \longrightarrow\supset,$$

  since $x \notin \Delta$. Similar arguments may be used for the other cases where the last step of π is a right rule.

- Case last step of π is of the form

$$\frac{\overset{\pi_1}{\vdash \Sigma; \Delta_1, x : A\ basis}}{\Sigma; \Delta_1, x : A \Rightarrow x : A}\ axiom,$$

  where $G = A$ is an atomic formula and $\Delta = (\Delta_1, x : A)$. Then, the following hH^{ULF}-derivation may be formed:

$$\frac{\overset{\pi_1}{\vdash \Sigma; \Delta_1, x : A\ basis} \quad \Sigma \vdash A\ af}{\Sigma; \Delta_1, x : A \xrightarrow{x:A} x : A}\ axiom,$$

since: from π_1 we may easily find a derivation of $\Sigma \vdash A\ af$; and $x : A \in \Delta$, thus the side condition of *axiom* is satisfied. From the derivation above, by using rule *choice*, a derivation of $\Sigma; \Delta_1, x : A \longrightarrow x : A$ may be formed. (Note that $x : A \in (\Delta_1, x : A)$ and $\Sigma; \Delta_1, x : A \xrightarrow{x:A} x : A$ is derivable in hH^{ULF}.)

- Case last step of π is of the form

$$\frac{\overset{\pi_2}{\Sigma; \Delta_1, x : G_1 \supset H \Rightarrow e_{ulf_1} : G_1} \quad \overset{\pi_3}{\Sigma; \Delta_1, x : G_1 \supset H, x_1 : H \Rightarrow a^{x_1}_{ulf_1} : G}}{\Sigma; \Delta_1, x : G_1 \supset H \Rightarrow apply(x, e_{ulf_1}, x_1.a^{x_1}_{ulf_1}) : G} \supset\Rightarrow,$$

where $\Delta = (\Delta_1, x : G_1 \supset H)$ and $x_1 \notin \Delta$. By the I.H.: (i) there is a hH^{ULF}-derivation σ_1 of $\Sigma; \Delta_1, x : G_1 \supset H \longrightarrow e_{ulf_1} : G_1$; (ii) G is atomic and there is a hH^{ULF}-derivation of $\Sigma; \Delta_1, x : G_1 \supset H, x_1 : H \xrightarrow{x_1:H} a^{x_1}_{ulf_1} : G$. By Lemma 3.10, since x_1 occurs exactly once in $a^{x_1}_{ulf_1}$— for $apply(x, e_{ulf_1}, x_1.a^{x_1}_{ulf_1})$ is uniform linear focused— there is a derivation σ_2 of the sequent

$$\Sigma; \Delta_1, x : G_1 \supset H \xrightarrow{x_1:H} a^{x_1}_{ulf_1} : G.$$

Thus, the following derivation may be formed:

$$\frac{\overset{\sigma_1}{\Sigma; \Delta_1, x : G_1 \supset H \longrightarrow e_{ulf_1} : G_1} \quad \overset{\sigma_2}{\Sigma; \Delta_1, x : G_1 \supset H \xrightarrow{x_1:H} a^{x_1}_{ulf_1} : G}}{\Sigma; \Delta_1, x : G_1 \supset H \xrightarrow{x:G_1 \supset H} apply(x, e_{ulf_1}, x_1.a^{x_1}_{ulf_1}) : G} \xrightarrow{\supset},$$

since $x_1 \notin (\Delta_1, x : G_1 \supset H)$ and $x : G_1 \supset H \in (\Delta_1, x : G_1 \supset H)$. For concluding the proof of this case, observe that, from the derivation above, by applying rule *choice*, for $x : G_1 \supset H \in (\Delta_1, x : G_1 \supset H)$, there is a hH^{ULF}-derivation of

$$\Sigma; \Delta_1, x : G_1 \supset H \longrightarrow apply(x, e_{ulf_1}, x_1.a^{x_1}_{ulf_1}) : G.$$

(Note that $x : G_1 \supset H \in (\Delta_1, x : G_1 \supset H)$ and

$$\Sigma; \Delta_1, x : G_1 \supset H \xrightarrow{x:G_1 \supset H} apply(x, e_{ulf_1}, x_1.a^{x_1}_{ulf_1}) : G$$

is derivable in hH^{ULF}.)

- The other cases where the last step of π is a left rule follow by similar arguments.

□

The following theorem shows how to transform a hH^{ULF}-derivation into a uniform linear focused derivation of hH.

Theorem 3.6

(1) *If $\Sigma; \Delta \longrightarrow e : G$ is derivable in hH^{ULF} then $\Sigma; \Delta \Rightarrow e : G$ has a uniform linear focused derivation.*

(2) *If $\Sigma; \Delta \xrightarrow{x:H} e : A$ is derivable in hH^{ULF} then e is atomic uniform linear focused of head variable x and*

  (a) *case $x : H \in \Delta$, then $\Sigma; \Delta \Rightarrow e : A$ is derivable in hH;*

  (b) *case $x \notin \Delta$, then $\Sigma; \Delta, x : H \Rightarrow e : A$ is derivable in hH and x occurs exactly once in e.*

Proof: Let π and σ be hH^{ULF}-derivations of the sequents $\Sigma; \Delta \longrightarrow e : G$ and $\Sigma; \Delta \xrightarrow{x:H} e : A$, respectively. The proof follows by simultaneous induction on the structure of the derivations π and σ.

- Case last step of π is of the form:

$$\frac{\overset{\sigma_1}{\Sigma; \Delta \xrightarrow{x:H} e : A}}{\Sigma; \Delta \longrightarrow e : A}\ choice, x : H \in \Delta.$$

  Then, by the I.H., e is (atomic) uniform linear focused and, since $x : H \in \Delta$, the sequent $\Sigma; \Delta \Rightarrow e : A$ is derivable.

- Case last step of π is of the form

$$\frac{\overset{\sigma_1}{\Sigma; \Delta, x : H \longrightarrow e : G_1}}{\Sigma; \Delta \longrightarrow lambda(x.e) : H \supset G_1}\ \longrightarrow\supset$$

  where $x \notin \Delta$ and $G = H \supset G_1$. Then, by the I.H., $\Sigma; \Delta, x : H \Rightarrow e : G_1$ is derivable and e is uniform linear focused. Thus, by using $\Rightarrow\supset$, $\Sigma; \Delta \Rightarrow lambda(x.e) : H \supset G_1$ is derivable and clearly $lambda(x.e)$ is uniform linear focused. (Similarly for the cases where the last step of π is of any other form.)

- Case last step of σ is of the form

$$\frac{\overset{\sigma_1}{\vdash \Sigma; \Delta\, basis} \quad \Sigma \vdash A\, af}{\Sigma; \Delta \xrightarrow{x:A} x : A}\ axiom,$$

  where either $x \notin \Delta$ or $x : A \in \Delta$. If $x : A \in \Delta$ then the following derivation may be formed:

$$\frac{\overset{\sigma_1}{\vdash \Sigma; \Delta\, basis}}{\Sigma; \Delta \Rightarrow x : A}\ axiom.$$

  If $x \notin \Delta$ then the following derivation may be formed:

$$\frac{\overset{\pi_1}{\vdash \Sigma; \Delta, x : A\, basis}}{\Sigma; \Delta, x : A \Rightarrow x : A}\ axiom,$$

  where π_1 may be obtained by combining σ_1 with the derivation of $\Sigma \vdash A\, af$, for $x \notin \Delta$. Clearly x occurs exactly once in x.

- Case last step of σ is of the form:

$$\frac{\overset{\sigma_1}{\Sigma;\Delta \longrightarrow e : G_1} \quad \overset{\sigma_2}{\Sigma;\Delta \overset{x_1:H}{\longrightarrow} e_1 : A}}{\Sigma;\Delta \overset{x:G_1 \supset H}{\longrightarrow} apply(x,e,x_1.e_1) : A} \overset{\supset}{\longrightarrow},$$

where $x_1 \notin \Delta$ and either $x \notin \Delta$ or $x : G_1 \supset H \in \Delta$. Since $x_1 \notin \Delta$, by the I.H., there are derivations π_1 and π_2 of the sequents:

$$\Sigma;\Delta \Rightarrow e : G_1;$$
$$\Sigma;\Delta, x_1 : H \Rightarrow e_1 : A,$$

where e is uniform linear focused, e_1 is atomic uniform linear focused of head variable x_1 and x_1 occurs exactly once in e_1. So, $apply(x,e,x_1.e_1)$ is atomic uniform linear focused of head variable x.

If $x \notin \Delta$, then, by weakening, there are derivations π_3 and π_4 of the sequents:

$$\Sigma;\Delta, x : G_1 \supset H \Rightarrow e : G_1;$$
$$\Sigma;\Delta, x_1 : H, x : G_1 \supset H \Rightarrow e_1 : G.$$

So, the following derivation may be formed:

$$\frac{\overset{\pi_3}{\Sigma;\Delta, x : G_1 \supset H \Rightarrow e : G_1} \quad \overset{\pi_4}{\Sigma;\Delta, x : G_1 \supset H, x_1 : H \Rightarrow e_1 : G}}{\Sigma;\Delta, x : G_1 \supset H \Rightarrow apply(x,e,x_1.e_1) : G} \supset\Rightarrow .$$

Notice that x occurs exactly once in $apply(x,e,x_1.e_1)$.

If $x : G_1 \supset H \in \Delta$, i.e. Δ is of the form $(\Delta_1, x : G_1 \supset H)$, then the following derivation may be formed:

$$\frac{\overset{\pi_1}{\Sigma;\Delta_1, x : G_1 \supset H \Rightarrow e : G_1} \quad \overset{\pi_2}{\Sigma;\Delta_1, x : G_1 \supset H, x_1 : H \Rightarrow e_1 : G}}{\Sigma;\Delta_1, x : G_1 \supset H \Rightarrow apply(x,e,x_1.e_1) : G} \supset\Rightarrow .$$

(The other cases, resulting from the other possible forms of σ, follow by similar arguments.)

□

Now the main result of this section is established.

Theorem 3.7 *$\Sigma;\Delta \Rightarrow e : G$ has a uniform linear focused derivation in hH iff $\Sigma;\Delta \longrightarrow e : G$ is derivable in hH^{ULF}.*

Proof: If $\Sigma;\Delta \Rightarrow e : G$ has a uniform linear focused derivation in hH, then, by Theorem 3.5, $\Sigma;\Delta \longrightarrow e : G$ is derivable in hH^{ULF}.

If $\Sigma;\Delta \longrightarrow e : G$ is derivable in hH^{ULF}, then, by Theorem 3.6, $\Sigma;\Delta \Rightarrow e : G$ has a uniform linear focused derivation in hH. □

The next section establishes a 1-1 correspondence between derivations in hH^{ULF} and expanded normal deductions in a restriction of NJ^{pt} to hereditary Harrop logic. So, by Theorem 3.7 we may conclude the 1-1 correspondence between uniform linear focused derivations and expanded normal deductions.

3.5 On the Bijection between Uniform Linear Focused Derivations and Expanded Normal Deductions.

This section establishes a 1-1 correspondence between uniform linear focused derivations, i.e. derivations in the sequent calculus formalisation of hereditary Harrop logic hH^{ULF}, and expanded normal deductions, i.e. derivations in the natural deduction system for first-order hereditary Harrop logic NN, presented in this section. In order to achieve this correspondence, we use an intermediate sequent calculus formalisation of hereditary Harrop logic called MM. MM may be regarded as a sequent calculus very similar to hH^{ULF}, it differs only from hH^{ULF} in how derivations are annotated with proof-terms. The form of annotating sequent calculus derivations used in MM follows Herbelin's [Her95]. Herbelin uses an alternative representation of λ-terms, called $\overline{\lambda}$-terms, for annotating sequent calculus derivations. Roughly, $\overline{\lambda}$-terms bring to the surface the *head variable* of a λ-term. Calculi similar to MM and NN are studied in [DP96b]; in fact this work addresses full first-order intuitionistic logic. Below, the work [DP96b] is sometimes referred to for proofs of results relating MM and NN.

We now define the calculus MM. The classes of objects of MM are the same as those of hH^{ULF}. They are defined by the same grammars, except for proof-terms. The proof-terms of MM are called *M-proof-terms*; they are defined as follows.

Definition 3.12 M**-Proof-Terms**

$$\begin{array}{lcl} M & ::= & (x; Ms) \mid par(M,M) \mid il(M) \mid ir(M) \mid lamb(x.M) \mid par_q(t,M) \mid lamb_q(x.M) \\ Ms & ::= & [] \mid [M|Ms] \mid fst(Ms) \mid snd(Ms) \mid appl_q(t,Ms) \end{array}$$

Below, M, possibly indexed, is used as a meta-variable ranging over M-proof-terms.

The forms of judgement of MM are the same as those of hH^{ULF}, except for sequents, which are replaced by the following two forms of judgement, that we also call *sequents*:

$$\begin{array}{ll} \text{(i)} & \Sigma; \Delta \Longrightarrow M : G; \\ \text{(ii)} & \Sigma; \Delta \overset{H}{\Longrightarrow} Ms : A. \end{array}$$

MM-judgements common to hH^{ULF} are derivable iff they are derivable in hH^{ULF}. The rules defining the derivable sequents of MM are shown in Fig. 3.7. The rules defining derivable sequents in MM are very similar to the rules defining derivable sequents in hH^{ULF}; the main difference being the proof-term annotation of the rules for deriving atomic sequents.

$$\frac{\Sigma;\Delta \Longrightarrow e_1 : G_1 \quad \Sigma;\Delta \Longrightarrow e_2 : G_2}{\Sigma;\Delta \Longrightarrow par(e_1, e_2) : G_1 \wedge G_2} \Longrightarrow\wedge$$

$$\frac{\Sigma;\Delta \Longrightarrow e : G_1 \quad \Sigma \vdash G_2\ gf}{\Sigma;\Delta \Longrightarrow il(e) : G_1 \vee G_2} \Longrightarrow\vee_l \qquad \frac{\Sigma;\Delta \Longrightarrow e : G_2 \quad \Sigma \vdash G_1\ gf}{\Sigma;\Delta \Longrightarrow ir(e) : G_1 \vee G_2} \Longrightarrow\vee_r$$

$$\frac{\Sigma;\Delta, x : H \Longrightarrow e : G}{\Sigma;\Delta \Longrightarrow lamb(x.e) : H \supset G} \Longrightarrow\supset,\ x \notin \Delta$$

$$\frac{\Sigma;\Delta \Longrightarrow e : [t/x]G \quad \Sigma \vdash t : \tau}{\Sigma;\Delta \Longrightarrow par_q(t, e) : \exists_{x:\tau} G} \Longrightarrow\exists$$

$$\frac{\Sigma, x : \tau;\Delta \Longrightarrow e : G \quad \vdash \Sigma;\Delta\ basis}{\Sigma;\Delta \Longrightarrow lamb_q(x.e) : \forall_{x:\tau} G} \Longrightarrow\forall,\ \ x \notin \Sigma$$

$$\frac{\Sigma;\Delta \overset{H}{\Longrightarrow} Ms : A}{\Sigma;\Delta \Longrightarrow (x; Ms) : A}\ select,\ x : H \in \Delta$$

$$\frac{\vdash \Sigma;\Delta\ basis \quad \Sigma \vdash A\ af}{\Sigma;\Delta \overset{A}{\Longrightarrow} [] : A}\ axiom$$

$$\frac{\Sigma;\Delta \overset{H_1}{\Longrightarrow} Ms : A \quad \Sigma \vdash H_2\ hf}{\Sigma;\Delta \overset{H_1 \wedge H_2}{\Longrightarrow} fst(Ms) : A} \overset{\wedge_l}{\Longrightarrow} \qquad \frac{\Sigma;\Delta \overset{H_2}{\Longrightarrow} Ms : A \quad \Sigma \vdash H_1\ hf}{\Sigma;\Delta \overset{H_1 \wedge H_2}{\Longrightarrow} snd(Ms) : A} \overset{\wedge_r}{\Longrightarrow}$$

$$\frac{\Sigma;\Delta \Longrightarrow M : G \quad \Sigma;\Delta \overset{H}{\Longrightarrow} Ms : A}{\Sigma;\Delta \overset{G \supset H}{\Longrightarrow} [M|Ms] : A} \overset{\supset}{\Longrightarrow}$$

$$\frac{\Sigma;\Delta \overset{[t/x]H}{\Longrightarrow} Ms : A \quad \Sigma \vdash t : \tau}{\Sigma;\Delta \overset{\forall_{x:\tau} H}{\Longrightarrow} appl_q(t, Ms) : A} \overset{\forall}{\Longrightarrow}$$

Figure 3.7: Rules for derivable sequents of MM.

In proof-terms of the form $lamb(x.M)$ and $lamb_q(x.M)$, x is called a *binder* of *scope* M and an occurrence of x in M is called *bound*. A non-bound occurrence of a variable is called *free*. The notation $x \notin M$ means that x has no free occurrences in M.

Below, in Definitions 3.13 and 3.14, we define mappings relating the sets of uniform linear focused proof-terms and M-proof-terms. These mappings are inverses of each other, as shown in Theorems 3.8 and 3.9, and they preserve derivability, as shown in Theorems 3.10 and 3.11.

In this subsection e is used for representing the class of uniform linear focused proof-terms and a for representing the class of atomic uniform linear focused proof-terms. e, possibly indexed, is also used for ranging over arbitrary uniform linear focused proof-terms. a, possibly indexed, with a variable x in superscript, is used for ranging over atomic uniform linear focused proof-terms of head variable x.

Definition 3.13 $\gamma : e \to M$ **and** $\gamma_1 : a \to Ms$

$$\begin{aligned}
&\gamma(pair(e_1, e_2)) =_{def} par(\gamma(e_1), \gamma(e_2))\\
&\gamma(inl(e)) =_{def} il(\gamma(e))\\
&\gamma(inr(e)) =_{def} ir(\gamma(e))\\
&\gamma(lambda(x.e)) =_{def} lamb(x.\gamma(e))\\
&\gamma(pair_q(t, e)) =_{def} par_q(t, \gamma(e))\\
&\gamma(lambda_q(x.e)) =_{def} lamb_q(x.\gamma(e))\\
&\gamma(a^x) =_{def} (x; \gamma_1(a^x))
\end{aligned}$$

$$\begin{aligned}
&\gamma_1(x) =_{def} []\\
&\gamma_1(splitl(x, x_1.a^{x_1})) =_{def} fst(\gamma_1(a^{x_1}))\\
&\gamma_1(splitr(x, x_1.a^{x_1})) =_{def} snd(\gamma_1(a^{x_1}))\\
&\gamma_1(apply(x, e, x_1.a^{x_1})) =_{def} [\gamma(e)|\gamma_1(a^{x_1})]\\
&\gamma_1(apply_q(x, t, x_1.a^{x_1})) =_{def} appl_q(t, \gamma_1(a^{x_1}))
\end{aligned}$$

Definition 3.14 $\delta : M \to e$

$$\begin{aligned}
&\delta(par(M_1, M_2)) =_{def} pair(\delta(M_1), \delta(M_2))\\
&\delta(il(M)) =_{def} inl(\delta(M))\\
&\delta(ir(M)) =_{def} inr(\delta(M))\\
&\delta(lamb(x.M)) =_{def} lambda(x.\delta(M))\\
&\delta(par_q(t, M)) =_{def} pair(t, \delta(M))\\
&\delta(lamb_q(x.M)) =_{def} lambda_q(x.\delta(M))\\
&\delta((x; [])) =_{def} x\\
&\delta((x; fst(Ms))) =_{def} splitl(x, x_1.\delta((x_1; Ms))),\ x_1 \notin Ms\\
&\delta((x; snd(Ms))) =_{def} splitr(x, x_1.\delta((x_1; Ms))),\ x_1 \notin Ms\\
&\delta((x; [M|Ms])) =_{def} apply(x, \delta(M), x_1.\delta((x_1; Ms))),\ x_1 \notin Ms\\
&\delta((x; appl_q(t, Ms))) =_{def} apply_q(x, t, x_1.\delta((x_1; Ms))),\ x_1 \notin Ms
\end{aligned}$$

Recall that two e-(proof-terms) are equal iff they are the same up to renaming of bound variables. Definition 3.14, above, requires equality of e-terms up to renaming of bound variables, otherwise it would be ill-formed, since a constraint of the form $x \notin Ms$ is satisfied by infinitely many variables.

Theorem 3.8 *For every uniform linear focused proof-term e, for every variable x and for every atomic uniform linear focused proof-term a^x of head variable x, the following identities hold:*

$$\begin{array}{ll} \text{(i)} & \delta\circ\gamma(e) = e; \\ \text{(ii)} & \delta((x; \gamma_1(a^x))) = a^x. \end{array}$$

Proof: By simultaneous induction on the structures of e and a^x.

(i) Case $e = apply(x, e_1, x_1.a_1^{x_1})$.

$$\begin{array}{lll} & \delta\circ\gamma(apply(x, e_1, x_1.a_1^{x_1})) & \\ = & \delta((x; [\gamma(e_1)|\gamma_1(a_1^{x_1})])) & \text{by def. of } \gamma \text{ and } \gamma_1 \\ = & apply(x, \delta(\gamma(e_1)), x_1.\delta((x_1; \gamma_1(a_1^{x_1})))) & \text{by def. of } \delta\text{, since } x_1 \notin \gamma_1(a_1^{x_1}) \\ = & apply(x, e_1, x_1.a_1^{x_1}) & \text{by I.H. (twice)} \end{array}$$

The other cases are similar.

(ii) Case $a^x = x$.

$$\begin{array}{lll} & \delta((x; \gamma_1(x))) & \\ = & \delta((x; [])) & \text{by def. of } \gamma_1 \\ = & x & \text{by def. of } \delta \end{array}$$

Case $a^x = apply(x, e_1, x_1.a_1^{x_1})$.

$$\begin{array}{lll} & \delta((x; \gamma_1(apply(x, e_1, x_1.a_1^{x_1})))) & \\ = & \delta((x; [\gamma(e_1)|\gamma_1(a_1^{x_1})])) & \text{by def. of } \gamma_1 \\ = & apply(x, \delta(\gamma(e_1)), x_1.\delta((x_1; \gamma_1(a_1^{x_1})))) & \text{by def. of } \delta\text{, since } x_1 \notin \gamma_1(a_1^{x_1}) \\ = & apply(x, e_1, x_1.a_1^{x_1}) & \text{by I.H. (twice)} \end{array}$$

The other cases are similar. □

Theorem 3.9 *For every M, Ms, x, the following identities hold:*

$$\begin{array}{ll} \text{(i)} & \gamma\circ\delta(M) = M; \\ \text{(ii)} & \gamma_1\circ\delta((x; Ms)) = Ms. \end{array}$$

Proof: By simultaneous induction on the structures of M and Ms.

(i) Case $M = (x; [M_1|Ms_1])$.

$$\begin{array}{lll} & \gamma\circ\delta((x; [M_1|Ms_1])) & \\ = & \gamma(apply(x, \delta(M_1), x_1.\delta((x_1; Ms_1)))) & \text{by def. of } \delta\text{, for every } x_1 \notin Ms_1 \\ = & (x; [\gamma(\delta(M_1))|\gamma_1(\delta((x_1; Ms_1)))]) & \text{by def. of } \gamma \text{ and } \gamma_1 \\ = & (x; [M_1|Ms_1]) & \text{by I.H. (twice)} \end{array}$$

The other cases are similar.

(ii) Case $Ms = []$.

$$\begin{array}{lll} & \gamma_1 \circ \delta((x;[])) & \\ = & \gamma_1(x) & \text{by def. of } \delta \\ = & [] & \text{by def. of } \gamma_1 \end{array}$$

Case $Ms = [M_1|Ms_1]$.

$$\begin{array}{lll} & \gamma_1 \circ \delta((x;[M_1|Ms_1])) & \\ = & \gamma_1(apply(x, \delta(M_1), x_1.\delta((x_1; Ms_1)))) & \text{by def. of } \delta \text{, for every } x_1 \notin Ms_1 \\ = & [\gamma(\delta(M_1))|\gamma_1(\delta((x_1; Ms_1)))] & \text{by def. of } \gamma_1 \\ = & [M_1|Ms_1] & \text{by I.H. (twice)} \end{array}$$

The other cases are similar. □

Theorem 3.10

(1) *If* $\Sigma; \Delta \longrightarrow e : G$ *is derivable in* hH^{ULF} *then* $\Sigma; \Delta \Longrightarrow \gamma(e) : G$ *is derivable in* MM.

(2) *If* $\Sigma; \Delta \xrightarrow{x:H} a^x : A$ *is derivable in* hH^{ULF} *then* $\Sigma; \Delta \xRightarrow{H} \gamma_1(a^x) : A$ *is derivable in* MM.

Proof: By simultaneous induction on the structure of the hH^{ULF}-derivations π_1 and π_2 of the sequents $\Sigma; \Delta \longrightarrow e : G$ and $\Sigma; \Delta \xrightarrow{x:H} a^x : A$, respectively.

Case the last step of π_1 is of the form:

$$\frac{\Sigma; \Delta \xrightarrow{x_1:H_1} e : A}{\Sigma; \Delta \longrightarrow e : A}\ choice,\ x_1 : H_1 \in \Delta.$$

Then, by Theorem 3.6, e is atomic uniform linear focused of head variable x_1. So, by the I.H., there is a MM-derivation of

$$\Sigma; \Delta \xRightarrow{H_1} \gamma_1(e) : A.$$

Thus, the following MM-derivation may be formed:

$$\frac{\Sigma; \Delta \xRightarrow{H_1} \gamma_1(e) : A}{\Sigma; \Delta \Longrightarrow (x_1; \gamma_1(e)) : A}\ select,$$

since $x_1 : H_1 \in \Delta$. Observe that the identity $\gamma(e) = (x_1; \gamma_1(e))$ holds, since e is atomic uniform linear focused of head variable x_1.

The cases where the last step of π_1 is of any other form are similar.

Case the last step of π_2 is of the form:

$$\frac{\Sigma; \Delta \longrightarrow e_1 : G_1 \quad \Sigma; \Delta \xrightarrow{x_1:H_1} a_1^{x_1} : A}{\Sigma; \Delta \xrightarrow{x:G_1 \supset H_1} apply(x, e_1, x_1.a_1^{x_1}) : A}\ \xrightarrow{\supset},$$

where $x_1 \notin \Delta$ and either $x : G_1 \supset H_1 \in \Delta$ or $x \notin \Delta$. Then, by the I.H., the following derivation in MM may be formed:

$$\frac{\Sigma; \Delta \Longrightarrow \gamma(e_1) : G_1 \quad \Sigma; \Delta \overset{H_1}{\Longrightarrow} \gamma_1(a_1^{x_1}) : A}{\Sigma; \Delta \overset{G_1 \supset H_1}{\Longrightarrow} [\gamma(e_1) | \gamma_1(a_1^{x_1})] : A} \overset{\supset}{\Longrightarrow}.$$

Note that the identity $\gamma_1(apply(x, e_1, x_1.a_1^{x_1})) = [\gamma(e_1)|\gamma_1(a_1^{x_1})]$ holds, by definition of γ_1. The cases where the last step of π_2 is of any other form are similar. □

Theorem 3.11

(1) *If* $\Sigma; \Delta \Longrightarrow M : G$ *is derivable in* MM *then* $\Sigma; \Delta \longrightarrow \delta(M) : G$ *is derivable in* hH^{ULF}.

(2) *If* $\Sigma; \Delta \overset{H}{\Longrightarrow} Ms : A$ *is derivable in* MM *then* $\Sigma; \Delta \overset{x:H}{\longrightarrow} \delta((x; Ms)) : A$ *is derivable in* hH^{ULF}*, for every* x *s.t. either* $x : H \in \Delta$ *or* $x \notin \Delta$.

Proof: By simultaneous induction on the structure of the MM-derivations π_1 and π_2 of the sequents $\Sigma; \Delta \Longrightarrow M : G$ and $\Sigma; \Delta \overset{H}{\Longrightarrow} Ms : A$, respectively.
Case the last step of π_1 is of the form:

$$\frac{\Sigma; \Delta \overset{H_1}{\Longrightarrow} Ms_1 : A}{\Sigma; \Delta \Longrightarrow (x_1; Ms_1) : A} \; select,$$

where $x_1 : H_1 \in \Delta$. Then, by the I.H., the sequent $\Sigma; \Delta \overset{x_1:H_1}{\longrightarrow} \delta((x_1; Ms_1)) : A$ is derivable in hH^{ULF}. So, the following hH^{ULF}-derivation may be formed:

$$\frac{\Sigma; \Delta \overset{x_1:H_1}{\longrightarrow} \delta((x_1; Ms_1)) : A}{\Sigma; \Delta \longrightarrow \delta((x_1; Ms_1)) : A} \; choice,$$

for $x_1 : H_1 \in \Delta$. The cases where the last step of π_1 is of any other form also follow easily by the I.H..
Case the last step of π_2 is of the form:

$$\frac{\vdash \Sigma; \Delta \; basis \quad \Sigma \vdash A \; af}{\Sigma; \Delta \overset{A}{\Longrightarrow} [] : A} \; axiom.$$

Then, for every variable x s.t. $x : A \in \Delta$ or $x \notin \Delta$, the following *axiom* may be formed in hH^{ULF}:

$$\frac{\vdash \Sigma; \Delta \; basis \quad \Sigma \vdash A \; af}{\Sigma; \Delta \overset{x:A}{\longrightarrow} x : A} \; axiom.$$

Note that $\delta((x; [])) = x$.

Case the last step of π_2 is of the form:

$$\frac{\Sigma;\Delta \Longrightarrow M_1 : G_1 \quad \Sigma;\Delta \overset{H_1}{\Longrightarrow} Ms_1 : A}{\Sigma;\Delta \overset{G_1 \supset H_1}{\Longrightarrow} [M_1|Ms_1] : A} \overset{\supset}{\Longrightarrow}.$$

Then, by the I.H., there is a hH^{ULF}-derivation of the sequent

$$\Sigma;\Delta \longrightarrow \delta(M_1) : G_1$$

and, for every variable x_1 s.t. $x_1 : H_1 \in \Delta$ or $x_1 \notin \Delta$, there is a hH^{ULF}-derivation of the sequent

$$\Sigma;\Delta \overset{x_1:H_1}{\longrightarrow} \delta((x_1; Ms_1)) : A.$$

So, for every variable x s.t. $x : G_1 \supset H_1 \in \Delta$ or $x \notin \Delta$, by choosing $x_2 \notin \Delta$, the following derivation may be formed

$$\frac{\Sigma;\Delta \longrightarrow \delta(M_1) : G_1 \quad \Sigma;\Delta \overset{x_2:H_1}{\longrightarrow} \delta((x_2; Ms_1)) : A}{\Sigma;\Delta \overset{x:G_1 \supset H_1}{\longrightarrow} apply(x, \delta(M_1), x_2.\delta((x_2; Ms_1))) : A} \overset{\supset}{\longrightarrow}.$$

Note that $\delta((x; [M_1|Ms_1])) = apply(x, \delta(M_1), x_2.\delta((x_2; Ms_1)))$, since $x_2 \notin Ms_1$.
The cases where the last step of π_2 is of any other form are similar. □

The next result establishes a 1-1 correspondence between forms of deriving a goal w.r.t. a basis in hH^{ULF} and MM.

Theorem 3.12 *The set of proof-terms e s.t. $\Sigma;\Delta \longrightarrow e : G$ is derivable in hH^{ULF} is in 1-1 correspondence with the set of proof-terms M s.t. $\Sigma;\Delta \Longrightarrow M : G$ is derivable in MM.*

Proof: We show that γ is a bijection between these two sets of proof-terms. First, we show that for every M s.t. $\Sigma;\Delta \Longrightarrow M : G$ is derivable in MM there exists e_1 s.t. $\Sigma;\Delta \longrightarrow e_1 : G$ is derivable in hH^{ULF} and $\gamma(e_1) = M$, thus showing the surjectivity of γ.

By Theorem 3.11, if $\Sigma;\Delta \Longrightarrow M : G$ is derivable in MM then $\Sigma;\Delta \longrightarrow \delta(M) : G$ is derivable in hH^{ULF}. For showing that $\delta(M)$ satisfies the conditions on e_1 above, it suffices to show that $\gamma(\delta(M)) = M$, which holds, since $\gamma \circ \delta = id_M$, by Theorem 3.9.

Now, for concluding the proof of the theorem, we show that γ is injective. If $\gamma(e_1) = \gamma(e_2)$ then $\delta(\gamma(e_1)) = \delta(\gamma(e_2))$ and thus, since $\delta \circ \gamma = id_e$ by Theorem 3.8, $e_1 = e_2$.

δ could also be proved to be a bijection between the two sets of proof-terms mentioned in this theorem, by using similar arguments to those used above and Theorem 3.10. □

Now, we turn our attention to the definition of the calculus NN. This calculus is a restriction of NJ^{pt} to hereditary Harrop logic, where only expanded normal deductions are allowed and the notions of well-formedness are encoded by means of derivable judgements. As for MM, the classes of objects of NN are the same as those of hH^{ULF}. They are defined by the same grammars, except for proof-terms. The proof-terms of NN are called *N-proof-terms*; they are defined as follows.

Definition 3.15 *N*-**Proof-Terms**

$$\begin{array}{lll} N & ::= & (N,N) \mid i(N) \mid j(N) \mid \lambda x.N \mid (t,N) \mid \lambda_q x.N \mid a; \\ a & ::= & x \mid fst(a) \mid snd(a) \mid app(a,N) \mid app_q(a,t). \end{array}$$

Note that the class of NN-proof-terms is a restriction of normal proof-terms of NJ^{pt}.

As for MM, the forms of judgement of NN are the same as those of hH^{ULF}, except for sequents, which are replaced by the following two forms of judgement, that we also call *sequents*:

$$\begin{array}{ll} \text{(i)} & \Sigma;\Delta \triangleright\triangleright N : G; \\ \text{(ii)} & \Sigma;\Delta \triangleright a : A. \end{array}$$

NN-judgements common to hH^{ULF} are derivable iff they are derivable in hH^{ULF}. The rules defining the derivable sequents of NN are shown in Fig. 3.8. As compared to NJ^{pt}, NN restricts the form of deductions allowed. Firstly, it allows only normal deductions; secondly, these normal deductions must be expanded. If the rule *change* would allow arbitrary H-formulae in the succedent, instead of allowing only atomic formulae, then NN would capture all normal deductions and not only those which are expanded. Roughly, the restriction on expanded normal forms reflects the restriction on the use of axioms in hH, which require the main formula to be atomic. Derivations in NN are called *expanded normal deductions.*

Definitions 3.16 and 3.17, below, define mappings between M-proof-terms and N-proof-terms. These mappings are inverses of each other, as shown in Theorems 3.13 and 3.14. Further, these mappings preserve derivability, as shown in Theorems 3.15 and 3.16.

Definition 3.16 $\Theta : M \to N$ *and* $\Theta_1 : a \times Ms \to N$

$$\begin{array}{l} \Theta(par(M_1,M_2)) =_{def} (\Theta(M_1),\Theta(M_2)) \\ \Theta(il(M)) =_{def} i(\Theta(M)) \\ \Theta(ir(M)) =_{def} j(\Theta(M)) \\ \Theta(lamb(x.M)) =_{def} \lambda x.\Theta(M) \\ \Theta(par_q(t,M)) =_{def} (t,\Theta(M)) \\ \Theta(lamb_q(x.M)) =_{def} \lambda_q x.\Theta(M) \\ \Theta((x;Ms)) =_{def} \Theta_1(x,Ms) \end{array}$$

$$\begin{array}{l} \Theta_1(a,[]) =_{def} a \\ \Theta_1(a,[M|Ms]) =_{def} \Theta_1(app(a,\Theta(M)),Ms) \\ \Theta_1(a,fst(Ms)) =_{def} \Theta_1(fst(a),Ms) \\ \Theta_1(a,snd(Ms)) =_{def} \Theta_1(snd(a),Ms) \\ \Theta_1(a,appl_q(t,Ms)) =_{def} \Theta_1(app_q(a,t),Ms) \end{array}$$

Definition 3.17 $\Psi : N \to M$ **and** $\Psi_1 : a \times Ms \to M$

$$\begin{aligned}
&\Psi((N_1, N_2)) =_{def} par(\Psi(N_1), \Psi(N_2))\\
&\Psi(i(N)) =_{def} il(\Psi(N))\\
&\Psi(j(N)) =_{def} ir(\Psi(N))\\
&\Psi(\lambda x.N) =_{def} lamb(x.\Psi(N))\\
&\Psi((t, N)) =_{def} par_q(t, \Psi(N))\\
&\Psi(\lambda_q x.N) =_{def} lamb_q(x.\Psi(N))\\
&\Psi(a) =_{def} \Psi_1(a, [])
\end{aligned}$$

$$\begin{aligned}
&\Psi_1(x, Ms) =_{def} (x; Ms)\\
&\Psi_1(fst(a), Ms) =_{def} \Psi_1(a, fst(Ms))\\
&\Psi_1(snd(a), Ms) =_{def} \Psi_1(a, snd(Ms))\\
&\Psi_1(app(a, N), Ms) =_{def} \Psi_1(a, [\Psi(N)|Ms])\\
&\Psi_1(app_q(a, t), Ms) =_{def} \Psi_1(a, appl_q(t, Ms))
\end{aligned}$$

Theorem 3.13 *The following identities hold:*

$$\begin{aligned}
&\text{(i)} \quad \Theta\circ\Psi = id_N;\\
&\text{(ii)} \quad \Psi\circ\Theta_1 = \Psi_1.
\end{aligned}$$

Proof: The proof follows by simultaneous induction on the structures of the argument and second argument, respectively. See [DP96b], for proof in the general case (not restricted to hH) of full intuitionistic first-order logic. □

Theorem 3.14 *The following identities hold:*

$$\begin{aligned}
&\text{(i)} \quad \Psi\circ\Theta = id_M;\\
&\text{(ii)} \quad \Theta\circ\Psi_1 = \Theta_1.
\end{aligned}$$

Proof: The proof follows by simultaneous induction on the structures of the argument and second argument, respectively. See [DP96b], for proof in the general case of full intuitionistic first-order logic. □

Theorem 3.15

(1) *If* $\Sigma; \Delta \Longrightarrow M : G$ *is derivable then* $\Sigma; \Delta \triangleright \triangleright \Theta(M) : G$ *is derivable.*

(2) *If* $\Sigma; \Delta \overset{H}{\Longrightarrow} Ms : A$ *is derivable and* $\Sigma; \Delta \triangleright a : H$ *then* $\Sigma; \Delta \triangleright \triangleright \Theta_1(a, Ms) : A$ *is derivable.*

$$\frac{\Sigma; \Delta \triangleright\triangleright N_1 : G_1 \quad \Sigma; \Delta \triangleright\triangleright N_2 : G_2}{\Sigma; \Delta \triangleright\triangleright (N_1, N_2) : G_1 \wedge G_2} \wedge I$$

$$\frac{\Sigma; \Delta \triangleright\triangleright N : G_1 \quad \Sigma \vdash G_2 \; gf}{\Sigma; \Delta \triangleright\triangleright i(N) : G_1 \vee G_2} \vee_l I \qquad\qquad \frac{\Sigma; \Delta \triangleright\triangleright N : G_2 \quad \Sigma \vdash G_1 \; gf}{\Sigma; \Delta \triangleright\triangleright j(N) : G_1 \vee G_2} \vee_r I$$

$$\frac{\Sigma; \Delta, x : H \triangleright\triangleright N : G}{\Sigma; \Delta \triangleright\triangleright \lambda x.N : H \supset G} \supset I, \; x \notin \Delta$$

$$\frac{\Sigma; \Delta \triangleright\triangleright N : [t/x]G \quad \Sigma \vdash t : \tau}{\Sigma; \Delta \triangleright\triangleright (t, N) : \exists_{x:\tau} G} \exists I$$

$$\frac{\Sigma, x : \tau; \Delta \triangleright\triangleright N : G \quad \vdash \Sigma; \Delta \; basis}{\Sigma; \Delta \triangleright\triangleright \lambda_q x.N : \forall_{x:\tau} G} \forall I, \;\; x \notin \Sigma$$

$$\frac{\Sigma; \Delta \triangleright a : A}{\Sigma; \Delta \triangleright\triangleright a : A} \; change$$

$$\frac{\vdash \Sigma; \Delta, x : H \; basis}{\Sigma; \Delta, x : H \triangleright x : H} \; axiom$$

$$\frac{\Sigma; \Delta \triangleright a : H_1 \wedge H_2}{\Sigma; \Delta \triangleright fst(a) : H_1} \wedge_l E \qquad \frac{\Sigma; \Delta \triangleright a : H_1 \wedge H_2}{\Sigma; \Delta \triangleright snd(a) : H_2} \wedge_r E$$

$$\frac{\Sigma; \Delta \triangleright a : G \supset H \quad \Sigma; \Delta \triangleright\triangleright N : G}{\Sigma; \Delta \triangleright app(a, N) : H} \supset E$$

$$\frac{\Sigma; \Delta \triangleright a : \forall_{x:\tau} H \quad \Sigma \vdash t : \tau}{\Sigma; \Delta \triangleright app_q(a, t) : [t/x]H} \forall E$$

Figure 3.8: Rules for derivable sequents of NN.

Proof: The proof follows by simultaneous induction on the structures of M and Ms, respectively. See [DP96b], for proof in the general case of full intuitionistic first-order logic. □

Theorem 3.16

(1) *If $\Sigma; \Delta \rhd\rhd N : G$ is derivable then $\Sigma; \Delta \longrightarrow \Psi(N) : G$ is derivable.*

(2) *If $\Sigma; \Delta \xRightarrow{H} Ms : A$ is derivable and $\Sigma; \Delta \rhd a : H$ then $\Sigma; \Delta \longrightarrow \Psi_1(a, Ms) : A$ is derivable.*

Proof: The proof follows by simultaneous induction on the structures of N and a, respectively. See [DP96b], for proof in the general case of full intuitionistic first-order logic. □

The following result establishes a 1-1 correspondence between forms of deriving a goal w.r.t. a basis in MM and NN.

Theorem 3.17 *The set of proof-terms M s.t. $\Sigma; \Delta \Longrightarrow M : G$ is derivable in MM is in 1-1 correspondence with the set of proof-terms N s.t. $\Sigma; \Delta \rhd\rhd N : G$ is derivable in NN.*

Proof: We show that Θ is a bijection between the two sets of proof-terms. First, we show that for every N s.t. $\Sigma; \Delta \rhd\rhd N : G$ is derivable in NN there exists M_1 s.t. $\Sigma; \Delta \Longrightarrow M_1 : G$ is derivable in MM and $\Theta(M_1) = N$, thus showing the surjectivity of Θ.

By Theorem 3.16, if $\Sigma; \Delta \rhd\rhd N : G$ is derivable in NN then $\Sigma; \Delta \Longrightarrow \Psi(N) : G$ is derivable in MM. For showing that $\Psi(N)$ satisfies the conditions on M_1 above, it suffices to show that $\Theta(\Psi(N)) = N$, which holds, since $\Theta \circ \Psi = id_N$ by Theorem 3.13.

Now, for concluding the proof of the theorem, we show that Θ is injective. If $\Theta(M_1) = \Theta(M_2)$ then $\Psi(\Theta(M_1)) = \Psi(\Theta(M_2))$ and thus, since $\Psi \circ \Theta = id_M$ by Theorem 3.14, $M_1 = M_2$.

Ψ could also be shown to be a bijection between the two sets of proof-terms of the theorem, by using similar arguments to those used above and Theorem 3.15. □

Now, we come to the main result of this section, i.e. uniform linear focused derivations are in a 1-1 correspondence to expanded normal deductions.

Theorem 3.18 *The set of proof-terms e s.t. $\Sigma; \Delta \longrightarrow e : G$ is derivable in hH^{ULF} is 1-1 correspondence with the set of proof-terms N s.t. $\Sigma; \Delta \rhd\rhd N : G$ is derivable in NN.*

Proof: By combining Theorems 3.12 and 3.17. □

We show in Proposition 3.4 below that the mapping ϕ, restricted to uniform linear focused proof-terms, is the same as $\Theta \circ \gamma$, thus itself a bijection between uniform linear focused derivations and expanded normal deductions. So, no two distinct uniform linear focused proof-terms may have the same image under ϕ.

Proposition 3.4 *The following identities hold:*

(i) $\phi(e) = \Theta(\gamma(e))$;

(ii) $[a/x]\phi(a_1^x) = \Theta_1(a, \gamma_1(a_1^x))$, *if x occurs freely in a_1^x exactly once*

Proof: By simultaneous induction on the structures of e and a_1^x.

(i) Case $e = apply(x_1, e_1, x_2.a_2^{x_2})$.

$$\begin{array}{lll} & \phi(apply(x_1, e_1, x_2.a_2^{x_2})) & \\ = & [app(x_1, \phi(e_1))/x_2]\phi(a_2^{x_2}) & \text{by def. of } \phi \\ = & \Theta_1(app(x_1, \phi(e_1)), \gamma_1(a_2^{x_2})) & \text{by I.H., since } x_2 \text{ occurs freely in } a_2^{x_2} \text{ exactly once} \\ = & \Theta_1(app(x_1, \Theta(\gamma(e_1))), \gamma_1(a_2^{x_2})) & \text{by I.H.} \\ = & \Theta_1(x_1, [\gamma(e_1)|\gamma_1(a_2^{x_2})]) & \text{by def. of } \Theta_1 \\ = & \Theta((x_1; [\gamma(e_1)|\gamma_1(a_2^{x_2})])) & \text{by def. of } \Theta \\ = & \Theta(\gamma(apply(x_1, e_1, x_2.a_2^{x_2}))) & \text{by def. of } \gamma \end{array}$$

(ii) Case $a_1^x = apply(x, e_1, x_1.a_2^{x_1})$.

$$\begin{array}{lll} & [a/x]\phi(apply(x, e_1, x_1.a_2^{x_1})) & \\ = & [a/x]([app(x, \phi(e_1))/x_1]\phi(a_2^{x_1})) & \text{by def. of } \phi \\ = & ([app(a, \phi(e_1))/x_1]\phi(a_2^{x_1})) & x \notin \phi(e_1) \text{ and } x \notin \phi(a_2^{x_1}) \\ = & \Theta_1(app(a, \phi(e_1)), \gamma_1(a_2^{x_1})) & \text{by I.H., since } x_1 \text{ occurs freely in } a_2^{x_1} \text{ exactly once} \\ = & \Theta_1(app(a, \Theta(\gamma(e_1))), \gamma_1(a_2^{x_1})) & \text{by I.H.} \\ = & \Theta_1(a, [\gamma(e_1)|\gamma_1(a_2^{x_1})]) & \text{by def. of } \Theta_1 \\ = & \Theta_1(a, \gamma(apply(x, e_1, x_1.a_2^{x_1}))) & \text{by def. of } \gamma \end{array}$$

Similar arguments apply in the other cases. □

We are now in conditions to give a simple argument for the uniqueness of a uniform linear focused form of a proof-term and a method for its calculation.

Theorem 3.19 *Every proof-term e which is not uniform linear focused is reducible by RS_{ulf} to a unique uniform linear focused proof-term.*

Proof: Let e be a proof-term which is not uniform linear focused. Then, by Theorem 3.4, e is reducible by RS_{ulf} to a uniform linear focused proof-term e_{ulf}. Now, let us suppose that e is also reducible by RS_{ulf} to a uniform linear focused proof-term e_{ulf_1}. Thus, by Theorem 2.9,

$$\phi(e_{ulf}) = \phi(e) = \phi(e_{ulf_1}).$$

So, by Proposition 3.4 and Theorems 3.9 and 3.14,

$$e_{ulf} = \delta(\Psi(\phi(e_{ulf}))) = \delta(\Psi(\phi(e_{ulf_1}))) = e_{ulf_1}.$$

□

Below the notation $ulf(e)$ is used to represent the uniform linear focused form of the proof-term e. By the theorem above, $ulf(e) = \delta(\Psi(\phi(e)))$.

3.6 Proof-Theoretic Semantics of FOPLP

This section defines a proof-theoretic semantics for the pure logic programming language FOPLP. In Sec. 3.2, we have already defined the notions of *program*, *goal* and goals *achievable* w.r.t. programs in FOPLP, which are recalled below.

- A *program* in FOPLP is a pair $\langle \Sigma, \Delta \rangle$, usually written $\Sigma; \Delta$, where Σ is a signature and Δ is a hH-context; a program $\Sigma; \Delta$ is *well-formed* iff the judgement $\vdash \Sigma; \Delta$ *basis* is derivable in hH.

- A *goal* in FOPLP is a G-formula; a goal G is *well-formed* w.r.t. the program $\Sigma; \Delta$ iff the judgement $\Sigma \vdash G\ gf$ is derivable in hH.

- A goal G is *achievable* w.r.t. a program $\Sigma; \Delta$ in FOPLP iff there exists a proof-term e s.t. $\Sigma; \Delta \Rightarrow e : G$ is derivable in hH; the proof-term e is called a *witness* for the achievement of G w.r.t. $\Sigma; \Delta$.

So, to complete the definition of a semantics for FOPLP, as argued in Sec. 3.2, we must define what are the different means of goal-achievement. In Sec. 3.4 is shown that uniform linear focused derivations of hH are in 1-1 correspondence with derivations in hH^{ULF}. Theorem 3.18 establishes a 1-1 correspondence between derivations in $hH^{ULF}$and expanded normal deductions. So, uniform linear focused derivations in hH are in 1-1 correspondence to expanded normal deductions. This 1-1 correspondence justifies our choice of the different means of achieving goals in FOPLP. We define the different means of achieving a goal G w.r.t. a program $\Sigma; \Delta$ in FOPLP as the uniform linear focused proof-terms e_{ulf} s.t. $\Sigma; \Delta \Rightarrow e_{ulf} : G$ is derivable in hH. So, witnesses which have the same image under ϕ, *i.e.* witnesses that correspond to the same expanded normal deduction, are regarded as the same means of achieving goals. This choice of the means of goal-achievement in FOPLP gives an immediate interpretation of FOPLP by means of the natural deduction system for first-order hereditary Harrop logic NN.

An *implementation* of FOPLP is any method that given a goal G and a program $\Sigma; \Delta$ enumerates the means for the achievement of G w.r.t. $\Sigma; \Delta$. Alternatively, an implementation of FOPLP may be defined as a method that given a goal and a program finds all expanded normal deductions of the goal w.r.t. the program in NN. Section 3.7 sketches a method for implementing FOPLP, by describing a method to find all derivations of a goal w.r.t. a program in hH^{ULF}, i.e. all uniform linear focused derivations of hH.

3.7 Towards an Implementation of FOPLP

According to Sec. 3.6, an implementation of FOPLP is a method that, for every goal G and program $\Sigma; \Delta$, finds every uniform linear focused proof-term e s.t. the sequent $\Sigma; \Delta \Rightarrow e : G$ is derivable in hH ; or, in other words, a method that finds every proof-term e s.t. the sequent $\Sigma; \Delta \longrightarrow e : G$ is derivable in hH^{ULF}.

This section outlines a method to search for proof-terms e s.t. $\Sigma; \Delta \longrightarrow e : G$ is derivable in hH^{ULF}. This procedure is described as a predicate of the form

$$search(G, \Sigma; \Delta, \Theta_{in}, \Theta_{out}, e, V_{in}, V_{out}),$$

where:

- $\vdash \Sigma; \Delta \; basis$ and $\Sigma \vdash G \; gf$ are derivable in hH;
- V_{in} and V_{out} are signatures s.t. $V_{in} \subseteq V_{out}$ and there is no x s.t. $x \in \Sigma$ and $x \in V_{out}$;
- Θ_{in} and Θ_{out} are substitutions s.t.: $\Theta_{out} = \Theta \circ \Theta_{in}$ for some Θ; for every $x \notin V_{out}$, $\Theta_{out}(x) = x$ and, for every $x : \tau \in V_{out}$, $\Sigma, V_{out} \vdash \Theta_{out}(x) : \tau$ is derivable.

Validity of the predicate *search* is defined by means of a collection of Horn clauses, it depends upon validity of the auxiliary predicate *search*1, which has the form:

$$search1(x : H, A, \Sigma; \Delta, \Theta_{in}, \Theta_{out}, e, V_{in}, V_{out}).$$

The notation $\Theta(\Delta)$ stands for the context obtained from Δ by replacing each element $x : H$ of Δ by $x : H_1$, where H_1 is the result of replacing the free occurrences of variables in H by their images under Θ. The notations $\Theta(G)$ and $\Theta(e)$ stand, respectively, for the result of replacing the free occurrences of variables in G and e by their images under Θ. The predicate *search* is such that:

- if $search(G, \Sigma; \Delta, \Theta_{in}, \Theta_{out}, e, V_{in}, V_{out})$ holds, then the sequent

  $$\Sigma, V_{out}; \Theta_{out}(\Delta) \longrightarrow \Theta_{out}(e) : \Theta_{out}(G)$$

  is derivable in hH^{ULF};
- if the sequent $\Sigma, V_{in}; \Theta_{in}(\Delta) \longrightarrow \Theta_{in}(e) : \Theta_{in}(G)$ is derivable in hH^{ULF} then there exists V_{out}, Θ_{out} and e_1 s.t. $\Theta_{out}(e_1) = \Theta_{in}(e)$ and $search(G, \Sigma; \Delta, \Theta_{in}, \Theta_{out}, e_1, V_{in}, V_{out})$, holds; in particular, when $V_{in} = \emptyset$ and $\Theta_{in} = identity$.

Recall that formulae and proof-terms are equal up to renaming of bound variables. In the definition of *search* below some constraints are satisfied simply by the renaming of bound variables.

The definition of *search* is by cases on the structure of G as shown below, in other words, a search for a derivation of G w.r.t. a program is guided by G.

- Case $G = G_1 \wedge G_2$, then a derivation of $\Sigma; \Delta \longrightarrow e : G$ must have as last step $\longrightarrow \wedge$. The clause corresponding to this case is:

$$
\begin{array}{l}
search(G_1 \wedge G_2, \Sigma; \Delta, \Theta_{in}, \Theta_{out}, pair(e_1, e_2), V_{in}, V_{out}) \text{ if} \\
\qquad search(G_1, \Sigma; \Delta, \Theta_{in}, \Theta, e_1, V_{in}, V) \text{ and} \\
\qquad search(G_2, \Sigma; \Delta, \Theta, \Theta_{out}, e_2, V, V_{out}).
\end{array}
$$

- Case $G = G_1 \vee G_2$, then a derivation of $\Sigma; \Delta \longrightarrow e : G$ must have as last step either rule $\longrightarrow \vee_l$ or rule $\longrightarrow \vee_r$. This case introduces two clauses in the definition of *search*:

$$
\begin{array}{l}
search(G_1 \vee G_2, \Sigma; \Delta, \Theta_{in}, \Theta_{out}, inl(e), V_{in}, V_{out}) \text{ if} \\
\qquad search(G_1, \Sigma; \Delta, \Theta_{in}, \Theta_{out}, e, V_{in}, V_{out}); \\
search(G_1 \vee G_2, \Sigma; \Delta, \Theta_{in}, \Theta_{out}, inr(e), V_{in}, V_{out}) \text{ if} \\
\qquad search(G_2, \Sigma; \Delta, \Theta_{in}, \Theta_{out}, e, V_{in}, V_{out}).
\end{array}
$$

Note that to find all derivations of $\Sigma; \Delta \longrightarrow e : G_1 \vee G_2$ both clauses must be considered, although there is a choice of which clause to consider first.

- Case $G = H \supset G_1$, then a derivation of $\Sigma; \Delta \longrightarrow e : G$ must have as last step a rule $\longrightarrow\supset$. The corresponding clause is:

$$
\begin{array}{l}
search(H \supset G_1, \Sigma; \Delta, \Theta_{in}, \Theta_{out}, lambda(x.e), V_{in}, V_{out}) \text{ if} \\
\qquad search(G_1, \Sigma; (\Delta, x : H), \Theta_{in}, \Theta_{out}, e, V_{in}, V_{out}) \text{ and} \\
\qquad x \notin \Delta.
\end{array}
$$

Recall that $x \notin \Delta$ means that there exists no H s.t. $x : H \in \Delta$.

- Case $G = \exists_{x:\tau} G_1$, then a derivation of $\Sigma; \Delta \longrightarrow e : G$ must have as last step rule $\longrightarrow \exists$. However, there is a choice of which term t of type τ to consider. In order to find all proof-terms e s.t. $\Sigma; \Delta \longrightarrow e : \exists_{x:\tau} G_1$ is derivable, all terms t of type τ must be considered. This problem may be solved by using unification of first-order terms. The variable x is treated as a *logical variable.* When attempting to form an axiom, a logical variable may be replaced by a term having the same type to make atomic formulae equal. The clause corresponding to this case is:

$$
\begin{array}{l}
search(\exists_{x:\tau} G_1, \Sigma; \Delta, \Theta_{in}, \Theta_{out}, pair_q(x, e), V_{in}, V_{out}) \text{ if} \\
\qquad search(G_1, \Sigma; \Delta, \Theta_{in}, \Theta_{out}, e, V_{in} \cup \{x : \tau\}, V_{out}) \text{ and} \\
\qquad x \notin V_{in} \text{ and } x \notin \Sigma.
\end{array}
$$

Recall that, $x \notin V_{in}$ ($x \notin \Sigma$) means that there is no τ s.t. $x : \tau$ is an element of V_{in} (Σ).

- Case $G = \forall_{x:\tau} G_1$, then a derivation of $\Sigma; \Delta \longrightarrow e : G$ must have as last step rule $\longrightarrow \forall$. The corresponding clause is:

$$\begin{array}{l} search(\forall_{x:\tau}G_1, \Sigma; \Delta, \Theta_{in}, \Theta_{out}, lambda_q(x.e), V_{in}, V_{out}) \text{ if} \\ \quad search(G_1, \Sigma \cup \{x : \tau\}; \Delta, \Theta_{in}, \Theta_{out}, e, V_{in}, V_{out}) \text{ and} \\ \quad x \notin V_{in} \text{ and } x \notin \Sigma. \end{array}$$

- Case $G = A$, then a derivation of $\Sigma; \Delta \longrightarrow e : G$ must have as last step rule a rule *choice*. The application of this rule involves the choice of a formula $x : H$ from the program $\Sigma; \Delta$. The problem of finding proof-terms e s.t. $\Sigma; \Delta \longrightarrow e : A$ is derivable depends upon the problem of finding proof-terms e s.t. $\Sigma; \Delta \xrightarrow{x:H} e : A$ is derivable. In order to find all proof-terms e s.t. $\Sigma; \Delta \xrightarrow{x:H} e : A$ is derivable, all $x : H$ in Δ must be considered. (Still, there is the choice of which order to follow in attempting formulae from $\Sigma; \Delta$.) The clause of *search* corresponding to this case is:

$$\begin{array}{l} search(A, \Sigma; \Delta, \Theta_{in}, \Theta_{out}, e, V_{in}, V_{out}) \text{ if} \\ \quad choice(x : H, \Delta) \text{ and} \\ \quad search1(x : H, A, \Sigma; \Delta, \Theta_{in}, \Theta_{out}, e, V_{in}, V_{out}). \end{array}$$

  A formula $choice(x : H, \Delta)$ holds iff $x : H$ is an element of Δ.

The definition of *search1* is by cases on the structure of the selected formula, in other words, a search for a derivation of an atomic goal is guided by the structure of the selected formula from the program.

- Case $H = H_1 \wedge H_2$, the last step of a derivation of $\Sigma; \Delta \xrightarrow{x:H} e : A$ is either $\xrightarrow{\wedge_l}$ or $\xrightarrow{\wedge_r}$. There are the following two clauses associated to this case:

$$\begin{array}{l} search1(x : H_1 \wedge H_2, A, \Sigma; \Delta, \Theta_{in}, \Theta_{out}, splitl(x, x_1.e), V_{in}, V_{out}) \text{ if} \\ \quad search1(x_1 : H_1, A, \Sigma; \Delta, \Theta_{in}, \Theta_{out}, e, V_{in}, V_{out}) \text{ and} \\ \quad x_1 \notin \Delta. \end{array}$$

$$\begin{array}{l} search1(x : H_1 \wedge H_2, A, \Sigma; \Delta, \Theta_{in}, \Theta_{out}, splitr(x, x_1.e), V_{in}, V_{out}) \text{ if} \\ \quad search1(x_1 : H_2, A, \Sigma; \Delta, \Theta_{in}, \Theta_{out}, e, V_{in}, V_{out}) \text{ and} \\ \quad x_1 \notin \Delta. \end{array}$$

  Note that to find every proof-term e, s.t.$\Sigma; \Delta \xrightarrow{x:H} e : A$ is derivable, both alternatives must be considered, but still having the choice of which alternative to consider first.

- Case $H = G \supset H_1$, then the last step of a derivation of $\Sigma; \Delta \xrightarrow{x:H} e : A$ must be a rule $\xrightarrow{\supset}$. Note that the left premise of $\xrightarrow{\supset}$ requires the search for proof-terms e_1 s.t. $\Sigma; \Delta \longrightarrow e_1 : G$ is derivable. The clause corresponding to this case is:

$$\begin{array}{l} search1(x : G \supset H_1, A, \Sigma; \Delta, \Theta_{in}, \Theta_{out}, apply(x, e, x_1.e_1), V_{in}, V_{out}) \text{ if} \\ \quad search(G, \Sigma; \Delta, \Theta_{in}, \Theta, e, V_{in}, V) \text{ and} \\ \quad search1(x_1 : H_1, A, \Sigma; \Delta, \Theta, \Theta_{out}, e_1, V, V_{out}) \text{ and } x_1 \notin \Delta. \end{array}$$

- Case $H = \forall_{x_1:\tau} H_1$, then the last step of a derivation of $\Sigma; \Delta \xrightarrow{x:H} e : A$ must be a rule $\xrightarrow{\forall}$. However, there is the choice of which term t of type τ to choose. As for the case where the goal is existentially quantified, unification may be used to solve this problem. The clause corresponding to this case is:

$$\begin{aligned}&search1(x : \forall_{x_1:\tau} H_1, A, \Sigma; \Delta, \Theta_{in}, \Theta_{out}, apply_q(x, x_1, x_2.e), V_{in}, V_{out}) \text{ if}\\ &\quad search1(x_2 : H_1, A, \Sigma; \Delta, \Theta_{in}, \Theta_{out}, e, V_{in} \cup \{x_1 : \tau\}, V_{out}) \text{ and}\\ &\quad x_1 \notin V_{in} \text{ and } x_1 \notin \Sigma.\end{aligned}$$

- Case $H = A_1$, then the last step of a derivation of $\Sigma; \Delta \xrightarrow{x:H} e : A$ must be an axiom. Recall that for forming an axiom it suffices that A_1 and A are unifiable. The clause corresponding to this case is:

$$\begin{aligned}&search1(x : A_1, A, \Sigma; \Delta, \Theta_{in}, \Theta_{out}, x, V_{in}, V_{out}) \text{ if}\\ &\quad unify(A_1, A, \Theta_{in}, \Theta_{out}, V_{in}, V_{out}, \Sigma).\end{aligned}$$

A formula $unify(A_1, A_2, \Theta_{in}, \Theta_{out}, V_{in}, V_{out}, \Sigma)$ holds iff $A_1 = pt_1...t_n$, $A = pt'_1...t'_n$, S is the set consisting of the pairs $\langle t_i, t'_i \rangle$, for $1 \leq i \leq n$, and the formula

$$unify(S, \Theta_{in}, \Theta_{out}, V_{in}, V_{out}, \Sigma),$$

whose meaning is defined in Sec. 2.2, holds.

Summarising, this section describes a method to search for means of goal-achievement in FOPLP. If the goal is compound, the goal is broken up, according to the rule corresponding to the outermost connective of the goal. If the goal is atomic, a formula from the program is selected and the structure of the selected formula determines how the search is to proceed. In order to find all the means of achieving a goal G w.r.t. a program $\Sigma; \Delta$ in FOPLP, it suffices to consider exhaustively all alternative "choices". The "choices" are as follows:

- case G is of the form $G_1 \vee G_2$ there is a "choice" of which of the rules $\longrightarrow \vee_l$, $\longrightarrow \vee_r$ to attempt;
- case G is of the form $\exists_{x:\tau} G_1$ there is a "choice" of which term $t : \tau$ to attempt;
- case G is atomic there is a "choice" of which formula $x : H$ of Δ to select, since there may be several formulae in Δ that may lead to derivations; once a formula $x : H$ of Δ has been selected, there are the following "choices":
  - case H is of the form $\forall_{x:\tau} H_1$ there is a "choice" of which term $t : \tau$ to attempt.
  - case H is of the form $H_1 \wedge H_2$ there is a "choice" of which of the rules $\longrightarrow \wedge_l$, $\longrightarrow \wedge_r$ to attempt.

Thus, any means of achieving a goal w.r.t. a program in FOPLP may be found by making appropriate "choices". This search procedure for FOPLP essentially corresponds to search procedures presented in [NM88, Nad93] that are used as bases for implementing the logic programming language λProlog.

3.8 Higher-Order Logic Programming

This section describes a higher-order logic programming language called HOPLP. The semantics of this language is defined by means of the proof theory of the calculus HH. HH is a formalisation of a higher-order hereditary Harrop logic; essentially, HH is a higher-order extension of hH, obtained by replacing the underlying set of first-order terms by the set of λ-terms. A calculus similar to HH is presented in [Mil90], except for the absence of proof-term annotations. This section presents the natural deduction system $NN^{\lambda norm}$, which is a formalisation of essentially the same higher-order hereditary Harrop logic formalised by HH, and describes an interpretation of HOPLP by means of $NN^{\lambda norm}$.

3.8.1 The Calculus HH for Higher-Order Hereditary Harrop Logic

The classes of objects of HH are the same as those of hH, except for the class t of first-order terms, which is replaced by the class Λ of λ-terms. The grammars defining the classes of objects of HH are shown in Fig. 3.9; roughly, they are obtained from those of hH by replacing Λ for t. After each grammar, in parentheses, is the intended meaning of each class of objects.

$$
\begin{array}{lcll}
\tau & ::= & s \mid (\tau \to \tau) & \text{(simple types)} \\
\Lambda & ::= & x \mid \lambda x : \tau.\Lambda \mid (\Lambda\Lambda) & (\lambda\text{-terms}) \\
\Sigma & ::= & \langle\rangle \mid \Sigma, x : \tau & \text{(signatures)} \\
A & ::= & p\Lambda\ldots\Lambda & \text{(atomic formulae)} \\
H & ::= & A \mid H \wedge H \mid G \supset H \mid \forall_{x:\tau} H & (H\text{-formulae}) \\
G & ::= & A \mid G \wedge G \mid G \vee G \mid H \supset G \mid \exists_{x:\tau} G \mid \forall_{x:\tau} G & (G\text{-formulae}) \\
\Delta & ::= & \langle\rangle \mid \Delta, x : H & \text{(contexts)} \\
e & ::= & pair(e,e) \mid inl(e) \mid inr(e) \mid lambda(x.e) & \\
 & \mid & pair_q(\Lambda, e) \mid lambda_q(x.e) & \\
 & \mid & x \mid splitl(x, x.e) \mid splitr(x, x.e) & \\
 & \mid & apply(x, e, x.e) \mid apply_q(x, \Lambda, x.e) & \text{(proof-terms)}
\end{array}
$$

s ranges over the set $\mathcal{S}$ of primitive types; x ranges over the set $\mathcal{X}$ of variables and p ranges over the set $\mathcal{P}$ of predicate symbols.

Figure 3.9: Classes of objects of HH.

The forms of judgement of HH are presented in Fig. 3.10. Each form of judgement (i) to

(i)	$\vdash \Sigma\ signature$	(signatures)
(ii)	$\Sigma \vdash \Lambda : \tau$	(terms of simple type)
(iii)	$\vdash \Sigma; \Delta\ basis$	(bases)
(iv)	$\Sigma \vdash A\ af$	(atomic formulae)
(v)	$\Sigma \vdash H\ hf$	(program formulae)
(vi)	$\Sigma \vdash G\ gf$	(goal formulae)
(vii)	$\Sigma; \Delta \Rightarrow e : G$	(proof-terms of a goal)
(viii)	$\Sigma \vdash \Lambda \rhd_\tau \Lambda$	(one step reduction of λ-terms)
(ix)	$\Sigma \vdash \Lambda \rhd_\tau^* \Lambda$	(zero or more steps reduction of λ-terms)
(x)	$\Sigma \vdash \Lambda \equiv_\tau \Lambda$	(convertibility of λ-terms)
(xi)	$\Sigma \vdash A \equiv A$	(equivalence of atomic formulae)
(xii)	$\Sigma \vdash H \equiv H$	(equivalence of program formulae)
(xiii)	$\Sigma \vdash G \equiv G$	(equivalence of goal formulae)

Figure 3.10: Judgement forms of HH.

(vii) has a corresponding form of judgement in hH, the only difference being that first-order terms are replaced by λ-terms. As in the calculus hH, judgements of the form $\Sigma; \Delta \Rightarrow e : G$ are called *sequents*. Sequents are the *main judgements* of HH; judgements of the other forms are called *auxiliary judgements*.

Observe that the forms of judgement (i), (ii), (viii), (ix) and (x) are common to the calculus λ^{ST}. The derivable judgements of these forms are those of λ^{ST}. The rules to define valid judgements of the forms (iii)-(vi) are obtained from the rules defining derivable judgements of the corresponding form in hH, replacing first-order terms by λ-terms. For example, the rules for derivable atomic formulae in HH are of the form:

$$\frac{\Sigma \vdash \Lambda_1 : \tau_1 \cdots \Sigma \vdash \Lambda_n : \tau_n}{\Sigma \vdash p\Lambda_1 ... \Lambda_n\ af},$$

where $p : \tau_1 \to ... \to \tau_n \to prop \in \mathcal{P}$. The rules defining derivable judgements of the forms (xi)-(xiii) are shown in Fig. 3.11.

Convertible λ-terms may be seen as alternative representations of the same object or, in other words, have the same denotation. So, in a logic based on λ-terms, having a term or another term convertible to it should be irrelevant, i.e. should not interfere with derivability of sequents. In the calculus HH this idea is captured by allowing the rules in Fig. 3.12 for deriving sequents, called *conversion rules.* The other rules for deriving sequents in HH are obtained from the rules for deriving sequents in hH, presented in Fig. 3.4, replacing first-order terms by λ-terms.

$$\frac{\Sigma \vdash \Lambda_1 \equiv_{\tau_1} \Lambda_1' \quad \ldots \quad \Sigma \vdash \Lambda_n \equiv_{\tau_n} \Lambda_n'}{\Sigma \vdash p\Lambda_1 \ldots \Lambda_n \equiv p\Lambda_1' \ldots \Lambda_n'}\ p : \tau_1 \to \ldots \to \tau_n \to prop \in \mathcal{P}$$

$$\frac{\Sigma \vdash H_1 \equiv H_3 \quad \Sigma \vdash H_2 \equiv H_4}{\Sigma \vdash H_1 \wedge H_2 \equiv H_3 \wedge H_4} \qquad \frac{\Sigma \vdash G_1 \equiv G_2 \quad \Sigma \vdash H_1 \equiv H_2}{\Sigma \vdash G_1 \supset H_1 \equiv G_2 \supset H_2}$$

$$\frac{\Sigma, x : \tau \vdash H_1 \equiv H_2}{\Sigma \vdash \forall_{x:\tau} H_1 \equiv \forall_{x:\tau} H_2}\ x \notin \Sigma$$

$$\frac{\Sigma \vdash G_1 \equiv G_3 \quad \Sigma \vdash G_2 \equiv G_4}{\Sigma \vdash G_1 \wedge G_2 \equiv G_3 \wedge G_4} \qquad \frac{\Sigma \vdash G_1 \equiv G_3 \quad \Sigma \vdash G_2 \equiv G_4}{\Sigma \vdash G_1 \vee G_2 \equiv G_3 \vee G_4}$$

$$\frac{\Sigma \vdash H_1 \equiv H_2 \quad \Sigma \vdash G_1 \equiv G_2}{\Sigma \vdash H_1 \supset G_1 \equiv H_2 \supset G_2} \qquad \frac{\Sigma, x : \tau \vdash G_1 \equiv G_2}{\Sigma \vdash \exists_{x:\tau} G_1 \equiv \exists_{x:\tau} G_2}\ x \notin \Sigma$$

$$\frac{\Sigma, x : \tau \vdash G_1 \equiv G_2}{\Sigma \vdash \forall_{x:\tau} G_1 \equiv \forall_{x:\tau} G_2}\ x \notin \Sigma$$

Figure 3.11: Rules defining convertible formulae.

$$\frac{\Sigma; \Delta, x : H_1 \Rightarrow e : G \quad \Sigma \vdash H_1 \equiv H}{\Sigma; \Delta, x : H \Rightarrow e : G}\ \equiv_l$$

$$\frac{\Sigma; \Delta \Rightarrow e : G_1 \quad \Sigma \vdash G_1 \equiv G}{\Sigma; \Delta \Rightarrow e : G}\ \equiv_r$$

Figure 3.12: Conversion rules of HH.

As for hH, the *principal part* of a derivation of a sequent is the tree obtained by erasing all derivations of auxiliary judgements.

In hH the proof-term of the endsequent of a derivation represents uniquely the principal part of such derivation. However, in HH this is no longer the case. Notice that the proof-terms of the sequent premiss and conclusion of a conversion rule are the same. Thus, the information contained in a proof-term of a HH-derivation is not enough to recover uniquely its principal part. Figure 3.13 shows two distinct derivations whose proof-terms are the same. Different derivations whose proof-terms are the same have a similar structure of logical rules; they differ at most in the places where conversion rules occur in the derivations.

Below is introduced the calculus HH°, which may be thought of as a calculus obtained from HH by constraining the use of conversion rules, so that conversion rules are only allowed to form axioms. For HH°, one may easily show that: the principal part of a derivation of a sequent is unique, or in other words, the proof-term of a sequent represents uniquely the principal part of a derivation of that sequent. The calculus HH° is used in Sec. 3.8.2 for defining the different

$$\dfrac{\dfrac{\overset{\pi_2}{\Sigma;\Delta \Rightarrow e_1 : [\Lambda/x]G_2} \quad \Sigma \vdash \Lambda : \tau}{\Sigma;\Delta \Rightarrow pair_q(\Lambda, e_1) : \exists_{x:\tau} G_2} \Rightarrow\exists \quad \dfrac{\Sigma, x : \tau \vdash G_2 \equiv G_1}{\Sigma \vdash \exists_{x:\tau} G_2 \equiv \exists_{x:\tau} G_1}}{\Sigma;\Delta \Rightarrow pair_q(\Lambda, e_1) : \exists_{x:\tau} G_1} \equiv_r \quad (*)$$

$$\dfrac{\dfrac{\overset{\pi_2}{\Sigma;\Delta \Rightarrow e_1 : [\Lambda/x]G_2} \quad \Sigma \vdash [\Lambda/x]G_2 \equiv [\Lambda/x]G_1}{\Sigma;\Delta \Rightarrow e_1 : [\Lambda/x]G_1} \equiv_r \quad \Sigma \vdash \Lambda : \tau}{\Sigma;\Delta \Rightarrow pair_q(\Lambda, e_1) : \exists_{x:\tau} G_1} \Rightarrow\exists \quad (**)$$

Figure 3.13: Two HH-derivations whose proof-terms are the same.

means of goal-achievement in HOPLP.

The calculus HH° is defined as HH except that axioms and conversion rules allowed in HH are replaced by the following new form of axiom:

$$\dfrac{\vdash \Sigma;\Delta, x : A_1\ basis \quad \Sigma \vdash A_1 \equiv A_2}{\Sigma;\Delta, x : A_1 \Rightarrow x : A_2}\ axiom - conv.$$

Theorems 3.20 and 3.21 show that a sequent is derivable in HH iff it is derivable in HH°.

Theorem 3.20 *Every sequent derivable in HH° is derivable in HH.*

Proof: Note that $axiom - conv$ is a derivable rule of HH, it suffices to combine an axiom with the appropriate conversion rule. □

Theorem 3.21 *Every sequent derivable in HH is derivable in HH°.*

Proof: Firstly, we prove that every HH-derivation π of a sequent $\Sigma;\Delta \Rightarrow e : G$ may be transformed into a derivation whose conversion rules occur only immediately below axioms or other conversion rules. The proof follows by induction on the structure of π. The interesting case is where the last step of π is a conversion rule. (The other cases follow easily by the I.H..) Let π_1 be the derivation of the sequent premiss. The proof follows by induction on the maximum number of *logical rules*[4] in a branch of π_1. The interesting cases are those where the last step of π_1 is a logical rule. Case the last step of π_1 is $\Rightarrow\supset$ and the last step of π is $\equiv_r$, the transformation on π below may be applied.

[4] A sequent rule of HH is called a *logical rule* if it is not a conversion rule.

$$\dfrac{\dfrac{\overset{\pi_2}{\Sigma;\Delta,x:H_2\Rightarrow e_1:G_2}}{\Sigma;\Delta\Rightarrow lambda(x.e_1):H_2\supset G_2}\Rightarrow\supset \quad \dfrac{\overset{\rho_1}{\Sigma\vdash H_2\equiv H_1}\quad \overset{\rho_2}{\Sigma\vdash G_2\equiv G_1}}{\Sigma\vdash H_2\supset G_2\equiv H_1\supset G_1}}{\Sigma;\Delta\Rightarrow lambda(x.e_1):H_1\supset G_1}\equiv_r$$

$$\downarrow$$

$$\dfrac{\dfrac{\dfrac{\overset{\pi_2}{\Sigma;\Delta,x:H_2\Rightarrow e_1:G_2}\quad \overset{\rho_2}{\Sigma\vdash G_2\equiv G_1}}{\Sigma;\Delta,x:H_2\Rightarrow e_1:G_1}\equiv_r \quad \overset{\rho_1}{\Sigma\vdash H_2\equiv H_1}}{\Sigma;\Delta,x:H_1\Rightarrow e_1:G_1}\equiv_l}{\Sigma;\Delta\Rightarrow lambda(x.e_1):H_1\supset G_1}\Rightarrow\supset$$

Note that, although an extra conversion step is introduced in the latter derivation of the endsequent, the maximum number of logical rules in a derivation of a sequent premiss of a conversion rule has been reduced by one. (The transformation corresponding to the case where the last step of π_1 is $\Rightarrow\exists$ corresponds to the transformation of derivation $(*)$ into derivation $(**)$ of Fig. 3.13.)

Other cases follow by using similar transformations on derivations, permuting conversion rules above logical rules.

Secondly, concluding the proof of the theorem, we show that a sequent $\Sigma;\Delta\Rightarrow e:G$ having a HH-derivation π, whose conversion rules are either immediately below axioms or other conversion rules, is derivable in HH°. The proof follows by induction on the structure of π. Case the last step of π is a logical rule, the result follows easily by the I.H.. Case the last step of π is either an axiom or a conversion rule, we show by induction on the number n of conversion rules in π that the sequent $\Sigma;\Delta\Rightarrow e:G$ is derivable by $axiom-conv$ in HH°. If $n=0$ then π is of the form:

$$\dfrac{\vdash\Sigma;\Delta_1,x:A\ basis}{\Sigma;\Delta_1,x:A\Rightarrow x:A}\ axiom.$$

Thus, the following HH°-derivation may be formed:

$$\dfrac{\vdash\Sigma;\Delta_1,x:A\ basis\quad \Sigma\vdash A\equiv A}{\Sigma;\Delta_1,x:A\Rightarrow x:A}\ axiom-conv.$$

If $n>0$, consider the last step of π to be of the form:

$$\dfrac{\Sigma;\Delta_1,x_1:H_1\Rightarrow x:A\quad \Sigma\vdash H_1\equiv H}{\Sigma;\Delta_1,x_1:H\Rightarrow x:A}\equiv_l .$$

By the I.H., $\Sigma;\Delta_1,x_1:H_1\Rightarrow x:A$ is derivable by $axiom-conv$. So, either (i) x_1 is the same as x and $\Sigma\vdash H_1\equiv A$ is derivable; or (ii) Δ_1 is of the form $(x:A_1,\Delta_2)$ and $\Sigma\vdash A_1\equiv A$ is derivable.

Case (i), the following HH°-derivation may be formed:

$$\dfrac{\vdash\Sigma;\Delta_1,x:H\ basis\quad \Sigma\vdash H\equiv A}{\Sigma;\Delta_1,x:H\Rightarrow x:A}\ axiom-conv.$$

Note that: (a) a derivation of $\Sigma\vdash H\equiv A$ may be easily constructed from derivations of $\Sigma\vdash H_1\equiv H$ and $\Sigma\vdash H_1\equiv A$; and (b) there is a derivation of $\vdash\Sigma;\Delta_1,x:H\ basis$, since

$\vdash \Sigma; \Delta_1, x : H_1$ *basis* is derivable (for $\Sigma; \Delta_1, x_1 : H_1 \Rightarrow x : A$ is derivable) and $\Sigma \vdash H_1 \equiv H$ is derivable.

Case (ii), the following $HH°$-derivation may be formed:

$$\frac{\vdash \Sigma; \Delta_2, x : A_1, x_1 : H \; basis \quad \Sigma \vdash A_1 \equiv A}{\Sigma; \Delta_2, x : A_1, x_1 : H \Rightarrow x : A} \; axiom - conv.$$

Note that a derivation of $\vdash \Sigma; \Delta_1, x : A_1, x_1 : H$ *basis* may be constructed from derivations of $\vdash \Sigma; \Delta_1, x : A_1, x_1 : H_1$ *basis* and $\Sigma \vdash H_1 \equiv H$.

The case where the last step of π is $\equiv_r$ follows by similar arguments. □

Although $HH°$ constrains the form of derivations allowed in HH, there are still forms of deriving a formula G w.r.t. a basis $\Sigma; \Delta$, which may be regarded as essentially the same, since if $\Sigma; \Delta \Rightarrow e : G$ is derivable in $HH°$ and e_1 is a proof-term only differing from e by convertible λ-terms, then $\Sigma; \Delta \Rightarrow e_1 : G$ is also derivable in $HH°$. This result is made precise with Definition 3.18 and Theorem 3.22.

Definition 3.18 (λ-convertible proof-terms) *The binary relation $\equiv_\lambda$ on proof-terms is the reflexive, symmetric, transitive and compatible closure of the relation R defined as: $e_1 R e_2$ iff one of the following holds:*

- $e_1 = pair_q(\Lambda_1, e)$, $e_2 = pair_q(\Lambda_2, e)$ *and Λ_1 and Λ_2 are convertible; or*
- $e_1 = apply_q(x, \Lambda_1, x_1.e)$, $e_2 = apply_q(x, \Lambda_2, x_1.e)$ *and Λ_1 and Λ_2 are convertible.*

Proof-terms $\equiv_\lambda$-related are called λ-convertible *proof-terms.*

Theorem 3.22 *If $e_1 \equiv_\lambda e_2$ then $\Sigma; \Delta \Rightarrow e_1 : G$ is derivable in $HH°$ iff $\Sigma; \Delta \Rightarrow e_2 : G$ is derivable in $HH°$.*

Proof: We show, by induction on the structure of e_1, that if $\Sigma; \Delta \Rightarrow e_1 : G$ is derivable in $HH°$, so is $\Sigma; \Delta \Rightarrow e_2 : G$. (The other implication is analogous.) Case $e_1 = pair_q(\Lambda_1, e)$. Then, the last step of a derivation of $\Sigma; \Delta \Rightarrow e_1 : G$ is of the form:

$$\frac{\Sigma; \Delta \Rightarrow e : [\Lambda_1/x]G_1 \quad \Sigma \vdash \Lambda_1 : \tau}{\Sigma; \Delta \Rightarrow pair_q(\Lambda_1, e) : \exists_{x:\tau} G_1} \Rightarrow \exists.$$

Since $e_1 \equiv_\lambda e_2$, it may be shown that $e_2 = pair(\Lambda_2, e_3)$, where Λ_2 is convertible to Λ_1 and $e \equiv_\lambda e_3$. So, by the I.H., there is a $HH°$-derivation of $\Sigma; \Delta \Rightarrow e_3 : [\Lambda_1/x]G_1$. By Theorem 3.20, there is a HH-derivation of $\Sigma; \Delta \Rightarrow e_3 : [\Lambda_1/x]G_1$, and by using $\equiv_r$, since Λ_1 is convertible to Λ_2, there is a HH-derivation of $\Sigma; \Delta \Rightarrow e_3 : [\Lambda_2/x]G_1$. Thus, by Theorem 3.21, there is a $HH°$-derivation of $\Sigma; \Delta \Rightarrow e_3 : [\Lambda_2/x]G_1$. So, the following $HH°$-derivation may be formed:

$$\frac{\Sigma; \Delta \Rightarrow e_3 : [\Lambda_2/x]G_1 \quad \Sigma \vdash \Lambda_2 : \tau}{\Sigma; \Delta \Rightarrow pair_q(\Lambda_2, e_3) : \exists_{x:\tau} G_1} \Rightarrow \exists.$$

The case where e_1 is of the form $apply(x, \Lambda, x_1.e)$ follows by similar arguments. The other cases follow directly by the I.H.. □

3.8.2 Proof-Theoretic Semantics of HOPLP

The motivation to study the calculi HH and HH° is to provide a proof-theoretic semantics for the higher-order logic programming language HOPLP. A semantics for HOPLP is defined below, following ideas similar to those used in Section 3.6, defining a semantics for the first-order language FOPLP.

Definition 3.19

- *A* program *in HOPLP is a pair* $\langle \Sigma, \Delta \rangle$, *usually written* $\Sigma; \Delta$, *where* Σ *is a signature and* Δ *is a* HH*-context. A program* $\Sigma; \Delta$ *is* well-formed *iff the judgement* $\vdash \Sigma; \Delta$ *basis is derivable in* HH.
- *A* goal *in HOPLP is a* G*-formula of* HH*; a goal* G *is* well-formed *w.r.t. a program* $\Sigma; \Delta$ *iff the judgement* $\Sigma \vdash G\ gf$ *is derivable in* HH.
- *A goal* G *is* achievable *w.r.t. a program* $\Sigma; \Delta$ *in HOPLP iff there is a proof-term* e *s.t.* $\Sigma; \Delta \Rightarrow e : G$ *is derivable in* HH*; the proof-term* e *is called a* witness *for the achievement of* G *w.r.t.* $\Sigma; \Delta$.

Usually, in logic programming, one is not only interested in knowing whether or not a goal is achievable w.r.t. a program, but also interested in knowing all the different means of achieving the goal w.r.t. the program.

As illustrated in Sec. 3.8.1, given a witness e for the achievement of a goal G w.r.t. a program $\Sigma; \Delta$, there may be several HH-derivations of the sequent $\Sigma; \Delta \Rightarrow e : G$, having distinct principal parts. However, the principal parts of any two HH°-derivations of the sequent $\Sigma; \Delta \Rightarrow e : G$ are the same. By Theorem 3.21, the calculus HH° is complete w.r.t. witnesses for HOPLP.

Theorem 3.22 shows that if e_1 is a witness, for the achievement of a goal G w.r.t. a program $\Sigma; \Delta$, and e_2 is a proof-term λ-convertible to e_1 then e_2 is also a witness for the achievement of G w.r.t. $\Sigma; \Delta$. Following the view that convertible λ-terms denote the same object, we regard λ-convertible proof-terms as essentially denoting the same object. So, we regard λ-convertible witnesses as the same means of goal-achievement.

A HH°-derivation is *uniform linear focused* if its proof-term is uniform linear focused, where uniform linear focused proof-terms of HH° are defined as for hH, the only difference being that first-order terms are replaced by λ-terms.

Exactly the same kind of arguments used to prove Theorem 3.4 may be used to prove that every HH°-derivation may be transformed into a uniform linear focused derivation. So, we regard witnesses that have the same uniform linear focused form as the same means of goal-achievement in HOPLP.

Definition 3.20 (complete set of witnesses for HOPLP) *A* complete set *S* of witnesses *for the achievement of a goal G w.r.t. a program $\Sigma;\Delta$ is a maximal set w.r.t.: (i) S consists of uniform linear focused proof-terms e s.t. $\Sigma;\Delta \Rightarrow e : G$ is derivable in HH°; (ii) no two members of S are λ-convertible.*

Definition 3.21 (implementation of HOPLP) *An* implementation *of HOPLP is a method that, given a goal G well-formed w.r.t. a well-formed program $\Sigma;\Delta$, finds a complete set of witnesses for the achievement of G w.r.t. $\Sigma;\Delta$.*

A method for implementing HOPLP may be described similarly to the method described in Sec. 3.7 for implementing FOPLP, the main difference being that unification on first-order terms needs to be replaced by unification of λ-terms. Recall the clause to deal with this case in FOPLP:

$$\begin{aligned}&search1(x : A_1, A, \Sigma;\Delta, \Theta_{in}, \Theta_{out}, x, V_{in}, V_{out}) \text{ if}\\ &\quad unify(A_1, A, \Theta_{in}, \Theta_{out}, V_{in}, V_{out}, \Sigma).\end{aligned}$$

Contrary to the case of FOPLP, the atomic formulae A_1 may have occurrences of λ-terms. So, in the case of HOPLP, a formula $unify(A_1, A_2, \Theta_{in}, \Theta_{out}, V_{in}, V_{out}, \Sigma)$ holds iff $A_1 = p\Lambda_1...\Lambda_n$, $A_2 = p\Lambda'_1...\Lambda'_n$, S is the set consisting of the pairs $\langle\Lambda_i, \Lambda'_i\rangle$, for $1 \leq i \leq n$, and the formula $unify(S, \Theta_{in}, \Theta_{out}, V_{in}, V_{out}, \Sigma)$, whose meaning is defined in Sec. 2.2, holds. Recall that, as opposed to unification of first-order terms, the set of most general unifiers for unifiable λ-terms may have more than one unifier. Provided an enumeration of unifiers does not enumerate unifiers Θ_{out_1} and Θ_{out_2} s.t., for every variable x, $\Theta_{out_1}(x)$ is λ-convertible to $\Theta_{out_2}(x)$ (as in the method presented in [Hue75]), the set of witnesses found by this method does not contain λ-convertible witnesses. However, since the enumeration of unifiers may be incomplete, it may be the case that not all witnesses in a complete set of witnesses are obtainable as ground instances of those unifiers.

Summarising, we have suggested a method for implementing HOPLP s.t.: if a goal is compound it breaks up the goal; if a goal is atomic it selects a formula from the program and proceeds by decomposing the selected formula; higher-order unification is used to deal with the choice of λ-terms to use in rules $\Rightarrow \exists$ and $\forall \Rightarrow$. Again, like the search procedure presented for FOPLP, this search procedure follows ideas similar to search procedures used as bases for implementing λProlog.

The language HOPLP and the related calculus HH constitute the basis of the integration of logic and functional programming suggested in the next chapter.

3.8.3 A Natural Deduction Interpretation of HOPLP

In this section is given an interpretation of HOPLP by means of the calculus $NN^{\lambda norm}$. This calculus may be thought of as an extension of the natural deduction system NN, where first-order terms are replaced by λ-terms. The logic formalised by $NN^{\lambda norm}$ is essentially the same higher-order hereditary Harrop logic formalised by HH, the difference being that, whereas in HH arbitrary formulae are allowed, in $NN^{\lambda norm}$ formulae are required to satisfy a normality constraint.

The classes of objects used in $NN^{\lambda norm}$ are the same as those of NN except for first-order terms which are replaced by λ-terms; their definitions are obtained from those in NN by replacing first-order terms by λ-terms. (Notice that the definitions of the classes of objects of $NN^{\lambda norm}$, except for proof-terms, are the same as those of HH.)

The *normal form of a formula* F in $NN^{\lambda norm}$, where F is either a G or H-formula, is the formula, written as $\lambda norm(F)$, obtained from F by replacing each λ-term Λ by its expanded normal form $\lambda norm(\Lambda)$. A formula is a *normal form* if all its λ-terms are in expanded normal form.

The forms of judgement of $NN^{\lambda norm}$ are (i)-(vi) of Fig. 3.10, which are common to HH, together with *sequents* of the forms:

$$\Sigma; \Delta \triangleright a : H;$$
$$\Sigma; \Delta \triangleright\triangleright N : G,$$

where the formulae in Δ, as well as H and G, are normal forms. The derivable judgements of forms (i)-(vi) are the same as those of HH. The rules for deriving $NN^{\lambda norm}$-sequents are those of NN, shown in Fig. 3.8, with the appropriate change of objects, except for the rules $\forall - Elim$ and $\exists - Intr$, which are replaced by the rules:

$$\frac{\Sigma; \Delta \triangleright a : \forall_{x:\tau} H \quad \Sigma \vdash \Lambda : \tau}{\Sigma; \Delta \triangleright app_q(a, \Lambda) : \lambda norm([\Lambda/x]H)} \ \forall - Elim;$$

$$\frac{\Sigma; \Delta \triangleright\triangleright N : \lambda norm([\Lambda/x]G) \quad \Sigma \vdash \Lambda : \tau}{\Sigma; \Delta \triangleright\triangleright (\Lambda, N) : \exists_{x:\tau} G} \ \exists - Intr.$$

As for HH, a proof-term differing only up to convertible λ-terms from a proof-term of a derivation is also a proof-term of a derivation, as shown below.

Definition 3.22 (λ-convertible proof-terms) *The binary relation $\equiv_\lambda$ on proof-terms is the reflexive, symmetric, transitive and compatible closure of the relation R defined as: $N_1 R N_2$ iff one of the following holds:*

- *$N_1 = (\Lambda_1, N)$, $N_2 = (\Lambda_2, N)$ and Λ_1 and Λ_2 are convertible; or*
- *$N_1 = app_q(a, \Lambda_1)$, $N_2 = app_q(a, \Lambda_2)$ and Λ_1 and Λ_2 are convertible.*

*Proof-terms $\equiv_\lambda$-related are called λ-*convertible *proof-terms.*

Theorem 3.23

(1) *If $N_1 \equiv_\lambda N_2$ then $\Sigma; \Delta \triangleright\triangleright N_1 : G$ is derivable in $NN^{\lambda norm}$ iff $\Sigma; \Delta \triangleright\triangleright N_2 : G$ is derivable in $NN^{\lambda norm}$.*

(2) *If* $a_1 \equiv_\lambda a_2$ *then* $\Sigma; \Delta \triangleright a_1 : H$ *is derivable in* $NN^{\lambda norm}$ *iff* $\Sigma; \Delta \triangleright a_2 : H$ *is derivable in* $NN^{\lambda norm}$.

Proof: By simultaneous induction on the structure of N_1 and a_1. Case $N_1 = (\Lambda_1, N)$. Then, a derivation of $\Sigma; \Delta \triangleright\triangleright N_1 : G$ is of the form:

$$\frac{\overset{\pi_1}{\Sigma; \Delta \triangleright\triangleright N : \lambda norm([\Lambda_1/x]G_1)} \quad \Sigma \vdash \Lambda_1 : \tau}{\Sigma; \Delta \triangleright\triangleright (\Lambda_1, N) : \exists_{x:\tau} G_1} \exists - Intr.$$

Since $N_1 \equiv_\lambda N_2$, it may be shown that $N_2 = (\Lambda_2, N_3)$, where Λ_2 is convertible to Λ_1 and $N \equiv_\lambda N_3$. So, $\lambda norm([\Lambda_2/x]G_1)$ is the same as $\lambda norm([\Lambda_1/x]G_1)$ and the following deduction may be formed:

$$\frac{\overset{\pi_1}{\Sigma; \Delta \triangleright\triangleright N : \lambda norm([\Lambda_2/x]G_1)} \quad \Sigma \vdash \Lambda_2 : \tau}{\Sigma; \Delta \triangleright\triangleright (\Lambda_2, N) : \exists_{x:\tau} G_1} \exists - Intr.$$

The case where a_1 is of the form $app_q(a, \Lambda_1)$ is similar to the case above. The other cases follow directly by the I.H.. □

If one takes two deductions in $NN^{\lambda norm}$ whose proof-terms are λ-convertible and deletes all the proof-term annotations one is left with the same structure. So, these two deductions may be thought of as two variants of the same deduction in the (proof-term)-free version of $NN^{\lambda norm}$. (Recall that proof-terms are introduced in Sec. 2.3 as a means of encoding deductions.) So, we regard deductions whose proof-terms are λ-convertible as equal deductions.

Below is introduced the calculus $HH^{\lambda norm}$, that is used as an intermediate step in the interpretation of HH into $NN^{\lambda norm}$. Essentially, $HH^{\lambda norm}$ is a restriction of HH where all formulae in derivations are required to be normal forms.

The classes of objects used in $HH^{\lambda norm}$ and their definitions are the same as those in HH. The forms of judgement of $HH^{\lambda norm}$ are (i)-(vi) of Fig. 3.10, which are common to HH and to $NN^{\lambda norm}$, together with *sequents* $\Sigma; \Delta \Rightarrow e : G$, where the formulae in Δ, as well as G, are normal forms. The derivable judgements of forms (i)-(vi) are the same as those of HH and $NN^{\lambda norm}$, where formulae are normal forms. The rules for deriving sequents are the same as those of HH except for *conversion* rules, which are not allowed, and for the rules $\forall \Rightarrow$ and $\Rightarrow \exists$, which are replaced by the rules:

$$\frac{\Sigma; \Delta, x_1 : \forall_{x:\tau} H, x_2 : \lambda norm([\Lambda/x]H) \Rightarrow e : G \quad \Sigma \vdash \Lambda : \tau}{\Sigma; \Delta, x_1 : \forall_{x:\tau} H \Rightarrow apply_q(x_1, \Lambda, x_2.e) : G} \forall \Rightarrow,\ x_2 \notin \Delta;$$

$$\frac{\Sigma; \Delta \Rightarrow e : \lambda norm([\Lambda/x]G) \quad \Sigma \vdash \Lambda : \tau}{\Sigma; \Delta \Rightarrow pair_q(\Lambda, e) : \exists_{x:\tau} G} \Rightarrow \exists.$$

Theorem 3.24 *Let* G *be a normal G-formula and* Δ *be a context whose formulae are normal forms. Then,* $\Sigma; \Delta \Rightarrow e : G$ *is derivable in* HH *iff it is derivable in* $HH^{\lambda norm}$.

Proof: Assume $\Sigma; \Delta \Rightarrow e : G$ has a derivation π in HH. It may be easily shown by induction on the structure of π that: the derivation obtained from π by replacing, in each sequent, formulae by their normal forms gives a derivation in $HH^{\lambda norm}$. For example, assume π contains a rule of the form:

$$\frac{\Sigma_1; \Delta_1 \Rightarrow e_1 : G_1 \quad \Sigma \vdash G_2 \equiv G_1}{\Sigma_1; \Delta_1 \Rightarrow e_1 : G_2} \Rightarrow \equiv_r .$$

Let Δ_2 be the context obtained from Δ_1 by replacing all formulae by their normal forms. Then, by the I.H., the sequent $\Sigma_1; \Delta_2 \Rightarrow e_1 : \lambda norm(G_1)$ is derivable in $HH^{\lambda norm}$, and so is the sequent $\Sigma_1; \Delta_2 \Rightarrow e_1 : \lambda norm(G_2)$, for $\lambda norm(G_2)$ is the same as $\lambda norm(G_1)$.

Conversely, it may be easily proved that if a sequent is derivable in $HH^{\lambda norm}$ then it is derivable in HH. The proof follows by induction on the structure of derivations in $HH^{\lambda norm}$. Notice that the rules $\forall \Rightarrow$ and $\Rightarrow \exists$ of $HH^{\lambda norm}$ are derivable in HH by combining the corresponding rules in HH together with appropriate *conversion* rules. □

Theorem 3.25 *The mapping ϕ^λ, obtained from ϕ, defined in Sec. 2.3.4, by replacing first-order terms by λ-terms, is a 1-1 correspondence between the uniform linear focused proof-terms e s.t. $\Sigma; \Delta \Rightarrow e : G$ is derivable in $HH^{\lambda norm}$ and the proof-terms N s.t. $\Sigma; \Delta \triangleright \triangleright N : G$ is derivable in $NN^{\lambda norm}$.*

Proof: Similar techniques to those used in Sec. 3.5, proving that ϕ is a 1-1 correspondence between the uniform linear focused proof-terms e s.t. $\Sigma; \Delta \Rightarrow e : G$ is derivable in hH and the proof-terms N s.t. $\Sigma; \Delta \triangleright \triangleright N : G$ is derivable in NN, may be used. □

For a normal G-formula G and a context Δ whose formulae are normal forms, $\Sigma; \Delta \triangleright \triangleright N : G$ is derivable in $NN^{\lambda norm}$ iff there exists a uniform linear focused proof-term e s.t. $\phi^\lambda(e) = N$ and $\Sigma; \Delta \Rightarrow e : G$ is derivable in HH, which in turn is equivalent to: $\Sigma; \Delta \Rightarrow e : G$ is derivable in HH°. Since we regard $NN^{\lambda norm}$-deductions with λ-convertible proof-terms as equal deductions, we may give the following interpretation of HOPLP by means of $NN^{\lambda norm}$. Given a goal G and a program $\Sigma; \Delta$, an implementation of HOPLP is any method that finds the $NN^{\lambda norm}$-deductions of $\lambda norm(G)$ w.r.t. Δ_1, the context obtained from Δ by replacing each formula by its normal form.

3.9 Summary

In this Chapter we set up the basis for our proposal of integrated logic and functional programmming (LFPL), described in Chapter 4. We define the logic programming language HOPLP, based on the intuitionistic hereditary Harrop logic with λ-terms rather than first-order terms (a fragment of the logic underlying λProlog). This language is defined by means of an LJ-based formalisation of its underlying logic, where proof-terms are used for representing derivations. The use of proof-terms is instrumental in the definition of the semantics of HOPLP (and LFPL), since the result of computations are proof-terms (as in Elf) encoding derivations of goals from programs. We define achievements of goals in HOPLP as proof-terms of uniform linear focused derivations, that we show to be in 1-1 correspondence to expanded normal deductions in an NJ-based formulation of the logic underlying HOPLP.

Chapter 4

Logical and Functional Programming

4.1 Introduction

The execution mechanism in the logic programming language HOPLP is defined in Sec. 3.8.2 as a search for a derivation of a goal w.r.t. a program in the calculus HH. The calculus HH uses the set of λ-terms as the set of underlying terms, thus functions are already allowed in HOPLP in the form of λ-abstractions. Usually, in functional programming names may be defined as abbreviations for functions [Jon87, Tho91], so that, instead of writing the entire expression representing a function, a name may be used to refer to a function.

Section 4.2 defines the programming language HOPLPD that extends HOPLP with an abbreviation mechanism for λ-terms. Mappings from programs and goals in HOPLPD into programs and goals in HOPLP are shown in Sec. 4.2; both mappings are many to one. Further, it is shown that witnesses for the achievement of a goal w.r.t. a program in HOPLPD are in 1-1 correspondence with witnesses for the achievement of the corresponding goal w.r.t. the corresponding program in HOPLP. We say that HOPLPD is a *conservative extension* of HOPLP, i.e. if a goal is achievable in HOPLP a corresponding goal is achievable in HOPLPD (extension) and if a goal is achievable in HOPLPD then the corresponding goal is achievable in HOPLP (conservative).

Typical integrations of logic and functional programming [AKN89, Han90] may be seen as extensions of logic programming that allow functions in the set of underlying terms to build formulae, providing an abbreviation mechanism to refer to functions. We propose another step in the integration of logic and functional programming, relating logical properties of functions with the logic programming part of the language, as explained below.

Often the type of a function is regarded as a first specification for a function. However, there are many logical aspects of functions that may not be captured within a simple theory of types. In the context of logic and functional programming, we propose a higher interaction between the logic and functional parts. We suggest the use of the logic language, underlying the logic programming part, for describing specifications of functions, coming from the functional programming part. Σ-types are used to attach a logical specification to a term of simple type, in a similar fashion to that used in the theory of deliverables [MB93].

Consider a program P and a goal G in a logic programming language which allows λ-terms, e.g. HOPLP. Let f be a function of type τ. Let F be a formula containing free occurrences of a variable y of type τ and let $[f/y]F$ be achievable w.r.t. the program P, in other words the function f meets the specification $\Sigma_{y:\tau}F$. Consider an attempt to achieve a goal w.r.t. P. Since $[f/y]F$ is achievable w.r.t. P, if a principle similar to Gentzen's *cut rule* is used, the problem of achieving G w.r.t. P may be transformed into the problem of achieving G w.r.t. the program P together with the formula $[f/y]F$. In case $[f/y]F$ is used in achieving G, such form of achieving G from P may be shorter than alternative forms of achieving G from P not using $[f/y]F$. Recall that in calculi allowing *cuts*, derivations with cuts may be significantly shorter than their cut-free variants [Boo84]. Our proposal to integrate logic and functional programming is based on the ideas described above, i.e. uses Σ-types to attach logical specifications to functions and uses a principle similar to the cut rule for accessing the logical content of a specification.

Section 4.6 defines a programming language that integrates logic and functional programming called LFPL. This language provides a mechanism for definitions of simple type as well as a mechanism for attaching logical specifications to terms of simple type based on the ideas described above. The language LFPL is defined in terms of the calculus HH^{def} presented in Sec. 4.5. The language LFPL is shown to be a conservative extension of HOPLP. The interpretation of proofs in LFPL as (cut-free) proofs in HOPLP is essentially a process of cut-elimination.

4.2 Simple Definitions and the Calculus HH'

Traditionally, functional programming provides an abbreviation mechanism that allows names to be introduced as abbreviations for expressions, thus making expressions more readable. In logic programming a similar idea may be used by introducing an abbreviation mechanism for terms of simple type to allow the writing of clearer logic programs. This section presents the calculus HH' which can be seen as an extension of HH with an abbreviation mechanism for terms of simple type.

A *simple definition* is a triple $\langle x, \Lambda, \tau\rangle$, written as $x =_{def} \Lambda : \tau$, where x is a variable, Λ is a λ-term and τ is a simple type; x is called the *definiendum* of the definition, Λ is called the *definiens* of the definition and τ is called the *type* of the definition.

Simple definitions are declared in the context-part of a program. *Contexts* in HH' are lists. The following notation is used for representing lists: $\langle\rangle$ represents the empty list; (E, L) represents the list whose head is E and whose tail is the list L; (L, E) represents the list obtained from the list L by adding E as last element; (L_1, L_2) represents the list obtained by appending the list L_2 at the end of L_1; the external parentheses are dropped when there is no danger of confusion. In HH', contexts Δ are lists defined as follows:

$$\Delta ::= \langle\rangle \mid \Delta, x : H \mid \Delta, x =_{def} \Lambda : \tau,$$

where H is a meta-variable ranging over the set of H-formulae of HH. The notation $x \notin \Delta$ means that there is no H s.t. $x : H$ is an element of the list Δ and there is no simple definition in Δ whose definiendum is x. The set of all definienda of simple definitions in a context Δ is written as $definienda(\Delta)$. Definienda of simple definitions may be used in building terms of simple type. The intended meaning for a simple definition in a context of the form $(\Delta_1, x =_{def} \Lambda : \tau, \Delta_2)$ is that in Δ_2 any occurrence of x may be replaced by Λ. Notice that the order of the components in a context is important. The other classes of objects in HH' are the same as those of HH and they are defined as in HH.

The forms of judgement of HH' are presented in Fig. 4.1. After each form of judgement of

	Judgements of HH'	Judgements of HH
(i)	$\vdash \Sigma\ signature$	$\vdash \Sigma\ signature$
(ii)	$\Sigma; \Delta \vdash \Lambda : \tau$	$\Sigma \vdash \Lambda : \tau$
(iii)	$\Sigma; \Delta \vdash \Lambda \rhd_\tau \Lambda$	$\Sigma \vdash \Lambda \rhd_\tau \Lambda$
(iv)	$\Sigma; \Delta \vdash \Lambda \rhd^*_\tau \Lambda$	$\Sigma \vdash \Lambda \rhd^*_\tau \Lambda$
(v)	$\Sigma; \Delta \vdash \Lambda \equiv_\tau \Lambda$	$\Sigma \vdash \Lambda \equiv_\tau \Lambda$
(vi)	$\vdash \Sigma; \Delta\ basis$	$\vdash \Sigma; \Delta\ basis$
(vii)	$\Sigma; \Delta \vdash A\ af$	$\Sigma \vdash A\ af$
(viii)	$\Sigma; \Delta \vdash H\ hf$	$\Sigma \vdash H\ hf$
(ix)	$\Sigma; \Delta \vdash G\ gf$	$\Sigma \vdash G\ gf$
(x)	$\Sigma; \Delta \vdash A \equiv A$	$\Sigma \vdash A \equiv A$
(xi)	$\Sigma; \Delta \vdash H \equiv H$	$\Sigma \vdash H \equiv H$
(xii)	$\Sigma; \Delta \vdash G \equiv G$	$\Sigma \vdash G \equiv G$
(xiii)	$\Sigma; \Delta \Rightarrow e : G$	$\Sigma; \Delta \Rightarrow e : G$

Figure 4.1: Judgements of HH'.

HH' is the corresponding form of judgement of HH. (Recall that the definition of contexts Δ is different in HH' and in HH.) Since definienda of simple definitions may be used in building terms of simple type, the definition of the terms of a simple type depends upon signatures as well as contexts. The notion of reduction on terms of simple type also depends on the context, since simple definitions may be used to replace definienda by definientia.

The derivable signatures of HH' are the same as those of HH. The rules defining derivable judgements of forms (ii), (iii) and (vi) are shown in Fig. 4.2. The rules defining derivable judgements of forms (iv), (v) and (vii)-(xiii) in HH' are similar to the rules defining derivable judgements of the corresponding form in HH; they are shown in Appendix C.

$$\frac{\vdash \Sigma, x : \tau; \Delta\ basis}{\Sigma, x : \tau; \Delta \vdash x : \tau}$$

$$\frac{\vdash \Sigma; \Delta_1, x =_{def} \Lambda : \tau, \Delta_2\ basis}{\Sigma; \Delta_1, x =_{def} \Lambda : \tau, \Delta_2 \vdash x : \tau}$$

$$\frac{\Sigma, x : \tau; \Delta \vdash \Lambda : \tau_1 \quad \vdash \Sigma; \Delta\ basis}{\Sigma; \Delta \vdash \lambda x : \tau.\Lambda : \tau \to \tau_1}\ x \notin \Sigma \qquad \frac{\Sigma; \Delta \vdash \Lambda : \tau_1 \to \tau \quad \Sigma; \Delta \vdash \Lambda_1 : \tau_1}{\Sigma; \Delta \vdash \Lambda\Lambda_1 : \tau}$$

$$\frac{\vdash \Sigma; \Delta_1, x =_{def} \Lambda : \tau, \Delta_2\ basis}{\Sigma; \Delta_1, x =_{def} \Lambda : \tau, \Delta_2 \vdash x \rhd_\tau \Lambda}$$

$$\frac{\Sigma, x : \tau; \Delta \vdash \Lambda : \tau_1 \quad \Sigma; \Delta \vdash \Lambda_1 : \tau}{\Sigma; \Delta \vdash (\lambda x : \tau.\Lambda)\Lambda_1 \rhd_{\tau_1} [\Lambda_1/x]\Lambda}\ x \notin \Sigma$$

$$\frac{\Sigma, x : \tau; \Delta \vdash \Lambda \rhd_{\tau_1} \Lambda_1 \quad \vdash \Sigma; \Delta\ basis}{\Sigma; \Delta \vdash \lambda x : \tau.\Lambda \rhd_{\tau \to \tau_1} \lambda x : \tau.\Lambda_1}\ x \notin \Sigma$$

$$\frac{\Sigma; \Delta \vdash \Lambda \rhd_{\tau \to \tau_1} \Lambda_1 \quad \Sigma; \Delta \vdash \Lambda_2 : \tau}{\Sigma; \Delta \vdash \Lambda\Lambda_2 \rhd_{\tau_1} \Lambda_1\Lambda_2} \qquad \frac{\Sigma; \Delta \vdash \Lambda \rhd_\tau \Lambda_1 \quad \Sigma; \Delta \vdash \Lambda_2 : \tau \to \tau_1}{\Sigma; \Delta \vdash \Lambda_2\Lambda \rhd_{\tau_1} \Lambda_2\Lambda_1}$$

$$\frac{\vdash \Sigma\ signature}{\vdash \Sigma; \langle\rangle\ basis} \qquad \frac{\vdash \Sigma; \Delta\ basis \quad \Sigma; \Delta \vdash H\ hf}{\vdash \Sigma; \Delta, x : H\ basis}\ x \notin \Delta$$

$$\frac{\vdash \Sigma; \Delta\ basis \quad \Sigma; \Delta \vdash \Lambda : \tau}{\vdash \Sigma; \Delta, x =_{def} \Lambda : \tau\ basis}\ x \notin \Sigma, x \notin \Delta$$

Figure 4.2: Rules for deriving judgements of forms (ii), (iii) and (vi) of HH'.

In Fig. 4.3 are defined the operations of substitution of a λ-term Λ for a variable x in a proof-term e (notation $[\Lambda/x]e$) and in a context Δ (notation $[\Lambda/x]\Delta$). These operations of substitution of a λ-term for a variable in an object may be thought of as the application of the substitution operation to the λ-terms used in forming that object. Note that the side conditions on the definitions of Fig. 4.3 may be satisfied simply by renaming of bound variables.

Let *Object* be either a G-formula or a H-formula or a proof-term or a HH'-context. The notation $[\Lambda_1/x_1][\Lambda_2/x_2]...[\Lambda_n/x_n]Object$ stands for $[\Lambda_1/x_1]([\Lambda_2/x_2](...([\Lambda_x/x_n]Object)...))$. Let Δ be a HH'-context having n simple definitions, $x_1 =_{def} \Lambda_1 : \tau_1, ..., x_n =_{def} \Lambda_n : \tau_n$, where for $1 \leq i < j \leq n$ the i-th definition occurs in Δ before the j-th definition. The notation $[\Delta]Object$

is an abbreviation for $[\Lambda_1/x_1]...[\Lambda_n/x_n]Object$.

The mapping μ, from lists to sets, is defined as the mapping that applied to a list returns the set consisting of the elements of the list. The mapping ξ, from HH'-contexts to HH-contexts, is defined in Fig. 4.4. The mapping $\aleph$, from HH'-contexts to HH-contexts, is defined as the composition of ξ with μ.

The following lemma shows that in the absence of simple definitions HH' and HH are essentially equivalent, the only difference being that HH'-contexts are lists whereas HH-contexts are sets.

Lemma 4.1 *Let $\vdash \Sigma; \Delta$ basis be derivable in HH', where Δ has no simple definitions. Then, $\Sigma; \Delta \Rightarrow e : G$ is derivable in HH' iff $\Sigma; \mu(\Delta) \Rightarrow e : G$ is derivable in HH.*

Proof: Two implications must be proved. If there is a HH'-derivation π of $\Sigma; \Delta \Rightarrow e : G$, then it may be easily proved, by induction on the structure of π, that there exists an HH-derivation of $\Sigma; \mu(\Delta) \Rightarrow e : G$, since in the absence of simple definitions each rule in HH' has a corresponding rule in HH.

If there is a HH-derivation π of $\Sigma; \mu(\Delta) \Rightarrow e : G$ then it may be proved, by induction on the structure of π, that $\Sigma; \Delta_1 \Rightarrow e : G$ is derivable in HH', for every HH'-context Δ_1 s.t. $\mu(\Delta_1) = \mu(\Delta)$. □

The following lemmas are used as auxiliary lemmas in proving that HH' is a conservative extension of HH.

Lemma 4.2 *If $\vdash \Sigma; \Delta_1, x =_{def} \Lambda : \tau, \Delta_2$ basis is derivable in HH' then $\vdash \Sigma; \Delta_1, [\Lambda/x]\Delta_2$ basis is derivable in HH'.*

Proof: Let π be a HH'-derivation of $\vdash \Sigma; \Delta_1, x =_{def} \Lambda : \tau, \Delta_2$ *basis*. The proof follows by induction on the structure of π. We consider only the case where π is of the following form:

$$\frac{\overset{\pi_1}{\vdash \Sigma; \Delta_1, x =_{def} \Lambda : \tau, \Delta_{21}\ basis} \quad \overset{\pi_2}{\Sigma; \Delta_1, x =_{def} \Lambda : \tau, \Delta_{21} \vdash H\ hf}}{\vdash \Sigma; \Delta_1, x =_{def} \Lambda : \tau, \Delta_{21}, x_1 : H\ basis},$$

where $\Delta_2 = (\Delta_{21}, x_1 : H)$ and $x_1 \notin (\Delta_1, x =_{def} \Lambda : \tau, \Delta_{21})$. (The other cases follow by similar arguments.) Since π_1 is a subderivation of π, by the I.H. there is a derivation of Judgement 4.1.

$$\vdash \Sigma; \Delta_1, [\Lambda/x]\Delta_{21}\ basis \tag{4.1}$$

From π_2, we may construct a derivation of Judgement 4.2, as sketched below.

$$\Sigma; \Delta_1, [\Lambda/x]\Delta_{21} \vdash [\Lambda/x]H\ hf \tag{4.2}$$

$$[\Lambda/x](p\Lambda_1 \ldots \Lambda_n) =_{def} p[\Lambda/x]\Lambda_1 \ldots [\Lambda/x]\Lambda_n$$
$$[\Lambda/x](G \supset H) =_{def} [\Lambda/x]G \supset [\Lambda/x]H$$
$$[\Lambda/x](H_1 \wedge H_2) =_{def} [\Lambda/x]H_1 \wedge [\Lambda/x]H_2$$
$$[\Lambda/x](\forall_{x_1:\tau}H) =_{def} \forall_{x_1:\tau}[\Lambda/x]H, x \neq x_1, x_1 \notin \Lambda$$

$$[\Lambda/x](G_1 \wedge G_2) =_{def} [\Lambda/x]G_1 \wedge [\Lambda/x]G_2$$
$$[\Lambda/x](G_1 \vee G_2) =_{def} [\Lambda/x]G_1 \vee [\Lambda/x]G_2$$
$$[\Lambda/x](H \supset G) =_{def} [\Lambda/x]H \supset [\Lambda/x]G$$
$$[\Lambda/x](\exists_{x_1:\tau}G) =_{def} \exists_{x_1:\tau}[\Lambda/x]G, x \neq x_1, x_1 \notin \Lambda$$
$$[\Lambda/x](\forall_{x_1:\tau}G) =_{def} \forall_{x_1:\tau}[\Lambda/x]G, x \neq x_1, x_1 \notin \Lambda$$

$$[\Lambda/x]pair(e_1, e_2) =_{def} pair([\Lambda/x]e_1, [\Lambda/x]e_2)$$
$$[\Lambda/x]inl(e) =_{def} inl([\Lambda/x]e)$$
$$[\Lambda/x]inr(e) =_{def} inr([\Lambda/x]e)$$
$$[\Lambda/x]lambda(x_1.e) =_{def} lambda(x_1.[\Lambda/x]e), x \neq x_1, x_1 \notin \Lambda$$
$$[\Lambda/x]lambda_q(x_1.e) =_{def} lambda_q(x_1.[\Lambda/x]e), x \neq x_1, x \notin \Lambda$$
$$[\Lambda/x]pair_q(\Lambda_1, e) =_{def} pair_q([\Lambda/x]\Lambda_1, [\Lambda/x]e)$$
$$[\Lambda/x]x_1 =_{def} x_1$$
$$[\Lambda/x]splitl(x_1, x_2.e) =_{def} splitl(x_1, x_2.[\Lambda/x]e), x \neq x_2, x_2 \notin \Lambda$$
$$[\Lambda/x]splitr(x_1, x_2.e) =_{def} splitr(x_1, x_2.[\Lambda/x]e), x \neq x_2, x_2 \notin \Lambda$$
$$[\Lambda/x]apply(x_1, e_1, x_2.e_2) =_{def} apply(x_1, [\Lambda/x]e_1, x_2.[\Lambda/x]e_2), x \neq x_2, x_2 \notin \Lambda$$
$$[\Lambda/x]apply_q(x_1, \Lambda_1, x_2.e) =_{def} apply_q(x_1, [\Lambda/x]\Lambda_1, x_2.[\Lambda/x]e), x \neq x_2, x_2 \notin \Lambda$$

$$[\Lambda/x]\langle\rangle =_{def} \langle\rangle$$
$$[\Lambda/x](x_1 : H, \Delta) =_{def} x_1 : [\Lambda/x]H, [\Lambda/x]\Delta$$
$$[\Lambda/x](x_1 =_{def} \Lambda_1 : \tau, \Delta) =_{def} (x_1 =_{def} \Lambda_1 : \tau, \Delta), x = x_1$$
$$[\Lambda/x](x_1 =_{def} \Lambda_1 : \tau, \Delta) =_{def} (x_1 =_{def} [\Lambda/x]\Lambda_1 : \tau, [\Lambda/x]\Delta), x \neq x_1 \text{ and } x_1 \notin \Lambda$$

Figure 4.3: The operations $[\Lambda/x]G, [\Lambda/x]e, [\Lambda/x]\Delta$.

$$\begin{aligned}
&\xi(\langle\rangle) =_{def} \langle\rangle\\
&\xi(x =_{def} \Lambda : \tau, \Delta) =_{def} \xi([\Lambda/x]\Delta)\\
&\xi(x : H, \Delta) =_{def} x : H, \xi(\Delta)
\end{aligned}$$

Figure 4.4: Mapping ξ.

Let H be an atomic formula $p\Lambda_1 ... \Lambda_n$. (The cases where H is a compound formula are simple.) Then, π_2 must be of the form:

$$\dfrac{\dfrac{\overset{\pi_{11}}{\Sigma; \Delta_1, x =_{def} \Lambda : \tau, \Delta_{21} \vdash \Lambda_1 : \tau_1} \quad ... \quad \overset{\pi_{1n}}{\Sigma; \Delta_1, x =_{def} \Lambda : \tau, \Delta_{21} \vdash \Lambda_n : \tau_n}}{\Sigma; \Delta_1, x =_{def} \Lambda : \tau, \Delta_{21} \vdash p\Lambda_1 ... \Lambda_n \; af}}{\Sigma; \Delta_1, x =_{def} \Lambda : \tau, \Delta_{21} \vdash p\Lambda_1 ... \Lambda_n \; hf}$$

where $p : \tau_1 \to ... \to \tau_n \to prop \in \mathcal{P}$.

For each $\Lambda_i : \tau_i$, where $1 \leq i \leq n$, a derivation of

$$\Sigma; \Delta_1, [\Lambda/x]\Delta_{21} \vdash [\Lambda/x]\Lambda_i : \tau_i$$

may be easily constructed from the derivation π_{1i} of

$$\Sigma; \Delta_1, x =_{def} \Lambda : \tau, \Delta_{21} \vdash \Lambda_i : \tau_i.$$

For example, consider the case where the last step of π_{1i} has the form:

$$\dfrac{\vdash \Sigma; \Delta_1, x =_{def} \Lambda : \tau, \Delta_{21} \; basis}{\Sigma; \Delta_1, x =_{def} \Lambda : \tau, \Delta_{21} \vdash x : \tau} \; .$$

By the I.H. there is a derivation of

$$\vdash \Sigma; \Delta_1, [\Lambda/x]\Delta_{21} \; basis.$$

Since $\vdash \Sigma; \Delta_1, x =_{def} \Lambda : \tau, \Delta_{21} \; basis$ is derivable, it is easy to show that there is a derivation of $\Sigma; \Delta_1 \vdash \Lambda : \tau$. Thus a derivation of $\Sigma; \Delta_1, [\Lambda/x]\Delta_{21} \vdash \Lambda : \tau$ may be easily constructed. Note that $[\Lambda/x]x = \Lambda$. (Cases corresponding to other forms of π_{1i} are simpler.)

From derivations of Judgements 4.1 and 4.2, the following derivation may be formed, as desired.

$$\dfrac{\vdash \Sigma; \Delta_1, [\Lambda/x]\Delta_{21} \; basis \quad \Sigma; \Delta_1, [\Lambda/x]\Delta_{21} \vdash [\Lambda/x]H \; hf}{\vdash \Sigma; \Delta_1, [\Lambda/x]\Delta_{21}, x_1 : [\Lambda/x]H \; basis} \; ,$$

since $x_1 \notin (\Delta_1, [\Lambda/x]\Delta_{21})$. Note that $[\Lambda/x](\Delta_{21}, x_1 : H) = ([\Lambda/x]\Delta_{21}, x_1 : [\Lambda/x]H)$, by definition.

$\square$

Lemma 4.3 *Let* $\vdash \Delta_1, x =_{def} \Lambda : \tau, \Delta_2$ *basis be derivable in* HH'. *Then, if the judgement*

$$\Sigma; \Delta_1, [\Lambda/x]\Delta_2 \vdash [\Lambda/x]\Lambda_1 : \tau_1$$

is derivable in HH' *then the judgement*

$$\Sigma; \Delta_1, x =_{def} \Lambda : \tau, \Delta_2 \vdash \Lambda_1 : \tau_1$$

is derivable in HH'.

Proof: The proof follows by induction on the structure of Λ_1.

Consider Λ_1 to be a variable. If $\Lambda_1 = x$ then the following derivation may be formed

$$\frac{\vdash \Sigma; \Delta_1, x =_{def} \Lambda : \tau, \Delta_2 \; basis}{\Sigma; \Delta_1, x =_{def} \Lambda : \tau, \Delta_2 \vdash x : \tau} .$$

If Λ_1 is a variable x_1 different from x then one of the following cases must hold: (i) Σ is of the form $\Sigma_1, x_1 : \tau_1$; or (ii) Δ_1 is of the form $\Delta_{11}, x_1 =_{def} \Lambda_2 : \tau_1, \Delta_{12}$; or (iii) $[\Lambda/x]\Delta_2$ is of the form $\Delta_{21}, x_1 =_{def} \Lambda_2 : \tau_1, \Delta_{22}$. Case (i) holds, the following derivation may be formed:

$$\frac{\vdash \Sigma_1, x_1 : \tau_1; \Delta_1, x =_{def} \Lambda : \tau, \Delta_2 \; basis}{\Sigma_1, x_1 : \tau_1; \Delta_1, x =_{def} \Lambda : \tau, \Delta_2 \vdash x_1 : \tau_1} .$$

Case (ii) holds, the following derivation may be formed:

$$\frac{\vdash \Sigma; \Delta_{11}, x_1 =_{def} \Lambda_2 : \tau_1, \Delta_{12}, x =_{def} \Lambda : \tau, \Delta_2 \; basis}{\Sigma; \Delta_{11}, x_1 =_{def} \Lambda_2 : \tau_1, \Delta_{12}, x =_{def} \Lambda : \tau, \Delta_2 \vdash x_1 : \tau_1} .$$

Case (iii) holds, it may be shown that Δ_2 must be of the form $\Delta_{31}, x_1 =_{def} \Lambda_3 : \tau_1, \Delta_{32}$, where $[\Lambda/x]\Delta_{31} = \Delta_{21}$, $[\Lambda/x]\Lambda_3 = \Lambda_2$ and $[\Lambda/x]\Delta_{32} = \Delta_{22}$. So, the following derivation may be formed:

$$\frac{\vdash \Sigma; \Delta_1, x =_{def} \Lambda : \tau, \Delta_{31}, x_1 =_{def} \Lambda_3 : \tau_1, \Delta_{32} \; basis}{\Sigma; \Delta_1, x =_{def} \Lambda : \tau, \Delta_{31}, x_1 =_{def} \Lambda_3 : \tau_1, \Delta_{32} \vdash x_1 : \tau_1} .$$

The cases where Λ_1 is not a variable follow by direct application of the I.H.. □

Lemma 4.4 *Let* $\Sigma; \Delta_1, x =_{def} \Lambda : \tau, \Delta_2 \Rightarrow e : G$ *be derivable in* HH'. *Then, the sequent* $\Sigma; \Delta_1, [\Lambda/x]\Delta_2 \Rightarrow [\Lambda/x]e : [\Lambda/x]G$ *is derivable in* HH'.

Proof: Let π be a HH'-derivation of $\Sigma; \Delta_1, x =_{def} \Lambda : \tau, \Delta_2 \Rightarrow e : G$. The proof follows by induction on the structure of π. Some cases are shown below.

Case last step of π is an axiom of the form:

$$\frac{\vdash \Sigma; \Delta_1, x =_{def} \Lambda : \tau, \Delta_{21}, x_1 : A, \Delta_{22} \; basis}{\Sigma; \Delta_1, x =_{def} \Lambda : \tau, \Delta_{21}, x_1 : A, \Delta_{22} \Rightarrow x_1 : A} \; axiom,$$

where $\Delta_2 = \Delta_{21}, x_1 : A, \Delta_{22}$, $e = x_1$ and $G = A$. Then, Lemma 4.2 shows that

$$\vdash \Sigma; \Delta_1, [\Lambda/x]\Delta_{21}, x_1 : [\Lambda/x]A, [\Lambda/x]\Delta_{22} \; basis$$

is derivable. The following derivation may be formed:

$$\frac{\vdash \Sigma; \Delta_1, [\Lambda/x]\Delta_{21}, x_1 : [\Lambda/x]A, [\Lambda/x]\Delta_{22}\ basis}{\Sigma; \Delta_1, [\Lambda/x]\Delta_{21}, x_1 : [\Lambda/x]A, [\Lambda/x]\Delta_{22} \Rightarrow x_1 : [\Lambda/x]A}\ axiom.$$

The proof of this case is concluded by using the following identities:

$$[\Lambda/x]\Delta_2 = [\Lambda/x]\Delta_{21}, x_1 : [\Lambda/x]A, [\Lambda/x]\Delta_{22};$$
$$[\Lambda/x]x_1 = x_1.$$

Case last step of π is $\Rightarrow \exists$:

$$\frac{\Sigma; \Delta_1, x =_{def} \Lambda : \tau, \Delta_2 \Rightarrow e_1 : [\Lambda_1/y]G_1 \quad \Sigma; \Delta_1, x =_{def} \Lambda : \tau, \Delta_2 \vdash \Lambda_1 : \tau_1}{\Sigma; \Delta_1, x =_{def} \Lambda : \tau, \Delta_2 \Rightarrow pair_q(\Lambda_1, e_1) : \exists_{y:\tau_1} G_1} \Rightarrow \exists,$$

where $e = pair_q(\Lambda_1, e_1)$ and $G = \exists_{y:\tau_1} G_1$. From a derivation of

$$\Sigma; \Delta_1, x =_{def} \Lambda : \tau, \Delta_2 \vdash \Lambda_1 : \tau_1,$$

we may construct a derivation of

$$\Sigma; \Delta_1, [\Lambda/x]\Delta_2 \vdash [\Lambda/x]\Lambda_1 : \tau_1.$$

(A sketch of this construction is given in the proof of Lemma 4.2.) By the I.H., there is a derivation of

$$\Sigma; \Delta_1, [\Lambda/x]\Delta_2 \Rightarrow [\Lambda/x]e_1 : [\Lambda/x]([\Lambda_1/y]G_1).$$

Using the identity

$$[\Lambda/x]([\Lambda_1/y]G_1) = [[\Lambda/x]\Lambda_1/y]([\Lambda/x]G_1),$$

which may be shown by using the substitution property of λ^{ST}, the following derivation may be formed

$$\frac{\Sigma; \Delta_1, [\Lambda/x]\Delta_2 \Rightarrow [\Lambda/x]e_1 : [[\Lambda/x]\Lambda_1/y]([\Lambda/x]G_1) \quad \Sigma; \Delta_1, [\Lambda/x]\Delta_2 \vdash [\Lambda/x]\Lambda_1 : \tau_1}{\Sigma; \Delta_1, [\Lambda/x]\Delta_2 \Rightarrow pair_q([\Lambda/x]\Lambda_1, [\Lambda/x]e_1) : \exists_{y:\tau_1} [\Lambda/x]G_1} \Rightarrow \exists.$$

The proof of this case is concluded by observing that the following identities hold:

$$[\Lambda/x]pair_q(\Lambda_1, e_1) = pair_q([\Lambda/x]\Lambda_1, [\Lambda/x]e_1);$$
$$[\Lambda/x](\exists_{y:\tau_1} G_1) = \exists_{y:\tau_1} [\Lambda/x]G_1.$$

The case where the last step of π is a rule $\forall \Rightarrow$ follows by similar arguments. The other cases follow directly from the I.H.. □

Lemma 4.5 *Let $\Sigma; \Delta_1, x =_{def} \Lambda : \tau, \Delta_2 \vdash G\ gf$ and $\Sigma; \Delta_1, [\Lambda/x]\Delta_2 \Rightarrow [\Lambda/x]e : [\Lambda/x]G$ be derivable in HH'. Then, $\Sigma; \Delta_1, x =_{def} \Lambda : \tau, \Delta_2 \Rightarrow e : G$ is derivable in HH'.*

Proof: Let π be a HH'-derivation of $\Sigma;\Delta_1,[\Lambda/x]\Delta_2 \Rightarrow [\Lambda/x]e : [\Lambda/x]G$. The proof follows by induction on the structure of π.

Case last step of π is an axiom of the form:

$$\frac{\vdash \Sigma;\Delta_1,\Delta_{21},x_1:A,\Delta_{22}\ basis}{\Sigma;\Delta_1,\Delta_{21},x_1:A,\Delta_{22} \Rightarrow x_1:A}\ axiom,$$

where $[\Lambda/x]\Delta_2 = (\Delta_{21},x_1:A,\Delta_{22})$, $[\Lambda/x]e = x_1$ and $[\Lambda/x]G = A$. It is easy to show that: Δ_2 must be of the form $(\Delta_{31},x_1:A_1,\Delta_{32})$, where $[\Lambda/x]\Delta_{31}=\Delta_{21}$, $[\Lambda/x]A_1 = A$, $[\Lambda/x]\Delta_{32}=\Delta_{22}$; and $e = x_1$. By hypothesis, $\Sigma;\Delta_1,x =_{def} \Lambda:\tau,\Delta_2 \vdash G\ gf$ is derivable, then there is a derivation of $\vdash \Sigma;\Delta_1,x=_{def}\Lambda:\tau,\Delta_2\ basis$. Since $[\Lambda/x]A_1 = A$, a derivation of

$$\Sigma;\Delta_1,x=_{def}\Lambda:\tau,\Delta_2 \vdash A_1 \equiv A$$

may be constructed. Thus, the following derivation may be formed:

$$\frac{\dfrac{\vdash \Sigma;\Delta_1,x=_{def}\Lambda:\tau,\Delta_2\ basis}{\Sigma;\Delta_1,x=_{def}\Lambda:\tau,\Delta_2 \Rightarrow x_1:A_1}\ axiom \quad \Sigma;\Delta_1,x=_{def}\Lambda:\tau,\Delta_2\vdash A_1\equiv A}{\Sigma;\Delta_1,x=_{def}\Lambda:\tau,\Delta_2 \Rightarrow x_1:A}\ \equiv_l .$$

Case last step of π is a rule $\Rightarrow\exists$ of the form:

$$\frac{\Sigma;\Delta_1,[\Lambda/x]\Delta_2 \Rightarrow e_1 : [\Lambda_1/y]G_1 \quad \Sigma;\Delta_1,[\Lambda/x]\Delta_2 \vdash \Lambda_1:\tau_1}{\Sigma;\Delta_1,[\Lambda/x]\Delta_2 \Rightarrow pair_q(\Lambda_1,e_1):\exists_{y:\tau_1}G_1}\Rightarrow\exists,$$

where $[\Lambda/x]e = pair_q(\Lambda_1,e_1)$ and $[\Lambda/x]G = \exists_{y:\tau_1}G_1$. It is easy to show that: (i) e is of the form $pair_q(\Lambda_2,e_2)$, where $[\Lambda/x]\Lambda_2=\Lambda_1$ and $[\Lambda/x]e_2 = e_1$; and (ii) G is of the form $\exists_{y:\tau_1}G_2$, where $[\Lambda/x]G_2 = G_1$. The identities below hold:

$$\begin{aligned}&[\Lambda_1/y]G_1 = [[\Lambda/x]\Lambda_2/y]([\Lambda/x]G_2) = [\Lambda/x]([\Lambda_2/y]G_2);\\ &[\Lambda/x]e_2 = e_1.\end{aligned}$$

So, by the I.H., there is a derivation of

$$\Sigma;\Delta_1,x=_{def}\Lambda:\tau,\Delta_2 \Rightarrow e_2:[\Lambda_2/y]G_2.$$

Since $\vdash \Sigma;\Delta_1,x=_{def}\Lambda:\tau,\Delta_2\ basis$ is derivable, by using Lemma 4.3, from the derivation of

$$\Sigma;\Delta_1,[\Lambda/x]\Delta_2 \vdash [\Lambda/x]\Lambda_2:\tau_1,$$

we may construct a derivation of

$$\Sigma;\Delta_1,x=_{def}\Lambda:\tau,\Delta_2\vdash\Lambda_2:\tau_1.$$

Thus, the following derivation may be formed:

$$\frac{\Sigma;\Delta_1,x=_{def}\Lambda:\tau,\Delta_2 \Rightarrow e_2:[\Lambda_2/y]G_2 \quad \Sigma;\Delta_1,x=_{def}\Lambda:\tau,\Delta_2\vdash\Lambda_2:\tau_1}{\Sigma;\Delta_1,x=_{def}\Lambda:\tau,\Delta_2 \Rightarrow pair_q(\Lambda_2,e_2):\exists_{y:\tau_1}G_2}\Rightarrow\exists$$

The case where the last step of π is a rule $\forall\Rightarrow$ follows by similar arguments. The other cases follow directly from the I.H.. □

Theorem 4.1 *Let $\Sigma; \Delta \vdash G\ gf$ be derivable in HH'. Then, the sequent $\Sigma; \Delta \Rightarrow e : G$ is derivable in HH' iff the sequent $\Sigma; \aleph(\Delta) \Rightarrow [\Delta]e : [\Delta]G$ is derivable in HH.*

Proof: The proof follows by induction on the number of simple definitions in Δ.

Case Δ contains no simple definitions. Then, by Lemma 4.1, $\Sigma; \Delta \Rightarrow e : G$ is derivable in HH' iff $\Sigma; \mu(\Delta) \Rightarrow e : [\Delta]G$ is derivable in HH. The proof of this case is concluded by observing that, for Δ contains no simple definitions: (i) $\aleph(\Delta) = \mu(\xi(\Delta)) = \mu(\Delta)$; and (ii) $[\Delta]e = e$.

Case Δ contains $n > 0$ simple definitions, Δ may be written as $\Delta_1, x =_{def} \Lambda : \tau, \Delta_2$, where Δ_2 contains no simple definitions.

(i) By Lemma 4.4, if

$$\Sigma; \Delta_1, x =_{def} \Lambda : \tau, \Delta_2 \Rightarrow e : G$$

is derivable in HH' then

$$\Sigma; \Delta_1, [\Lambda/x]\Delta_2 \Rightarrow [\Lambda/x]e : [\Lambda/x]G$$

is derivable in HH' .

(ii) By Lemma 4.5, for $\Sigma; \Delta \vdash G\ gf$ is derivable, if

$$\Sigma; \Delta_1, [\Lambda/x]\Delta_2 \Rightarrow [\Lambda/x]e : [\Lambda/x]G$$

is derivable in HH' then

$$\Sigma; \Delta_1, x =_{def} \Lambda : \tau, \Delta_2 \Rightarrow e : G$$

is derivable in HH'.

So, from (i) and (ii), since $(\Delta_1, [\Lambda/x]\Delta_2)$ has one simple definition fewer than $(\Delta_1, x =_{def} \Lambda : \tau, \Delta_2)$, by the I.H.,

$$\Sigma; \Delta_1, [\Lambda/x]\Delta_2 \Rightarrow [\Lambda/x]e : [\Lambda/x]G$$

is derivable in HH' iff

$$\Sigma; \aleph(\Delta_1, [\Lambda/x]\Delta_2) \Rightarrow [\Delta_1, [\Lambda/x]\Delta_2][\Lambda/x]e : [\Delta_1, [\Lambda/x]\Delta_2][\Lambda/x]G$$

is derivable in HH. For concluding the proof it suffices to observe that the following identities hold:

$$\begin{aligned}
&\aleph(\Delta_1, [\Lambda/x]\Delta_2) = \aleph(\Delta_1, x =_{def} \Lambda : \tau, \Delta_2);\\
&[\Delta_1, [\Lambda/x]\Delta_2][\Lambda/x]e = [\Delta_1, x =_{def} \Lambda : \tau, \Delta_2]e;\\
&[\Delta_1, [\Lambda/x]\Delta_2][\Lambda/x]G = [\Delta_1, x =_{def} \Lambda : \tau, \Delta_2]G,
\end{aligned}$$

since Δ_2 contains no simple definitions. □

Theorem 4.1 gives a means to interpret a logic programming language based on the calculus HH' into HOPLP. Consider the programming language HOPLPD based on HH' defined as follows.

- A *program* is a pair $\langle \Sigma, \Delta \rangle$, usually written $\Sigma; \Delta$, where Σ is a signature and Δ is a HH'-context; $\Sigma; \Delta$ is a *well-formed* program iff $\vdash \Sigma; \Delta$ *basis* is derivable in HH'.

- A *goal* is a G-formula; G is a *well-formed* goal w.r.t. a program $\Sigma; \Delta$ iff $\Sigma; \Delta \vdash G\ gf$ is derivable in HH'.

- A goal G is *achievable* w.r.t. a program $\Sigma; \Delta$ iff there exists a proof-term e s.t. $\Sigma; \Delta \Rightarrow e : G$ is derivable in HH'; the proof-term e is called a *witness* for the achievement of G w.r.t $\Sigma; \Delta$.

Theorem 4.1 guarantees that for every witness for the achievement of $[\Delta]G$ w.r.t. $\Sigma; \Delta$ in HOPLP there is a witness for the achievement of G w.r.t. $\Sigma; \Delta$ in HOPLPD. Now, a *complete set S of witnesses* for the achievement of a goal G w.r.t. a program $\Sigma; \Delta$ in HOPLPD is defined as a maximal set s.t.: (i) the elements of S are uniform linear focused proof-terms e s.t. $\Sigma; \Delta \Rightarrow e : G$ is derivable in HH' ; (ii) no two members e_1, e_2 of S are such that $[\Delta]e_1$ and $[\Delta]e_2$ are λ-convertible. Then HOPLP and HOPLPD may be regarded, in logical terms, as essentially the same language. The differences between the two languages are the mechanisms provided for writing programs and goals, e.g. HOPLPD allows an abbreviation mechanism for terms of simple type whereas HOPLP does not have such mechanism.

The programming language LeFun, as described in [AKN89], is a language that integrates logic and functional programming. LeFun provides a mechanism for simple definitions that is essentially the same as the mechanism for simple definitions of HOPLPD. In LeFun a program is essentially a list of logical formulae together with a list of definitions of the form $f =_{def} t$, where f is an identifier and t is a λ-term. The computation mechanism is called *residuation*, which is essentially a form of resolution, where not all unification problems on λ-terms are solved; those unification problems involving function application to arguments that are not fully instantiated are left to be verified as constraints.

4.3 The calculi HH^{cut} and $HH^{cut'}$

So far, the achievement of a goal G w.r.t. a program P has been considered to be a search for a special form of derivation of a sequent $P \Rightarrow G$ in calculi having no rule similar to Gentzen's *cut* rule [Gen35]. The *cut* rule in Gentzen's LJ may be written in the form:

$$\frac{\Gamma \Rightarrow C \quad \Gamma, C \Rightarrow B}{\Gamma \Rightarrow B}\ cut,$$

where Γ is a set of formulae and B, C are formulae. The formula C is called the *cut formula.* This rule may be read as follows. For proving that B is a logical consequence of Γ it suffices to prove that there exists a C s.t. C is a logical consequence of Γ and B is a logical consequence of Γ, C.

Using *cuts* for constructing derivations allows much shorter derivations of some judgements, see *e.g.* [WW92]. In [Boo84] are shown formulae whose shortest cut-free derivations are exponentially longer than their derivations using *cuts*.

The problem of using a *cut* rule in proof-search is to decide when and how a *cut* rule should be applied; in other words, what the adequate lemmas are to use in proving a theorem and when they should be applied. Usually, the lemmas are established based on experience.

In logic programming lemmas should not be established during proof-search. Instead, the programmer should know what lemmas may be useful and define names for the proofs of those lemmas. Then, during the search for a proof of a formula these formulae, previously established, may be used several times without having to be proved. In parallel to a mathematician who wants to prove a theorem, in order to achieve a goal, firstly some lemmas are proved and then those lemmas are combined to achieve the initial goal.

In this thesis the ideas described above are realised in the integrated logical and functional programming language LFPL, defined in Sec. 4.6. The semantics of LFPL is described in terms of the proof theory of the calculus HH^{def}, defined in Sec. 4.5. The language LFPL may be interpreted into HOPLP by means of an interpretation of HH^{def} into HH that essentially corresponds to a process of *cut elimination*, as shown in Sec. 4.5.

This section defines the calculi HH^{cut} and $HH^{cut'}$; they are essentially extensions of HH and HH', respectively, with a *cut* rule. These calculi are used as intermediate calculi in the interpretation of HH^{def} into HH described in Sec. 4.5. This interpretation is also used in relating the programming languages LFPL and HOPLP. The remainder of this section defines the calculi HH^{cut} and $HH^{cut'}$ and presents some results needed for interpreting HH^{def} into HH.

As noted above, the calculus HH^{cut} is an extension of HH with a *cut* rule. The *cut* rule may only allow cut formulae which are simultaneously H-formulae and G-formulae, otherwise one or more premises of this rule would be ill-formed sequents.

All the classes of objects used in HH are also used in HH^{cut}; their definitions are the same except for the class of proof-terms e. The proof-terms e of HH^{cut} are all the proof-terms of HH together with the proof-terms that may obtained by the rule: if e_1,e_2 are proof-terms of HH^{cut} then $let\ x = e_1\ in\ e_2$ is a proof-term of HH^{cut}, where x is a variable. Proof-terms of the form $let\ x = e_1\ in\ e_2$ are used to annotate *cut* rules. In proof-terms of the form $let\ x = e_1\ in\ e_2$, x is a *binder* of *scope* e_2; any occurrence of x in e_2 is said to be *bound*. As before, proof-terms are equal up to renaming of bound variables.

The operation $[\Lambda/x]e$ of substitution of a variable by a term of simple type on HH^{cut} proof-terms is defined as for HH, with the following extra case to deal with the *let* constructor:

$$[\Lambda/x](let\ x_1 = e\ in\ e_1) =_{def} (let\ x_1 = [\Lambda/x]e\ in\ [\Lambda/x]e_1),\ x \neq x_1,\ x_1 \notin \Lambda.$$

The class I of *intersection formulae* is defined by the following grammar:

$$I ::= A \mid I \wedge I \mid I \supset I \mid \forall_{x:\tau} I.$$

The class of I-formulae is the maximal subclass s.t. it is simultaneously a subclass of H-formulae and a subclass of G-formulae.

The forms of judgement in HH^{cut} are the same as those in HH. The derivable judgements of HH^{cut} are the same as those of HH except for sequents. For constructing derivations of sequents in HH^{cut} all the rules of HH are allowed, as well as the *cut* rule, presented in Fig. 4.5.

$$\frac{\Sigma; \Delta \Rightarrow e : I \quad \Sigma; \Delta, x : I \Rightarrow e_1 : G}{\Sigma; \Delta \Rightarrow let\ x = e\ in\ e_1 : G}\ cut,\ x \notin \Delta$$

Figure 4.5: Cut rule of HH^{cut}.

The result below shows that HH^{cut} is an extension of HH .

Lemma 4.6 *If* $\Sigma; \Delta \Rightarrow e : G$ *is derivable in* HH *then* $\Sigma; \Delta \Rightarrow e : G$ *is derivable in* HH^{cut}.

Proof: The proof is by induction on the structure of the derivation of $\Sigma; \Delta \Rightarrow e : G$ in HH. Notice that all the rules allowed in HH are also allowed in HH^{cut}. □

Now a result converse to Lemma 4.6 is addressed, i.e. a mapping from derivable sequents in HH^{cut} into derivable sequents in HH.

Let HH^{cut° be the calculus obtained from HH^{cut} by replacing the conversion rules and axioms by the new form of axiom, $axiom - conv$, shown in Fig. 4.6.

$$\frac{\vdash \Sigma; \Delta, x : A\ basis \quad \Sigma \vdash A \equiv A_1}{\Sigma; \Delta, x : A \Rightarrow x : A_1}\ axiom - conv$$

Figure 4.6: The rule $axiom - conv$.

The following result is an analogue to Theorem 3.21 relating HH and HH°.

Theorem 4.2 *A sequent is derivable in* HH^{cut} *iff it is derivable in* HH^{cut°.

Proof: If a sequent is derivable in HH^{cut° then it is derivable in HH^{cut} since $axiom - conv$ is derivable in HH^{cut}.

In order to prove the other implication, first is proved that: every HH^{cut}-derivation π of a sequent $\Sigma; \Delta \Rightarrow e : G$ may be transformed into a derivation whose conversion rules occur either immediately below axioms or other conversion rules. The proof is obtained, analogously to

Theorem 3.21, by showing that every conversion rule may be moved above both logical rules and *cuts*. For example, consider π to have the form below.

$$\dfrac{\dfrac{\overset{\pi_1}{\Sigma; \Delta \Rightarrow e_1 : I} \quad \overset{\pi_2}{\Sigma; \Delta, x : I \Rightarrow e_2 : G_1}}{\Sigma; \Delta \Rightarrow let\ x = e_1\ in\ e_2 : G_1}\ cut \quad \overset{\pi_3}{\Sigma \vdash G_1 \equiv G}}{\Sigma; \Delta \Rightarrow let\ x = e_1\ in\ e_2 : G}\ \equiv_r$$

Thus, the following derivation may be formed.

$$\dfrac{\overset{\pi_1}{\Sigma; \Delta \Rightarrow e_1 : I} \quad \dfrac{\overset{\pi_2}{\Sigma; \Delta, x : I \Rightarrow e_2 : G_1} \quad \overset{\pi_3}{\Sigma \vdash G_1 \equiv G}}{\Sigma; \Delta, x : I \Rightarrow e_2 : G}\ \equiv_r}{\Sigma; \Delta \Rightarrow let\ x = e_1\ in\ e_2 : G}\ cut$$

Secondly, following the proof of Theorem 3.21, it may be shown that: every sequent derivable in HH^{cut}, by using only axioms and conversion rules, is derivable in HH^{cut° by using *axiom − conv*. □

It may be readily shown that in HH^{cut° the proof-term of a derivation identifies uniquely the derivation, up to renaming of bound variables.

In order to prove cut elimination first is shown the admissibility of a form of *contraction* in HH. Since proof-terms are being used for representing derivations, the transformations on derivations required for showing admissibility of contraction are captured at the level of proof-terms. If proof-terms are regarded as the expressions of a functional language, see [Wad93], then the operation of contraction on proof-terms corresponds to substitution of a variable by another variable inside an expression.

The operation $\{x/x\}e$ of contraction on proof-terms is defined in Fig. 4.7.

Theorem 4.3 (Contraction Admissibility) *Let $\Sigma \vdash H \equiv H_1$ be derivable in HH. If $\Sigma; \Delta, x : H, x_1 : H_1 \Rightarrow e : G$ is derivable in HH then $\Sigma; \Delta, x : H \Rightarrow \{x/x_1\}e : G$ is derivable in HH.*

Proof: Let π be a derivation of $\Sigma; \Delta, x : H, x_1 : H_1 \Rightarrow e : G$ in HH. (By Theorem 3.21 it suffices to concentrate on HH°.) The proof follows by induction on the structure of π. Some cases are considered below.

Case last step of π is *axiom − conv* of the form:

$$\dfrac{\vdash \Sigma; \Delta, x : H, x_1 : H_1\ basis \quad \Sigma \vdash H_1 \equiv H_2}{\Sigma; \Delta, x : H, x_1 : H_1 \Rightarrow x_1 : H_2}\ axiom - conv.$$

Then, from a derivation of the judgement

$$\vdash \Sigma; \Delta, x : H, x_1 : H_1\ basis$$

$$
\begin{aligned}
&\{x/x_1\}pair(e_1,e_2) =_{def} pair(\{x/x_1\}e_1, \{x/x_1\}e_2)\\
&\{x/x_1\}inl(e) =_{def} inl(\{x/x_1\}e)\\
&\{x/x_1\}inr(e) =_{def} inr(\{x/x_1\}e)\\
&\{x/x_1\}lambda(x_2.e) =_{def} lambda(x_2.\{x/x_1\}e),\ x \neq x_2,\ x_1 \neq x_2\\
&\{x/x_1\}pair_q(\Lambda, e) =_{def} pair_q(\Lambda, \{x/x_1\}e)\\
&\{x/x_1\}lambda_q(x_2.e) =_{def} lambda_q(x_2.\{x/x_1\}e),\ x \neq x_2,\ x_1 \neq x_2\\
&\{x/x_1\}x_1 =_{def} x\\
&\{x/x_1\}x_2 =_{def} x_2,\ x_1 \neq x_2\\
&\{x/x_1\}splitl(x_1, x_2.e) =_{def} splitl(x, x_2.\{x/x_1\}e),\ x \neq x_2,\ x_1 \neq x_2\\
&\{x/x_1\}splitl(x_2, x_3.e) =_{def} splitl(x_2, x_3.\{x/x_1\}e),\ x_1 \neq x_2,\ x \neq x_3,\ x_1 \neq x_3\\
&\{x/x_1\}splitr(x_1, x_2.e) =_{def} splitr(x, x_2.\{x/x_1\}e),\ x \neq x_2,\ x_1 \neq x_2\\
&\{x/x_1\}splitr(x_2, x_3.e) =_{def} splitr(x_2, x_3.\{x/x_1\}e),\ x_1 \neq x_2,\ x \neq x_3,\ x_1 \neq x_3\\
&\{x/x_1\}apply(x_1, e, x_2.e_1) =_{def} apply(x, \{x/x_1\}e, x_2.\{x/x_1\}e_1),\ x_1 \neq x_2,\ x \neq x_2\\
&\{x/x_1\}apply(x_2, e, x_3.e_1) =_{def} apply(x_2, \{x/x_1\}e, x_3.\{x/x_1\}e_1),\ x_1 \neq x_2,\ x \neq x_3,\ x_1 \neq x_3\\
&\{x/x_1\}apply_q(x_1, \Lambda, x_2.e) =_{def} apply_q(x, \Lambda, x_2.\{x/x_1\}e),\ x \neq x_2,\ x_1 \neq x_2\\
&\{x/x_1\}apply_q(x_2, \Lambda, x_3.e) =_{def} apply_q(x_2, \Lambda, x_3.\{x/x_1\}e),\ x \neq x_2,\ x \neq x_3,\ x_1 \neq x_3
\end{aligned}
$$

Figure 4.7: Contraction on proof-terms.

a derivation of $\vdash \Sigma; \Delta, x : H\ basis$ may be easily constructed. From the derivations of $\Sigma \vdash H \equiv H_1$ and $\Sigma \vdash H_1 \equiv H_2$ it is simple to construct a derivation of $\Sigma \vdash H \equiv H_2$. Thus the following derivation may be formed:

$$\frac{\vdash \Sigma; \Delta, x : H\ basis \quad \Sigma \vdash H \equiv H_2}{\Sigma; \Delta, x : H \Rightarrow x : H_2}\ axiom - conv.$$

Note that $\{x/x_1\}x_1 = x$.

Case last step of π is of the form:

$$\frac{\Sigma; \Delta, x : H, x_1 : H_1, x_2 : H_2 \Rightarrow e : G}{\Sigma; \Delta, x : H, x_1 : H_1 \Rightarrow lambda(x_2.e) : H_2 \supset G} \Rightarrow\supset,$$

where $x_2 \notin (\Delta, x : H, x_1 : H_1)$. By the I.H., there is a derivation of

$$\Sigma; \Delta, x : H, x_2 : H_2 \Rightarrow \{x/x_1\}e : G.$$

Thus, the derivation below may be formed.

$$\frac{\Sigma; \Delta, x : H, x_2 : H_2 \Rightarrow \{x/x_1\}e : G}{\Sigma; \Delta, x : H \Rightarrow lambda(x_2.\{x/x_1\}e) : H_2 \supset G} \Rightarrow\supset$$

Note that $\{x/x_1\}lambda(x_2.e) = lambda(x_2.\{x/x_1\}e)$, since $x \neq x_2$ and $x_1 \neq x_2$.

Case H_1 is of the form $G_2 \supset H_2$ and the last step of π is of the form:

$$\frac{\Sigma; \Delta, x : H, x_1 : G_2 \supset H_2 \Rightarrow e : G_2 \quad \Sigma; \Delta, x : H, x_1 : G_2 \supset H_2, x_2 : H_2 \Rightarrow e_1 : G}{\Sigma; \Delta, x : H, x_1 : G_2 \supset H_2 \Rightarrow apply(x_1, e, x_2.e_1) : G} \supset\Rightarrow,$$

where $x_2 \notin (\Delta, x : H, x_1 : G_2 \supset H_2)$.

Since the judgement $\Sigma \vdash H \equiv H_1$ is derivable, it must be the case that H is of the form $G_3 \supset H_3$ and the judgements $\Sigma \vdash G_3 \equiv G_2$ and $\Sigma \vdash H_3 \equiv H_2$ are derivable. By the I.H., there are derivations of the sequents:

$$\Sigma; \Delta, x : G_3 \supset H_3 \Rightarrow \{x/x_1\}e : G_2;$$
$$\Sigma; \Delta, x : G_3 \supset H_3, x_2 : H_2 \Rightarrow \{x/x_1\}e_1 : G.$$

Thus, Derivations 4.3 and 4.4 may be formed.

$$\frac{\Sigma; \Delta, x : G_3 \supset H_3 \Rightarrow \{x/x_1\}e : G_2 \quad \Sigma \vdash G_2 \equiv G_3}{\Sigma; \Delta, x : G_3 \supset H_3 \Rightarrow \{x/x_1\}e : G_3} \equiv_r \tag{4.3}$$

$$\frac{\Sigma; \Delta, x : G_3 \supset H_3, x_2 : H_2 \Rightarrow \{x/x_1\}e_1 : G \quad \Sigma \vdash H_2 \equiv H_3}{\Sigma; \Delta, x : G_3 \supset H_3, x_2 : H_3 \Rightarrow \{x/x_1\}e_1 : G} \equiv_l \tag{4.4}$$

By using $\supset\Rightarrow$, the Derivations 4.3 and 4.4 may be put together to form a derivation of the sequent

$$\Sigma; \Delta, x : G_3 \supset H_3 \Rightarrow apply(x, \{x/x_1\}e, x_2.\{x/x_1\}e_1) : G.$$

Note that $\{x/x_1\}apply(x_1, e, x_2.e_1) = apply(x, \{x/x_1\}e, x_2.\{x/x_1\}e_1)$, since $x \neq x_2$ and $x_1 \neq x_2$.

Proofs for the cases where π is of any other form follow by similar arguments. □

In Fig. 4.8 is presented a list of rules on HH^{cut}-proof-terms; these rules encode the transformations on derivations used below for proving admissibility of the *cut* rule.

Definition 4.1 (RS_{cut}) *RS_{cut} is the rewriting system on HH^{cut}-proof-terms consisting of the rules in Fig. 4.8. The rewrite relation induced by RS_{cut} is called $\triangleright_{cut}$. A proof-term e_1 is* reducible *by RS_{cut} to a proof-term e_2 if the pair (e_1, e_2) is in the transitive closure of $\triangleright_{cut}$.*

Theorem 4.4 (Cut Elimination) *Let $\Sigma; \Delta \Rightarrow e : G$ be derivable in HH^{cut}. Then, there exists e_1 s.t. $\Sigma; \Delta \Rightarrow e_1 : G$ is derivable in HH and either e is equal to e_1 or e is reducible by RS_{cut} to e_1.*

Proof: The proof sketched below follows closely the proof in [Dra88], showing admissibility of the cut rule in the system GHPC.

Let π be a HH^{cut}-derivation of $\Sigma; \Delta \Rightarrow e : G$. Then, If π has no *cuts* then it may be easily proved that $\Sigma; \Delta \Rightarrow e : G$ is derivable in HH.

Otherwise, the proof follows by induction on the number of *cuts* of maximal *degree* in π, where the *degree* of a *cut*

$$\frac{\overset{\pi_1}{\Sigma_1; \Delta_1 \Rightarrow e_1 : I} \quad \overset{\pi_2}{\Sigma_1; \Delta_1, x : I \Rightarrow e_2 : G_1}}{\Sigma_1; \Delta_1 \Rightarrow let\ x = e_1\ in\ e_2 : G_1}\ cut$$

$$
\begin{array}{l}
let\ x = e\ in\ pair(e_1, e_2) \triangleright pair(let\ x = e\ in\ e_1, let\ x = e\ in\ e_2) \\
let\ x = e\ in\ inl(e_1) \triangleright inl(let\ x = e\ in\ e_1) \\
let\ x = e\ in\ inr(e_1) \triangleright inr(let\ x = e\ in\ e_1) \\
let\ x = e\ in\ lambda(x_1.e_1) \triangleright lambda(x_1.let\ x = e\ in\ e_1),\ x \neq x_1,\ x_1 \notin e \\
let\ x = e\ in\ pair_q(\Lambda, e_1) \triangleright pair_q(\Lambda, let\ x = e\ in\ e_1) \\
let\ x = e\ in\ lambda_q(x_1.e_1) \triangleright lambda_q(x_1.let\ x = e\ in\ e_1),\ x \neq x_1,\ x_1 \notin e \\
let\ x = x_1\ in\ e \triangleright \{x_1/x\}e \\
let\ x = splitl(x_1, x_2.e)\ in\ e_1 \triangleright splitl(x_1, x_2.let\ x = e\ in\ e_1),\ x \neq x_2,\ x_2 \notin e_1 \\
let\ x = splitr(x_1, x_2.e)\ in\ e_1 \triangleright splitr(x_1, x_2.let\ x = e\ in\ e_1),\ x \neq x_2,\ x_2 \notin e_1 \\
let\ x = apply(x_1, e, x_2.e_1)\ in\ e_2 \triangleright apply(x_1, e, x_2.let\ x = e_1\ in\ e_2),\ x \neq x_2,\ x_2 \notin e_2 \\
let\ x = apply_q(x_1, \Lambda, x_2.e)\ in\ e_1 \triangleright apply_q(x_1, \Lambda, x_2.let\ x = e\ in\ e_1),\ x \neq x_2,\ x_2 \notin e_1
\end{array}
$$

$$
\begin{array}{l}
let\ x = e\ in\ x_1 \triangleright x_1,\ x \neq x_1 \\
let\ x = e\ in\ splitl(x_1, x_2.e_1) \triangleright splitl(x_1, x_2.let\ x = e\ in\ e_1),\ x \neq x_1,\ x \neq x_2,\ x_2 \notin e \\
let\ x = e\ in\ splitr(x_1, x_2.e_1) \triangleright splitr(x_1, x_2.let\ x = e\ in\ e_1),\ x \neq x_1,\ x \neq x_2,\ x_2 \notin e \\
let\ x = e\ in\ apply(x_1, e_1, x_2.e_2) \triangleright \\
\qquad apply(x_1, let\ x = e\ in\ e_1, x_2.let\ x = e\ in\ e_2),\ x \neq x_1,\ x \neq x_2,\ x_2 \notin e \\
let\ x = e\ in\ apply_q(x_1, \Lambda, x_2.e_1) \triangleright \\
\qquad apply_q(x_1, \Lambda, x_2.let\ x = e\ in\ e_1),\ x \neq x_1,\ x \neq x_2,\ x_2 \notin e
\end{array}
$$

$$
\begin{array}{l}
let\ x = pair(e_1, e_2)\ in\ splitl(x, x_1.e_3) \triangleright \\
\qquad let\ x_1 = e_1\ in\ (let\ x = pair(e_1, e_2)\ in\ e_3),\ x \neq x_1,\ x_1 \notin e_1,\ x_1 \notin e_2 \\
let\ x = pair(e_1, e_2)\ in\ splitr(x, x_1.e_3) \triangleright \\
\qquad let\ x_1 = e_2\ in\ (let\ x = pair(e_1, e_2)\ in\ e_3),\ x \neq x_1,\ x_1 \notin e_1,\ x_1 \notin e_2 \\
let\ x = lambda(x_3.e_3)\ in\ apply(x, e_1, x_2.e_2) \triangleright \\
\qquad let\ x_2 = (let\ x_3 = (let\ x = lambda(x_3.e_3)\ in\ e_1)\ in\ e_3)\ in\ (let\ x = lambda(x_3.e_3)\ in\ e_2), \\
\qquad x \neq x_2,\ x_2 \notin e_3 \\
let\ x = lambda_q(x_1.e)\ in\ apply_q(x, \Lambda, x_2.e_1) \triangleright \\
\qquad let\ x_2 = [\Lambda/x_1]e\ in\ (let\ x = lambda_q(x_1.e)\ in\ e_1),\ x \neq x_2,\ x_2 \notin e
\end{array}
$$

Figure 4.8: Rules of the rewriting system RS_{cut}.

is defined as the triple $\langle c, h_1, h_2 \rangle$, where: c is the logical complexity of the cut formula I, i.e. the number of $\forall, \wedge, \supset$ in I; h_1 is the height of π_1; and h_2 is the height of π_2. (The lexicographic ordering from the left is used to compare degrees of *cuts*.) Let us assume that the *cut* rule above is a *cut* of maximal degree in π. We consider below some forms for π_1 and π_2 and show how to transform such derivations into derivations whose *cuts* are of *lower degree*.

Case π is of the form:

$$\dfrac{\dfrac{\vdash \Sigma_1; \Delta_2, x_1 : H \ basis \quad \Sigma_1 \vdash H \equiv I}{\Sigma_1; \Delta_2, x_1 : H \Rightarrow x_1 : I}\ axiom-conv \qquad \Sigma_1; \Delta_2, x_1 : H, x : I \Rightarrow e_2 : G_1}{\Sigma_1; \Delta_2, x_1 : H \Rightarrow let\ x = x_1\ in\ e_2 : G_1}\ cut.$$

By Theorem 4.3, for the judgements $\Sigma_1; \Delta_2, x_1 : H, x : I \Rightarrow e_2 : G_1$ and $\Sigma_1 \vdash H \equiv I$ are derivable, there is a HH-derivation of the sequent

$$\Sigma_1; \Delta_2, x_1 : H \Rightarrow \{x_1/x\}e_2 : G_1$$

and $let\ x = x_1\ in\ e_2 \rhd_{cut} \{x_1/x\}e_2$.

The cases where the last steps of π_1 and π_2 do not both introduce the cut formula I follow easily by induction. The more interesting cases are when both last steps introduce the cut formula. We consider the case where I is of the form $I_1 \supset I_2$, i.e. π_1 is of the form

$$\dfrac{\overset{\pi_3}{\Sigma_1; \Delta_1, x_3 : I_1 \Rightarrow e_3 : I_2}}{\Sigma_1; \Delta_1 \Rightarrow lambda(x_3.e_3) : I_1 \supset I_2}\ \Rightarrow\supset,$$

where $x_3 \notin \Delta_1$, and π_2 is of the form

$$\dfrac{\overset{\pi_4}{\Sigma_1; \Delta_1, x : I_1 \supset I_2 \Rightarrow e_4 : I_1} \quad \overset{\pi_5}{\Sigma_1; \Delta_1, x : I_1 \supset I_2, x_2 : I_2 \Rightarrow e_5 : G_1}}{\Sigma_1; \Delta_1, x : I_1 \supset I_2 \Rightarrow apply(x, e_4, x_2.e_5) : G_1}\ \supset\Rightarrow,$$

where $x_2 \notin (\Delta_1, x : I_1 \supset I_2)$. In this case π is of the form of Derivation 4.5.

$$\dfrac{\overset{\pi_1}{\Sigma_1; \Delta_1 \Rightarrow lambda(x_3.e_3) : I_1 \supset I_2} \quad \overset{\pi_2}{\Sigma_1; \Delta_1, x : I_1 \supset I_2 \Rightarrow apply(x, e_4, x_2.e_5) : G_1}}{\Sigma_1; \Delta_1 \Rightarrow let\ x = lambda(x_3.e_3)\ in\ apply(x, e_4, x_2.e_5) : G_1}\ cut \tag{4.5}$$

Derivations 4.6 and 4.7 may be formed.

$$\dfrac{\dfrac{\overset{\pi_3}{\Sigma_1; \Delta_1, x_3 : I_1 \Rightarrow e_3 : I_2}}{\Sigma_1; \Delta_1 \Rightarrow lambda(x_3.e_3) : I_1 \supset I_2}\ \Rightarrow\supset \quad \overset{\pi_4}{\Sigma_1; \Delta_1, x : I_1 \supset I_2 \Rightarrow e_4 : I_1}}{\Sigma_1; \Delta_1 \Rightarrow let\ x = lambda(x_3.e_3)\ in\ e_4 : I_1}\ cut \tag{4.6}$$

$$\dfrac{\dfrac{\overset{\pi_6}{\Sigma_1; \Delta_1, x_3 : I_1, x_2 : I_2 \Rightarrow e_3 : I_2}}{\Sigma_1; \Delta_1, x_2 : I_2 \Rightarrow lambda(x_3.e_3) : I_1 \supset I_2}\ \Rightarrow\supset \quad \overset{\pi_5}{\Sigma_1; \Delta_1, x : I_1 \supset I_2, x_2 : I_2 \Rightarrow e_5 : G_1}}{\Sigma_1; \Delta_1, x_2 : I_2 \Rightarrow let\ x = lambda(x_3.e_3)\ in\ e_5 : G_1}\ cut \tag{4.7}$$

Derivation π_6 may be obtained from π_3 by weakening, since $x_2 \notin \Delta_1$, having the same height as π_3. The *cut* rules in Derivations 4.6 and 4.7 have degree lower than the *cut* rule in Derivation 4.5, since the heights of their right premises are at least one fewer than the height of π_2.

Applying the *cut* rule to the endsequents of Derivation 4.6 and π_3 one obtains Derivation 4.8.

$$\frac{\Sigma_1;\Delta_1 \Rightarrow \mathit{let}\ x = \mathit{lambda}(x_3.e_3)\ \mathit{in}\ e_4 : I_1 \quad \overset{\pi_3}{\Sigma_1;\Delta_1, x_3 : I_1 \Rightarrow e_3 : I_2}}{\Sigma_1;\Delta_1 \Rightarrow \mathit{let}\ x_3 = (\mathit{let}\ x = \mathit{lambda}(x_3.e_3)\ \mathit{in}\ e_4)\ \mathit{in}\ e_3 : I_2}\ \mathit{cut} \tag{4.8}$$

This *cut* rule has degree lower than *cut* rule in Derivation 4.5 since the logical complexity of I_1 is at least one fewer than the logical complexity of $I_1 \supset I_2$.

Applying a *cut* rule to the endsequents of Derivations 4.7, 4.8 one obtains a derivation of

$$\Sigma_1;\Delta_1 \Rightarrow \mathit{let}\ x_2 = (\mathit{let}\ x_3 = (\mathit{let}\ x = \mathit{lambda}(x_3.e_3)\ \mathit{in}\ e_4)\ \mathit{in}\ e_3)\ \mathit{in}\ (\mathit{let}\ x = \mathit{lambda}(x_3.e_3)\ \mathit{in}\ e_5) : G_1.$$

This *cut* rule has degree lower than the *cut* rule in Derivation 4.5 since the logical complexity of I_2 is at least one fewer than the logical complexity of $I_1 \supset I_2$. So, by the I.H., the proof-term of the sequent above reduces by RS_{cut} to a proof-term e_6, and so does e, s.t. $\Sigma_1;\Delta_1 \Rightarrow e_6 : G_1$ is derivable in HH.

□

From the proof above, we may extract an argument for weak normalisation of well-typed proof-terms in RS_{cut}, *i.e.* every proof-term of a derivable sequent in HH^{cut} is reducible in RS_{cut} to the proof-term of a derivable sequent in HH. (See the Appendix A of [Dra88] for a strong normalisation argument of a set of rules essentially including those encoded by the rules of RS_{cut}.)

We have shown that if $\Sigma;\Delta \Rightarrow e : G$ is derivable in HH^{cut} then there exists e_1, possibly more than one, s.t. $\Sigma;\Delta \Rightarrow e_1 : G$ is derivable in HH. Below, we write $cut(e)$ meaning a e_1 in these conditions; e_1 may be thought of as the cut-free form of e when a particular strategy for choosing redexes is followed.

The remainder of this section defines the calculus $HH^{cut'}$ and shows how $HH^{cut'}$ is related to HH^{cut}.

The calculus $HH^{cut'}$ is an extension of HH' with a *cut* rule. Essentially, the process of going from HH' to $HH^{cut'}$ is the same as that described above to go from HH to HH^{cut}. The classes of objects used in $HH^{cut'}$ are those used in HH' together with the class of I-formulae. The definitions of the classes of objects used in $HH^{cut'}$ are the same as for HH' with the exception of proof-terms e that allow the extra constructor *let* mentioned above. The forms of judgement of $HH^{cut'}$ are the same as those of HH'. The derivable judgements of $HH^{cut'}$ are the same as those of HH' except for sequents. For deriving sequents in $HH^{cut'}$ the *cut* rule shown below is also allowed.

$$\frac{\Sigma;\Delta_1 \Rightarrow e : I \quad \Sigma;\Delta_1, x : I, \Delta_2 \Rightarrow e_1 : G}{\Sigma;\Delta_1,\Delta_2 \Rightarrow \mathit{let}\ x = e\ \mathit{in}\ e_1 : G}\ \mathit{cut}$$

The differences between the *cut* rule for $HH^{cut'}$ and the *cut* rule for HH^{cut} are due to the difference between contexts in $HH^{cut'}$, which are lists, and contexts in HH^{cut}, which are sets. A result analogous to Theorem 4.1, relating HH' and HH, holds between $HH^{cut'}$ and HH^{cut}.

Theorem 4.5 *Let $\Sigma; \Delta \vdash G\ gf$ be derivable in $HH^{cut'}$. Then, the sequent $\Sigma; \Delta \Rightarrow e : G$ is derivable in $HH^{cut'}$ iff the sequent $\Sigma; \aleph(\Delta) \Rightarrow [\Delta]e : [\Delta]G$ is derivable in HH^{cut}.*

Proof: A proof for this result may be obtained by similar arguments to those used for proving Theorem 4.1. □

Using Lemma 4.6 and Theorem 4.5 one may show $HH^{cut'}$ to be an extension of HH. Combining theorems 4.5 and 4.4 one shows that $HH^{cut'}$ is conservative w.r.t. HH. Thus, $HH^{cut'}$ is a conservative extension of HH.

4.4 Definitions in Integrated Logical and Functional Programming and Σ-types

Consider logic and functional programming from a type-theoretic perspective. Traditionally, functional programming is based on non-dependent types; thus, the logical contents that may be attached to a function is fairly simple. However, in the presence of richer type theories more information may be attached to functions.

In the context of integrated logical and functional programming, if more elaborate types are attached to the functions being defined, during the search for a proof it may suffice to look at the information contained in a type attached to a function, rather than having to use the function itself. So, in the presence of a definition there may be two alternative ways of achieving a goal: (i) by replacing occurrences of the definiendum by its definiens; or (ii) by using the type attached to the definiendum. In the first case a definition is merely being used as an abbreviation mechanism, which corresponds to the traditional use of definitions in functional programming, *i.e.* the definition mechanism introduced in Section 4.2. In the second case the logical properties of the definiendum, i.e. the type attached to the definiendum, may be used to achieve a goal, possibly providing shorter ways of achieving the goal.

The idea above can be realised in a type theory like the Extended Calculus of Constructions [Luo94], where a Σ-type may be used to attach a logical specification to a term of simple type. Traditionally, in dependent type theories [ML84, CH88] terms and proofs are identified. However, in this thesis terms of simple type and proof-terms are kept separate. The presence of Σ-types allows us to maintain the distinction in a structured way, following [MB93].

The notation generally used for Σ-types is $\Sigma_{x:T_1}T_2$, where T_1, and T_2 are types. Given a Σ-type $\Sigma_{x:T_1}T_2$, an element of this type is a pair (t_1, t_2) s.t. t_1 is of type T_1 and t_2 is of type

$[t_1/x]T_2$. In this thesis, the set D of Σ-types allowed is as follows:

$$D ::= I \mid \Sigma_{x:\tau} D,$$

where τ is a simple type and I is an I-formula, as defined in the previous section. (Logical formulae are regarded as types). The operation of substitution $[\Lambda/x]D$ on D-formulae is defined as for H-formulae, when D is an I-formula, and is defined as $\Sigma_{y:\tau}[\Lambda/x]D_1$, when D is of the form $\Sigma_{y:\tau}D_1$ and $y \neq x$ and $y \notin \Lambda$. (As usual, in a formula $\Sigma_{y:\tau}D$, y is a *binder* whose *scope* is D and occurrences of y in D are called *bound*; D-formulae are considered equal up to renaming of bound variables.)

A *(elementary) definition of dependent type* is of the form:

$$x =_{def} e : D,$$

where x is a variable, e is a proof-term and D is a D-formula. Generally, when D is of the form $\Sigma_{y:\tau}I$, e is a pair $pair_q(\Lambda, e_1)$, where Λ is a term of simple type τ and e_1 is a proof-term of the formula $[\Lambda/y]I$.

Let us consider contexts Δ, extending the contexts used in HH', allowing definitions of dependent type. Let us consider the rule below, resembling the cut rule of $HH^{cut'}$, as a rule for dealing with definitions of dependent type in the context.

$$\frac{\Sigma; \Delta_1 \Rightarrow e : D \quad \Sigma; \Delta_1, x : D, \Delta_2 \Rightarrow e_1 : G}{\Sigma; \Delta_1, x =_{def} e : D, \Delta_2 \Rightarrow e_1 : G}\ def$$

The main difference between the *def* rule and the *cut* rule of $HH^{cut'}$ is the presence of a definition of dependent type in the conclusion sequent, whose type corresponds to the cut formula. (Note that D occurs in the succedent of the left premise and in the antecedent of the right premise. Also note that the proof-term of the conclusion sequent is the same as the proof-term of the right premiss. In the conclusion sequent, the information that x is defined as e is kept solely in the context.) Let us consider proof-search of a goal G w.r.t. a program $\Sigma; \Delta$ in hypothetical languages based on calculi containing the *cut* rule of $HH^{cut'}$ and the *def* rule. The *cut* rule of $HH^{cut'}$ may be applied at any point in the search, with the possibility of choosing any well-formed formula for cut formula. The *def* rule may also be applied at any point in the search, as long as there is at least one definition of dependent type in the program, but the formula used as cut formula must be the type of the definition being broken up.

Having a rule such as the *def* rule for dealing with definitions of dependent type requires rules to deal with Σ-types, both in the succedent and in the antecedent of sequents. A rule for dealing with Σ-types in the succedent can be easily obtained by interpreting a Σ-type as an existentially quantified formula. A rule for dealing with Σ-types in the antecedent is a bit more problematic. One possible form of defining such rule is by enlarging the form of proof-term annotations allowed. *Projections* may be allowed both in proof-terms annotating formulae in the program and in formulae themselves, as used in the preliminary exposition [Pin94] of these ideas for an integrated logical and functional language. Another alternative, that used in this thesis, is the use of *patterns* in place of projections.

Traditionally, the first and second projections are used to access the components of a pair. A *pattern* is a simple generalisation of a variable. A pattern allows the definition of names for the components of a pair. The set of *patterns* p is defined as follows:

$$p ::= x \mid (x, p),$$

where x ranges over variables. A pattern of the form $(x_1, (x_2, ...(x_n, x)...))$ is usually written as $(x_1, x_2, ..., x_n, x)$; the variables $x_1, x_2, ..., x_n$ are called its *simple type variables* and the variable x is called its *proof-term variable.*

The *definitions of dependent type* allowed in this thesis are of the form:

$$p =_{def} e : D.$$

For example, an elementary definition of the form

$$x =_{def} pair_q(\Lambda, e) : \Sigma_{y:\tau} I,$$

may now be written as the non-elementary definition:

$$(x_1, p) =_{def} pair_q(\Lambda, e) : \Sigma_{y:\tau} I,$$

defining a name x_1 for the term of simple type Λ.

A simple definition $x =_{def} \Lambda : \tau$ is said to be *implicit* in a definition of dependent type of the form $(x_1, p) =_{def} pair_q(\Lambda_1, e) : \Sigma_{y:\tau_1} D$ if either $x_1 = x$, $\Lambda_1 = \Lambda$ and $\tau_1 = \tau$ or $x =_{def} \Lambda : \tau$ is implicit in $p =_{def} e : [\Lambda_1/y]D$. The *definienda (of simple type) of a pattern p* are inductively defined as follows: (i) if p is a pattern of the form (x, p_1) then x is a definiendum of p and any definiendum of p_1 is also a definiendum of p; (ii) if p is a variable then p has no definienda.

A rule similar to the *def* rule shown above to deal with definitions of dependent type is the following rule:

$$\frac{\Sigma; \Delta_1 \Rightarrow e : D \quad \Sigma; \Delta_1, p : D, \Delta_2 \Rightarrow e_1 : G}{\Sigma; \Delta_1, p =_{def} e : D, \Delta_2 \Rightarrow e_1 : G}.$$

This rule requires patterns annotating formulae in the context. This rule is not invertible, since there may be implicit simple definitions in the conclusion which are not available in the premises. For avoiding these problems, the rule to deal with definitions of dependent type is split into two rules, described in Fig 4.9, one for definitions of Σ-type, that keeps available abbreviations for terms of simple type, and the other rule for definitions of I-type, similar to the *def* rule shown above.

Section 4.5 defines the calculus HH^{def}. This calculus allows definitions of dependent type of the form described in this section. HH^{def} also allows simple definitions not implicit in definitions of dependent type. The calculus HH^{def} is used in Section 4.6 to define a proof-theoretic semantics for an integrated logical and functional language.

$$\frac{\Sigma; \Delta_1, x =_{def} \Lambda : \tau, p =_{def} e : [x/y]D, \Delta_2 \Rightarrow e_1 : G}{\Sigma; \Delta_1, (x, p) =_{def} pair_q(\Lambda, e) : \Sigma_{y:\tau} D, \Delta_2 \Rightarrow e_1 : G}\ def_\Sigma$$

$$\frac{\Sigma; \Delta_1 \Rightarrow e : I \quad \Sigma; \Delta_1, x : I, \Delta_2 \Rightarrow e_1 : G}{\Sigma; \Delta_1, x =_{def} e : I, \Delta_2 \Rightarrow e_1 : G}\ def_I$$

Figure 4.9: Rules for definitions of dependent type.

4.5 The Calculus HH^{def}

This section defines the calculus HH^{def}, which may be thought of as an extension of HH' with the mechanism for definitions of dependent type described in Sec. 4.4. The calculus HH^{def} is used in Sec. 4.6 to define a proof-theoretic semantics for the integrated logical and functional programming language LFPL. In this section is shown an interpretation of derivable judgements of HH^{def} as derivable judgements of HH. The converse is also shown, i.e. how to interpret derivable judgements of HH as derivable judgements of HH^{def}.

Figure 4.10 presents the grammars of the several classes of objects used in HH^{def}. The judgements of the calculus HH^{def} are presented in Fig. 4.11.

$$\begin{array}{lcll}
\tau & ::= & s \mid \tau \to \tau & \text{(simple types)} \\
\Lambda & ::= & x \mid \lambda x : \tau.\Lambda \mid \Lambda\Lambda & (\lambda\text{-terms}) \\
\Sigma & ::= & \langle\rangle \mid \Sigma, x : \tau & \text{(signatures)} \\
H & ::= & A \mid H \wedge H \mid G \supset H \mid \forall_{x:\tau} H & \text{(hereditary Harrop formulae)} \\
G & ::= & A \mid G \wedge G \mid G \vee G \mid H \supset G \mid \exists_{x:\tau} G \mid \forall_{x:\tau} G & \text{(hereditary Harrop goals)} \\
I & ::= & A \mid I \wedge I \mid I \supset I \mid \forall_{x:\tau} I & \text{(intersection formulae)} \\
D & ::= & I \mid \Sigma_{x:\tau} D & \text{(definition types)} \\
e & ::= & pair(e, e) \mid inl(e) \mid inr(e) \mid lambda(x.e) & \\
 & \mid & pair_q(\Lambda, e) \mid lambda_q(x.e) \mid x \mid splitl(x, x.e) & \\
 & \mid & splitr(x, x.e) \mid apply(x, e, x.e) \mid apply_q(x, \Lambda, x.e) & \text{(proof-terms)} \\
p & ::= & x \mid (x, p) & \text{(patterns)} \\
\Delta & ::= & \langle\rangle \mid \Delta, x : H \mid \Delta, x =_{def} \Lambda : \tau \mid \Delta, p =_{def} e : D & \text{(programs)}
\end{array}$$

s ranges over the set $\mathcal{S}$ of primitive types; x ranges over the infinitely denumerable set $\mathcal{X}$ of variables; A ranges over the set of atomic formulae of HH.

Figure 4.10: Classes of objects of HH^{def}.

Note that although the sets of HH'-contexts and HH^{def}-contexts are not the same, the notation used for meta-variables ranging over these two sets of objects is the same, i.e. Δ possibly indexed.

(i)	$\vdash \Sigma\ signature$	(signatures)
(ii)	$\vdash \Sigma; \Delta\ basis$	(bases)
(iii)	$\Sigma; \Delta \vdash \Lambda : \tau$	(λ-terms of a type)
(iv)	$\Sigma; \Delta \vdash \Lambda \rhd_\tau \Lambda$	(one step reduction on λ-terms)
(v)	$\Sigma; \Delta \vdash \Lambda \rhd_\tau^* \Lambda$	(zero or more steps reduction on λ-terms)
(vi)	$\Sigma; \Delta \vdash \Lambda \equiv_\tau \Lambda$	(convertible λ-terms)
(vii)	$\Sigma; \Delta \vdash A\ af$	(atomic formulae)
(viii)	$\Sigma; \Delta \vdash H\ hf$	(hereditary Harrop formulae)
(ix)	$\Sigma; \Delta \vdash G\ gf$	(hereditary Harrop goals)
(x)	$\Sigma; \Delta \vdash A \equiv A$	(convertible atomic formulae)
(xi)	$\Sigma; \Delta \vdash H \equiv H$	(convertible H-formulae)
(xii)	$\Sigma; \Delta \vdash G \equiv G$	(convertible G-formulae)
(xiii)	$\Sigma; \Delta \Rightarrow e : G$	(proof-terms of a G-formula)

Figure 4.11: Judgement forms of HH^{def}.

The notation $x \notin \Delta$, when Δ is a HH^{def}-context, indicates that: (i) there is no formula H s.t. $x : H$ is in Δ; (ii) there is no simple definition of the form $x =_{def} \Lambda : \tau$ in Δ; (iii) there is no definition of dependent type in Δ of the form $(x_1, ..., x_n) =_{def} e : D$ s.t. $x = x_i$, for some $1 \leq i \leq n$, or x occurs freely in e.

The rules defining derivable judgements of HH^{def} of the forms (i) and (v)-(xii) are obtained from the rules of HH' defining the corresponding forms of judgement, where contexts are regarded as HH^{def}-contexts. The rules defining derivable judgements of HH^{def} of the forms (ii)-(iv) and (xii) are the rules obtained from the rules defining derivable judgements of the corresponding form in HH', by regarding contexts as HH^{def}-contexts, together with the extra rules shown in Fig. 4.12. The rules defining derivable sequents of HH^{def} are shown in Fig. 4.13.

The following meta-theoretical properties of HH^{def} are used in various places in the remainder of this thesis.

Proposition 4.1 *Let the sequent $\Sigma; \Delta \Rightarrow e : G$ be derivable in HH^{def}. Then, the judgements $\vdash \Sigma; \Delta\ basis$ and $\Sigma; \Delta \vdash G\ gf$ are derivable in HH^{def}.*

Proof: Follows easily by induction on the structure of the derivation of $\Sigma; \Delta \Rightarrow e : G$. □

Proposition 4.2 *Let the judgement $\vdash \Sigma; \Delta_1, x =_{def} e : I, \Delta_2\ basis$ be derivable in HH^{def}. Then, the sequent $\Sigma; \Delta_1 \Rightarrow e : I$ is derivable in HH^{def}.*

$$\frac{\vdash \Sigma; \Delta, x =_{def} \Lambda : \tau, p =_{def} e : [x/y]D \; basis}{\vdash \Sigma; \Delta, (x,p) =_{def} pair_q(\Lambda, e) : \Sigma_{y:\tau} D \; basis}$$

$$\frac{\vdash \Sigma; \Delta \; basis \quad \Sigma; \Delta \Rightarrow e : I}{\vdash \Sigma; \Delta, x =_{def} e : I \; basis} \; x \notin \Delta$$

$$\frac{\Sigma; \Delta_1, x =_{def} \Lambda : \tau, p =_{def} e : [x/y]D, \Delta_2 \vdash \Lambda_1 : \tau_1}{\Sigma; \Delta_1, (x,p) =_{def} pair_q(\Lambda, e) : \Sigma_{y:\tau} D, \Delta_2 \vdash \Lambda_1 : \tau_1}$$

$$\frac{\Sigma; \Delta_1, x =_{def} \Lambda : \tau, p =_{def} e : [x/y]D, \Delta_2 \vdash \Lambda_1 \triangleright_{\tau_1} \Lambda_2}{\Sigma; \Delta_1, (x,p) =_{def} pair_q(\Lambda, e) : \Sigma_{y:\tau} D, \Delta_2 \vdash \Lambda_1 \triangleright_{\tau_1} \Lambda_2}$$

Figure 4.12: Rules of HH^{def}.

Proof: By analysis of the rules allowed for deriving bases. □

Proposition 4.3 (Weakening) *Let π be a HH^{def}-derivation of the sequent $\Sigma; \Delta_1, \Delta_2 \Rightarrow e : G$ and let the judgement $\Sigma; \Delta_1 \vdash H \; hf$ be derivable in HH^{def}. Then, for every x s.t. $x \notin (\Delta_1, \Delta_2)$, the sequent $\Sigma; \Delta_1, x : H, \Delta_2 \Rightarrow e : G$ is derivable in HH^{def}.*

Proof: The proof follows by induction on the structure of π. Case π is an axiom a similar step may be taken for deriving $\Sigma; \Delta_1, x : H, \Delta_2 \Rightarrow e : G$, the only difference being that in the latter case one needs to construct a derivation of the judgement $\vdash \Sigma; \Delta_1, x : H, \Delta_2 \; basis$. Such a derivation may be easily obtained from the derivation of $\vdash \Sigma; \Delta_1, \Delta_2 \; basis$ and the derivation of $\Sigma; \Delta_1 \vdash H \; hf$, since $x \notin (\Delta_1, \Delta_2)$. The other cases follow easily by the I.H.. □

Proposition 4.4 (Contraction) *Let π be a HH^{def}-derivation of the sequent*

$$\Sigma; \Delta_1, x : I, z : I, \Delta_2 \Rightarrow e : G,$$

where $z \notin \Delta_2$. Then, the sequent $\Sigma; \Delta_1, x : I, \Delta_2 \Rightarrow \{x/z\}e : G$ is derivable in HH^{def}.

Proof: The proof follows by induction on the structure of π.

Consider the case where π is an axiom of the form:

$$\frac{\vdash \Sigma; \Delta_1, x : I, z : I, \Delta_2 \; basis}{\Sigma; \Delta_1, x : I, z : I, \Delta_2 \Rightarrow z : I} \; axiom.$$

Since $z \notin \Delta_2$, a derivation of $\vdash \Sigma; \Delta_1, x : I, \Delta_2 \; basis$ may be easily constructed. Thus, the following derivation may be formed:

$$\frac{\vdash \Sigma; \Delta_1, x : I, \Delta_2 \; basis}{\Sigma; \Delta_1, x : I, \Delta_2 \Rightarrow x : I} \; axiom.$$

$$\frac{\Sigma;\Delta \Rightarrow e_1 : G_1 \quad \Sigma;\Delta \Rightarrow e_2 : G_2}{\Sigma;\Delta \Rightarrow pair(e_1,e_2) : G_1 \wedge G_2} \Rightarrow\wedge$$

$$\frac{\Sigma;\Delta \Rightarrow e : G_1 \quad \Sigma;\Delta \vdash G_2\ gf}{\Sigma;\Delta \Rightarrow inl(e) : G_1 \vee G_2} \Rightarrow\vee_l \qquad \frac{\Sigma;\Delta \Rightarrow e : G_2 \quad \Sigma;\Delta \vdash G_1\ gf}{\Sigma;\Delta \Rightarrow inr(e) : G_1 \vee G_2} \Rightarrow\vee_r$$

$$\frac{\Sigma;\Delta_1, x : H, \Delta_2 \Rightarrow e : G}{\Sigma;\Delta_1,\Delta_2 \Rightarrow lambda(x.e) : H \supset G} \Rightarrow\supset, x \notin \Delta_2$$

$$\frac{\Sigma;\Delta \Rightarrow e : [\Lambda/x]G \quad \Sigma;\Delta \vdash \Lambda : \tau}{\Sigma;\Delta \Rightarrow pair_q(\Lambda, e) : \exists_{x:\tau} G} \Rightarrow\exists$$

$$\frac{\Sigma, x : \tau;\Delta \Rightarrow e : G \quad \vdash \Sigma;\Delta\ basis}{\Sigma;\Delta \Rightarrow lambda_q(x.e) : \forall_{x:\tau} G} \Rightarrow\forall, x \notin \Sigma$$

$$\frac{\vdash \Sigma;\Delta_1, x : A, \Delta_2\ basis}{\Sigma;\Delta_1, x : A, \Delta_2 \Rightarrow x : A}\ axiom$$

$$\frac{\Sigma;\Delta_1, x : H_1 \wedge H_2, x_1 : H_1, \Delta_2 \Rightarrow e : G}{\Sigma;\Delta_1, x : H_1 \wedge H_2, \Delta_2 \Rightarrow splitl(x, x_1.e) : G} \wedge_l \Rightarrow, x_1 \notin \Delta_2$$

$$\frac{\Sigma;\Delta_1, x : H_1 \wedge H_2, x_1 : H_2, \Delta_2 \Rightarrow e : G}{\Sigma;\Delta_1, x : H_1 \wedge H_2, \Delta_2 \Rightarrow splitr(x, x_1.e) : G} \wedge_r \Rightarrow, x_1 \notin \Delta_2$$

$$\frac{\Sigma;\Delta_1, x : G_1 \supset H, \Delta_2 \Rightarrow e_1 : G_1 \quad \Sigma;\Delta_1, x : G_1 \supset H, x_1 : H, \Delta_2 \Rightarrow e : G}{\Sigma;\Delta_1, x : G_1 \supset H, \Delta_2 \Rightarrow apply(x, e_1, x_1.e) : G} \supset\Rightarrow, x_1 \notin \Delta_2$$

$$\frac{\Sigma;\Delta_1, x : \forall_{x_2:\tau} H, x_1 : [\Lambda/x_2]H, \Delta_2 \Rightarrow e : G \quad \Sigma,\Delta_1 \vdash \Lambda : \tau}{\Sigma;\Delta_1, x : \forall_{x_2:\tau} H, \Delta_2 \Rightarrow apply_q(x, \Lambda, x_1.e) : G} \forall\Rightarrow, x_1 \notin \Delta_2$$

$$\frac{\Sigma;\Delta_1, x : H_1, \Delta_2 \Rightarrow e : G \quad \Sigma;\Delta_1 \vdash H_1 \equiv H}{\Sigma;\Delta_1, x : H, \Delta_2 \Rightarrow e : G} \equiv_l \qquad \frac{\Sigma;\Delta \Rightarrow e : G_1 \quad \Sigma;\Delta \vdash G_1 \equiv G}{\Sigma;\Delta \Rightarrow e : G} \equiv_r$$

$$\frac{\Sigma;\Delta_1, x =_{def} \Lambda : \tau, p =_{def} e : [x/y]D, \Delta_2 \Rightarrow e_1 : G}{\Sigma;\Delta_1, (x,p) =_{def} pair_q(\Lambda, e) : \Sigma_{y:\tau} D, \Delta_2 \Rightarrow e_1 : G}\ def_{\Sigma}$$

$$\frac{\Sigma;\Delta_1 \Rightarrow e : I \quad \Sigma;\Delta_1, x : I, \Delta_2 \Rightarrow e_1 : G}{\Sigma;\Delta_1, x =_{def} e : I, \Delta_2 \Rightarrow e_1 : G}\ def_I$$

Figure 4.13: Rules for derivable sequents of HH^{def}.

Note that the identity $\{x/z\}z = z$ holds.

The other cases, except those where the last step of π is either def_Σ or def_I, may be proved by similar arguments to those used in proving the corresponding cases in Lemma 4.3 (admissibility of contraction in HH).

Case the last step of π is a rule def_I of the form:

$$\frac{\Sigma; \Delta_1, x : I, z : I, \Delta_{21} \Rightarrow e_1 : I_1 \quad \Sigma; \Delta_1, x : I, z : I, \Delta_{21}, x_1 : I_1, \Delta_{22} \Rightarrow e : G}{\Sigma; \Delta_1, x : I, z : I, \Delta_{21}, x_1 =_{def} e_1 : I_1, \Delta_{22} \Rightarrow e : G} \; def_I.$$

By the I.H., there are derivations of the sequents:

$$\Sigma; \Delta_1, x : I, \Delta_{21} \Rightarrow \{x/z\}e_1 : I_1;$$
$$\Sigma; \Delta_1, x : I, \Delta_{21}, x_1 : I_1, \Delta_{22} \Rightarrow \{x/z\}e : G.$$

Since z has no free occurrences in e_1, by hypothesis, it is easy to show that $\{x/z\}e_1$ is equal to e_1. Thus, the following derivation may be formed:

$$\frac{\Sigma; \Delta_1, x : I, \Delta_{21} \Rightarrow e_1 : I_1 \quad \Sigma; \Delta_1, x : I, \Delta_{21}, x_1 : I_1, \Delta_{22} \Rightarrow \{x/z\}e : G}{\Sigma; \Delta_1, x : I, \Delta_{21}, x_1 =_{def} e_1 : I_1, \Delta_{22} \Rightarrow \{x/z\}e : G} \; def_I .$$

The case where the last step of π is def_Σ follows directly by the I.H.. □

The remainder of this section studies relations between HH^{def} and HH. These relations are used in Sec. 4.6 to interpret the programming language LFPL, based on HH^{def}, by means of the programming language HOPLP, based on HH.

Definition 4.2 *The mapping ψ from HH^{def}-contexts to $HH^{cut'}$-contexts is defined as follows:*

$\psi(\langle\rangle) =_{def} \langle\rangle;$
$\psi(\Delta, x =_{def} \Lambda : \tau) =_{def} \psi(\Delta), x =_{def} \Lambda : \tau;$
$\psi(\Delta, x : H) =_{def} \psi(\Delta), x : H;$
$\psi(\Delta, (x, p) =_{def} pair_q(\Lambda, e) : \Sigma_{y:\tau} D) =_{def} \psi(\Delta, x =_{def} \Lambda : \tau, p =_{def} e : [x/y]D);$
$\psi(\Delta, x =_{def} e : I) =_{def} \psi(\Delta).$

Lemma 4.7 *Let Δ be a HH^{def}-context. Then:*

(i) $\psi(\Delta) = (\Delta_1, x : H, \Delta_2)$ *iff* $\Delta = (\Delta_3, x : H, \Delta_4)$ *and* $\psi(\Delta_3) = \Delta_1$ *and* $\psi(\Delta_4) = \Delta_2$*;*

(ii) if $(x =_{def} \Lambda : \tau) \in \psi(\Delta)$ *then either* $(x =_{def} \Lambda : \tau) \in \Delta$ *or exists* $(p =_{def} e : D) \in \Delta$ *s.t.* $x =_{def} \Lambda : \tau$ *is implicit in* $p =_{def} e : D$.

Proof: By analysis of the definition of ψ. □

The lemma below shows how to interpret some derivable judgements of HH^{def} into derivable judgements of $HH^{cut'}$, when HH^{def}-contexts are interpreted into $HH^{cut'}$ via ψ.

Lemma 4.8 *If the HH^{def}-judgements in the left column of the table below are derivable in HH^{def} then the corresponding $HH^{cut'}$-judgements in the right column are derivable in $HH^{cut'}$.*

(i)	$\vdash \Sigma\ signature$	$\vdash \Sigma\ signature$
(ii)	$\vdash \Sigma; \Delta\ basis$	$\vdash \Sigma; \psi(\Delta)\ basis$
(iii)	$\Sigma; \Delta \vdash \Lambda : \tau$	$\Sigma; \psi(\Delta) \vdash \Lambda : \tau$
(iv)	$\Sigma; \Delta \vdash A\ af$	$\Sigma; \psi(\Delta) \vdash A\ af$
(v)	$\Sigma; \Delta \vdash H\ hf$	$\Sigma; \psi(\Delta) \vdash H\ hf$
(vi)	$\Sigma; \Delta \vdash G\ gf$	$\Sigma; \psi(\Delta) \vdash G\ gf$

Proof: The proof is by simultaneous induction on the structure of the derivations of the HH^{def}-judgements on the left column. We analyse some cases below.

Case the last step of the derivation of the HH^{def}-judgement on line (ii) is of the form:

$$\frac{\vdash \Sigma\ signature}{\vdash \Sigma; \langle\rangle\ basis}.$$

Then, by the I.H., there is a derivation of $\vdash \Sigma\ signature$ in $HH^{cut'}$. (Observe that the derivable signatures of HH^{def} are the same as those of $HH^{cut'}$.) Thus, the following $HH^{cut'}$-derivation may be formed:

$$\frac{\vdash \Sigma\ signature}{\vdash \Sigma; \langle\rangle\ basis}.$$

Case the last step of the derivation of the HH^{def}-judgement on line (ii) is of the form:

$$\frac{\vdash \Sigma; \Delta_1\ basis \quad \Sigma; \Delta_1 \vdash \Lambda : \tau}{\vdash \Sigma; \Delta_1, x =_{def} \Lambda : \tau\ basis},$$

where $x \notin \Sigma$ and $x \notin \Delta_1$. By the I.H., there are $HH^{cut'}$-derivations of judgements 4.9 and 4.10.

$$\vdash \Sigma; \psi(\Delta_1)\ basis \tag{4.9}$$

$$\Sigma; \psi(\Delta_1) \vdash \Lambda : \tau \tag{4.10}$$

From derivations of 4.9 and 4.10, the following $HH^{cut'}$-derivation may be formed:

$$\frac{\vdash \Sigma; \psi(\Delta_1)\ basis \quad \Sigma; \psi(\Delta_1) \vdash \Lambda : \tau}{\vdash \Sigma; \psi(\Delta_1), x =_{def} \Lambda : \tau\ basis},$$

since $x \notin \Sigma$ and also $x \notin \psi(\Delta_1)$, which may be proved from $x \notin \Delta_1$. Note that

$$\psi(\Delta_1, x =_{def} \Lambda : \tau) = (\psi(\Delta_1), x =_{def} \Lambda : \tau).$$

Case the last step of the derivation of the HH^{def}-judgement on line (ii) is of the form:

$$\frac{\vdash \Sigma; \Delta_1\ basis \quad \Sigma; \Delta_1 \vdash H\ hf}{\vdash \Sigma; \Delta_1, x : H\ basis},$$

where $x \notin \Delta_1$. By the I.H., there are $HH^{cut'}$-derivations of the judgements:

$$\vdash \Sigma; \psi(\Delta_1)\ basis;$$
$$\Sigma; \psi(\Delta_1) \vdash H\ hf.$$

So, the following $HH^{cut'}$-derivation may be formed:

$$\frac{\vdash \Sigma; \psi(\Delta_1)\ basis \quad \Sigma; \psi(\Delta_1) \vdash H\ hf}{\vdash \Sigma; \psi(\Delta_1), x : H\ basis},$$

since, from $x \notin \Delta_1$, it may be easily shown that $x \notin \psi(\Delta_1)$. Note that

$$\psi(\Delta_1, x : H) = (\psi(\Delta_1), x : H).$$

Case the last step of the derivation of the HH^{def}-judgement on line (ii) is of the form:

$$\frac{\vdash \Sigma; \Delta_1\ basis \quad \Sigma; \Delta_1 \Rightarrow e : I}{\vdash \Sigma; \Delta_1, x =_{def} e : I\ basis},$$

where $x \notin \Delta_1$. Then, by the I.H., there is a $HH^{cut'}$-derivation of $\vdash \Sigma; \psi(\Delta_1)\ basis$ and note that

$$\psi(\Delta_1, x =_{def} e : I) = \psi(\Delta_1).$$

Case the last step of the derivation of the HH^{def} -judgement on line (ii) is of the form:

$$\frac{\vdash \Sigma; \Delta_1, x =_{def} \Lambda : \tau, p =_{def} e : [x/y]D\ basis}{\vdash \Sigma; \Delta_1, (x, p) =_{def} pair_q(\Lambda, e) : \Sigma_{y:\tau} D\ basis}.$$

Then, by the I.H., there is a $HH^{cut'}$-derivation of

$$\vdash \Sigma; \psi(\Delta_1, x =_{def} \Lambda : \tau, p =_{def} e : [x/y]D)\ basis.$$

The proof of this case is concluded by observing that the following identity holds:

$$\psi(\Delta_1, (x, p) =_{def} pair_q(\Lambda, e) : \Sigma_{y:\tau} D) = \psi(\Delta_1, x =_{def} \Lambda : \tau, p =_{def} e : [x/y]D).$$

The last case we consider is that where the derivation of the HH^{def}-judgement on line (iii) is of the form:

$$\frac{\Sigma; \Delta_1, x_1 =_{def} \Lambda_1 : \tau_1, p =_{def} e : [x_1/y]D, \Delta_2 \vdash \Lambda_2 : \tau_2}{\Sigma; \Delta_1, (x_1, p) =_{def} pair_q(\Lambda_1, e) : \Sigma_{y:\tau_1} D, \Delta_2 \vdash \Lambda_2 : \tau_2}.$$

By the I.H., there is a $HH^{cut'}$-derivation of

$$\Sigma; \psi(\Delta_1, x_1 =_{def} \Lambda_1 : \tau_1, p =_{def} e : [x_1/y]D, \Delta_2) \vdash \Lambda_2 : \tau_2;$$

it is easy to show that the following identity holds:

$$\psi(\Delta_1, x_1 =_{def} \Lambda_1 : \tau_1, p =_{def} e : [x_1/y]D, \Delta_2) = \psi(\Delta_1, (x_1, p) =_{def} pair_q(\Lambda_1, e) : \Sigma_{y:\tau_1} D, \Delta_2).$$

□

Lemma 4.9 *Let π be a HH^{def}-derivation of $\Sigma; \Delta \Rightarrow e : G$ with no def_I rules. Then, the sequent $\Sigma; \psi(\Delta) \Rightarrow e : G$ is derivable in $HH^{cut'}$.*

Proof: The proof is by induction on the structure of π. Case the last step of π is an axiom, then, by using Lemma 4.8, a similar axiom may be formed in $HH^{cut'}$. The cases where the last step of π is either a right or left rule or a conversion rule may be mimicked in $HH^{cut'}$. The case where the last step of π is def_Σ follows by using the I.H. and the definition of ψ. $\square$

Lemma 4.9 shows an interpretation of HH^{def}-sequents having derivations with no def_I rules into derivable sequents of $HH^{cut'}$. Theorem 4.6, below, presents an interpretation of arbitrary derivable sequents of HH^{def} into derivable sequents of $HH^{cut'}$. The mapping ν, that takes a HH^{def}-context and a $HH^{cut'}$-proof-term and gives a $HH^{cut'}$-proof-term, is defined as follows:

$$\begin{aligned}
&\nu(\langle\rangle, e) =_{def} e;\\
&\nu((x : H, \Delta), e) =_{def} \nu(\Delta, e);\\
&\nu((x =_{def} \Lambda : \tau, \Delta), e) =_{def} \nu(\Delta, e);\\
&\nu((x =_{def} e : I, \Delta), e_1) =_{def} let\ x = e\ in\ \nu(\Delta, e_1);\\
&\nu((x, p) =_{def} pair_q(\Lambda, e) : \Sigma_{y:\tau} D, \Delta), e_1) =_{def} \nu((p =_{def} e : [x/y]D, \Delta), e_1).
\end{aligned}$$

The following lemma is used in proving Theorem 4.6.

Lemma 4.10 *Let $\vdash \Sigma; \Delta$ basis be derivable in HH^{def}. Then, if $\Sigma; \psi(\Delta) \Rightarrow e : G$ is derivable in $HH^{cut'}$ then $\Sigma; \psi(\Delta) \Rightarrow \nu(\Delta, e) : G$ is derivable in $HH^{cut'}$.*

Proof: The proof follows by induction on the number n of definitions of dependent type in Δ.

Case $n = 0$ then $\nu(\Delta, e) = e$, hence the result is trivial.

Case $n \geq 1$ then Δ may be written in the form $(\Delta_1, p =_{def} e_1 : D, \Delta_2)$, where Δ_1 contains no definitions of dependent type. The proof follows by induction on the structure of p.

Case $p = x$ then it may be easily shown that $\psi(\Delta) = \psi(\Delta_1, \Delta_2)$. So by the I.H., there is a derivation of

$$\Sigma; \psi(\Delta_1, \Delta_2) \Rightarrow \nu((\Delta_1, \Delta_2), e) : G.$$

It may be shown that $\psi(\Delta_1, \Delta_2) = (\psi(\Delta_1), \psi(\Delta_2))$ and, for Δ_1 contains no definitions of dependent type, $\nu((\Delta_1, \Delta_2), e) = \nu(\Delta_2, e)$. Thus, by weakening, there is a derivation π_1 of the sequent:

$$\Sigma; \psi(\Delta_1), x : D, \psi(\Delta_2) \Rightarrow \nu(\Delta_2, e) : G.$$

By hypothesis, $\vdash \Sigma; \Delta$ *basis* is derivable. So, by Proposition 4.2, there is a HH^{def}-derivation π_2 of $\Sigma; \Delta_1 \Rightarrow e_1 : D$. Since Δ_1 contains no definitions of dependent type, π_2 may contain no def_I rules. So, by Lemma 4.9, there is a $HH^{cut'}$-derivation π_3 of $\Sigma; \psi(\Delta_1) \Rightarrow e_1 : D$. Now, the following $HH^{cut'}$-derivation may be formed:

$$\frac{\overset{\pi_3}{\Sigma; \psi(\Delta_1) \Rightarrow e_1 : D} \quad \overset{\pi_1}{\Sigma; \psi(\Delta_1), x : D, \psi(\Delta_2) \Rightarrow \nu(\Delta_2, e) : G}}{\Sigma; \psi(\Delta_1), \psi(\Delta_2) \Rightarrow let\ x = e_1\ in\ \nu(\Delta_2, e) : G}\ cut.$$

For concluding the proof, observe that the identity

$$\nu((\Delta_1, x =_{def} e_1 : D, \Delta_2), e) = (let\ x = e_1\ in\ \nu(\Delta_2, e))$$

holds, since Δ_1 has no definitions of dependent type.

The case where p is of the form (x_1, p_1) follows easily by the I.H.. $\square$

Theorem 4.6 *If π is a HH^{def}-derivation of sequent $\Sigma; \Delta \Rightarrow e : G$ then $\Sigma; \psi(\Delta) \Rightarrow \nu(\Delta, e) : G$ is derivable in $HH^{cut'}$.*

Proof: The proof follows by induction on the structure of π. Some cases are analysed below.

Case the last step of π is an axiom:

$$\frac{\vdash \Sigma; \Delta_1, x : H, \Delta_2\ basis}{\Sigma; \Delta_1, x : H, \Delta_2 \Rightarrow x : H}\ axiom.$$

By Lemma 4.8, there is a $HH^{cut'}$-derivation of $\vdash \Sigma; \psi(\Delta_1, x : H, \Delta_2)\ basis$. By Lemma 4.7, the identity $\psi(\Delta_1, x : H, \Delta_2) = (\psi(\Delta_1), x : H, \psi(\Delta_2))$ holds. So, the following $HH^{cut'}$-derivation may be formed:

$$\frac{\vdash \Sigma; \psi(\Delta_1), x : H, \psi(\Delta_2)\ basis}{\Sigma; \psi(\Delta_1), x : H, \psi(\Delta_2) \Rightarrow x : H}\ axiom.$$

By hypothesis, $\Sigma; \Delta_1, x : H, \Delta_2 \Rightarrow e : G$ is derivable; so, by Proposition 4.1, there is a derivation of $\vdash \Sigma; \Delta_1, x : H, \Delta_2\ basis$. Thus, using Lemma 4.10, there is a $HH^{cut'}$-derivation of the sequent:

$$\Sigma; \psi(\Delta_1), x : H, \psi(\Delta_2) \Rightarrow \nu((\Delta_1, x : H, \Delta_2), x) : H.$$

Case the last step of π is def_I:

$$\frac{\Sigma; \Delta_1 \Rightarrow e : I \quad \Sigma; \Delta_1, x : I, \Delta_2 \Rightarrow e_1 : G}{\Sigma; \Delta_1, x =_{def} e : I, \Delta_2 \Rightarrow e_1 : G}\ def_I.$$

By the I.H., and by Lemma 4.7, there are $HH^{cut'}$-derivations of the sequents:

$$\Sigma; \psi(\Delta_1) \Rightarrow \nu(\Delta_1, e) : I;$$
$$\Sigma; \psi(\Delta_1), x : I, \psi(\Delta_2) \Rightarrow \nu((\Delta_1, x : I, \Delta_2), e_1) : G.$$

Thus, by using a *cut* rule in $HH^{cut'}$, there is a derivation of

$$\Sigma; \psi(\Delta_1), \psi(\Delta_2) \Rightarrow PT_1 : G,$$

where PT_1 is the proof-term:

$$let\ x = \nu(\Delta_1, e)\ in\ \nu((\Delta_1, x : I, \Delta_2), e_1).$$

Note that $\psi(\Delta_1, x =_{def} e : I, \Delta_2) = (\psi(\Delta_1), \psi(\Delta_2))$. It remains to prove that from a $HH^{cut'}$-derivation of proof-term PT_1, we may construct a $HH^{cut'}$-derivation of proof-term PT_2, defined as:

$$\nu((\Delta_1, x =_{def} e : I, \Delta_2), e_1).$$

The proof is by induction on the number n of definitions of dependent type in Δ_1.

Case $n = 0$ then it is easy to show that PT_1 and PT_2 are equal to

$$let\ x = e\ in\ \nu(\Delta_2, e_1).$$

Case $n \geq 1$ then Δ_1 may be written as $(\Delta_{11}, p =_{def} e_2 : D, \Delta_{12})$, where Δ_{11} contains no definitions of dependent type. Assume without loss of generality that p is a variable x_2. Proof-term PT_1 is equal to the proof-term

$$let\ x = (let\ x_2 = e_2\ in\ \nu(\Delta_{12}, e))\ in\ (let\ x_2 = e_2\ in\ \nu((\Delta_{12}, x : I, \Delta_2), e_1)),$$

for Δ_{11} contains no definitions of dependent type. It is easy to show that an $HH^{cut'}$-derivation with this proof-term may be permuted to a derivation with the following proof-term

$$let\ x_2 = e_2\ in\ (let\ x = \nu(\Delta_{12}, e)\ in\ \ \nu((\Delta_{12}, x : I, \Delta_2), e_1)).$$

(Figure 4.14 shows derivations corresponding to this kind of permutation.) Since Δ_{12} contains one definition of dependent type fewer than Δ_1, by I.H., we may construct a derivation of proof term:

$$let\ x_2 = e_2\ in\ \nu((\Delta_{12}, x =_{def} e : I, \Delta_2), e_1),$$

which may be easily shown equal to PT_2.

Proofs for the cases where the last step of π is of any other form may be obtained by similar arguments to those used above; they require permutations of each rule with cut rules. □

$$\dfrac{\dfrac{\Sigma; \Delta_1 \Rightarrow e_2 : I_1 \quad \Sigma; \Delta_1, x_2 : I_1, \Delta_2 \Rightarrow e : I}{\Sigma; \Delta_1, \Delta_2 \Rightarrow let\ x_2 = e_2\ in\ e : I}\, cut \quad \dfrac{\Sigma; \Delta_1, \Delta_2, x : I, \Delta_3 \Rightarrow e_2 : I_1 \quad \Sigma; \Delta_1, \Delta_2, x : I, \Delta_3, x_2 : I_1, \Delta_4 \Rightarrow e_1 : G}{\Sigma; \Delta_1, \Delta_2, x : I, \Delta_3, \Delta_4 \Rightarrow let\ x_2 = e_2\ in\ e_1 : G}\, cut}{\Sigma; \Delta_1, \Delta_2, \Delta_3, \Delta_4 \Rightarrow let\ x = (let\ x_2 = e_2\ in\ e)\ in\ (let\ x_2 = e_2\ in\ e_1) : G}\, cut$$

$$\downarrow$$

$$\dfrac{\Sigma; \Delta_1 \Rightarrow e_2 : I_1 \quad \dfrac{\Sigma; \Delta_1, x_2 : I_1, \Delta_2 \Rightarrow e : I \quad \Sigma; \Delta_1, x_2 : I_1, \Delta_2, x : I, \Delta_3, \Delta_4 \Rightarrow e_1 : G}{\Sigma; \Delta_1, x_2 : I_1, \Delta_2, \Delta_3, \Delta_4 \Rightarrow let\ x = e\ in\ e_1) : G}\, cut}{\Sigma; \Delta_1, \Delta_2, \Delta_3, \Delta_4 \Rightarrow let\ x_2 = e_2\ in\ (let\ x = e\ in\ e_1) : G}\, cut$$

Figure 4.14: A permutation of *cuts* in $HH^{cut'}$.

Corollary 4.1 *If* $\Sigma; \Delta \Rightarrow e : G$ *is derivable in* HH^{def} *then*

$$\Sigma; \aleph(\psi(\Delta)) \Rightarrow cut([\psi(\Delta)]\nu(\Delta, e)) : [\psi(\Delta)]G$$

is derivable in HH.

Proof: If $\Sigma; \Delta \Rightarrow e : G$ is derivable in HH^{def} then, by Theorem 4.6,

$$\Sigma; \psi(\Delta) \Rightarrow \nu(\Delta, e) : G$$

is derivable in $HH^{cut'}$. By Theorem 4.5,

$$\Sigma; \aleph(\psi(\Delta)) \Rightarrow [\psi(\Delta)]\nu(\Delta, e) : [\psi(\Delta)]G$$

is derivable in HH^{cut}. By Theorem 4.4,

$$\Sigma; \aleph(\psi(\Delta)) \Rightarrow cut([\psi(\Delta)]\nu(\Delta, e)) : [\psi(\Delta)]G$$

is derivable in HH. □

Now a problem converse to Theorem 4.6 is addressed, roughly, how to transform derivable sequents of $HH^{cut'}$ into derivable sequents of HH^{def}.

Lemma 4.11 *Let $\vdash \Sigma; \Delta$ basis be derivable in HH^{def}. Then:*

(*i*) *if $\Sigma; \psi(\Delta) \vdash \Lambda : \tau$ is derivable in $HH^{cut'}$ then $\Sigma; \Delta \vdash \Lambda : \tau$ is derivable in HH^{def};*

(*ii*) *if $\Sigma; \psi(\Delta) \vdash \Lambda \triangleright_\tau \Lambda_1$ is derivable in $HH^{cut'}$ then $\Sigma; \Delta \vdash \Lambda \triangleright_\tau \Lambda_1$ is derivable in HH^{def}.*

Proof: The proof of (i) follows by induction on the structure of the $HH^{cut'}$-derivation π of $\Sigma; \psi(\Delta) \vdash \Lambda : \tau$. Consider the case where the last step of π is of the form:

$$\frac{\vdash \Sigma; \Delta_1, x =_{def} \Lambda : \tau, \Delta_2 \; basis}{\Sigma; \Delta_1, x =_{def} \Lambda : \tau, \Delta_2 \vdash x : \tau} \; .$$

By Lemma 4.7, either $\Delta = (\Delta_3, x =_{def} \Lambda : \tau, \Delta_4)$ or $\Delta = (\Delta_3, p =_{def} e : D, \Delta_4)$ and $x =_{def} \Lambda : \tau$ is implicit in $p =_{def} e : D$. Case Δ is of the first form a similar step may be used in HH^{def}. Case Δ is of the latter form, we assume, without loss of generality, $p =_{def} e : D$ to be of the form $(x, p_1) =_{def} pair_q(\Lambda, e_1) : \Sigma_{y:\tau} D_1$. Then, the following steps in HH^{def} may be formed:

$$\frac{\dfrac{\vdash \Sigma; \Delta_3, x =_{def} \Lambda : \tau, p_1 =_{def} e_1 : [x/y]D_1, \Delta_4 \; basis}{\Sigma; \Delta_3, x =_{def} \Lambda : \tau, p_1 =_{def} e_1 : [x/y]D_1, \Delta_4 \vdash x : \tau}}{\Sigma; \Delta_3, (x, p_1) =_{def} pair_q(\Lambda, e_1) : \Sigma_{y:\tau} D_1, \Delta_4 \vdash x : \tau} \; ,$$

where a derivation of the uppermost judgement may be obtained from the derivation of $\vdash \Sigma; \Delta$ *basis*, that exists by hypothesis.

If the last step of π is a rule of any other form a similar step may be used in HH^{def}.

A proof of (ii) may be obtained by similar arguments to those used for proving (i). □

Theorem 4.7 *Let $\vdash \Sigma; \Delta$ basis be derivable in HH^{def}. Then, if $\Sigma; \psi(\Delta) \Rightarrow e : G$ is derivable in $HH^{cut'}$ with no* cut *rules then $\Sigma; \Delta \Rightarrow e : G$ is derivable in HH^{def}.*

Proof: Let π be a $HH^{cut'}$-derivation of $\Sigma; \psi(\Delta) \Rightarrow e : G$. The proof follows by induction on the structure of π. Since π is *cut*-free, the last step of π may not be a *cut* rule. Some cases are considered below.

Case the last step of π is of the form:

$$\frac{\vdash \Sigma; \Delta_1, x : H, \Delta_2 \; \textit{basis}}{\Sigma; \Delta_1, x : H, \Delta_2 \Rightarrow x : H}$$

By Lemma 4.7, the HH^{def}-context Δ is of the form $(\Delta_3, x : H, \Delta_4)$. By hypothesis, the judgement $\vdash \Sigma; \Delta_3, x : H, \Delta_4 \; \textit{basis}$ is derivable in HH^{def}. So, the following HH^{def}-derivation may be formed:

$$\frac{\vdash \Sigma; \Delta_3, x : H, \Delta_4 \; \textit{basis}}{\Sigma; \Delta_3, x : H, \Delta_4 \Rightarrow x : H} \; \textit{axiom}.$$

Case the last step of π is $\Rightarrow \exists$:

$$\frac{\Sigma; \psi(\Delta) \Rightarrow e_1 : [\Lambda/y]G_1 \quad \Sigma; \psi(\Delta) \vdash \Lambda : \tau}{\Sigma; \psi(\Delta) \Rightarrow \textit{pair}_q(\Lambda, e_1) : \exists_{y:\tau} G_1} \Rightarrow \exists.$$

By Lemma 4.11, there is a HH^{def}-derivation of the judgement $\Sigma; \Delta \vdash \Lambda : \tau$. By the I.H., there is a HH^{def}-derivation of the sequent $\Sigma; \Delta \Rightarrow e_1 : [\Lambda/y]G_1$. Thus, the following HH^{def}-derivation may be formed:

$$\frac{\Sigma; \Delta \Rightarrow e_1 : [\Lambda/y]G_1 \quad \Sigma; \Delta \vdash \Lambda : \tau}{\Sigma; \Delta \Rightarrow \textit{pair}_q(\Lambda, e_1) : \exists_{y:\tau} G_1} \Rightarrow \exists.$$

Case the last step of π is $\equiv_l$:

$$\frac{\Sigma; \psi(\Delta) \Rightarrow e : G_1 \quad \Sigma; \psi(\Delta) \vdash G_1 \equiv G}{\Sigma; \psi(\Delta) \Rightarrow e : G} \equiv_l .$$

An HH^{def}-derivation of the judgement

$$\Sigma; \Delta \vdash G_1 \equiv G$$

may be constructed by induction on the structure of the $HH^{cut'}$-derivation of

$$\Sigma; \psi(\Delta) \vdash G_1 \equiv G,$$

essentially by using part (ii) of Lemma 4.11. By the I.H., there is a HH^{def}-derivation of the sequent

$$\Sigma; \Delta \Rightarrow e : G_1.$$

So, the following derivation in HH^{def} may be formed:

$$\frac{\Sigma; \Delta \Rightarrow e : G_1 \quad \Sigma; \Delta \vdash G_1 \equiv G}{\Sigma; \Delta \Rightarrow e : G} \equiv_l .$$

The cases corresponding to the other possible forms of π follow by similar arguments. □

Corollary 4.2 *Let $\vdash \Sigma; \Delta$ basis be derivable in HH^{def}. Then, if the sequent*

$$\Sigma; \aleph(\psi(\Delta)) \Rightarrow e : [\psi(\Delta)]G$$

is derivable in HH then, for every e_1 s.t. $[\psi(\Delta)]e_1 = e$, the sequent $\Sigma; \Delta \Rightarrow e_1 : G$ is derivable in HH^{def}.

Proof: If $\Sigma; \aleph(\psi(\Delta)) \Rightarrow e : [\psi(\Delta)]G$ is derivable in HH then, by Lemma 4.6, $\Sigma; \aleph(\psi(\Delta)) \Rightarrow e : [\psi(\Delta)]G$ is derivable in HH^{cut}. Thus, by Theorem 4.5, $\Sigma; \psi(\Delta) \Rightarrow e_1 : G$ is derivable in $HH^{cut'}$, for every $e_1 = [\psi(\Delta)]e$, since $\Sigma; \psi(\Delta) \vdash G$ is derivable in $HH^{cut'}$. It may be easily shown that a $HH^{cut'}$-derivation whose proof-term is of the form $[\Delta]e$, where Δ is a $HH^{cut'}$-basis and e is a proof-term in HH, has no *cuts*. So, by Theorem 4.7, $\Sigma; \Delta \Rightarrow e_1 : G$ is derivable in HH^{def}, for every $e_1 = [\psi(\Delta)]e$. □

From Corollary 4.2 one may easily prove that: if the sequent $\Sigma; \Delta \Rightarrow e : G$ is derivable in HH then, for every list Δ_1 consisting of the elements of the set Δ, the sequent $\Sigma; \Delta_1 \Rightarrow e : G$ is derivable in HH^{def}. So, combining this result with Corollary 4.1, one shows that HH^{def} is a conservative extension of HH.

We conclude this section by showing that the family of rules of *backchaining*, defined below, is admissible in HH^{def}. The rules in this family may be thought of as instances of Miller's rule of backchaining presented in [Mil90]. Backchaining is used in Sec. 4.7 for showing that some HH^{def}-sequents are derivable.

Definition 4.3 *The family of rules of* backchaining (BC), *indexed by the natural numbers, is defined as follows:*

Index $n = 0$

$$\frac{\Sigma; \Delta_1, x : G \supset A, \Delta_2 \Rightarrow e : G}{\Sigma; \Delta_1, x : G \supset A, \Delta_2 \Rightarrow bc(x, [], e) : A}\ BC,$$

Index $n \geq 1$

$$\frac{\Sigma; \Delta_1, x : H, \Delta_2 \Rightarrow e : [\Lambda_n/x_n] \ldots [\Lambda_1/x_1]G \quad \Sigma; \Delta_1 \vdash \Lambda_1 : \tau_1 \quad \ldots \quad \Sigma; \Delta_1 \vdash \Lambda_n : \tau_n}{\Sigma; \Delta_1, x : H, \Delta_2 \Rightarrow bc(x, [\Lambda_1, \ldots, \Lambda_n], e) : [\Lambda_n/x_n] \ldots [\Lambda_1/x_1]A}\ BC,$$

where $H = \forall_{x_1:\tau_1} \ldots \forall_{x_n:\tau_n}(G \supset A)$ and
$bc(x, [], e) =_{def} apply(x, e, z.z)$, $z \neq x$, $z \notin e$;
$bc(x, [\Lambda|\Lambda s], e) =_{def} apply_q(x, \Lambda, z.bc(z, \Lambda s, e))$, $z \neq x$, $z \notin e$.

Proposition 4.5 (admissibility of BC) *If π is a derivation of a HH^{def}-sequent constructed by using HH^{def}-rules and BC then π may be transformed into a derivation of the same sequent using HH^{def}-rules only.*

Proof: The proof follows by induction on the structure of π. The cases where the last step of π is not a BC rule are easy. Consider the last step of π is a BC rule. The proof follows by induction on the index of the BC rule.

(i) Case the index is zero, π is of the form:

$$\frac{\overset{\pi_1}{\Sigma; \Delta_1, x : G \supset A, \Delta_2 \Rightarrow e : G}}{\Sigma; \Delta_1, x : G \supset A, \Delta_2 \Rightarrow bc(x, [], e) : A}\ BC.$$

Then, by the I.H., π_1 may be transformed into a HH^{def}-derivation π_2 of $\Sigma; \Delta_1, x : G \supset A, \Delta_2 \Rightarrow e : G$. From π_2 a derivation of

$$\vdash \Sigma; \Delta_1, x : G \supset A, \Delta_2\ basis$$

may be obtained, *cf* Proposition 4.1. From such derivation, we may derive

$$\vdash \Sigma; \Delta_1, x : G \supset A, z : A, \Delta_2\ basis,$$

for every z s.t. $z \neq x$ and $z \notin (\Delta_1, \Delta_2)$. So, the derivation below may be formed.

$$\frac{\overset{\pi_2}{\Sigma; \Delta_1, x : G \supset A, \Delta_2 \Rightarrow e : G} \quad \dfrac{\vdash \Sigma; \Delta_1, x : G \supset A, z : A, \Delta_2\ basis}{\Sigma; \Delta_1, x : G \supset A, z : A, \Delta_2 \Rightarrow z : A}\ axiom}{\Sigma; \Delta_1, x : G \supset A, \Delta_2 \Rightarrow apply(x, e, z.z) : A}\ \supset\Rightarrow$$

Note that $bc(x, [], e) = apply(x, e, z.z)$, for $z \neq x$ and $x \notin e$.

(ii) Case the index of the last step of π is greater than zero, π is of the form:

$$\frac{\overset{\pi_1}{\Sigma; \Delta_1, x : H, \Delta_2 \Rightarrow e : [\Lambda_n/x_n] \ldots [\Lambda_2/x_2][\Lambda_1/x_1]G} \quad \Sigma; \Delta_1 \vdash \Lambda_1 : \tau_1 \quad \ldots \quad \Sigma; \Delta_1 \vdash \Lambda_n : \tau_n}{\Sigma; \Delta_1, x : H, \Delta_2 \Rightarrow bc(x, [\Lambda_1, \Lambda_2, \ldots, \Lambda_n], e) : [\Lambda_n/x_n] \ldots [\Lambda_2/x_2][\Lambda_1/x_1]A}\ BC,$$

where $H = \forall_{x_1:\tau_1} H_1$ and $H_1 = \forall_{x_2:\tau_2} \ldots \forall_{x_n:\tau_n}(G \supset A)$. A derivation π_2 of the sequent

$$\Sigma; \Delta_1, x : H, z : [\Lambda_1/x_1]H_1, \Delta_2 \Rightarrow e : [\Lambda_n/x_n] \ldots [\Lambda_2/x_2][\Lambda_1/x_1]G,$$

where $z \neq x$ and $z \notin (\Delta_1, \Delta_2)$, may be obtained by weakening from π_1. By the I.H., the derivation below may be transformed into a derivation with no instances of BC, since the instance of BC has index one smaller than n. In the derivation below $\boxed{\text{TJs}}$ represents the $n-1$ judgements $\Sigma; \Delta_1 \vdash \Lambda_i : \tau_i$ for $2 \leq i \leq n$.

$$\frac{\dfrac{\overset{\pi_2}{\Sigma; \Delta_1, x : H, z : [\Lambda_1/x_1]H_1, \Delta_2 \Rightarrow e : [\Lambda_n/x_n] \ldots [\Lambda_2/x_2][\Lambda_1/x_1]G} \quad \boxed{\text{TJs}}}{\Sigma; \Delta_1, x : H, z : [\Lambda_1/x_1]H_1, \Delta_2 \Rightarrow bc(z, [\Lambda_2, \ldots, \Lambda_n], e)) : [\Lambda_n/x_n] \ldots [\Lambda_2/x_2][\Lambda_1/x_1]A}\ BC}{\Sigma; \Delta_1, x : H, \Delta_2 \Rightarrow apply_q(x, \Lambda_1, z.bc(z, [\Lambda_2, \ldots, \Lambda_n], e)) : [\Lambda_n/x_n] \ldots [\Lambda_2/x_2][\Lambda_1/x_1]A}\ \forall\Rightarrow$$

Note that the following identities hold:

$[\Lambda_1/x_1]H_1 = \forall_{x_2:\tau_2} \ldots \forall_{x_n:\tau_n}([\Lambda_1/x_1]G \supset [\Lambda_1/x_1]A)$;
$bc(x, [\Lambda_1, \Lambda_2, \ldots, \Lambda_n], e) = apply_q(x, \Lambda_1, z.bc(z, [\Lambda_2, \ldots, \Lambda_n], e))$, for $z \neq x$ and $z \notin e$.

□

4.6 Proof-Theoretic Semantics of the Integrated Logical and Functional Programming Language LFPL

This section defines the semantics of a programming language that integrates logic and functional programming called LFPL. This language may be seen as a language that extends HOPLP with simple definitions (i.e. abbreviations for terms of simple type) and some forms of definitions of dependent type, namely definitions asserting logical properties of functions.

We require the language LFPL to satisfy the following constraints:

1. goals and programs in LFPL should be interpretable as goals and programs in HOPLP;
2. the language LFPL should be *conservative* w.r.t. HOPLP, i.e. if a goal is achievable w.r.t. a program in LFPL then the interpretation of the goal into HOPLP should be achievable w.r.t. the interpretation of the program into HOPLP; roughly, there should be no more goals achievable w.r.t. a program in LFPL than there are in HOPLP;
3. the language LFPL should be *complete* for the means of goal-achievement in HOPLP, i.e., given a goal G and a program $\Sigma;\Delta$ in LFPL, for every means of achieving the interpretation of G w.r.t. the interpretation of $\Sigma;\Delta$ in HOPLP, there should be a means of achieving G w.r.t. $\Sigma;\Delta$ in LFPL.

The semantics of LFPL is defined below by means of the calculus HH^{def}.

Definition 4.4 *A* program *in LFPL is a pair $\langle\Sigma,\Delta\rangle$, usually written $\Sigma;\Delta$, where Σ is a signature and Δ is a HH^{def}-context; a program $\Sigma;\Delta$ is* well-formed *iff the judgement $\vdash \Sigma;\Delta$ basis is derivable in HH^{def}.*

A goal *in LFPL is a G-formula; G is* well-formed *w.r.t. a program $\Sigma;\Delta$ iff the judgement $\Sigma;\Delta \vdash G\ gf$ is derivable in HH^{def}.*

A goal G is achievable *w.r.t. a program $\Sigma;\Delta$ iff there exists a proof-term e s.t. the sequent $\Sigma;\Delta \Rightarrow e : G$ is derivable in HH^{def}; the proof-term e is called a* witness *for the achievement of G w.r.t $\Sigma;\Delta$.*

In Section 3.2, in the context of first-order logic programming, it is argued that a semantics for a logic programming language needs to define what are the different means of goal-achievement. So, in order to complete the definition of LFPL , we must define, given a goal G and a program $\Sigma;\Delta$, what are the different means for the achievement of G w.r.t. $\Sigma;\Delta$.

First is shown that LFPL , as defined so far, meets some of the requirements mentioned above; this is shown by using results presented in Sec. 4.5 relating the calculi HH and HH^{def}.

Theorem 4.8 *Let $\Sigma;\Delta$ be a well-formed program in LFPL. Then:*

(1) *$\Sigma;\aleph(\psi(\Delta))$ is a well-formed program in HOPLP;*

(2) *if G is well-formed w.r.t. $\Sigma; \Delta$ in LFPL then $[\psi(\Delta)]G$ is a well-formed goal w.r.t. the program $\Sigma; \aleph(\psi(\Delta))$ in HOPLP.*

Proof:

(1) By Lemma 4.8, there is a derivation of $\vdash \Sigma; \psi(\Delta)$ in $HH^{cut'}$. Using arguments similar to those used in proving Lemmas 4.1 and 4.2, that show how to transform a HH'-basis into a HH-basis, one may show that $\vdash \Sigma; \aleph(\psi(\Delta))$ *basis* is derivable in HH.

(2) As above, by using Lemma 4.8, a $HH^{cut'}$-derivation of $\Sigma; \psi(\Delta) \vdash G$ may be obtained, and from such derivation it may be shown that $\Sigma \vdash [\psi(\Delta)]G$ is derivable in HH. □

Theorem 4.8 shows an interpretation of goals and programs in LFPL as goals and programs in HOPLP, respectively; thus, constraint 1 imposed on the language LFPL is satisfied.

Theorem 4.9 *Let G be achievable w.r.t. $\Sigma; \Delta$ in LFPL. Then, $[\psi(\Delta)]G$ is achievable w.r.t. $\Sigma; \aleph(\psi(\Delta))$ in HOPLP.*

Proof: If G is achievable w.r.t. $\Sigma; \Delta$ in LFPL then there exists a proof-term e s.t. the sequent $\Sigma; \Delta \Rightarrow e : G$ is derivable in HH^{def}. So, by Corollary 4.1, the sequent

$$\Sigma; \aleph(\psi(\Delta)) \Rightarrow cut([\psi(\Delta)]\nu(\Delta, e)) : [\psi(\Delta)]G$$

is derivable in HH. Thus, $cut([\psi(\Delta)]\nu(\Delta, e))$ is a witness for the achievement of $[\psi(\Delta)]G$ w.r.t. $\Sigma; \aleph(\psi(\Delta))$ in HOPLP. □

The proof of Theorem 4.9 shows how to interpret witnesses in LFPL as witnesses in HOPLP. Given a witness e for the achievement of G w.r.t. $\Sigma; \Delta$ in LFPL, $cut([\psi(\Delta)]\nu(\Delta, e))$ is a witness for the achievement of $[\psi(\Delta)]G$ w.r.t. $\Sigma; \aleph(\psi(\Delta))$ in HOPLP. For the remainder of this thesis, this interpretation of witnesses is taken as the standard interpretation of witnesses in LFPL as witnesses in HOPLP. Theorem 4.9 shows that constraint 2 imposed on the language LFPL is satisfied.

By Corollary 4.2, if e is a witness for the achievement of $[\psi(\Delta)]G$ w.r.t. $\Sigma; \aleph(\psi(\Delta))$ in HOPLP then there exists a witness for the achievement of G w.r.t. $\Sigma; \Delta$ in LFPL; thus showing that constraint 3 imposed on LFPL is satisfied, i.e. LFPL is complete for witnesses w.r.t. HOPLP. However, constraint 3 is satisfied in an excessive way, since, under the standard interpretation, several witnesses in LFPL may be interpreted as the same witness of a complete set of witnesses in HOPLP.

Recall that a complete set of witnesses for goal-achievement in HOPLP is a maximal set w.r.t. the conditions: its members are uniform linear focused witnesses of the goal w.r.t. the program and no two members of the set are λ-convertible. Recall also that the notation *ulf*(e) stands for the uniform linear focused form of e. Below we define the concepts of *complete* and *non-redundant* sets of witnesses for goal-achievement in LFPL . These concepts are such that any complete and non-redundant set of witnesses for the achievement of G w.r.t. $\Sigma; \Delta$ in LFPL is in 1-1 correspondence with any complete set of witnesses for the achievement of $[\psi(\Delta)]G$ w.r.t. $\Sigma; \aleph(\psi(\Delta))$ in HOPLP.

Definition 4.5 *Let $\Sigma; \Delta \vdash G\ gf$ be derivable in LFPL.*

A set S of witnesses for the achievement of G w.r.t. $\Sigma; \Delta$ is complete *iff for every uniform linear focused proof-term e_{ulf} s.t. $\Sigma; \aleph(\psi(\Delta)) \Rightarrow e_{ulf} : [\psi(\Delta)]G$ is derivable in HH there exists a witness e in S s.t. $\Sigma; \Delta \Rightarrow e : G$ is derivable in HH^{def} and $ulf(cut([\psi(\Delta)]\nu(\Delta, e))) = e_{ulf}$.*

A set S of witnesses for the achievement of G w.r.t. $\Sigma; \Delta$ is non-redundant *iff there are no two witnesses e_1, e_2 in S s.t. $ulf(cut([\psi(\Delta)]\nu(\Delta, e_1))) \equiv_\lambda ulf(cut([\psi(\Delta)]\nu(\Delta, e_2)))$.*

Definition 4.6 *An* implementation *of LFPL is any method that given a goal G well-formed w.r.t. a program $\Sigma; \Delta$ in LFPL finds a complete set of witnesses for the achievement of G w.r.t. $\Sigma; \Delta$ in LFPL; it is called* excessive *if the complete set of witnesses is redundant.*

The relation between HOPLP and LFPL may be described as follows. Suppose there is a problem that may be formulated as a LFPL program and goal. Such problem could also be formulated in HOPLP, since there is an interpretation of LFPL by means of HOPLP. However, the problem in LFPL may have a more natural formulation, since there are definition mechanisms provided in LFPL that are not provided in HOPLP. Having the problem formulated both in LFPL and HOPLP the forms of achieving the goal w.r.t. the program in both languages may be very different. In HOPLP, only the canonical form of reasoning, corresponding to uniform linear focused derivations, is allowed for goal-achievement. In LFPL other forms of reasoning are allowed, namely those corresponding to *cut* rules. These forms of reasoning are interpretable into the canonical form of reasoning, but often the canonical forms correspond to much longer derivations.

4.7 Solving Problems in LFPL: an Example

In this section is shown a formulation of the following problem in LFPL. Given a natural number n and a list of natural numbers L, find an element of L greater than or equal to n, if there is any. A formulation of the problem above is used to illustrate various aspects of the definition mechanisms allowed in LFPL, as well as some issues raised by an implementation for LFPL. In [Pin94] is shown a formulation of a problem in a calculus whose features are mainly present in HH^{def}. This problem is based upon an example presented by Boolos, in [Boo84], to show that cut-free derivations may be hyper-exponentially longer than their counterparts using cuts. This problem could be formulated similarly in LFPL; it could be used to illustrate that there are witnesses for the achievement of a goal w.r.t. a program in LFPL representing derivations which are hyper-exponentially shorter than the derivations represented by the corresponding witnesses in HOPLP.

Proof-search in HH^{def} is studied in Chap. 5, from the perspective of an implementation for LFPL. In the example described in this section some derivations in HH^{def} are constructed to show that certain goals are achievable w.r.t. programs. The intuition behind the construction of these derivations comes from goal-directed search, for compound goals, and *backchaining* together with the idea that a definition of dependent type should be used if any of its definienda occurs in the goal, for atomic goals.

The problem of "given a list of natural numbers and a natural number n, finding an element of the list greater than or equal to n", may be formulated in LFPL as the problem of achieving a goal G w.r.t. a program $\Sigma; \Delta$, as follows.

The types of natural numbers and lists of natural numbers are represented as primitive types *nat* and *lnat*. The signature Σ consists of the following pairs:

$$\begin{aligned}&0 : \tau,\\ &s : nat \to nat \to nat,\\ &nil : lnat,\\ &cons : nat \to lnat \to lnat.\end{aligned}$$

The *constructors* of primitive types are represented as variables. If polymorphism were allowed, the polymorphic type of lists could be defined and the type *lnat* obtained as a particular case of lists. The usual abbreviations for lists are used, i.e. $[]$ for *nil*, $[x|x_1]$ for $cons\ x\ x_1$, and so on.

The set of predicates $\mathcal{P}$ contains the following predicate symbols:

$$\begin{aligned}&geq : nat \to nat \to prop,\\ &member_{geq} : nat \to lnat \to nat \to prop.\end{aligned}$$

Intuitively, *geq* represents the relation *greater than or equal to* over the natural numbers and $member_{geq}$ represents a relation on triples $\langle n_1, ns, n_2\rangle$ that holds if the natural number n_2 is an element of the list ns greater than or equal to n_1.

In what follows $\forall_{x_1,x_2,\ldots,x_n:\tau}F$, where F is either a G or a H-formula, is used as an abbreviation for $\forall_{x_1:\tau}\forall_{x_2:\tau}\ldots\forall_{x_n:\tau}F$. The context Δ is defined as follows:

$$\begin{aligned}&z_1 : \forall_{x:nat} geq(x, 0),\\ &z_2 : \forall_{x_1,x_2:nat}(geq(x_1, x_2) \supset geq(sx_1, sx_2)),\\ &(+_2, z) =_{def} pair_q(\lambda x.ssx, e) : \Sigma_{f:nat\to nat}\forall_{x_1,x_2:nat}(geq(x_1, x_2) \supset geq(fx_1, fx_2)),\\ &z_3 : \forall_{x_1,x_2:nat}\forall_{x:lnat}(geq(x_2, x_1) \supset member_{geq}(x_1, [x_2|x], x_2)),\\ &z_4 : \forall_{x_1,x_2,x_3:nat}\forall_{x:lnat}(member_{geq}(x_1, x, x_3) \supset member_{geq}(x_1, [x_2|x], x_3)),\end{aligned}$$

where

$$\begin{aligned}&e = lambda(x_1.lambda(x_2.lambda(w_1.e_1))),\\ &e_1 = bc(z_2, [sx_1, sx_2], bc(z_2, [x_1, x_2], w_1)).\end{aligned}$$

The definition of dependent type in Δ defines the name $+_2$ for the function $\lambda x.ssx$ and describes one of its logical properties, monotonicity, i.e. if a natural number n_1 is greater than or equal to n_2 then the result of applying $+_2$ to n_1 is greater than or equal to the result of applying $+_2$ to n_2.

Below, a context $(\Delta_1, x : H, \Delta_2)$ is sometimes written simply as (Δ_1, x, Δ_2) without ambiguity, since x determines uniquely H.

It may be shown that the judgement:

$$\vdash \Sigma; \Delta \; basis$$

is derivable in HH^{def}. A derivation of this judgement requires a derivation of Sequent 4.11.

$$\Sigma; z_1, z_2 \Rightarrow e : [\lambda x.ssx/f]\forall_{x_1,x_2:nat}(geq(x_1,x_2) \supset geq(fx_1, fx_2)) \tag{4.11}$$

A derivation of Sequent 4.11 may be uniquely recovered from the proof-term $pair_q(\lambda x.ssx, e)$, up to *conversion* rules.

The problem of given a natural number n and a list of natural numbers ns finding a natural number n_2 in ns greater than or equal to n_1 may be formulated as the goal formula:

$$G = \exists_{x:nat} member_{geq}(n_1^r, ns^r, x),$$

where n_1^r is a representation of n_1 and ns^r is a representation of ns. A natural number n_2 in ns greater than or equal to n_1 exists iff G is achievable w.r.t. $\Sigma; \Delta$ in LFPL. Values for n_2 may be extracted from witnesses for the achievement of G w.r.t. $\Sigma; \Delta$. We study below the case where G is the goal formula $\exists_{x:\tau} member_{geq}(+_2 0, [sss0], x)$.

It may be shown that judgement $\Sigma; \Delta \vdash G \; gf$ is derivable in HH^{def}. Let us attempt to achieve G w.r.t. $\Sigma; \Delta$, i.e. search for a proof-term $?_1$ s.t. there is an HH^{def}-derivation of the sequent $\Sigma; \Delta \Rightarrow ?_1 : G$. Following goal-directed proof-search, the goal is broken up until it becomes atomic, as follows:

$$\frac{\Sigma; \Delta \Rightarrow ?_2 : member_{geq}(+_2 0, [sss0], !_1) \quad \Sigma; \Delta \vdash !_1 : nat}{\Sigma; \Delta \Rightarrow ?_1 : \exists_{x:nat} member_{geq}(+_2 0, [sss0], x)} \Rightarrow \exists,$$

provided the indeterminate $?_1$ is made equal to $(!_1, ?_2)$. Now we search for a proof-term $?_2$ and a term $!_1$ of type nat. The goal is atomic; backchaining is attempted as follows:

$$\frac{\Sigma; \Delta \Rightarrow ?_3 : geq([sss0], +_2 0)}{\Sigma; \Delta \Rightarrow ?_2 : member_{geq}(+_2 0, [sss0], !_1)} \; BC,$$

where $?_2 = bc(z_3, [+_2 0, sss0, []], ?_3)$ and $!_1 = sss0$. The terms $+_2 0$ and $sss0$ may be easily shown to have type nat. Now the set of indeterminates contains solely $?_3$, since $!_1$ is fully determined as the term $sss0$ of type nat. Case there exists $?_3$ s.t. the sequent

$$\Sigma; \Delta \Rightarrow ?_3 : geq(sss0, +_2 0)$$

is derivable, the natural number corresponding to $sss0$ is a possible answer to the original query.

For determining $?_3$ two different approaches are pursued. A first approach to find $?_3$ is by substituting the definiendum $+_2$ by its definiens followed by β-reduction, through an instance of a *conversion* rule, and then performing *backchaining* as below.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\overset{\theta}{\vdash \Sigma; z_1, w_2 : geq(s0,0), z_2, \ldots\ basis}}{\Sigma; z_1, w_2 : geq(s0,0), z_2, \ldots \Rightarrow w_2 : geq(sss0,0)}\ axiom}{\Sigma; \Delta \Rightarrow apply_q(z_2, ss0, w_2.w_2) : geq(s0,0)}\ \forall\Rightarrow}{\Sigma; \Delta \Rightarrow bc(z_2, [s0,0], apply_q(z_2, ss0, w_2.w_2)) : geq(s0,0)}\ BC}{\Sigma; \Delta \Rightarrow bc(z_2, [ss0, s0], bc(z_2, [s0,0], apply_q(z_2, ss0, w_2.w_2))) : geq(sss0, ss0)}\ BC \quad \boxed{CJ_1}}{\Sigma; \Delta \Rightarrow bc(z_2, [ss0, s0], bc(z_2, [s0,0], apply_q(z_2, ss0, w_2.w_2))) : geq(sss0, +_2 0)}\ \equiv_r$$

Derivations of the auxiliary judgements in rules $\forall \Rightarrow$ and BC may be easily constructed. Derivation θ is guaranteed to exist since $\vdash \Sigma; \Delta\ basis$ is derivable. $\boxed{CJ_1}$ is an abbreviation for the judgement

$$\Sigma; \Delta \vdash geq(sss0, ss0) \equiv geq(sss0, +_2 0).$$

A derivation for $\boxed{CJ_1}$ may be obtained by substituting $+_2$ by $\lambda x.ssx$, followed by β-reduction.

This approach finds Proof-Term 4.12 for $?_3$.

$$bc(z_2, [ss0, s0], bc(z_2, [s0,0], apply_q(z_2, ss0, w_2.w_2))) \tag{4.12}$$

This form of searching for $?_3$ essentially reflects the implementation suggested in Sec. 3.7 for HOPLP, the difference being the use of definitions of simple type to replace definienda by definientia.

A second approach to find $?_3$ is by using the definition of dependent type in the program. (Intuitively, we take the occurrence of $+_2$ in the goal as a suggestion for using the definition of dependent type of which $+_2$ is a definiendum.) An instance of def_Σ is attempted, followed by an instance of def_I, followed by *backchaining* on the type of the definition, as shown below.

$$\dfrac{\dfrac{\boxed{S_1} \quad \dfrac{\Sigma; \ldots, +_2 =_{def} \lambda x.sxx : nat \to nat, z : I_1, \ldots \Rightarrow ?_4 : geq(s0,0)}{\Sigma; \ldots, +_2 =_{def} \lambda x.sxx : nat \to nat, z : I_1, \ldots \Rightarrow bc(z, [s0,0], ?_4) : geq(sss0, +_2 0)}\ BC^*}{\Sigma; \ldots, +_2 =_{def} \lambda x.sxx : nat \to nat, z =_{def} e : I_1, \ldots \Rightarrow bc(z, [s0,0], ?_4) : geq(sss0, +_2 0)}\ def_I}{\Sigma; \ldots, (+_2, z) =_{def} (\lambda x.sxx, e) : D_1, \ldots \Rightarrow bc(z, [s0,0], ?_4) : geq(sss0, +_2 0)}\ def_\Sigma$$

In the derivation above, the rule BC^* stands for a combination of *backchaining* with an instance of a conversion rule. D_1 abbreviates the D-formula

$$\Sigma_{f:\tau\to\tau} \forall_{x_1,x_2:\tau} (geq(x_1,x_2) \supset geq(fx_1, fx_2))$$

and I_1 abbreviates the I-formula

$$[+_2/f](\forall_{x_1,x_2:\tau}(geq(x_1,x_2) \supset geq(fx_1, fx_2))).$$

$\boxed{S_1}$ is an abbreviation for the sequent:

$$\Sigma; z_1, z_2, +_2 =_{def} \lambda x.sxx : nat \rightarrow nat \Rightarrow e : I_1.$$

A derivation for $\boxed{S_1}$ is guaranteed to exist; it may be easily obtained from a derivation of Sequent 4.11. Under this approach $?_3$ is made equal to $bc(z, [s0, 0], ?_4)$ and the new problem is determining $?_4$, which may be done by using the formula annotated by z_1 as below.

$$\dfrac{\dfrac{\dfrac{\rho}{\vdash \Sigma; \ldots, z_1, w_2, \ldots\ basis}}{\Sigma; z_1, w_2, \ldots \Rightarrow w_2 : geq(s0, 0)}\ axiom}{\Sigma; z_1, \ldots \Rightarrow apply_q(z_1, s0, w_2.w_2) : geq(s0, 0)}\ \forall \Rightarrow$$

(The existence of derivation ρ is guaranteed, since $\vdash \Sigma; \Delta\ basis$ is derivable. The instance of $\forall \Rightarrow$ requires a derivation for a judgement of the form $\Sigma; \ldots \vdash s0 : nat$, which may be easily constructed.) So, this approach finds Proof-Term 4.13 for $?_3$.

$$bc(z, [s0, 0], apply_q(z_1, s0, w_2.w_2)) \tag{4.13}$$

The result of applying the interpretation of LFPL-witnesses into HOPLP-witnesses to Proof-Term 4.13 is Proof-Term 4.14.

$$bc(z_2, [ss0, s0], bc(z_2, [s0, 0], apply(z_1, s0, w_2.w_2))) \tag{4.14}$$

The result of mapping Proof-Term 4.12 into HOPLP is Proof-Term 4.12 itself, which is equal to Proof-Term 4.14. So, any non-redundant set of witnesses for the achievement of G w.r.t. $\Sigma; \Delta$ cannot contain simultaneously the Proof-Terms 4.12 and 4.13, for they map into the same uniform linear focused proof-term of HOPLP.

The semantics for LFPL makes no constraint on the form of the terms occurring in witnesses for goal-achievement. In general, we are only interested in extracting from a witness the terms for the variables existentially quantified in the initial goal. The example above illustrates that by using definitions some reductions on terms of simple type may be avoided. For example, the proof-term found for $?_2$, which does not correspond to an existentially quantified variable of the goal, has an occurrence of the term $+_2 0$.

The work involved in finding Proof-Terms 4.12 and 4.13 is essentially the same. However, one may formulate goals for which the use of definitions of dependent type is a means of finding shorter derivations.

For example, let us consider forms of achieving the goal

$$G_1 = geq(\underbrace{+_2 \ldots +_2}_{n\ times} 0, \underbrace{+_2 \ldots +_2}_{m\ times} 0),$$

where $m \leq n$, w.r.t. $\Sigma; \Delta$ by using derivations that (i) make no use or (ii) that use the definition of dependent type in Δ.

The only form, up to conversion rules, of deriving G_1 from $\Sigma;\Delta$ making no uses of the definition of dependent type in Δ is by a derivation of the following form:

$$\dfrac{\dfrac{\dfrac{\begin{matrix}\dfrac{\dfrac{\overset{\pi_1}{\vdash \Sigma;\Delta_2\ basis}}{\Sigma;\Delta_2 \Rightarrow w : G_4}\,axiom}{\Sigma;\Delta_1 \Rightarrow e_2 : G_4}\,\forall\Rightarrow \\ \vdots\ {\scriptstyle 2m-1\ times\ BC} \\ \Sigma;\Delta_1 \Rightarrow e_1 : G_3\end{matrix}}{\Sigma;\Delta_1 \Rightarrow e : G_2}\,BC \qquad \overset{\pi_2}{\Sigma;\Delta_1 \vdash G_2 \equiv G_1}}{\Sigma;\Delta_1 \Rightarrow e : G_1}\,\equiv_r}{\Sigma;\Delta \Rightarrow e : G_1}\,def_\Sigma$$

where:

$$\begin{aligned}
G_2 &= geq(\underbrace{s...s}_{2n}0, \underbrace{s...s}_{2m}0);\\
G_3 &= geq(\underbrace{s...s}_{2n-1}0, \underbrace{s...s}_{2m-1}0);\\
G_4 &= geq(\underbrace{s......s}_{2n-2m)}0, 0);\\
e &= bc(z_2, [\underbrace{s...s}_{2n}0, \underbrace{s...s}_{2m}0], e_1);\\
e_1 &= bc(z_2, [\underbrace{s...s}_{2n-1}0, \underbrace{s...s}_{2m-1}0], ...e_2...);\\
e_2 &= apply_q(z_1, \underbrace{s......s}_{2n-2m}0, w.w).
\end{aligned}$$

Derivation π_1 is guaranteed to exist since $\vdash \Sigma;\Delta\ basis$ is derivable and π_2 essentially corresponds to normalisation of the terms

$$\underbrace{+_2...+_2}_{n}0 \text{ and } \underbrace{+_2...+_2}_{m}0$$

to the terms

$$\underbrace{s...s}_{2n}0 \text{ and } \underbrace{s...s}_{2m}0.$$

By using the definition of dependent type in Δ, G_1 may be derived from $\Sigma;\Delta$ in the following way:

$$\dfrac{\dfrac{\overset{\pi_1}{\Sigma;\Delta_2 \Rightarrow e_1 : I} \qquad \dfrac{\begin{matrix}\dfrac{\dfrac{\overset{\pi_2}{\vdash \Sigma;\Delta_3\ basis}}{\Sigma;\Delta_3 \Rightarrow w : G_3}\,axiom}{\Sigma;\Delta_1 \Rightarrow e_3 : G_3}\,\forall\Rightarrow \\ \vdots\ {\scriptstyle m-1\ times\ BC} \\ \Sigma;\Delta_1 \Rightarrow e_2 : G_2\end{matrix}}{\Sigma;\Delta_1 \Rightarrow e : G_1}\,BC}{\Sigma;\Delta_1 \Rightarrow e : G_1}\,def_I}{\Sigma;\Delta \Rightarrow e : G_1}\,def_\Sigma$$

where:

$$
\begin{aligned}
G_2 &= geq(\underbrace{+_2 \ldots +_2}_{n-1} 0, \underbrace{+_2 \ldots +_2}_{m-1} 0);\\
G_3 &= geq(\underbrace{+_2 \ldots +_2}_{n-m} 0, 0);\\
e &= bc(z_2, [\underbrace{+_2 \ldots +_2}_{n} 0, \underbrace{+_2 \ldots +_2}_{m} 0], e_2);\\
e_2 &= bc(z_2, [\underbrace{+_2 \ldots +_2}_{n-1} 0, \underbrace{+_2 \ldots +_2}_{m-1} 0], \ldots e_3 \ldots);\\
e_3 &= apply_q(z_1, \underbrace{+_2 \ldots +_2}_{n-m} 0, w.w).
\end{aligned}
$$

Both π_1 and π_2 are guaranteed to exist since $\vdash \Sigma; \Delta\ basis$ is derivable.

Roughly, the first form of achieving G_1 needs normalisation of the terms

$$\underbrace{+_2 +_2 \ldots +_2}_{n} 0 \text{ and } \underbrace{+_2 +_2 \ldots +_2}_{m} 0,$$

and needs $2m$ times backchaining on the formula annotated by z_2. The second form of achieving G_1 (using the definition of dependent type), needs no normalisation and needs only m times backchaining on the formula resulting from the type of the definition. (Note that in both cases backchaining involves the same work, *i.e.* two applications of $\forall \Rightarrow$, one application of $\supset\Rightarrow$ and an axiom.)

Chapter 5

Implementing LFPL

5.1 Introduction

This chapter is concerned with studying means of implementing the integrated logical and functional programming language LFPL. As defined in Sec 4.6, an implementation of LFPL is a procedure that given a program $\Sigma; \Delta$ and a goal G finds a complete set of witnesses. A witness is a proof-term e s.t. the sequent $\Sigma; \Delta \Rightarrow e : G$ is derivable in HH^{def}. So, an implementation for LFPL may be thought of as a procedure to find derivations in HH^{def}.

Finding a witness for the achievement of G w.r.t. $\Sigma; \Delta$ requires a construction of a derivation whose endsequent is of the form $\Sigma; \Delta \Rightarrow ? : G$. The proof-term obtained for ? is the desired witness. For constructing a derivation of $\Sigma; \Delta \Rightarrow ? : G$, one may attempt rules for deriving sequents whose conclusion has $\Sigma; \Delta$ as antecedent and G as the succedent formula. For using LFPL one first writes a program and then queries about the program, i.e. asks whether or not some goals are achievable w.r.t. the program. Given a program $\Sigma; \Delta$ the first step is to check if the judgement $\vdash \Sigma; \Delta$ *basis* is derivable in HH^{def}. If so, some simplifications may be made when attempting to achieve a goal G w.r.t. $\Sigma; \Delta$. For example, if def_I is attempted there is no need to find a derivation for the left premiss, since such derivation must exist for the judgement $\vdash \Sigma; \Delta$ *basis* to be derivable in HH^{def}, as shown in Proposition 4.2. Other simplifications may be made when an axiom is being attempted. In this case there is no need to prove that the antecedent is a derivable basis, since this is guaranteed from the facts that $\Sigma; \Delta$ is a derivable basis and the rules for deriving sequents preserve bases.

A rule def_Σ is not very convenient for proof-search. Notice that such rule requires the replacement of a definition of dependent type by a simple definition together with a definition of dependent type. A rule def_I has similar inconveniences. Each time def_I is used, for obtaining the antecedent of the left premiss, a definition of dependent type needs to be deleted from the context and an annotated formula needs to be added to the context.

So, for studying forms of implementing LFPL, we define a new calculus called $HH^{def'}$.

This new calculus may be thought of as a calculus obtained from HH^{def} by replacing the rules def_{Σ} and def_{I} by a new form of rule def_{contr}, more suitable for proof-search.

For studying forms of implementing LFPL in terms of $HH^{def'}$, the calculus $HH^{def'}$ must be s.t.: (i) all derivable sequents of $HH^{def'}$ are interpretable into derivable sequents of HH^{def}; (ii) there are enough $HH^{def'}$-derivations for producing complete sets of witnesses. Section 5.2 shows a means of interpreting derivable sequents of $HH^{def'}$ as derivable sequents of HH^{def} and shows that this interpretation is surjective. Thus, criteria (i) and (ii) are verified and an implementation for LFPL may be defined in terms of $HH^{def'}$.

Section 5.3.1 studies the class of *extended uniform linear focused derivations* of $HH^{def'}$, which is complete for LFPL, i.e. extended uniform linear focused derivations are sufficient to find complete sets of witnesses. This class of derivations imposes constraints on the use of def_{contr} similar to the constraints on left rules, i.e. uniformity (the formula in the succedent of the conclusion is atomic), focusing (the side formula is the main formula of the inference above) and linearity (the side formula is used exactly once).

The calculus def'^{EULF} allows exactly the extended uniform linear focused derivations of $HH^{def'}$ having different proof-terms. Thus, def'^{EULF} is complete for LFPL. However, def'^{EULF} is redundant, i.e. there are different derivations in def'^{EULF} whose interpretations into HH are the same.

In Sec. 5.4 are sketched two search procedures for def'^{EULF}, *Proc. 1* and *Proc. 2*. Both procedures find complete [1] sets of witnesses for goal-achievement in LFPL; so, they constitute implementations of LFPL. They differ in the use of definitions of dependent type during search, i.e. in when to attempt def_{contr}. For attempting to use a definition of dependent type, both procedures require the goal to be atomic. *Proc. 2* imposes a further constraint in the use of a definition of dependent type: a definiendum of the definition must occur in the goal. *Proc. 2* has a search space smaller than *Proc. 1*; however, *Proc. 1* may find some derivations which are shorter than their counterparts found by *Proc. 2*. Section 5.4 gives a characterisation of the derivations that may be found by *Proc. 1* and may not be found by *Proc. 2*.

Procedures *Proc. 1* and *Proc. 2* are excessive implementations of LFPL, since they both find redundant sets of witnesses. Section 5.4 puts forward some ideas that may be integrated in *Proc. 1* and *Proc. 2* to eliminate redundancy.

5.2 The Calculus $HH^{def'}$

This section introduces the calculus $HH^{def'}$. The calculus $HH^{def'}$ is essentially the same as HH^{def}, except for the rules to deal with definitions of dependent type, which are replaced by rules more convenient for proof-search. The calculus $HH^{def'}$ is sound for derivable formulas and complete for proof-terms deriving a formula w.r.t. HH^{def}.

[1] Completeness is understood within the limitations of depth-first search and unification of λ-terms.

The classes of objects used in $HH^{def'}$ are the same as those used in HH^{def}. They are defined by the same grammars except for proof-terms. In $HH^{def'}$ are allowed proof-terms of the form

$$contr(x, x.e).$$

This form of proof-term is used in the new form of rule def_{contr} of $HH^{def'}$. In proof-terms of the form $contr(x, x_1.e)$, the variable x_1 is called a *binder* of *scope* e; any occurrence of x_1 in e is said to be *bound*.

The forms of judgement in $HH^{def'}$ are the same as those in HH^{def}. Except for sequents, we want the derivable judgements in $HH^{def'}$ to be the same as the derivable judgements in HH^{def}. However, HH^{def}-derivable bases depend upon derivable sequents. One method of solving this problem is by introducing two different forms of sequents as follows:

$$\text{(i) } \Sigma; \Delta \Rightarrow_{basis} e : G;$$
$$\text{(ii) } \Sigma; \Delta \Rightarrow e : G.$$

The form of sequent (i) is used only in defining derivable bases; it corresponds to the form of sequents used in the non-principal part of a sequent derivation in HH^{def}. The derivable sequents of form (i) are the same as those of HH^{def}, *i.e.* $\Sigma; \Delta \Rightarrow_{basis} e : G$ is derivable in $HH^{def'}$ iff $\Sigma; \Delta \Rightarrow e : G$ is derivable in HH^{def}. (When there is no danger of confusion, sequents of form (i) are simply called *sequents* and written with the symbol $\Rightarrow$.) Sequents of form (ii) correspond to the sequents used in the principal part of sequent derivations in HH^{def}. Sequents of form (ii) are simply called *sequents*. The rules for deriving sequents are the rules allowed in HH^{def}, except rules def_{Σ} and def_I, together with the new form of rule def_{contr}, shown in Fig. 5.1, where *Ipart* is defined as follows:

$$Ipart(x =_{def} e : I) =_{def} \langle I, x\rangle;$$
$$Ipart((x, p) =_{def} pair_q(\Lambda, e) : \Sigma_{y:\tau} D) =_{def} Ipart(p =_{def} e : [x/y]D).$$

In the rule def_{contr} of Fig. 5.1, the formula I is called the *side formula* and $p =_{def} e : D$ is called the *main definition*.

$$\frac{\Sigma; \Delta_1, p =_{def} e : D, x_1 : I, \Delta_2 \Rightarrow e_1 : G}{\Sigma; \Delta_1, p =_{def} e : D, \Delta_2 \Rightarrow contr(x, x_1.e_1) : G}\ def_{contr}\ ,$$

$$Ipart(p =_{def} e : D) = \langle I, x\rangle \text{ and } x_1 \notin \Delta_2$$

Figure 5.1: def_{contr} rule.

When searching for a proof-term ? s.t. a sequent

$$\Sigma; \Delta_1, p =_{def} e : D, \Delta_2 \Rightarrow ? : G$$

is derivable, def_{contr} may be attempted. For doing so, it suffices to calculate $Ipart(p =_{def} e : D)$, obtaining a pair $\langle I, x\rangle$, and add I (the *I-part* of the definition), annotated by a new variable, to the context. Intuitively, a rule def_{contr}, when read from conclusion to premises, may be thought of as a form of making a copy of the I-part of the type of the main definition. This rule leaves definitions of dependent type unchanged.

As mentioned before, in order to guarantee that an implementation for LFPL may be sought amongst search procedures for $HH^{def'}$, the calculus $HH^{def'}$ must satisfy two properties. The first property is a form of soundness result w.r.t LFPL, i.e. if a G-formula G is derivable in $HH^{def'}$ w.r.t. a basis $\Sigma;\Delta$ then G is achievable w.r.t. $\Sigma;\Delta$ in LFPL. The second property is a form of completeness result w.r.t LFPL, i.e. it is possible to find complete sets of witnesses, for the achievement of a goal G w.r.t. a program $\Sigma;\Delta$ in LFPL, within the set of proof-terms for deriving G w.r.t. $\Sigma;\Delta$ in $HH^{def'}$.

Theorems 5.1 and 5.2, below, state the fundamental results for proving that the two properties above mentioned hold for $HH^{def'}$. These theorems show methods of interpreting derivable sequents of $HH^{def'}$ as derivable sequents of HH^{def} and vice-versa.

Figure 5.2 defines the mapping $d'd$ from $HH^{def'}$-proof-terms into HH^{def}-proof-terms. This

$$\begin{aligned}
&d'd(pair(e_1, e_2)) =_{def} pair(d'd(e_1), d'd(e_2))\\
&d'd(inl(e)) =_{def} inl(d'd(e))\\
&d'd(inr(e)) =_{def} inr(d'd(e))\\
&d'd(lambda(x.e)) =_{def} lambda(x.d'd(e))\\
&d'd(pair_q(\Lambda, e)) =_{def} pair_q(\Lambda, d'd(e))\\
&d'd(lambda_q(x.e)) =_{def} lambda_q(x.d'd(e))\\
&d'd(x) =_{def} x\\
&d'd(contr(x, x_1.e)) =_{def} \{x/x_1\}d'd(e)\\
&d'd(splitl(x, x_1.e)) =_{def} splitl(x, x_1.d'd(e))\\
&d'd(splitr(x, x_1.e)) =_{def} splitr(x, x_1.d'd(e))\\
&d'd(apply(x, e, x_1.e_1)) =_{def} apply(x, d'd(e), x_1.d'd(e_1))\\
&d'd(apply_q(x, \Lambda, x_1.e)) =_{def} apply_q(x, \Lambda, x_1.d'd(e))
\end{aligned}$$

Figure 5.2: Mapping $d'd$ from $HH^{def'}$-proof-terms to HH^{def}-proof-terms.

mapping uses the mapping on HH^{def}-proof-terms associated with contraction, see Proposition 4.4. Recall that contraction on proof-terms essentially replaces free occurrences of a variable by another variable.

Let G be a derivable formula w.r.t. $\Sigma;\Delta$ in $HH^{def'}$. Then, by Theorem 5.1, G is derivable w.r.t. $\Sigma;\Delta$ in HH^{def}, thus G is achievable w.r.t. $\Sigma;\Delta$ in LFPL. The following auxiliary lemma is used in proving Theorem 5.1; it provides a means of contracting a formula in the context with the I-part of a definition of dependent type, eliminating the definition.

Lemma 5.1 *If there is a HH^{def}-derivation of $\Sigma; \Delta_1, x =_{def} e : I, z : I, \Delta_2 \Rightarrow e_1 : G$, where $z \notin \Delta_2$, then $\Sigma; \Delta_1, x : I, \Delta_2 \Rightarrow \{x/z\}e_1 : G$ is derivable in HH^{def}.*

Proof: Let π be a HH^{def}-derivation of $\Sigma; \Delta_1, x =_{def} e : I, z : I, \Delta_2 \Rightarrow e_1 : G$. The proof follows by induction on the structure of π.

Case π is an axiom of the form:

$$\frac{\vdash \Sigma; \Delta_1, x =_{def} e : I, z : I, \Delta_2 \ basis}{\Sigma; \Delta_1, x =_{def} e : I, z : I, \Delta_2 \Rightarrow z : G} \ axiom.$$

(Note that $I = G$.) From a derivation of the judgement

$$\vdash \Sigma; \Delta_1, x =_{def} e : I, z : I, \Delta_2 \ basis,$$

one may easily construct a derivation of the judgement

$$\vdash \Sigma; \Delta_1, x : I, z : I, \Delta_2 \ basis.$$

So, for $z \notin \Delta_2$, there is a derivation of

$$\vdash \Sigma; \Delta_1, x : I, \Delta_2 \ basis.$$

Thus, the following derivation may be formed:

$$\frac{\vdash \Sigma; \Delta_1, x : I, \Delta_2 \ basis}{\Sigma; \Delta_1, x : I, \Delta_2 \Rightarrow x : G} \ axiom.$$

Note that the identity $\{x/z\}z = x$ holds.

Case the last step of π is a rule def_I of the form:

$$\frac{\Sigma; \Delta_1 \Rightarrow e : I \quad \Sigma; \Delta_1, x : I, z : I, \Delta_2 \Rightarrow e_1 : G}{\Sigma; \Delta_1, x =_{def} e : I, z : I, \Delta_2 \Rightarrow e_1 : G} \ def_I.$$

Then, by Proposition 4.4, since $z \notin \Delta_2$, from the derivation of the left premiss, one may construct a HH^{def}-derivation of

$$\Sigma; \Delta_1, x : I, \Delta_2 \Rightarrow \{x/z\}e_1 : G.$$

Case the last step of π is a rule def_I of the form:

$$\frac{\Sigma; \Delta_1, x =_{def} e : I, z : I, \Delta_{21} \Rightarrow e_2 : I_1 \quad \Sigma; \Delta_1, x =_{def} e : I, z : I, \Delta_{21}, x_1 : I_1, \Delta_{22} \Rightarrow e_1 : G}{\Sigma; \Delta_1, x =_{def} e : I, z : I, \Delta_{21}, x_1 =_{def} e_2 : I_1, \Delta_{22} \Rightarrow e_1 : G} \ def_I.$$

By the I.H., there are HH^{def}-derivations of the sequents:

(i) $\Sigma; \Delta_1, x : I, \Delta_{21} \Rightarrow \{x/z\}e_2 : I_1$;
(ii) $\Sigma; \Delta_1, x : I, \Delta_{21}, x_1 : I_1, \Delta_{22} \Rightarrow \{x/z\}e_1 : G$.

It may be shown that, since $z \notin e_2$, the identity $\{x/z\}e_2 = e_2$ holds. Thus, the derivation below may be formed.

$$\frac{\Sigma; \Delta_1, x : I, \Delta_{21} \Rightarrow e_2 : I_1 \quad \Sigma; \Delta_1, x : I, \Delta_{21}, x_1 : I_1, \Delta_{22} \Rightarrow \{x/z\}e_1 : G}{\Sigma; \Delta_1, x : I, \Delta_{21}, x_1 =_{def} e_2 : I_1, \Delta_{22} \Rightarrow \{x/z\}e_1 : G}\ def_I.$$

The remainder cases follow directly by the I.H.. □

Theorem 5.1 *If $\Sigma; \Delta \Rightarrow e : G$ is derivable in $HH^{def'}$ then $\Sigma; \Delta \Rightarrow d'd(e) : G$ is derivable in HH^{def}.*

Proof: Let π be a $HH^{def'}$-derivation of $\Sigma; \Delta \Rightarrow e : G$. The proof follows by induction on the height[2] (of the principal part) of π.

Case the height of π is 1, then π is an axiom, say:

$$\frac{\vdash \Sigma; \Delta_1, x : H, \Delta_2\ basis}{\Sigma; \Delta_1, x : H, \Delta_2 \Rightarrow x : H}\ axiom.$$

Then, the same sequence of inferences may be performed in $HH^{def'}$. (Recall that derivable bases of $HH^{def'}$ are the same as derivable bases of HH^{def}.)

Case the height of π is greater than 1 and the last step of π is of the form:

$$\frac{\overset{\pi_1}{\Sigma; \Delta_1, p =_{def} e : D, x_1 : I, \Delta_2 \Rightarrow e_1 : G}}{\Sigma; \Delta_1, p =_{def} e : D, \Delta_2 \Rightarrow contr(x, x_1.e_1) : G}\ def_{contr}\ ,$$

where $Ipart(p =_{def} e : D) = \langle I, x \rangle$ and $x_1 \notin \Delta_2$.

It may be easily shown that there is a derivation π_2, whose height is smaller or equal to the height of π_1, of the sequent

$$\Sigma; \Delta_1, x'_1 =_{def} \Lambda_1 : \tau_1, ..., x'_n =_{def} \Lambda_n : \tau_n, x =_{def} e_2 : I, x_1 : I, \Delta_2 \Rightarrow e_1 : G,$$

where

$$(p =_{def} e : D) = ((x'_1, ...(x'_n, x)...) =_{def} pair_q(\Lambda_1, ...pair_q(\Lambda_n, e_2)...) : D).$$

So, by the I.H., there is a HH^{def}-derivation of

$$\Sigma; \Delta_1, x'_1 =_{def} \Lambda_1 : \tau_1, ..., x'_n =_{def} \Lambda_n : \tau_n, x =_{def} e_2 : I, x_1 : I, \Delta_2 \Rightarrow d'd(e_1) : G.$$

Since $x_1 \notin \Delta_2$, by Lemma 5.1, there is a HH^{def}-derivation ρ_1 of

$$\Sigma; \Delta_1, x'_1 =_{def} \Lambda_1 : \tau_1, ..., x'_n =_{def} \Lambda_n : \tau_n, x : I, \Delta_2 \Rightarrow \{x/x_1\}d'd(e_1) : G.$$

[2]The *height* of a derivation is 1, if it is an axiom, and is 1 + the maximum of the height of the premisses, for any other rule.

From the derivation π_2, by using Proposition 4.1, there is a HH^{def}-derivation ρ_2 of

$$\Sigma; \Delta_1, x'_1 =_{def} \Lambda_1 : \tau_1, ..., x'_n =_{def} \Lambda_n : \tau_n \Rightarrow e_2 : I.$$

From ρ_1 and ρ_2, by using a rule def_I, there is a HH^{def}-derivation of

$$\Sigma; \Delta_1, x'_1 =_{def} \Lambda_1 : \tau_1, ..., x'_n =_{def} \Lambda_n : \tau_n, x =_{def} e_2 : I, \Delta_2 \Rightarrow \{x/x_1\}d'd(e_1) : G.$$

Now, by using n times def_Σ, a HH^{def}-derivation of

$$\Sigma; \Delta_1, p =_{def} e : D, \Delta_2 \Rightarrow \{x/x_1\}d'd(e_1) : G$$

may be formed. Note that $d'd(contr(x, x_1.e_1)) = \{x/x_1\}d'd(e_1)$.

The other cases where the height of π is greater than 1 follow easily from the I.H.. □

Theorem 5.2 below shows that every witness for goal-achievement in LFPL may be obtained in $HH^{def'}$. Lemma 5.2 is used in proving Theorem 5.2. This lemma is proved by induction on the *principal* def_I*-height* of a derivation, where the *principal* def_I*-height* of a sequent derivation π in HH^{def} is inductively defined on the structure of π as follows:

- case the last step of π is an axiom, the principal def_I-height of π is 1;
- case the last step of π is def_I, the principal def_I-height of π is 1 plus the principal def_I-height of the derivation of the principal (right) premiss;
- otherwise, the principal def_I-height of π is 1 plus the maximum of the principal def_I-heights of the derivations of the premisses.

Lemma 5.2 *Let $\Sigma; \Delta_1 \Rightarrow e : I$ be derivable in HH^{def} and let π be a HH^{def}-derivation of the sequent*

$$\Sigma; \Delta_1, x : I, \Delta_2 \Rightarrow e_1 : G.$$

Then, for every x_1 s.t. $x_1 \notin (\Delta_1, x : I, \Delta_2)$, the sequent

$$\Sigma; \Delta_1, x =_{def} e : I, x_1 : I, \Delta_2 \Rightarrow \{x_1/x\}e_1 : G$$

has a HH^{def}-derivation of principal def_I-height equal to the principal def_I-height of π.

Proof: The proof follows by induction on the structure of π.

Case the last step of π is an axiom of the form:

$$\frac{\vdash \Sigma; \Delta_1, x : I, \Delta_2 \; basis}{\Sigma; \Delta_1, x : I, \Delta_2 \Rightarrow x : I} \; axiom.$$

Then, from a derivation of $\vdash \Sigma; \Delta_1, x : I, \Delta_2 \; basis$, a derivation of $\vdash \Sigma; \Delta_1, x : I, x_1 : I, \Delta_2 \; basis$ may be constructed, since $\Sigma; \Delta_1 \vdash I \; hf$ is derivable and $x_1 \notin (\Delta_1, x : I, \Delta_2)$. From π and from a derivation of $\Sigma; \Delta_1 \Rightarrow e : I$, that exists by hypothesis, a derivation of

$$\vdash \Sigma; \Delta_1, x =_{def} e : I, x_1 : I, \Delta_2 \; basis$$

may be easily constructed. So, the following derivation may be formed.

$$\frac{\vdash \Sigma; \Delta_1, x =_{def} e : I, x_1 : I, \Delta_2 \; basis}{\Sigma; \Delta_1, x =_{def} e : I, x_1 : I, \Delta_2 \Rightarrow x_1 : I} \; axiom.$$

Observe that $\{x_1/x\}x = x_1$ by definition. The derivation above has principal def_I-height 1, which is equal to the principal def_I-height of π.

Case last step of π is an axiom of the form:

$$\frac{\vdash \Sigma; \Delta_1, x : I, \Delta_{21}, x_2 : I_1, \Delta_{22} \; basis}{\Sigma; \Delta_1, x : I, \Delta_{21}, x_2 : I_1, \Delta_{22} \Rightarrow x_2 : I_1} \; axiom.$$

Similar arguments to those above may be used to construct a derivation of

$$\vdash \Sigma; \Delta_1, x =_{def} e : I, x_1 : I, \Delta_{21}, x_2 : I_1, \Delta_{22} \; basis.$$

Then, the following derivation of principal def_I-height 1 may be formed.

$$\frac{\vdash \Sigma; \Delta_1, x =_{def} e : I, x_1 : I, \Delta_{21}, x_2 : I_1, \Delta_{22} \; basis}{\Sigma; \Delta_1, x =_{def} e : I, x_1 : I, \Delta_{21}, x_2 : I_1, \Delta_{22} \Rightarrow x_2 : I_1} \; axiom.$$

Note that the identity $\{x_1/x\}x_2 = x_2$ holds, since $x \neq x_2$. The case where the main formula of an axiom is in Δ_1 is similar to this case.

Case last step of π is a def_I rule of the form:

$$\frac{\overset{\pi_1}{\Sigma; \Delta_1, x : I, \Delta_{21} \Rightarrow e_2 : I_1} \quad \overset{\pi_2}{\Sigma; \Delta_1, x : I, \Delta_{21}, x_2 : I_1, \Delta_{22} \Rightarrow e_1 : G}}{\Sigma; \Delta_1, x : I, \Delta_{21}, x_2 =_{def} e_2 : I_1, \Delta_{22} \Rightarrow e_1 : G} \; def_I.$$

Then, by the I.H., there is a derivation σ_1, having the same principal def_I-height as π_2, of sequent 5.1, for every $x_1 \notin (\Delta_1, x : I, \Delta_{21}, x_2 : I_1, \Delta_{22})$.

$$\Sigma; \Delta_1, x =_{def} e : I, x_1 : I, \Delta_{21}, x_2 : I_1, \Delta_{22} \Rightarrow \{x_1/x\}e_1 : G \tag{5.1}$$

Derivation 5.2 may be formed, where σ_2 exists by hypothesis and σ_3 may be obtained from π_1 by weakening, since $x_1 \notin (\Delta_1, x : I, \Delta_{21})$.

$$\frac{\overset{\sigma_2}{\Sigma; \Delta_1 \Rightarrow e : I} \quad \overset{\sigma_3}{\Sigma; \Delta_1, x : I, x_1 : I, \Delta_{21} \Rightarrow e_2 : I_1}}{\Sigma; \Delta_1, x =_{def} e : I, x_1 : I, \Delta_{21} \Rightarrow e_2 : I_1} \; def_I \tag{5.2}$$

Putting together Derivation 5.2 and derivation σ_1 of Sequent 5.1, by using def_I, one constructs a derivation of Sequent 5.3.

$$\Sigma; \Delta_1, x =_{def} e : I, x_1 : I, \Delta_{21}, x_2 =_{def} e_2 : I_1, \Delta_{22} \Rightarrow \{x_1/x\}e_1 : G \tag{5.3}$$

Note that the principal def_I-height of such derivation is the same as the principal def_I-height of π, i.e. 1 plus the principal def_I-height of σ_1.

The other cases follow directly by the I.H.. □

Theorem 5.2 *If $\Sigma; \Delta \Rightarrow e : G$ is derivable in HH^{def} then there exists e_1 s.t. $\Sigma; \Delta \Rightarrow e_1 : G$ is derivable in $HH^{def'}$ and $d'd(e_1) = e$.*

Proof: Let π be a HH^{def}-derivation of $\Sigma; \Delta \Rightarrow e : G$. The proof follows by induction on the principal def_I-height of π. The cases where the last step of π is neither def_Σ nor def_I follow easily by the I.H.. Below are studied the cases where the last step of π is either def_Σ or def_I.

Case the last step of π is def_I:

$$\frac{\Sigma; \Delta_1 \Rightarrow e : I \quad \overset{\pi_1}{\Sigma; \Delta_1, x : I, \Delta_2 \Rightarrow e_1 : G}}{\Sigma; \Delta_1, x =_{def} e : I, \Delta_2 \Rightarrow e_1 : G} \; def_I.$$

Let $x_1 \notin (\Delta_1, x : I, \Delta_2)$. Then, by Lemma 5.2, there is an HH^{def}-derivation, whose principal def_I-height is the same as that of π_1, of the sequent

$$\Sigma; \Delta_1, x =_{def} e : I, x_1 : I, \Delta_2 \Rightarrow \{x_1/x\}e_1 : G.$$

So, by the I.H., there exists e_2 s.t. $d'd(e_2) = \{x_1/x\}e_1$ and there is a $HH^{def'}$-derivation of the sequent

$$\Sigma; \Delta_1, x =_{def} e : I, x_1 : I, \Delta_2 \Rightarrow e_2 : G.$$

Thus, the following $HH^{def'}$-derivation may be formed:

$$\frac{\Sigma; \Delta_1, x =_{def} e : I, x_1 : I, \Delta_2 \Rightarrow e_2 : G}{\Sigma; \Delta_1, x =_{def} e : I, \Delta_2 \Rightarrow contr(x, x_1.e_2) : G} \; def_{contr} \, .$$

Note that $Ipart(x =_{def} e : I) = \langle I, x\rangle$. Since $d'd(e_2) = \{x_1/x\}e_1$, $d'd(contr(x, x_1.e_2))$ is equal to $\{x/x_1\}(\{x_1/x\}e_1)$, which may be shown equal to e_1, for x_1 may have no free occurrences in e_1.

Case the last step of π is def_Σ:

$$\frac{\Sigma; \Delta_1, x =_{def} \Lambda : \tau, p =_{def} e : [x/y] : D, \Delta_2 \Rightarrow e_1 : G}{\Sigma; \Delta_1, (x, p) =_{def} pair(\Lambda, e) : \Sigma_{y:\tau} D, \Delta_2 \Rightarrow e_1 : G} \; def_\Sigma.$$

By the I.H., there exists e_2 s.t. $d'd(e_2) = e_1$ and the sequent

$$\Sigma; \Delta_1, x =_{def} \Lambda : \tau, p =_{def} e : [x/y]D, \Delta_2 \Rightarrow e_2 : G$$

has a $HH^{def'}$-derivation σ. Then, it may be shown by induction on the structure of σ, as sketched below, that Sequent 5.4 is derivable in $HH^{def'}$.

$$\Sigma; \Delta_1, (x, p) =_{def} pair_q(\Lambda, e) : \Sigma_{y:\tau} D, \Delta_2 \Rightarrow e_2 : G \tag{5.4}$$

The most interesting case is when the last step of σ is of the form:

$$\frac{\Sigma; \Delta_1, x =_{def} \Lambda : \tau, p =_{def} e : [x/y]D, x_2 : I, \Delta_2 \Rightarrow e_3 : G}{\Sigma; \Delta_1, x =_{def} \Lambda : \tau, p =_{def} e : [x/y]D, \Delta_2 \Rightarrow contr(x_1, x_2.e_3) : G} \; def_{contr} \, ,$$

where $Ipart(p =_{def} e : [x/y]D) = \langle I, x_1\rangle$. In this case a similar step may be used for deriving Sequent 5.4, since $Ipart((x, p) =_{def} pair_q(\Lambda, e) : \Sigma_{y:\tau} D) = Ipart(p =_{def} e : [x/y]D)$, by definition of *Ipart*. □

5.3 A Complete Class of $HH^{def'}$-derivations for LFPL

The previous section shows that an implementation for LFPL may be sought amongst search procedures for $HH^{def'}$. This section studies a complete class of $HH^{def'}$-derivations for LFPL, i.e. a class of $HH^{def'}$-derivations within which complete sets of witnesses for goal-achievement in LFPL may be obtained.

5.3.1 Extended Uniform Linear Focused Derivations

The work in developing HH^{def} would be wasted if we were to consider implementations of LFPL that search for derivations using no definitions of dependent type. (Recall that by using definitions of dependent type, derivations for goal-achievement may become much shorter.) In some sense, we want to use definitions of dependent type whenever possible, so long as they are relevant for achieving a goal.

Below is studied a complete class of derivations for LFPL, called the class of *extended uniform linear focused* (EULF) *derivations.* Roughly, this class of derivations is an extension of ULF-derivations, in the context of HH, that allows some constrained forms of def_{contr}. The constraints imposed on def_{contr} may be thought of as: (i) uniformity (the formula in the succedent's conclusion is atomic); (ii) focusing (in a derivation of the premiss of a def_{contr} rule, the main formula of the last step is the side formula of def_{contr}); (iii) linearity (each time the I-part of a definition of dependent type is required, a new copy must be made by using a def_{contr} rule). These constraints are imposed in analogy to the constraints on left rules. (Recall that uniform linear focused derivations are isomorphic to expanded normal deductions and there are efficient methods to search for these derivations.) Theorem 5.3 below shows that every derivation in $HH^{def'}$ may be transformed into a EULF-derivation by means of permutations.

Definition 5.1 *The grammar defining the sets of* extended uniform focused *proof-terms* e_{euf} *and* atomic extended uniform focused *proof-terms* $a^{x_i}_{euf}$ *of* head variable x_i *of* $HH^{def'}$ *is as follows:*

$$\begin{array}{rcl} e_{euf} & ::= & a^{x_i}_{euf} \mid pair(e_{euf}, e_{euf}) \mid inl(e_{euf}) \mid inr(e_{euf}) \\ & \mid & lambda(x.e_{euf}) \mid pair_q(\Lambda, e_{euf}) \mid lambda_q(x.e_{euf}); \\ a^{x_i}_{euf} & ::= & x_i \mid contr(x_i, x_j.a^{x_j}_{euf}) \\ & \mid & splitl(x_i, x_j.a^{x_j}_{euf}) \mid splitr(x_i, x_j.a^{x_j}_{euf}) \\ & \mid & apply(x_i, e_{euf}, x_j.a^{x_j}_{euf}) \mid apply_q(x_i, \Lambda, x_j.a^{x_j}_{euf}). \end{array}$$

Definition 5.2 *A proof-term is called* extended linear *(*extended affine*) if every variable x bound by a left constructor or by contr occurs exactly once (at most once) in the scope of x.*

Definition 5.3 *A proof-term is called* extended uniform linear focused *if it is extended linear and extended uniform focused.*

Definition 5.4 *A derivation of a sequent is called* extended uniform linear focused *(extended uniform focused) if its proof-term is extended uniform linear focused (extended uniform focused).*

Theorem 5.3 provides a means of showing completeness of EULF-derivations; it shows that if a sequent $\Sigma; \Delta \Rightarrow e : G$ is derivable in $HH^{def'}$ then there exists a sequence of proof-term transformations, preserving derivability, for obtaining an extended uniform linear focused proof-term e_1 s.t. $\Sigma; \Delta \Rightarrow e_1 : G$ is derivable in $HH^{def'}$.

Definition 5.5 (RS_{eulf}) *RS_{eulf} is the rewriting system on $HH^{def'}$-proof-terms consisting of the rules in Figs. 2.8, 2.9, 2.10 and 2.11, where proof-terms are seen as $HH^{def'}$-proof-terms, together with the rules in Fig. 5.3 involving the constructor contr. The rewrite relation induced by RS_{eulf} is called $\triangleright_{eulf}$. A proof-term e_1* is reducible *by RS_{eulf} to a proof-term e_2 if the pair (e_1, e_2) is in the transitive closure of $\triangleright_{eulf}$.*

(i') $contr(x, x_1.pair(e_1, e_2)) \triangleright pair(contr(x, x_1.e_1), contr(x, x_1.e_2))$

(ii') $contr(x, x_1.inl(e)) \triangleright inl(contr(x, x_1.e))$,

(iii') $contr(x, x_1.inr(e)) \triangleright inr(contr(x, x_1.e))$,

(iv') $contr(x, x_1.lambda(x_2.e)) \triangleright lambda(x_2.contr(x, x_1.e)), x_2 \neq x,\ x_2 \neq x_1$

(v') $contr(x, x_1.pair_q(\Lambda, e)) \triangleright pair_q(\Lambda, contr(x, x_1.e))$

(vi') $contr(x, x_1.lambda_q(x_2.e)) \triangleright lambda_q(x_2.contr(x, x_1.e)), x_2 \neq x,\ x_2 \neq x_1$

(vii') $contr(x, x_1.splitl(x_2, x_3.e)) \triangleright splitl(x_2, x_3.contr(x, x_1.e)), x_1 \neq x_2,\ x_3 \neq x,\ x_1 \neq x_3$

(viii') $contr(x, x_1.splitr(x_2, x_3.e)) \triangleright splitr(x_2, x_3.contr(x, x_1.e)), x_1 \neq x_2,\ x_3 \neq x,\ x_1 \neq x_3$

(ix') $contr(x, x_1.apply(x_2, e, x_3.e_1)) \triangleright apply(x_2, contr(x, x_1.e), x_3.contr(x, x_1.e_1))$,
$x_1 \neq x_2,\ x_3 \neq x,\ x_1 \neq x_3$

(x') $contr(x, x_1.apply_q(x_2, \Lambda, x_3.e_1)) \triangleright apply_q(x_2, \Lambda, x_3.contr(x, x_1.e_1)), x_1 \neq x_2,\ x_3 \neq x,\ x_1 \neq x_3$

(xi') $contr(x, x_1.contr(x_2, x_3.e)) \triangleright contr(x_2, x_3.contr(x, x_1.e)), x_1 \neq x_2,\ x_3 \neq x,\ x_1 \neq x_3$

(xii') $splitl(x, x_1.contr(x_2, x_3.e)) \triangleright contr(x_2, x_3.splitl(x, x_1.e)),\ x_1 \neq x_2,\ x_3 \neq x,\ x_1 \neq x_3$

(xiii') $splitr(x, x_1.contr(x_2, x_3.e)) \triangleright contr(x_2, x_3.splitr(x, x_1.e)),\ x_1 \neq x_2,\ x_3 \neq x,\ x_1 \neq x_3$

(xiv') $apply(x, e, x_1.contr(x_2, x_3.e_1)) \triangleright contr(x_2, x_3.apply(x, e, x_1.e_1))\ x_1 \neq x_2,\ x_3 \neq x,\ x_1 \neq x_3$

(xv') $apply_q(x, \Lambda, x_1.contr(x_2, x_3.e)) \triangleright contr(x_2, x_3.apply_q(x, \Lambda, x_1.e))\ x_1 \neq x_2,\ x_3 \neq x,\ x_1 \neq x_3$

(xvi') $contr(x, x_1.e) \triangleright e, x_1 \notin e$

(xvii') $contr(x, x_1.e) \triangleright contr(x, x_1.contr(x, x_2.e_1)), x_2 \notin e$,
e_1 is obtained from e by replacing one of the occurrences of x_1 by x_2.

Figure 5.3: Permutations involving def_{contr}.

Lemma 5.3 *For every rule $e_1 \triangleright e_2$ of RS_{eulf}, if $\Sigma; \Delta \Rightarrow e_1 : G$ is derivable then $\Sigma; \Delta \Rightarrow e_2 : G$ is derivable.*

Proof: By similar arguments to the proof of Theorem 2.6 □

Before proving Theorem 5.3 the following auxiliary result is shown.

Lemma 5.4 *Let e_{euf_1} be an extended uniform focused proof-term. Then, every proof-term of the form*

$$contr(x, x_1.e_{euf_1}),$$

which is not extended uniform focused, is reducible by RS_{eulf} to an extended uniform focused proof-term e_{euf}. Further, if e_{euf_1} is of the form $a^{x_2}_{euf}$, for some $x_2 \neq x_1$, then e_{euf} is of the form $a^{x_2}_{euf_1}$.

Proof: The proof is by induction on the structure of e_{euf_1}.

(i) If e_{euf_1} is of the form $lambda(x_2.e_{euf_2})$, then rule (iv') may be applied to $contr(x, x_1.e_{euf_1})$ obtaining

$$lambda(x_2.contr(x, x_1.e_{euf_2})).$$

If $contr(x, x_1.e_{euf_2})$ is extended uniform focused, then $lambda(x_2.e_{euf_2})$ is reducible by RS_{eulf} to the extended uniform focused proof-term $lambda(x_2.contr(x, x_1.e_{euf_2}))$. Otherwise, by the I.H., $contr(x, x_1.e_{euf_2})$ is reducible by RS_{eulf} to an extended uniform focused proof-term e_{euf_3}; so, $contr(x, x_1.e_{euf_1})$ is reducible by RS_{eulf} to the extended uniform focused proof-term $lambda(x_2.e_{euf_3})$.

(ii) The other cases where the outermost constructor of e_{euf_1} is a right constructor follow by similar arguments to those used in (i).

(iii) Case e_{euf_1} is of the form $a^{x_2}_{euf}$.

(a) It may not be the case that $a^{x_2}_{euf} = x_2$ and $x_2 = x_1$, otherwise the proof-term $contr(x, x_1.e_{euf_1})$ is extended uniform focused.

(b) If $a^{x_2}_{euf} = x_2$ and $x_2 \neq x_1$ then $contr(x, x_1.x_2)$ is reducible, by rule (xvi'), to x_2, which is an extended uniform focused proof-term of the form $a^{x_2}_{euf_1}$.

(c) It may not be the case that $a^{x_2}_{euf} = apply(x_2, e_{euf_2}, x_3.a^{x_3}_{euf_1})$ and $x_2 = x_1$, otherwise the proof-term $contr(x, x_1.a^{x_2}_{euf})$ is extended uniform focused.

(d) If $a^{x_2}_{euf} = apply(x_2, e_{euf_2}, x_3.a^{x_3}_{euf_1})$ and $x_2 \neq x_1$, then $contr(x, x_1.e_{euf_1})$ is reducible, by rule (ix'), to

$$apply(x_2, contr(x, x_1.e_{euf_2}), x_3.contr(x, x_1.a^{x_3}_{euf_1})).$$

By the I.H.: $contr(x, x_1.e_{euf_2})$ is equal or is reducible by RS_{eulf} to an extended uniform focused proof-term e_{euf_3}; and $contr(x, x_1.a^{x_3}_{euf_1})$ is reducible by RS_{eulf} to a uniform focused proof-term of the form $a^{x_3}_{euf_2}$. Thus, the proof-term

$$apply(x_2, e_{euf_3}, x_3.a^{x_3}_{euf_2})$$

is extended uniform focused of the form $a^{x_2}_{euf_1}$ and $contr(x, x_1.e_{uf_1})$ is reducible by RS_{eulf} to it.

(e) The other cases where the outermost constructor of $a^{x_2}_{euf}$ is either a left constructor or *contr* follow by similar arguments to those used in (a)-(d).

□

Theorem 5.3 *EULF-derivations are complete for $HH^{def'}$.*

Proof: The arguments used for proving this result are very similar to those used for proving Theorem 3.4. We give a sketch of the proof below.

(i) The first step is showing that every proof-term e, which is not extended uniform focused, is reducible by RS_{eulf} to a uniform focused proof-term. The proof follows by induction on the structure of e.

Case the outermost constructor of e is a right constructor, the proof follows immediately by the I.H..

Case e is of the form $contr(x, x_1.e_1)$. Then, e_1 is equal or is, by the I.H., reducible by RS_{eulf} to an extended uniform focused proof-term. So, $contr(x, x_1.e_1)$ is extended uniform focused or is, by Lemma 5.4, reducible by RS_{eulf} to an extended uniform focused proof-term.

The cases corresponding to left constructors follow by lemmas similar to Lemma 5.4.

(ii) Secondly is shown that every proof-term e is reducible by RS_{eulf} to an extended affine proof-term, by induction on the structure of e. Case the outermost constructor of e is a right constructor, the result follows by immediate use of the I.H.. Case e is of the form $contr(x, x_1.e_1)$, the proof follows by induction on the number of occurrences of x_1 in e_1. Case there is at most one occurrence of x_1 in e_1, e is already an extended affine proof-term. Case there is more than one occurrence of x_1 in e_1. Then, rule (xvii') transforms $contr(x, x_1.e_1)$ into $contr(x, x_1.contr(x, x_2.e_2))$, where e_2 results from e_1 by replacing one occurrence of x_1 by x_2. Thus, by the I.H., proof-term $contr(x, x_1.contr(x, x_2.e_2))$ is reducible by RS_{eulf} to an extended affine proof-term.

From (ii) above, every proof-term e is or is reducible by RS_{eulf} to an extended affine proof-term e_1. By (i) above, e_1 is equal or is reducible by RS_{eulf} to an extended uniform focused proof-term e_2. We claim that e_2 must still be extended affine, which may be shown by verifying that each rule used for proving (ii) does not increase the number of occurrences of variables bound by left constructors or by *contr*. It is easy to verify that a proof-term which is simultaneously extended uniform focused and extended affine is extended uniform linear focused (EULF). So, every proof-term is or is reducible by RS_{eulf} to an EULF-proof-term.

So, if $\Sigma; \Delta \Rightarrow e : G$ is derivable in $HH^{def'}$, then either e is a EULF-proof-term, and the result is trivial, or is reducible by RS_{eulf} to a EULF-proof-term e_1 and, by Lemma 5.3, one may easily show that $\Sigma; \Delta \Rightarrow e_1 : G$ is derivable in $HH^{def'}$. □

5.3.2 The Calculus def'^{EULF}

Now, we focus our study on methods of implementing LFPL within search procedures for EULF-derivations. Our first step is defining the calculus def'^{EULF}. This calculus captures exactly all EULF-derivations of $HH^{def'}$ having distinct proof-terms. Ideas similar are used in Sec. 3.4; there, the calculus hH^{ULF} is introduced to capture exactly the ULF-derivations of hH.

The classes of objects of def'^{EULF} are the same as those of $HH^{def'}$. The forms of judgement of def'^{EULF} are the same as those of $HH^{def'}$ except for sequents. In def'^{EULF}, sequents are replaced by the following two forms of sequent:

$$\begin{array}{ll} \text{(i)} & \Sigma; \Delta \longrightarrow e_{eulf} : G; \\ \text{(ii)} & \Sigma; \Delta \overset{\Delta;x:H}{\longrightarrow} a^x_{eulf} : A. \end{array}$$

Sequents of form (i) are called *goal sequents* and sequents of form (ii) are called *program sequents.* These two forms of sequents are similar to those used in hH^{ULF}, the difference being that program sequents require a context, called the *side context,* as an extra argument. The side context is used for guaranteeing the well-formedness of derivable sequents. (When there is no danger of confusion, the side context is omitted.)

The rules defining derivable goal sequents in def'^{EULF} are $choice_{ddt}$, in Fig. 5.4, and the rules for deriving goal sequents in hH^{ULF}, except *choice* that is replaced by the new form of rule *choice* in Fig. 5.4. The rules defining derivable program sequents are shown in Fig. 5.4.

In def'^{EULF} the replacement of a formula by a λ-convertible formula is only allowed at the axioms. A rule $choice_{ddt}$ selects a definition of dependent type from the program and focuses on its I-part.

Theorem 5.4 shows that if e is a EULF-proof-term for deriving G w.r.t. $\Sigma; \Delta$ in $HH^{def'}$ then e is a proof-term for deriving G w.r.t. $\Sigma; \Delta$ in def'^{EULF}. Theorem 5.5 shows the converse, i.e. if e is a proof-term for deriving G w.r.t. $\Sigma; \Delta$ in def'^{EULF} then e is a EULF-proof-term for deriving G w.r.t. $\Sigma; \Delta$ in $HH^{def'}$. Lemma 5.5, below, is used for proving Theorem 5.4.

Lemma 5.5 *If $\Sigma; \Delta_1, x : H, \Delta_2 \overset{\Delta_1;x:H}{\longrightarrow} a^x_{eulf} : A$ is derivable in def'^{EULF}, where $x \notin \Delta_2$, and x occurs only once in a^x_{eulf}, then $\Sigma; \Delta_1, \Delta_2 \overset{\Delta_1;x:H}{\longrightarrow} a^x_{eulf} : A$ is also derivable in def'^{EULF}.*

Proof: See the proof of Lemma 3.10 (a similar result for hH^{ULF}). □

Theorem 5.4 *Let π be a $HH^{def'}$-derivation of $\Sigma; \Delta \Rightarrow e_{eulf} : G$. Then, $\Sigma; \Delta \longrightarrow e_{eulf} : G$ is derivable in def'^{EULF}. Further, if e_{eulf} is of the form a^x_{eulf}, then G is atomic, $\Delta = (\Delta_1, x : H, \Delta_2)$ and $\Sigma; \Delta \overset{\Delta_1;x:H}{\longrightarrow} a^x_{eulf} : G$ is derivable.*

Proof: By using similar arguments to those used for proving Theorem 3.21, it may be shown that every $HH^{def'}$-derivation may be transformed into a derivation where conversion rules

$$\frac{\Sigma;\Delta \xrightarrow{\Delta_1;x:H} e:A}{\Sigma;\Delta \longrightarrow e:A}\ choice \qquad \Delta=(\Delta_1,x:H,\Delta_2)$$

$$\frac{\Sigma;\Delta \xrightarrow{\Delta_1,p=_{def}e:D;x_1:I} e:A}{\Sigma;\Delta \longrightarrow contr(x,x_1.e):A}\ choice_{ddt} \qquad \begin{array}{r} x_1\notin\Delta,\ \Delta=(\Delta_1,p=_{def}e:D,\Delta_2) \\ \text{and } Ipart(p=_{def}e:D)=\langle I,x\rangle \end{array}$$

$$\frac{\Sigma;\Delta\vdash A_1\equiv A \quad \Sigma;\Delta_1\vdash A_1\ af}{\Sigma;\Delta \xrightarrow{\Delta_1;x:A_1} x:A}\ axiom \qquad \begin{array}{r} \text{either } \Delta=(\Delta_1,x:A_1,\Delta_2) \text{ or} \\ \Delta=(\Delta_1,\Delta_2) \text{ and } x\notin\Delta \end{array}$$

$$\frac{\Sigma;\Delta \xrightarrow{\Delta_1;x_1:H_1} e:A \quad \Sigma;\Delta_1\vdash H_2\ hf}{\Sigma;\Delta \xrightarrow{\Delta_1;x:H_1\wedge H_2} splitl(x,x_1.e):A}\ \xrightarrow{\wedge_l} \qquad \begin{array}{r} x_1\notin\Delta \text{ and either} \\ \Delta=(\Delta_1,x:H_1\wedge H_2,\Delta_2) \\ \text{or } \Delta=(\Delta_1,\Delta_2) \text{ and } x\notin\Delta \end{array}$$

$$\frac{\Sigma;\Delta \xrightarrow{\Delta_1;x_1:H_2} e:A \quad \Sigma;\Delta_1\vdash H_1\ hf}{\Sigma;\Delta \xrightarrow{\Delta_1;x:H_1\wedge H_2} splitr(x,x_1.e):A}\ \xrightarrow{\wedge_r} \qquad \begin{array}{r} x_1\notin\Delta \text{ and either} \\ \Delta=(\Delta_1,x:H_1\wedge H_2,\Delta_2) \\ \text{or } \Delta=(\Delta_1,\Delta_2) \text{ and } x\notin\Delta \end{array}$$

$$\frac{\Sigma;\Delta \longrightarrow e_1:G_1 \quad \Sigma;\Delta \xrightarrow{\Delta_1;x_1:H} e:A}{\Sigma;\Delta \xrightarrow{\Delta_1;x:G_1\supset H} apply(x,e_1,x_1.e):A}\ \xrightarrow{\supset} \qquad \begin{array}{r} x_1\notin\Delta \text{ and either} \\ \Delta=(\Delta_1,x:G_1\wedge H,\Delta_2) \\ \text{or } \Delta=(\Delta_1,\Delta_2) \text{ and } x\notin\Delta \end{array}$$

$$\frac{\Sigma;\Delta \xrightarrow{\Delta_1;x_1:[\Lambda/y]H} e:A \quad \Sigma;\Delta_1\vdash\Lambda:\tau}{\Sigma;\Delta \xrightarrow{\Delta_1;x:\forall_{y:\tau}H} apply_q(x,\Lambda,x_1.e):A}\ \xrightarrow{\forall} \qquad \begin{array}{r} x_1\notin\Delta \text{ and either} \\ \Delta=(\Delta_1,x:\forall_{y:\tau}H,\Delta_2) \\ \text{or } \Delta=(\Delta_1,\Delta_2) \text{ and } x\notin\Delta \end{array}$$

Figure 5.4: The rule *choice* and the rules for deriving program sequents of def'^{EULF}.

occur only below axioms or conversion rules. Let π be a derivation of $\Sigma;\Delta\Rightarrow e_{eulf}:G$ where conversion rules occur only below axioms or conversion rules. The proof follows by induction on the structure of π.

Case the last step of π is either an axiom or a conversion rule, it may be proved by induction on the number of conversion rules in π that: G is atomic, e_{eulf} is a variable x, $\Delta=(\Delta_1,x:A,\Delta_2)$ and $\Sigma;\Delta_1,x:A,\Delta_2\vdash A\equiv G$ is derivable. So, the following def'^{EULF}-derivation may be formed:

$$\frac{\dfrac{\Sigma;\Delta_1,x:A,\Delta_2\vdash A\equiv G \quad \Sigma;\Delta_1\vdash A\ af}{\Sigma;\Delta_1,x:A,\Delta_2 \xrightarrow{\Delta_1;x:A} x:G}\ axiom}{\Sigma;\Delta_1,x:A,\Delta_2\longrightarrow x:G}\ choice.$$

Case the last step of π is of the form:

$$\frac{\overset{\pi_1}{\Sigma;\Delta_1,p=_{def}e:D,x_1:I,\Delta_2\Rightarrow a^{x_1}_{eulf_1}:G}}{\Sigma;\Delta_1,p=_{def}e:D,\Delta_2\Rightarrow contr(x,x_1.a^{x_1}_{eulf_1}):G}\ def_{contr}\ ,$$

where $Ipart(p =_{def} e : D) = \langle I, x\rangle$ and $x_1 \notin \Delta_2$. By the I.H., G is atomic and there is a derivation of

$$\Sigma; \Delta_1, p =_{def} e : D, x_1 : I, \Delta_2 \overset{\Delta_1, p=_{def} e:D; x_1:I}{\longrightarrow} a^{x_1}_{eulf_1} : G.$$

By Lemma 5.5, since $x_1 \notin \Delta_2$ and x_1 occurs only once in $a^{x_1}_{eulf_1}$ ($contr(x, x_1.a^{x_1}_{eulf_1})$ is EULF), there is a derivation of

$$\Sigma; \Delta_1, p =_{def} e : D, \Delta_2 \overset{\Delta_1, p=_{def} e:D; x_1:I}{\longrightarrow} a^{x_1}_{eulf_1} : G.$$

So, the following derivation may be formed:

$$\frac{\Sigma; \Delta_1, p =_{def} e : D, \Delta_2 \overset{\Delta_1, p=_{def} e:D; x_1:I}{\longrightarrow} a^{x_1}_{eulf_1} : G}{\Sigma; \Delta_1, p =_{def} e : D, \Delta_2 \longrightarrow contr(x, x_1.a^{x_1}_{eulf_1}) : G}\ choice_{ddt},$$

since $Ipart(p =_{def} e : D) = \langle I, x\rangle$ and $x_1 \notin (\Delta_1, p =_{def} e : D)$, which may be shown from the fact that the antecedent of the endsequent of π_1 is a derivable basis.

Case last step of π is of the form:

$$\frac{\overset{\pi_1}{\Sigma; \Delta_1, x : G_1 \supset H_1, \Delta_2 \Rightarrow e_{eulf_1} : G_1} \quad \overset{\pi_2}{\Sigma; \Delta_1, x : G_1 \supset H_1, x_1 : H_1, \Delta_2 \Rightarrow a^{x_1}_{eulf_1} : G}}{\Sigma; \Delta_1, x : G_1 \supset H_1, \Delta_2 \Rightarrow apply(x, e_{eulf_1}, x_1.a^{x_1}_{eulf_1}) : G}\ \supset\Rightarrow,$$

where $x_1 \notin \Delta_2$. By the I.H.,

$$\Sigma; \Delta_1, x : G_1 \supset H_1, x_1 : H_1, \Delta_2 \overset{\Delta_1, x:G_1 \supset H_1; x_1:H_1}{\longrightarrow} a^{x_1}_{eulf_1} : G.$$

is derivable; so, G is atomic. Yet by the I.H., there is a derivation π_3 of

$$\Sigma; \Delta_1, x : G_1 \supset H_1, \Delta_2 \longrightarrow e_{eulf_1} : G_1.$$

But, x_1 occurs exactly once in a_{eulf_1}, for $apply(x, e_{eulf_1}, x_1.a_{eulf_1})$ is EULF. So, since $x_1 \notin \Delta_2$, by Lemma 5.5, there is a derivation π_4 of

$$\Sigma; \Delta_1, x : G_1 \supset H_1, \Delta_2 \overset{\Delta_1, x:G_1 \supset H_1; x_1:H_1}{\longrightarrow} a^{x_1}_{eulf_1} : G.$$

The following derivation may be formed:

$$\frac{\dfrac{\overset{\pi_3}{\Sigma; \Delta_1, x : G_1 \supset H_1, \Delta_2 \longrightarrow e_{eulf_1} : G_1} \quad \overset{\pi_5}{\Sigma; \Delta_1, x : G_1 \supset H_1, \Delta_2 \overset{\Delta_1; x_1:H_1}{\longrightarrow} a^{x_1}_{eulf_1} : G}}{\Sigma; \Delta_1, x : G_1 \supset H_1, \Delta_2 \overset{\Delta_1; x:G_1 \supset H_1}{\longrightarrow} apply(x, e_{eulf_1}, x_1.a^{x_1}_{eulf_1}) : G}\ \overset{\supset}{\longrightarrow}}{\Sigma; \Delta_1, x : G_1 \supset H_1, \Delta_2 \longrightarrow apply(x, e_{eulf_1}, x_1.a^{x_1}_{eulf_1}) : G}\ choice,$$

where π_5 can be easily obtained from π_4.

The other cases corresponding to other left rules follow by similar arguments. □

Theorem 5.5

(1) *If $\Sigma; \Delta \longrightarrow e : G$ is derivable in def'^{EULF} then $\Sigma; \Delta \Rightarrow e : G$ is derivable in $HH^{def'}$ and e is EULF.*

(2) *If $\Sigma; \Delta \overset{\Delta_1;x;H}{\longrightarrow} e : A$ is derivable in def'^{EULF} then e is atomic extended uniform linear focused of head variable x and either*

- **(a)** *$\Delta = (\Delta_1, x : H, \Delta_2)$ and $\Sigma; \Delta \Rightarrow e : A$ is derivable in $HH^{def'}$; or*
- **(b)** *$x \notin \Delta$, $\Delta = (\Delta_1, \Delta_2)$, the sequent $\Sigma; \Delta_1, x : H, \Delta_2 \Rightarrow e : A$ is derivable in $HH^{def'}$ and x occurs exactly once in e.*

Proof: Let σ_1 and σ_2 be def'^{EULF}-derivations of $\Sigma; \Delta \longrightarrow e : G$ and $\Sigma; \Delta \overset{\Delta_1;x;H}{\longrightarrow} e : A$, respectively. The proof follows by simultaneous induction on the structure of the derivations σ_1 and σ_2. We consider below the case where the last step of σ_1 is $choice_{ddt}$:

$$\frac{\Sigma; \Delta_1, p =_{def} e : D, \Delta_2 \overset{\Delta_1, p =_{def} e:D; x_1 : I}{\longrightarrow} e_1 : A}{\Sigma; \Delta_1, p =_{def} e : D, \Delta_2 \longrightarrow contr(x, x_1.e_1) : A}\ choice_{ddt},$$

where $Ipart(p =_{def} e : D) = \langle I, x \rangle$ and $x_1 \notin (\Delta_1, p =_{def} e : D, \Delta_2)$. By the I.H., since $x_1 \notin (\Delta_1, p =_{def} e : D, \Delta_2)$, there is a derivation of the sequent

$$\Sigma; \Delta_1, p =_{def} e : D, x_1 : I, \Delta_2 \Rightarrow e_1 : A,$$

where e_1 is atomic extended uniform linear focused of head variable x_1 and x_1 occurs exactly once in e_1. Thus, for concluding the proof of this case, the following derivation may be formed:

$$\frac{\Sigma; \Delta_1, p =_{def} e : D, x_1 : I, \Delta_2 \Rightarrow e_1 : A}{\Sigma; \Delta_1, p =_{def} e : D, \Delta_2 \Rightarrow contr(x, x_1.e_1) : A}\ def_{contr}\ ,$$

since $Ipart(p =_{def} e : D) = \langle I, x \rangle$ and $x_1 \notin \Delta_2$. (Note that $contr(x, x_1.e_1)$ is EULF, for e_1 is atomic extended uniform linear focused of head variable x_1 and x_1 occurs exactly once in e_1.)

The last case we consider is that where the last step of σ_2 is of the form:

$$\frac{\Sigma; \Delta \longrightarrow e_1 : G_1 \quad \Sigma; \Delta \overset{\Delta_1;x_1;H_1}{\longrightarrow} e_2 : A}{\Sigma; \Delta \overset{\Delta_1;x:G_1 \supset H_1}{\longrightarrow} apply(x, e_1, x_1.e_2) : A}\ \overset{\supset}{\longrightarrow},$$

where $x_1 \notin \Delta$ and either $\Delta = (\Delta_1, x : G_1 \supset H_1, \Delta_2)$ or $\Delta = (\Delta_1, \Delta_2)$ and $x \notin \Delta$. By the I.H., there is a derivation of Sequent 5.5 and e_1 is EULF.

$$\Sigma; \Delta \Rightarrow e_1 : G_1 \tag{5.5}$$

By the I.H., since $x_1 \notin \Delta$, $\Delta = (\Delta_1, \Delta_{21})$, Sequent 5.6 is derivable and e_2 is an atomic extended uniform linear focused proof-term of head variable x_1, where x_1 occurs exactly once.

$$\Sigma; \Delta_1, x_1 : H_1, \Delta_{21} \Rightarrow e_2 : A \tag{5.6}$$

If $\Delta = (\Delta_1, x : G_1 \supset H_1, \Delta_2)$, then $\Delta_{21} = (x : G_1 \supset H_1, \Delta_2)$. Thus, from a derivation of Sequent 5.6, one may easily construct a derivation of the sequent:

$$\Sigma; \Delta_1, x : G_1 \supset H_1, x_1 : H_1, \Delta_2 \Rightarrow e_2 : A.$$

So, the following derivation may be formed:

$$\frac{\Sigma; \Delta_1, x : G_1 \supset H_1, \Delta_2 \Rightarrow e_1 : G_1. \quad \Sigma; \Delta_1, x : G_1 \supset H_1, x_1 : H_1, \Delta_2 \Rightarrow e_2 : A}{\Sigma; \Delta_1, x : G_1 \supset H_1, \Delta_2 \Rightarrow apply(x, e_1, x_1.e_2) : A} \supset\Rightarrow .$$

Note that $apply(x, e_1, x_1.e_2)$ is atomic extended uniform linear focused of head variable x, for e_1 is EULF and e_2 is an atomic extended uniform linear focused proof-term of head variable x_1, where x_1 occurs exactly once.

If $x \notin \Delta$ and $\Delta = (\Delta_1, \Delta_2)$ then $\Delta_{21} = \Delta_2$. From derivations of Sequents 5.5 and 5.6 one may easily construct derivations of the sequents:

$$\Sigma; \Delta_1, x : G_1 \supset H_1, \Delta_2 \Rightarrow e_1 : G_1;$$
$$\Sigma; \Delta_1, x : G_1 \supset H_1, x_1 : H_1, \Delta_2 \Rightarrow e_2 : A.$$

So the following derivation may be formed:

$$\frac{\Sigma; \Delta_1, x : G_1 \supset H_1, \Delta_2 \Rightarrow e_1 : G_1. \quad \Sigma; \Delta_1, x : G_1 \supset H_1, x_1 : H_1, \Delta_2 \Rightarrow e_2 : A}{\Sigma; \Delta_1, x : G_1 \supset H_1, \Delta_2 \Rightarrow apply(x, e_1, x_1.e_2) : A} \supset\Rightarrow$$

As above, $apply(x, e_1, x_1.e_2)$ is atomic extended uniform linear focused of head variable x. Since $x \notin \Delta$, x has no occurrences in e_1 and x has no occurrences in e_2. Thus, x occurs exactly once in $apply(x, e_1, x_1.e_2)$. □

Theorems 5.4 and 5.5 guarantee that an implementation of LFPL may be described as a search procedure for derivations in def'^{EULF}. Procedures to search for def'^{EULF}-derivations are described in the next section. Proof-search in def'^{EULF} has a smaller search space than proof-search in $HH^{def'}$. For example, if a goal G is compound the only rule that may be used in def'^{EULF} for deriving G is the rule that introduces its main connective; in $HH^{def'}$ the rule introducing the main connective of G could be used, but left rules or def_{contr} could also be used for breaking up formulae or definitions of dependent type in the program.

5.4 Towards an Implementation of LFPL

An implementation of LFPL, as defined in Sec. 4.6, is a procedure that given a goal G and a program $\Sigma; \Delta$ is capable of finding a complete set of witnesses for the achievement of G w.r.t. $\Sigma; \Delta$. As argued in the previous section, an implementation of LFPL may be described as a search procedure for def'^{EULF}. This section presents two possible approaches of implementing LFPL.

Let $\Sigma; \Delta \longrightarrow e_{eulf} : G$ be a derivable sequent of def'^{EULF}. Then, the sequence of transformations below may be applied to obtain the interpretation of this sequent into HH. (After each sequent is mentioned the calculus of which the sequent is a judgement and the theorem that justifies its interpretation into the sequent below.)

$\Sigma; \Delta \longrightarrow e_{eulf} : G$	(def'^{EULF})	by Theorem 5.5
$\Sigma; \Delta \Rightarrow e_{eulf} : G$	$(HH^{def'})$	by Theorem 5.1
$\Sigma; \Delta \Rightarrow d'd(e_{eulf}) : G$	(HH^{def})	by Theorem 4.6
$\Sigma; \psi(\Delta) \Rightarrow \nu(\Delta, d'd(e_{eulf})) : G$	$(HH^{cut'})$	by Theorem 4.5
$\Sigma; \aleph(\psi(\Delta)) \Rightarrow [\psi(\Delta)]\nu(\Delta, d'd(e_{eulf})) : [\psi(\Delta)]G$	(HH^{cut})	by Theorem 4.4
$\Sigma; \aleph(\psi(\Delta)) \Rightarrow cut([\psi(\Delta)]\nu(\Delta, d'd(e_{eulf}))) : [\psi(\Delta)]G$	(HH)	by Corollary 3.3
$\Sigma; \aleph(\psi(\Delta)) \Rightarrow ulf(cut([\psi(\Delta)]\nu(\Delta, d'd(e_{eulf})))) : [\psi(\Delta)]G$	(HH)	

The proof-term

$$ulf(cut([\psi(\Delta)]\nu(\Delta, d'd(e_{eulf}))))$$

is called the *ulf-form* of e_{eulf}; $d'd(e_{eulf})$ is a witness for the achievement of G w.r.t. $\Sigma; \Delta$ in LFPL. Also, by Corollary 4.2, the ulf-form e of e_{eulf} is a witness for the achievement of G w.r.t. $\Sigma; \Delta$ in LFPL. Further, it may be easily shown that $\Sigma; \Delta \Rightarrow e : G$ is derivable in def'^{EULF}. So, if there is a def'^{EULF}-derivation π_1 of G using rules $choice_{ddt}$ then there exists a def'^{EULF}-derivation π_2 of G using no such rules s.t. the ulf-forms of π_1 and π_2 are the same. Thus, any set containing simultaneously the proof-terms of all uniform linear focused derivations of G and proof-terms of derivations of G using rules $choice_{ddt}$ is redundant.

Let us first concentrate on two procedures to find complete sets of witnesses for the achievement of a goal w.r.t. a program in LFPL. These procedures follow ideas similar to those described in Sec. 3.7 for proof-search in hH^{ULF}. In the first procedure (*Proc. 1*) are defined predicates:

$$search(G, (\Sigma; \Delta), \Theta_{in}, \Theta_{out}, e, V_{in}, V_{out});$$
$$search1(x : H, A, \Delta_1, (\Sigma; \Delta), \Theta_{in}, \Theta_{out}, e, V_{in}, V_{out}),$$

As in Sec. 3.7, Θ_{in} and Θ_{out} are mappings from variables to λ-terms and V_{in}, V_{out} are signatures. Notice that $search1$ requires an extra argument for the side context.

The predicate *search* is s.t. if a sequent

$$\Sigma; \Delta \longrightarrow e : G$$

is derivable in def'^{EULF} then there exist Θ, V, e_1 s.t. the formula

$$search(G, (\Sigma; \Delta), identity, \Theta, e_1, \emptyset, V),$$

holds, where $e = \Theta(e_1)$. The definition of predicates *search* and *search*1 is the same as for hH^{ULF} with the exceptions mentioned below. The predicate *search* has a new alternative form of achieving atomic goals. When a goal is atomic two rules may be used: (i) a rule *choice*, for selecting a formula from the program (*derivation via formula*), corresponding to the case permitted in hH^{ULF} ; (ii) a rule $choice_{ddt}$ for selecting a definition of dependent type from the program (*derivation via definition*), corresponding to the following clause in the definition of *search*:

$$\begin{aligned}&search(A, (\Sigma; \Delta), \Theta_{in}, \Theta_{out}, contr(x, x_1.e_1), V_{in}, V_{out}) \text{ if}\\ &\quad choiceddt(\Delta, I, x, \Delta_1, \Delta_2, p, e, D) \text{ and}\\ &\quad search1(x_1 : I, A, \Delta_1, (\Sigma; \Delta), \Theta_{in}, \Theta_{out}, e_1, V_{in}, V_{out}) \text{ and}\\ &\quad x_1 \notin \Delta.\end{aligned}$$

The clause defining validity of *choiceddt* is:

$$\begin{aligned}&choiceddt(\Delta, I, x, \Delta_1, \Delta_2, p, e, D) \text{ if}\\ &\quad \Delta = (\Delta_1, p =_{def} e : D, \Delta_2) \text{ and}\\ &\quad Ipart(p =_{def} e : D) = \langle I, x\rangle.\end{aligned}$$

Another clause that needs modification is that in the definition of *search*1 corresponding to the use of the axiom. Now, definitions of simple type may be used to replace definienda by definientia. The new clause is:

$$\begin{aligned}&search1(x : A_1, A, \Delta_1, (\Sigma; \Delta), \Theta_{in}, \Theta_{out}, x, V_{in}, V_{out}) \text{ if}\\ &\quad unify(A_1, A, \Theta_{in_1}, \Theta_{out}, V_{in}, V_{out}, \Sigma \cup \Sigma_1),\end{aligned}$$

where: Θ_{in_1} is the function, from variables to λ-terms, s.t. if x is a variable for which there is a definition in Δ, implicit or explicit, with $x =_{def} \Lambda : \tau$, then $\Theta_{in_1}(x) = \Lambda$; otherwise $\Theta_{in_1}(x) = \Theta_{in}(x)$; and Σ_1 consists of all pairs $x : \tau$ s.t. $x =_{def} \Lambda : \tau$, for some Λ, is either an implicit or explicit simple definition in Δ_1.

Below is described a second procedure (*Proc. 2*) to search for derivations in def'^{EULF}. This procedure is similar to that described above except for the use of definitions of dependent type. In *Proc. 2*, a definition of dependent type is used during search if any of its definienda occurs in the goal; in other words, the logical properties of a definiendum are used in achieving a goal only if the goal has some occurrence of that definiendum. The procedure, *Proc. 2*, for finding def'^{EULF}-derivations, based on the criterion above for using definitions of dependent type, may be defined as the procedure obtained from *Proc. 1* by replacing the clause to deal with definitions of dependent type by the following clause:

$$\begin{aligned}&search(A, (\Sigma; \Delta), \Theta_{in}, \Theta_{out}, contr(x, x_1.e_1), V_{in}, V_{out}) \text{ if}\\ &\quad choiceddt1(A, \Delta, I, x, \Delta_1, \Delta_2, p, e, D, \Theta_{in}) \text{ and}\\ &\quad search1(x_1 : I, A, \Delta_1, (\Sigma; \Delta), \Theta_{in}, \Theta_{out}, e_1, V_{in}, V_{out}) \text{ and}\\ &\quad x_1 \notin \Delta.\end{aligned}$$

The clause defining validity of *choiceddt*1 is:

$$\begin{aligned}
&choiceddt1(A, \Delta, I, x, \Delta_1, \Delta_2, p, e, D, \Theta_{in}) \text{ if} \\
&\qquad \Delta = (\Delta_1, p =_{def} e : D, \Delta_2) \text{ and} \\
&\qquad Ipart(p =_{def} e : D) = \langle I, x \rangle \text{ and} \\
&\qquad x_1 \in definienda(p) \text{ and} \\
&\qquad x_1 \in \Theta_{in}(A).
\end{aligned}$$

Proc. 2 has the virtue, as compared to *Proc. 1*, of cutting down the alternative derivations via definitions of dependent type. Rather than attempting exhaustively all definitions of dependent type, a goal is used to determine which definitions of dependent type should be attempted. However, *Proc. 2* is not capable of finding all def'^{EULF}-derivations of a formula.

Definition 5.6 *A rule $choice_{ddt}$ is* sensible *if there is a definiendum of the main definition occurring in the succedent formula of the conclusion sequent. A derivation is* sensible *if all its $choice_{ddt}$-rules are sensible.*

All the derivations found by *Proc. 2* are sensible, however not all sensible derivations are found by *Proc. 2*. Roughly, the sensible derivations which may not be found by *Proc. 2* are those derivations which may be found by *Proc. 1* by using the clause to deal with definitions of dependent type in the following conditions:

(i) the proposition $search(A, (\Sigma; \Delta), \Theta_{in}, \Theta_{out}, contr(x, x_1.e_1), V_{in}, V_{out})$ holds because the following propositions hold:

$$\begin{aligned}
&choiceddt(\Delta, I, x, \Delta_1, \Delta_2, p, e, D); \\
&search1(x_1 : I, A, \Delta_1, (\Sigma; \Delta), \Theta_{in}, \Theta_{out}, e_1, V_{in}, V_{out}); \\
&x_1 \not\in \Delta;
\end{aligned}$$

(ii) no definiendum of p occurs in A;

(iii) some variable x of V_{in} occurring in A is s.t. a definiendum of p occurs in $\Theta_{out}(x)$.

In the conditions above A has no occurrences of a definiendum of p. However, the corresponding derivation uses a rule $choice_{ddt}$ whose formula in the conclusion's succedent is $\Theta_{out}(A)$, which has some definienda of p occurring in it. Below is studied a sufficient condition for a rule $choice_{ddt}$ to be sensible.

Definition 5.7 *The set of* strictly positive subformulae *of an H-formula H (notation $sps(H)$) w.r.t. a basis $\Sigma; \Delta$ is defined as follows:*

$$\begin{aligned}
&sps(A) =_{def} \{A\} \\
&sps(G \supset H) =_{def} sps(H) \\
&sps(H_1 \wedge H_2) =_{def} sps(H_1) \cup sps(H_2) \\
&sps(\forall_{x:\tau} H) =_{def} sps([\Lambda/x]H), \textit{ if } \Sigma; \Delta \vdash \Lambda : \tau \textit{ is derivable.}
\end{aligned}$$

Definition 5.8 *Let* $\Sigma; \Delta \xrightarrow{\Delta_1;x;H} a^{x}_{eulf} : A$ *be derivable by* π *in* def'^{EULF}. *The* derivation of the main axiom *and the* principal strictly positive subformula *of* H *for* π *are inductively defined on the structure of* π *as follows:*

- *Case* π *is of the form:*

$$\frac{\overset{\pi_1}{\Sigma; \Delta \vdash A_1 \equiv A} \quad \overset{\pi_2}{\Sigma; \Delta_1 \vdash A_1\ af}}{\Sigma; \Delta \xrightarrow{\Delta_1;x:A_1} x : A}\ axiom,$$

where $H = A_1$ *and* $a^{x}_{eulf} = x$. *Then, the* derivation of the main axiom *for* π *is* π *and the* principal strictly positive subformula *of* H *for* π *is* H.

- *Case* π *is of the form:*

$$\frac{\overset{\pi_1}{\Sigma; \Delta \xrightarrow{\Delta_1;x_1:H_1} a^{x_1}_{eulf_1} : A} \quad \overset{\pi_2}{\Sigma; \Delta_1 \vdash H_2\ hf}}{\Sigma; \Delta \xrightarrow{\Delta_1;x:H_1 \wedge H_2} splitl(x, x_1.a^{x_1}_{eulf_1}) : A}\ \xrightarrow{\wedge_l},$$

where $H = H_1 \wedge H_2$ *and* $a^{x}_{eulf} = splitl(x, x_1.a^{x_1}_{eulf_1})$. *Then, the* derivation of the main axiom *and the* principal strictly positive subformula *of* H *for* π *are respectively the derivation of the main axiom and the principal strictly positive subformula of* H_1 *for* π_1.

- *Case* π *is of the form:*

$$\frac{\overset{\pi_1}{\Sigma; \Delta \xrightarrow{\Delta_1;x_1:H_2} a^{x_1}_{eulf_1} : A} \quad \overset{\pi_2}{\Sigma; \Delta_1 \vdash H_1\ hf}}{\Sigma; \Delta \xrightarrow{\Delta_1;x:H_1 \wedge H_2} splitr(x, x_1.a^{x_1}_{eulf_1}) : A}\ \xrightarrow{\wedge_r},$$

where $H = H_1 \wedge H_2$ *and* $a^{x}_{eulf} = splitr(x, x_1.a^{x_1}_{eulf_1})$. *Then, the* derivation of the main axiom *and the* principal strictly positive subformula *of* H *for* π *are respectively the derivation of the main axiom and the principal strictly positive subformula of* H_2 *for* π_1.

- *Case* π *is of the form:*

$$\frac{\overset{\pi_1}{\Sigma; \Delta \longrightarrow e_{eulf} : G} \quad \overset{\pi_2}{\Sigma; \Delta \xrightarrow{\Delta_1;x_1:H_1} a^{x_1}_{eulf_1} : A}}{\Sigma; \Delta \xrightarrow{\Delta_1;x:G \supset H_1} apply(x, e_{eulf}.x_1.a^{x_1}_{eulf_1}) : A}\ \xrightarrow{\supset},$$

where $H = G \supset H_1$ *and* $a^{x}_{eulf} = apply(x, e_{eulf}, x_1.a^{x_1}_{eulf_1})$. *Then, the* derivation of the main axiom *and the* principal strictly positive subformula *of* H *for* π *are respectively the derivation of the main axiom and the principal strictly positive subformula of* H_1 *for* π_2.

- *Case* π *is of the form:*

$$\frac{\overset{\pi_1}{\Sigma; \Delta \xrightarrow{\Delta_1;x_1:[\Lambda/y]H_1} a^{x_1}_{eulf_1} : A} \quad \overset{\pi_2}{\Sigma; \Delta_1 \vdash \Lambda : \tau}}{\Sigma; \Delta \xrightarrow{\Delta_1;x:\forall_{y:\tau}H_1} apply_q(x, \Lambda, x_1.a^{x_1}_{eulf_1}) : A}\ \xrightarrow{\forall},$$

where $H = \forall_{y:\tau} H_1$ *and* $a^{x}_{eulf} = apply_q(x, \Lambda, x_1.a^{x_1}_{eulf_1})$. *Then,* the derivation of the main axiom *and the* principal strictly positive subformula *of H for π are respectively the derivation of the main axiom and the principal strictly positive subformula of $[\Lambda/y]H_1$ for π_1.*

It is easy to show that if π is the derivation of the main axiom for a derivation π_1 of $\Sigma; \Delta \overset{\Delta_1;x;H}{\longrightarrow} a^{x}_{eulf} : A$ then π is of the form:

$$\frac{\overset{\pi_2}{\Sigma; \Delta \vdash A_1 \equiv A} \quad \overset{\pi_3}{\Sigma; \Delta_1 \vdash A_1\ af}}{\Sigma; \Delta \overset{\Delta_1;x_1;A_1}{\longrightarrow} x_1 : A}\ axiom,$$

for some x_1, A_1, π_2 and π_3.

Definition 5.9 *A definition of dependent type*

$$(x_1, ..., x_n, x) =_{def} pair_q(\Lambda_1, ...pair_q(\Lambda_n, e)...)) : \Sigma_{y_1:\tau_1}...\Sigma_{y_n:\tau_n} I$$

is called reasonable *if $n \geq 1$ and for every $A \in sps(I)$ there is $1 \leq i \leq n$ s.t. $y_i \in A$.*

Not all definitions of dependent type are reasonable. However, when definitions of dependent type are used for declaring logical properties of functions almost all definitions may be replaced by reasonable definitions, essentially equivalent to them. For example, definitions whose type is an I-formula are not reasonable. However, for declaring logical properties of functions the types of definitions need to be of form $\Sigma_{y:\tau} D$. Other situation where a definition is not reasonable is when its type is of the form $\Sigma_{y_1:\tau_1}...\Sigma_{y_n:\tau_n} I$ and $y_i \notin I$, for $1 \leq i \leq n$. In this case the type of the definition is logically equivalent to I and so, by the argument above, this definition is not useful for declaring logical properties of a function. The last situation where a definition may be not reasonable is when the type of the definition is of the form $\Sigma_{y_1:\tau_1}...\Sigma_{y_n:\tau_n} I$, where I is either of the form (i) $I_1 \wedge I_2$ or of the form (ii) $I_1 \supset I_2$ and not simultaneously $y_i \in I_1$ and $y_j \in I_2$, for $1 \leq i, j \leq n$. In case (i) the definition could be easily replaced by two definitions. In case (ii), if $y_i \in I_2$ the ideas above may be attempted to express the definition by means of reasonable definitions. For case (ii), where $y_i \in I_1$, the ideas above are not sufficient to replace the definition by reasonable definitions.

Consider the following two forms of D-formulae:

(i) $\Sigma_{f:\tau_1 \to \tau_2} \forall_{x:\tau_1} (I_1 \supset I_2)$, where $x \in I_1$ and fx occurs in I_2;

(ii) $\Sigma_{g:\tau_1 \to \tau_2 \to \tau_3} \forall_{x_1:\tau_1} (I_1 \supset (\forall_{x_2:\tau_2} (I_2 \supset I_3)))$, where $x_1 \in I_1$, $x_2 \in I_2$ and gx_1x_2 occurs in I_3.

Types of forms (i) and (ii) correspond to the types of first and second order deliverables [MB93], the difference being that in the theory of deliverables the I-formulae used above may be replaced by arbitrary formulae. A definition whose type is either of form (i) or (ii) is reasonable, for f occurs in I_2 and g occurs in I_3, respectively.

Definition 5.10 *Let $\Sigma; \Delta \vdash \Lambda_1 \equiv_\tau \Lambda_2$ be derivable by π. A variable x is* necessary *for Λ_1 in π if there is a free occurrence x^* of x in Λ_1 and π has no rules of the forms:*

$$\frac{\vdash \Sigma_1; \Delta_1, x =_{def} \Lambda : \tau, \Delta_2 \; basis}{\Sigma_1; \Delta_1, x =_{def} \Lambda : \tau, \Delta_2 \vdash x^* \rhd_\tau \Lambda} \; ;$$

$$\frac{\Sigma_1, x_1 : \tau; \Delta_1 \vdash \Lambda_3 : \tau_1 \quad \Sigma_1; \Delta_1 \vdash \Lambda_4 : \tau}{\Sigma_1; \Delta_1 \vdash (\lambda x_1 : \tau.\Lambda_3)\Lambda_4 \rhd_{\tau_1} [\Lambda_4/x_1]\Lambda_3} \; where\ x^* \in \Lambda_4\ and\ x_1 \notin \Lambda_3.$$

Using no rules of the first form guarantees that, case x has a definiens, x^* is not replaced by its definiens. Use of no rules of the second form guarantees that x^* may not disappear with β-reductions.

Lemma 5.6 *Let $\Sigma; \Delta \vdash \Lambda_1 \rhd_\tau \Lambda_2$ be derivable by π. Then, if x is necessary for Λ_1 in π then $x \in \Lambda_2$.*

Proof: Let x be a necessary variable for Λ_1 in π. The proof follows by induction on the structure of π. For example, case the last step of π is of the form:

$$\frac{\Sigma, x_1 : \tau_1; \Delta \vdash \Lambda_3 : \tau \quad \Sigma; \Delta \vdash \Lambda_4 : \tau_1}{\Sigma; \Delta \vdash (\lambda x_1 : \tau_1.\Lambda_3)\Lambda_4 \rhd_\tau [\Lambda_4/x_1]\Lambda_3} \; .$$

By definition of necessary variables, there is a free occurrence x^* of x in $(\lambda x_1 : \tau_1.\Lambda_3)\Lambda_4$, *i.e.* $x^* \in \Lambda_3$ or $x^* \in \Lambda_4$. If $x^* \in \Lambda_3$ then $x^* \in [\Lambda_4/x_1]\Lambda_3$, since $x_1 \neq x$. If $x^* \in \Lambda_4$ then $x_1 \in \Lambda_3$, otherwise x is not necessary for $(\lambda x_1 : \tau_1.\Lambda_3)\Lambda_4$ in π. Thus, $x^* \in [\Lambda_4/x_1]\Lambda_3$. □

Definition 5.11 *Let $\Sigma; \Delta \vdash (p\Lambda_1 ... \Lambda_n) \equiv (p\Lambda'_1 ... \Lambda'_n)$ be derivable by π, i.e., for $1 \leq i \leq n$, there is a derivation π_i of $\Sigma; \Delta \vdash \Lambda_i \equiv_{\tau_i} \Lambda'_i$. Then, a variable x is* necessary *for $p\Lambda_1 ... \Lambda_n$ in π, if, for some $1 \leq i \leq n$, x is necessary for Λ_i in π_i.*

Lemma 5.7 *Let $\Sigma; \Delta \vdash A_1 \equiv A_2$ be derivable by π. Then, if x is necessary for A_1 in π then $x \in A_2$.*

Proof: Follows easily from Lemma 5.6. □

Theorem 5.6 *Let π be a derivation using the choice$_{ddt}$ rule 5.7, whose main definition is reasonable of the form $(x_1, ..., x_n, x) =_{def} (\Lambda_1, ..., \Lambda_n, e) : \Sigma_{y_1:\tau_1} ... \Sigma_{y_n:\tau_n} I$.*

$$\frac{\overset{\pi_1}{\Sigma; \Delta \xrightarrow{z:[x_1/y_1]...[x_n/y_n]I} a^z_{eulf} : A}}{\Sigma; \Delta \longrightarrow contr(x, z.a^z_{eulf}) : A} \; choice_{ddt} \tag{5.7}$$

Let the derivation of the main axiom for π_1 be:

$$\frac{\Sigma; \Delta \vdash \overset{\pi_2}{A_1} \equiv A \quad \Sigma; \Delta_1 \vdash A_1 \; af}{\Sigma; \Delta \xrightarrow{z_1:A_1} z_1 : A} \; axiom.$$

Then, for some $1 \leq i \leq n$, x_i occurs in the principal strictly positive subformula A_1 of $[x_1/y_1]...[x_n/y_n]I$. Further, if x_i is necessary for A_1 in π_2 then the choice$_{ddt}$ rule 5.7 is sensible.

Proof: Since the main definition of rule 5.7 is reasonable, every strictly positive subformula of I has an occurrence of an y_j, for $1 \leq j \leq n$. So, every strictly positive subformula of $[x_1/y_1]...[x_n/y_n]I$ has an occurrence of an x_j. In particular A_1 has an occurrence of a definiendum, let us call it x. Now, assume that x is a necessary variable of A_1 in π_2. Then, by Lemma 5.7, x occurs in A. Thus, rule 5.7 is sensible, since x occurs in A and it is a definiendum of the main definition. □

Theorem 5.6 gives a sufficient condition for a rule *choice*$_{ddt}$ to be sensible, in case its main definition is reasonable. On one hand, an implementation of LFPL based on *Proc. 2* may be more efficient than an implementation based on *Proc. 1*, since for achieving atomic goals the definitions of dependent type to attempt are dictated by the occurrences of definienda in the goal, so there may be fewer definitions to attempt. But, on the other hand, there are some derivations via definitions found by *Proc. 1* that may be missed with *Proc. 2*; however, these are not many, as hinted by Theorem 5.6. In future work we intend to make precise the benefits and costs of implementations based on *Proc. 1* and *Proc. 2*.

The two procedures sketched above to search for def'^{EULF}-derivations are capable of finding complete sets of witnesses. However, these complete sets of witnesses may be redundant. For example, consider a search for a witness of an atomic goal A w.r.t. a program $\Sigma; \Delta$. Only two rules may be used to derive an atomic formula; they are *choice* and *choice*$_{ddt}$. Suppose a witness e is found by using *choice*$_{ddt}$ for a definition of dependent type $p =_{def} e_1 : D$ in the program. Then, there must be a formula in Δ leading to a derivation of A, whose ulf-form is the same as the ulf-form of e. Below we put forward some ideas that may be used in eliminating redundancy.

Consider a search of a $HH^{def'}$-derivation for an atomic formula. The only rules that may be applied are *choice* and *choice*$_{ddt}$, i.e. a derivation via a formula or a derivation via a definition. The procedure sketched above imposes no constraints on the order in which components (formulae or definitions of dependent type) in the context should be selected.

Many times in logic programming a user is only interested in knowing some of the means for achieving a goal, rather than knowing all of them. So, the order in which alternative derivations are searched becomes important. Shorter derivations should be reported first.

A motivation to introduce definitions of dependent type is that their use may introduce shortcuts in deriving a formula; so a derivation via a definition should be preferred to a derivation via a formula. Thus, we suggest that the choice of components from the context to derive an atomic goal should be such that definitions of dependent type are selected before formulae. Still there is a choice for the order in which to select definitions and the order in which to select formulae.

Let us consider a search for a witness of an atomic goal A w.r.t. a program $\Sigma; \Delta$, where there is a procedure *choice* that has already selected components of Δ producing the non-redundant set S_e of witnesses. Assume the next component selected by *choice* is C. In this setting, the predicate *search* for atomic goals could be described by the following clause:

```
$search(A, (\Sigma; \Delta), \Theta_{in}, \Theta_{out}, e, V_{in}, V_{out})$ if
    ($C = (x : H)$ and
     $search1(x : H, A, (\Sigma; \Delta), \Theta_{in}, \Theta_{out}, e, V_{in}, V_{out})$ and
      $not\ in(ulf(e), S_e))$
    )
    or
    ($C = (p =_{def} e_1 : D)$ and
     $Ipart(p =_{def} e_1 : D, I, x)$ and
     $search1(x_1 : I, A, (\Sigma; \Delta), \Theta_{in}, \Theta_{out}, e_2, V_{in}, V_{out})$ and
     $e = contr(x, x_1.e_2)$ and
     $not\ in(ulf(e), S_e))$ and
     $x_1 \notin \Delta$
    )
```

An expression $ulf(e)$ stands for the ulf-form of e. A proposition $in(ulf(e), S_e)$ holds iff the ulf-form of e is λ-convertible to a proof-term in the set S_e. In practice both e and S_e may require the use of free variables of simple type, those introduced to deal with existential goals and universally quantified formulae in the program. Such requirement complicates the test for membership of a witness in S_e, demanding pattern-matching on λ-terms.

In future work we intend to make precise these ideas for eliminating redundancy of complete sets of witnesses for the achievement of a goal w.r.t. a program in LFPL. We hope to improve on the ideas described above by integrating the test for redundancy with the choice of a component of the program to proceed search; so that, some components of the program need not be attempted, either because they lead to derivations whose ulf-forms are identical to other ulf-forms obtained before or because they fail to produce a derivation for the same reason as some other component of the program attempted before.

This section finishes by applying the ideas described above to find a complete and non-redundant set of witnesses for the example shown in Sec. 4.7. There, the problem is achieving the goal

$$G = \exists_{x:\tau} member_{geq}(+_2 0, [sss0], x)$$

w.r.t. the program $\Sigma; \Delta$. (See Sec. 4.7 for the definition of $\Sigma; \Delta$.)

Using either *Proc. 1* or *Proc. 2*, a proof-term e is a witness for the achievement of G w.r.t. $\Sigma; \Delta$ iff there exists $\Theta_?$, $e_?$, $V_?$ s.t. proposition (i) is valid and $\Theta_?(e_?) = e$.

(i) $search(\exists_{x:\tau} member_{geq}(+_2 0, [sss0], x), (\Sigma; \Delta), id, \Theta_?, e_?, \emptyset, V_?)$, where *id* represents the identity function on variables and $\emptyset$ represents the empty signature. Proposition (i) is valid iff proposition (ii) is valid, where $e_? = (x, e_{?_1})$.

$$\text{(ii)} \quad search(member_{geq}(+_2 0, [sss0], x), (\Sigma; \Delta), id, \Theta_?, e_{?_1}, \{x : nat\}, V_?)$$

Now a component of the program must be selected. The definiendum of the unique definition of dependent type in the program, $+_2$, occurs in the goal, so *Proc. 1* and *Proc. 2* behave in the same way. Either the definition of dependent type or any program formula may be selected for showing validity of (ii). However, the only means of making (ii) valid is by selecting the formula annotated by z_3 in the program. So, after a few steps, for (ii) to be valid, proposition (iii) must be valid.

$$\text{(iii)} \quad search1(z_9 : H_1, member_{geq}(+_2 0, [sss0], x), \Delta_1, (\Sigma; \Delta), id, \Theta_?, e_{?_2}, V_1, V_?),$$

where:

- $e_{?_1} = apply_q(z_4, x_1, z_7.apply_q(z_7, x_2, z_8.apply_q(z_8, x_3, z_9.e_{?_2})))$;
- $H_1 = geq(x_2, x_1) \supset member_{geq}(x_1, [x_2|x_3], x_2)$;
- Δ_1 is the context consisting of the components of Δ that appear before the formula annotated by z_3;
- $V_1 = \{x : nat, x_1 : nat, x_2 : nat, x_3 : lnat\}$.

For proposition (iii) to be valid, there must exist $\Theta_{?_1}$ and $V_{?_1}$ s.t. propositions (iv) and (v) are valid.

$$\begin{aligned} &\text{(iv)} \quad search1(z_{10} : member_{geq}(x_1, [x_2|x_3], x_2), \\ &\qquad\qquad member_{geq}(+_2 0, [sss0], x), \Delta_1, (\Sigma; \Delta), id, \Theta_{?_1}, e_{?_3}, V_1, V_{?_1}); \\ &\text{(v)} \quad search(geq(x_2, x_1), (\Sigma; \Delta), \Theta_{?_1}, \Theta_?, e_{?_4}, V_{?_1}, V_?), \end{aligned}$$

where: $e_{?_2} = apply(z_9, e_{?_4}, z_{10}.e_{?_3})$. Proposition (iv) may be shown valid performing the following instantiations:

$$\begin{aligned} &e_{?_3} = z_{10}; \\ &\Theta_{?_1} = id - \{x_1 \mapsto +_2 0; x_2 \mapsto sss0; x_3 \mapsto nil; x \mapsto sss0\}; \\ &V_{?_1} = V_1. \end{aligned}$$

($\Theta_{?_1}$ represents the function *id* except for the values associated to x_1, x_2, x_3, x which are now respectively $+_2 0, sss0, nil, sss0$.)

Validity of (v) may be shown either by selecting a formula or the definition of dependent type, since the definiendum occurs in the goal, for $\Theta_{?_1}(x_1) = +_2 0$. Notice that, in case $\Theta_{?_1}(x_1)$ was still an indeterminate the definition of dependent type could not be used at this point in *Proc. 2*.

First we attempt showing validity of (v) through the definition of dependent type. Doing so, (v) is valid only if (vi) is valid.

$$\text{(vi)} \quad search1(z_{13} : H_2, geq(x_2, x_1), \Delta_2, (\Sigma; \Delta), \Theta_{?_1}, \Theta_?, e_{?_5}, V_2, V_?),$$

where:

- $e_{?_4} = contr(z, z_{11}.apply(z_{11}, x_4, z_{12}.apply(z_{12}, x_5, z_{13}.e_{?_5})))$;
- $H_2 = geq(x_4, x_5) \supset geq(+_2 x_4, +_2 x_5)$;
- Δ_2 is the subcontext of Δ that appears before the definition of dependent type, followed by the definition of simple type $+_2 =_{def} \lambda x.ssx : nat \to nat$.
- $V_2 = V_1 \cup \{x_4 : nat, x_5 : nat\}$.

For proposition (vi) to be valid, there must exist $\Theta_{?_2}$ and $V_{?_2}$ s.t. propositions (vii) and (viii) are valid:

$$\text{(vii)} \quad search1(z_{14} : geq(+_2 x_4, +_2 x_5), geq(x_2, x_1), \Delta_2, (\Sigma; \Delta), \Theta_{?_1}, \Theta_{?_2}, e_{?_6}, V_2, V_{?_2});$$
$$\text{(viii)} \quad search(geq(x_4, x_5), (\Sigma; \Delta), \Theta_{?_2}, \Theta_?, e_{?_7}, V_{?_2}, V_?),$$

where: $e_{?_5} = apply(z_{13}, e_{?_7}, z_{14}.e_{?_6})$.

Proposition (vii) may be shown valid by performing the following instantiations:

$$\begin{aligned} &e_{?_6} = z_{14}; \\ &\Theta_{?_2} = \Theta_{?_1} - \{x_4 \mapsto s0; x_5 \mapsto 0\}; \\ &V_{?_2} = V_2. \end{aligned}$$

By using *Proc. 1*, validity of (viii) may be shown either by selecting a formula or the definition of dependent type. But, by using *Proc. 2*, validity of (viii) may only be shown using a formula, since $\Theta_{?_2}$ applied either to x_4 or x_5 produces no occurrences of $+_2$. In fact validity of (viii) may not be shown through the definition of dependent type. The only form of showing validity of (viii) is by making the following instantiations for the indeterminates:

$$\begin{aligned} &e_{?_7} = apply_q(z_1, x_6, z_{15}.z_{15}); \\ &\Theta_? = \Theta_{?_2} - \{x_6 \mapsto s0\}; \\ &V_? = V_{?_2} \cup \{x_6 : nat\}. \end{aligned}$$

There is only other alternative form of showing validity of (v), which follows by selecting the formula annotated by z_2. This alternative produces the following instantiations for the indeterminates:

$$\begin{aligned} &e_{?_4} = apply_q(z_2, x_4, z_{11}.apply_q(z_{11}, x_5, z_{12}.apply(z_{12}, e_{?_5}, z_{13}.z_{13}))); \\ &e_{?_5} = apply_q(z_2, x_6, z_{14}.apply_q(z_{14}, x_7, z_{15}.apply(z_{15}, apply_q(z_1, x_8, z_{16}.)), z_{17}.z_{17})); \\ &\Theta_? = \Theta_{?_1} - \{x_4 \mapsto ss0, x_5 \mapsto s0, x_6 \mapsto s0, x_7 \mapsto 0, x_8 \mapsto s0\}; \\ &V_? = V_1 \cup \{x_4 : nat, x_5 : nat, x_6 : nat, x_7 : nat, x_8 : nat\}. \end{aligned}$$

The ulf-forms corresponding to the two possible instantiations of $e_{?_4}$ are the same; they are equal to:

$$bc(z_2, [ss0, s0], bc(z_2, [s0, 0], apply_q(z_1, s0, z_5.z_5))).$$

Thus, a complete and non-redundant set of witnesses for the achievement of G w.r.t. $\Sigma; \Delta$ contains only one witness.

Summarising, we have sketched two methods for implementing LFPL. Both methods are goal-directed for compound goals. For atomic goals, either a formula or a definition of dependent type of the program is selected and search proceeds by breaking it up. The difference between both methods is in the selection of definitions of dependent type. Whereas in *Proc. 1* arbitrary definitions of dependent type may be selected, *Proc. 2* allows only the selection of a definition of dependent type if any of its definienda occurs in the goal. So, the search space in *Proc. 2* is much smaller than the search space of *Proc. 1*, but there may be derivations using definitions of dependent type, potentially shorter, which are not found by *Proc. 2*. However, as shown in Theorem 5.6, these derivations are not many.

Running both methods for all possible choices produces redundant sets of witnesses. In future work we intend to address mechanisms that permit to abandon search at particular choices that lead to witnesses whose ulf-form is equal to the ulf-form of another witness found before.

Chapter 6

Conclusions and Open Problems

6.1 Conclusions

This thesis proposes the language LFPL as a new approach for integrating logic and functional programming. In contrast to other approaches (such as ALF, Babel, Curry and Escher), LFPL keeps functions and formulae at separate levels; as in type theory, formulae are used for expressing logical properties (specifications) of functions. Sometimes, the specification of a function may suffice for achieving goals, thus obviating the necessity of replacing the function name by its definiens and subsequent normalisation.

Often, in programming, we write a specification and attempt to derive an implementation of it. In the context of LFPL, specifications of functions are thought of as lemmas that may be used for goal-achievement. Firstly, it needs to be shown that functions meet their specifications. Secondly, those specifications may be used, as many times as desired, for goal-achievement.

We take a proof-theoretic view of logic programming, essentially following [MNPS91, Mil90], where a goal (formula) G is achievable w.r.t. a program (set of formulae) P if there is a proof of G from assumptions P. The language FOPLP is defined by means of the sequent calculus system hH for first-order hereditary Harrop logic, a fragment of the logic underlying λProlog[NM88].

When defining a semantics of a logic programming language, we must define what are the different means of goal-achievement. The quest for an answer to this question for an integrated logical and functional language led us to studying this question in the simpler setting of the first-order pure logic programming language FOPLP.

The different means of achieving a goal G w.r.t. a program P in FOPLP are the proof-terms e of the uniform linear focused derivations of G w.r.t. P. We show that such derivations are in a 1-1 correspondence, through Prawitz's mapping ϕ[Pra65], to the expanded normal deductions of G w.r.t. P, or, in other words, the deductions of G w.r.t. P in the natural deduction system NN for first-order hereditary Harrop logic, presented in Sec. 3.5. Further, we show how to transform each derivation d in hH into a uniform linear focused derivation u, essentially by means of Kleene's permutations [Kle52], so that d and u are interpreted by ϕ as the same expanded normal deduction.

The language HOPLP is defined by means of HH, a sequent calculus for the higher-order hereditary Harrop logic with λ-terms rather than first-order terms. This language follows ideas similar to FOPLP. The different means of achieving a goal G w.r.t. a program P correspond to the λ-normal proof-terms of the uniform linear focused derivations of G w.r.t. to P in HH. This class of derivations is in 1-1 correspondence to the expanded normal deductions of G' (the λ-normal form of G) w.r.t. P' (the program obtained by replacing the formulae in P by their λ-normal forms) in $NN^{\lambda norm}$, a natural deduction system for higher-order hereditary Harrop logic.

We take HOPLP as our starting point for integrating logic and functional programming, since we may express directly in HOPLP arbitrary relations, by means of formulae, and functions, by means of λ-abstractions. Roughly, LFPL is obtained from HOPLP by allowing a mechanism for defining names for λ-terms (simple definitions) and a mechanism for declaring specifications of functions (definitions of dependent type).

Traditionally, logic programming constructs cut-free derivations for goal-achievement. In LFPL, the use of a definition $p =_{def} e : D$ of dependent type for goal-achievement may be thought of as the use of a (constrained) cut, whose cut-formula is the specification D declared in the definition and whose derivation of D from other formulae in the program is determined by e, thus building a derivation with the subformula property. There are cases, as illustrated in Sec. 4.7, where the use of definitions of dependent type permits more efficiency in goal-achievement. When attempting to achieve a goal G, the occurrences of function names in G control the uses of cuts. If there is a definiendum occurring in G then a cut, whose cut formula is the specification of that definiendum, is attempted for achieving G.

Section 4.6 shows that LFPL is both sound and complete for HOPLP, in the sense that a goal G is achievable w.r.t. a program P in LFPL iff G' (the interpretation of G in HOPLP) is achievable w.r.t. P' (the interpretation of P in HOPLP) in HOPLP. Further, the different means of goal-achievement of G w.r.t. P in LFPL are in a 1-1 correspondence with the different means of goal-achievement of G' w.r.t. P' in HOPLP. So, LFPL is interpretable by means of the natural deduction system $NN^{\lambda norm}$. The programming language LFPL may be thought of as an equivalent language to HOPLP, the difference being that LFPL allows extra mechanisms for writing programs, making them clearer, and uses those mechanisms for achieving goals more efficiently.

The means of goal-achievement in LFPL are interpreted in HOPLP essentially by cut-elimination. For each means of goal-achievement in LFPL using definitions of dependent type there is a means of goal-achievement in LFPL using no such definitions, whose interpretation in HOPLP is the same. Sometimes, as shown in Sec. 4.7, the use of definitions of dependent type provides shorter forms of goal-achievement, since the corresponding derivations using no such definitions are (exponentially) longer.

Computation in LFPL may be divided into two steps, as in [PW92]: the first step is proof-search, where, due to the use of *cuts*, the proofs found may be shorter than in HOPLP; and the second step is an extraction of the witnesses for the existentially quantified variables in the goal from the proof obtained in the first step. The extraction step may require normalisation of some parts of the proof. However, this normalisation may be done lazily, for only the witnesses for the existentially quantified variables in the goal need to be exhibited.

This thesis uses systems with proof-terms annotating formulae for defining logic programming languages. These systems permit a simple type-theoretic account of such languages. By viewing propositions as types, a program (set of annotated formulae) is a type assignment (context) and a goal (formula) is a type. Achieving a goal G w.r.t. a program P is a search for an object (proof-term) of type G under type-assignment P. This view of logic programming follows similar ideas to the language Elf [Pfe89], based on $\lambda\Pi$-calculus (the theory of dependent types underlying LF), where the results of computations are objects of goal-type. The language LFPL may be thought of as a dependent type theory that provides some definition mechanisms, where the definitions and types allowed are restricted in such a way that search for objects of a type may be done efficiently by means of goal-directed proof-search.

6.2 Open Problems

Allowing cuts for building derivations introduces redundancy in the means of achieving goals w.r.t. programs. Recall that in LFPL, a derivation using def_I (cuts) and its ulf-form, essentially obtained by cut elimination followed by reduction to ulf-form, are regarded as the same means of goal-achievement, *cf* Sec. 4.6. The implementation of LFPL, suggested in Sec. 5.4, deals with this redundancy problem in a very inefficient way. Each time witnesses of goal-achievement are found, their ulf-forms are calculated and compared with the ulf-forms of the witnesses already known. An interesting question still to be resolved is whether or not, when searching for alternative witnesses, a set S of witnesses may be used for pruning the search space, either positively, by ensuring that the ulf-forms that may be found within some branches of the search space are already in S, or negatively, by ensuring that there are no possible witnesses within some branches of the search space.

The redundancy problem mentioned above is also of interest in the wider context of knowledge bases. Consider a knowledge base that has redundant information and keeps control of how such redundant information is related. How can one use effectively the information about redundancy for withdrawing conclusions from the knowledge base? (In our case: a knowledge base is a program; the redundant information appears in the form of the formulae in the definitions of dependent type; and proof-terms (definientia) of definitions of dependent type show how those formulae are related to other formulae in the program.)

In Sec. 5.4 are sketched two search procedures, *Proc. 1* and *Proc. 2*, for implementing LFPL. They differ only in the components (formulae or definitions of dependent type) of the program that may be selected for achieving atomic goals. *Proc. 1* allows the selection of any component of the program and is capable of finding all EULF-derivations. *Proc. 2* also permits the selection of any formula in the program, but reduces significantly the choice of definitions of dependent type, as compared to *Proc. 1*, since it only permits the selection of a definition of dependent type if some of its definienda occur in the goal, *i.e.* *Proc. 2* finds only sensible derivations. The drawback is that some EULF-derivations may not be found with *Proc. 2*. Section 5.4 gives a characterisation of the EULF-derivations which are not sensible (Theorem 5.6) and describes which sensible derivations are not found by *Proc. 2*. We would like to investigate other possible forms of characterising the derivations found by *Proc. 1* which are not found by *Proc. 2* that may help clarify the relation between advantages of using *Proc. 2*, by pruning the search space, and drawbacks of using *Proc. 2*, by failing to find some EULF-derivations.

Typed λ-calculus is a weak type system for representing functions. (Recall that only *extended polynomials* may be represented in typed λ-calculus, as shown in [Sch75].) In future work, we intend to extend LFPL with inductive datatypes and recursion and allow the programmer to use some induction principles for constructing derivations of properties universally quantified, over inductive types, in specifications; derivations using these induction principles should have induction-free counterparts, so that derivations using definitions of dependent type are interpretable as derivations using no such definitions. Within such extensions of LFPL we hope to able to express more complex examples than the example presented in Sec. 4.7 that benefit from our proposal for integrating logic and functional programming.

An interesting exercise would be to give a precise interpretation of the language LFPL by means of the Extended Calculus of Constructions (ECC) [Luo94]. The ECC provides the impredicative type of propositions, where formulae may be interpreted, and a cumulative hierarchy of predicative universes, that we may use for interpreting simple types. In this setting, ECC's Σ-types may be used to interpret directly the types of definitions of dependent type. Having an interpretation of LFPL into ECC, there is a precise framework for investigating relations between definitions of dependent type and deliverables[MB93].

By taking the view of logic programming as a means of performing meta-logical studies, it would be pertinent to investigate relations between the definition mechanisms allowed in LFPL, namely the mechanism for making definitions of dependent type, and the module system suggested in [HP91] for Elf.

The semantics of the logic programming language FOPLP is defined by means of hH, a sequent calculus for first-order hereditary Harrop logic. However, there is a clear interpretation of FOPLP by means of the natural deduction system NN, defined in Sec. 3.5. In order to achieve this interpretation the intermediate calculus MM [1] is used; this is essentially a version

[1]The calculus MM is a fragment of the permutation-free calculus for intuitionistic first-order logic, based on

of the calculus hH where permutation variants of a derivation are identified, *i.e.* a permutation-free calculus. (Recall that, derivations in MM are in 1-1 correspondence to expanded normal deductions.) We would like to follow a similar approach to investigate possible characterisations of disjunctive logic programming [NL95] by means of natural deduction systems. The study of permutation-free calculi is also of particular interest in the context of languages based on linear logic, *cf* [And92a, Mil94, GP94, PH94], where there are usually many permutation variants of a derivation. We would also like to investigate this topic.

Other problems left open throughout this thesis or motivated by this thesis, that we would like to study in future work, comprehend:

- Investigations on the conjecture: proofs in LJ are interpreted (in Prawitz's sense) as the same deduction in NJ iff they are permutable (in Kleene's sense), in other words, if $\Sigma; \Delta \Rightarrow e_1 : F$ and $\Sigma; \Delta \Rightarrow e_2 : F$ are derivable in LJ^{pt}, then $e_1 \cong_e e_2$ iff $\phi(e_1) = \phi(e_2)$. (This result has so far been shown for the implicational fragment, *cf* Sec. 2.3.4.)

- Can the rewriting systems RS_{uf} and RS_{ulf} be transformed into strongly normalising and Church-Rosser rewriting systems, within which every proof-term is reducible to a uniform focused and a uniform linear focused, respectively, proof-term?

- Is there a 1-1 correspondence between uniform focused derivations (notice that the linearity constraint has been dropped) in a sequent calculus for intuitionistic logic and natural deduction systems that allow deductions to be directed acyclic graphs, rather than trees? (Recall that the linearity constraint in witnesses for achieving a goal, *e.g.* in FOPLP, comes from the fact that side formulae of left rules may be used only once. This restriction is essentially motivated by implementation issues, since it would be very costly to add to the program all the side formulae of the left rules used in a derivation for achieving a goal, which would need to be retracted in case of backtracking.)

- An interpretation (normalisation procedure) of HH^{cut} -derivations as uniform linear focused derivations of HH may be described at the level of proof-terms by means of the weakly normalising rewriting systems RS_{cut} and RS^{λ}_{ulf} (RS_{ulf} with λ-terms rather than first-order terms), whose normal forms are respectively HH-proof-terms and uniform linear focused proof-terms. (This interpretation essentially corresponds to the interpretation of LFPL-witnesses as means of goal-achievement in HOPLP.) We intend to study new procedures for normalising proof-terms of HH^{cut} as uniform linear focused proof-terms of HH, by interleaving both rewriting systems. In particular, we are interested in efficient procedures for deciding whether or not two proof-terms have the same normal form; such procedures are useful for dealing with the redundancy issue in implementations of LFPL. Following ideas similar to those in [Wad93], we may describe functional languages whose

[Her95], presented in [DP96b].

execution mechanism is normalisation of proof-terms to cut-free uniform (linear) focused form. The study of procedures for normalising proof-terms of HH^{cut} (and extensions thereof) as proof-terms of HH is also of interest for implementing such languages.

Appendix A

Relations amongst the main calculi

$$\begin{tikzcd}
HH^{def} \arrow[d, "(6)"] & & \\
HH^{cut} \arrow[d, two heads, shift left, "(5)"] & & \\
HH \arrow[u, hook, shift left] \arrow[rr, "(4)"] & & NN^{\lambda norm} \\
hH \arrow[u, hook] \arrow[r, two heads, shift left, "(2)"] \arrow[d, hook] & hH^{ULF} \arrow[l, hook, shift left] \arrow[r, leftrightarrow, "(3)"] & NN \arrow[u, hook] \arrow[d, hook] \\
LJ^{pt} \arrow[rr, "(1)"] & & NJ^{pt}
\end{tikzcd}$$

$\longrightarrow$: mapping

$\hookrightarrow$: embedding

$\twoheadrightarrow$: surjection

$\longleftrightarrow$: bijection

(1) - Theorem 2.7
(2) - Corollary 3.3 and Theorem 3.7
(3) - Theorem 3.18
(4) - Theorems 3.24 and 3.25
(5) - Theorem 4.4
(6) - Theorems 4.5 and 4.6

Appendix B

Derivations in hH

Below is presented the hH-derivation[1] corresponding to witness (i) of Fig. 3.5 for the achievement of the goal

$$G = A_1 \supset (((A_2 \supset A_3) \supset A_2) \supset A_3)$$

w.r.t. the basis $\Sigma; \Delta$, where $\Delta = x : A_1 \supset (A_2 \supset A_3), x_1 : A_1$.

Consider the following abreviations.

$$\begin{array}{lll}
\Delta_1 & =_{def} & \Delta, x_2 : A_1, x_3 : (A_2 \supset A_3) \supset A_2 \\
e & =_{def} & apply(x, x_1, x_4.e_1) \\
e_1 & =_{def} & lambda(x_2.e_2) \\
e_2 & =_{def} & lambda(x_3.e_3) \\
e_3 & =_{def} & apply(x_4, e_4, x_5.x_5) \\
e_4 & =_{def} & apply(x_3, e_5, x_6.x_6) \\
e_5 & =_{def} & lambda(x_7.e_6) \\
e_6 & =_{def} & apply(x_4, x_7, x_8.x_8)
\end{array}$$

Let π_1 be the hH-derivation:

$$\dfrac{\dfrac{\dfrac{\vdash \Sigma; \Delta_1, x_4 : A_2 \supset A_3, x_7 : A_2 \; basis}{\Sigma; \Delta_1, x_4 : A_2 \supset A_3, x_7 : A_2 \Rightarrow x_7 : A_2}\,axiom \quad \dfrac{\vdash \Sigma; \Delta_1, x_4 : A_2 \supset A_3, x_7 : A_2, x_8 : A_3 \; basis}{\Sigma; \Delta_1, x_4 : A_2 \supset A_3, x_7 : A_2, x_8 : A_3 \Rightarrow x_8 : A_3}\,axiom}{\Sigma; \Delta_1, x_4 : A_2 \supset A_3, x_7 : A_2 \Rightarrow e_6 : A_3}\,{\Rightarrow\supset}}{\Sigma; \Delta_1, x_4 : A_2 \supset A_3 \Rightarrow e_5 : A_2 \supset A_3}\,{\Rightarrow\supset}$$

Let π_2 be the hH-derivation:

$$\dfrac{\overset{\pi_1}{\Sigma; \Delta_1, x_4 : A_2 \supset A_3 \Rightarrow e_5 : A_2 \supset A_3} \quad \dfrac{\vdash \Sigma; \Delta_1, x_4 : A_2 \supset A_3, x_6 : A_2 \; basis}{\Sigma; \Delta_1, x_4 : A_2 \supset A_3, x_6 : A_2 \Rightarrow x_6 : A_2}\,axiom}{\Sigma; \Delta_1, x_4 : A_2 \supset A_3 \Rightarrow e_4 : A_2}\,{\supset\Rightarrow}$$

[1]Recall that hH-derivations are unique up to the name of bound variables and the derivations of auxiliary judgements.

Let π_3 be the hH-derivation:

$$\dfrac{\overset{\pi_2}{\Sigma;\Delta_1, x_4 : A_2 \supset A_3 \Rightarrow e_4 : A_2} \quad \dfrac{\vdash \Sigma;\Delta_1, x_4 : A_2 \supset A_3, x_5 : A_3 \ \textit{basis}}{\Sigma;\Delta_1, x_4 : A_2 \supset A_3, x_5 : A_3 \Rightarrow x_5 : A_3}\ \textit{axiom}}{\Sigma;\Delta, x_2 : A_1, x_3 : (A_2 \supset A_3) \supset A_2, x_4 : A_2 \supset A_3 \Rightarrow e_3 : A_3}\ {\supset\Rightarrow}$$

Then, the following hH-derivation, whose proof-term is witness (i) of Fig. 3.5, may be formed.

$$\dfrac{\dfrac{\vdash \Sigma;\Delta \ \textit{basis}}{\Sigma;\Delta \Rightarrow x_1 : A_1}\ \textit{axiom} \quad \dfrac{\dfrac{\overset{\pi_3}{\Sigma;\Delta, x_4 : A_2 \supset A_3, x_2 : A_1, x_3 : (A_2 \supset A_3) \supset A_2 \Rightarrow e_3 : A_3}}{\Sigma;\Delta, x_4 : A_2 \supset A_3, x_2 : A_1 \Rightarrow e_2 : ((A_2 \supset A_3) \supset A_2) \supset A_3}\ {\Rightarrow\supset}}{\Sigma;\Delta, x_4 : A_2 \supset A_3 \Rightarrow e_1 : A_1 \supset (((A_2 \supset A_3) \supset A_2) \supset A_3)}\ {\Rightarrow\supset}}{\Sigma;\Delta \Rightarrow e : A_1 \supset (((A_2 \supset A_3) \supset A_2) \supset A_3)}\ {\supset\Rightarrow}$$

Appendix C

Derivable judgements of HH'

$$\frac{\Sigma;\Delta \vdash \Lambda_1 : \tau_1 \cdots \Sigma;\Delta \vdash \Lambda_n : \tau_n}{\Sigma;\Delta \vdash p\Lambda_1 ... \Lambda_n\ af}\ p : \tau_1 \rightarrow ... \rightarrow \tau_n \rightarrow prop \in \mathcal{P}$$

Well-formed atomic formulae.

$$\frac{\Sigma;\Delta \vdash A\ af}{\Sigma;\Delta \vdash A\ hf} \qquad \frac{\Sigma;\Delta \vdash H_1\ hf \quad \Sigma;\Delta \vdash H_2\ hf}{\Sigma;\Delta \vdash H_1 \wedge H_2\ hf}$$

$$\frac{\Sigma;\Delta \vdash H\ hf \quad \Sigma;\Delta \vdash G\ gf}{\Sigma;\Delta \vdash G \supset H\ hf} \qquad \frac{\Sigma, x:\tau;\Delta \vdash H\ hf \quad \vdash \Sigma;\Delta\ basis}{\Sigma;\Delta \vdash \forall_{x:\tau} H\ hf}\ x \notin \Sigma$$

Well-formed program formulae.

$$\frac{\Sigma;\Delta \vdash A\ af}{\Sigma;\Delta \vdash A\ gf} \qquad \frac{\Sigma;\Delta \vdash G_1\ gf \quad \Sigma;\Delta \vdash G_2\ gf}{\Sigma;\Delta \vdash G_1 \wedge G_2\ gf}$$

$$\frac{\Sigma;\Delta \vdash G_1\ gf \quad \Sigma;\Delta \vdash G_2\ gf}{\Sigma;\Delta \vdash G_1 \vee G_2\ gf} \qquad \frac{\Sigma;\Delta \vdash G\ gf \quad \Sigma;\Delta \vdash H\ hf}{\Sigma;\Delta \vdash H \supset G\ gf}$$

$$\frac{\Sigma, x:\tau;\Delta \vdash G\ gf \quad \vdash \Sigma;\Delta\ basis}{\Sigma;\Delta \vdash \exists_{x:\tau} G\ gf}\ x \notin \Sigma \qquad \frac{\Sigma, x:\tau;\Delta \vdash G\ gf \quad \vdash \Sigma;\Delta\ basis}{\Sigma;\Delta \vdash \forall_{x:\tau} G\ gf}\ x \notin \Sigma$$

Well-formed goal formulae.

Figure C.1: Well-formed HH'-formulae.

$$\frac{\Sigma;\Delta \vdash \Lambda \rhd_\tau \Lambda_1}{\Sigma;\Delta \vdash \Lambda \rhd^*_\tau \Lambda_1}$$

$$\frac{\Sigma;\Delta \vdash \Lambda : \tau}{\Sigma;\Delta \vdash \Lambda \rhd^*_\tau \Lambda} \qquad \frac{\Sigma;\Delta \vdash \Lambda \rhd^*_\tau \Lambda_1 \quad \Sigma;\Delta \vdash \Lambda_1 \rhd^*_\tau \Lambda_2}{\Sigma;\Delta \vdash \Lambda \rhd^*_\tau \Lambda_2}$$

$$\frac{\Sigma;\Delta \vdash \Lambda \rhd^*_\tau \Lambda_1}{\Sigma;\Delta \vdash \Lambda \equiv_\tau \Lambda_1}$$

$$\frac{\Sigma;\Delta \vdash \Lambda_1 \equiv_\tau \Lambda}{\Sigma;\Delta \vdash \Lambda \equiv_\tau \Lambda_1} \qquad \frac{\Sigma;\Delta \vdash \Lambda \equiv_\tau \Lambda_2 \quad \Sigma;\Delta \vdash \Lambda_2 \equiv_\tau \Lambda_1}{\Sigma;\Delta \vdash \Lambda \equiv_\tau \Lambda_1}$$

Figure C.2: Reduction in zero or more steps and conversion on HH'-terms.

$$\frac{\Sigma;\Delta \vdash \Lambda_1 \equiv_{\tau_1} \Lambda'_1 ... \Sigma;\Delta \vdash \Lambda_n \equiv_{\tau_n} \Lambda'_n}{\Sigma;\Delta \vdash p\Lambda_1...\Lambda_n \equiv p\Lambda'_1...\Lambda'_n} \quad p : \tau_1 \rightarrow ... \rightarrow \tau_n \rightarrow prop \in \mathcal{P}$$

$$\frac{\Sigma;\Delta \vdash H_1 \equiv H_3 \quad \Sigma;\Delta \vdash H_2 \equiv H_4}{\Sigma;\Delta \vdash H_1 \wedge H_2 \equiv H_3 \wedge H_4} \qquad \frac{\Sigma;\Delta \vdash G_1 \equiv G_2 \quad \Sigma;\Delta \vdash H_1 \equiv H_2}{\Sigma;\Delta \vdash G_1 \supset H_1 \equiv G_2 \supset H_2}$$

$$\frac{\Sigma, x:\tau;\Delta \vdash H_1 \equiv H_2 \quad \vdash \Sigma;\Delta\ basis}{\Sigma;\Delta \vdash \forall_{x:\tau} H_1 \equiv \forall_{x:\tau} H_2} \quad x \notin \Sigma$$

$$\frac{\Sigma;\Delta \vdash G_1 \equiv G_3 \quad \Sigma;\Delta \vdash G_2 \equiv G_4}{\Sigma;\Delta \vdash G_1 \wedge G_2 \equiv G_3 \wedge G_4} \qquad \frac{\Sigma;\Delta \vdash G_1 \equiv G_3 \quad \Sigma;\Delta \vdash G_2 \equiv G_4}{\Sigma;\Delta \vdash G_1 \vee G_2 \equiv G_3 \vee G_4}$$

$$\frac{\Sigma;\Delta \vdash H_1 \equiv H_2 \quad \Sigma;\Delta \vdash G_1 \equiv G_2}{\Sigma;\Delta \vdash H_1 \supset G_1 \equiv H_2 \supset G_2} \qquad \frac{\Sigma, x:\tau;\Delta \vdash G_1 \equiv G_2 \quad \vdash \Sigma;\Delta\ basis}{\Sigma;\Delta \vdash \exists_{x:\tau} G_1 \equiv \exists_{x:\tau} G_2} \quad x \notin \Sigma$$

$$\frac{\Sigma, x:\tau;\Delta \vdash G_1 \equiv G_2 \quad \vdash \Sigma;\Delta\ basis}{\Sigma;\Delta \vdash \forall_{x:\tau} G_1 \equiv \forall_{x:\tau} G_2} \quad x \notin \Sigma$$

Figure C.3: Convertible HH'-formulae.

$$\frac{\Sigma;\Delta \Rightarrow e_1 : G_1 \quad \Sigma;\Delta \Rightarrow e_2 : G_2}{\Sigma;\Delta \Rightarrow pair(e_1,e_2) : G_1 \wedge G_2} \Rightarrow\wedge$$

$$\frac{\Sigma;\Delta \Rightarrow e : G_1 \quad \Sigma;\Delta \vdash G_2\ gf}{\Sigma;\Delta \Rightarrow inl(e) : G_1 \vee G_2} \Rightarrow\vee_l \qquad \frac{\Sigma;\Delta \Rightarrow e : G_2 \quad \Sigma;\Delta \vdash G_1\ gf}{\Sigma;\Delta \Rightarrow inr(e) : G_1 \vee G_2} \Rightarrow\vee_r$$

$$\frac{\Sigma;\Delta_1, x : H, \Delta_2 \Rightarrow e : G}{\Sigma;\Delta_1,\Delta_2 \Rightarrow lambda(x.e) : H \supset G} \Rightarrow\supset$$

$$\frac{\Sigma;\Delta \Rightarrow e : [\Lambda/x]G \quad \Sigma;\Delta \vdash \Lambda : \tau}{\Sigma;\Delta \Rightarrow pair_q(\Lambda, e) : \exists_{x:\tau} G} \Rightarrow\exists$$

$$\frac{\Sigma, x : \tau;\Delta \Rightarrow e : G \quad \vdash \Sigma;\Delta\ basis}{\Sigma;\Delta \Rightarrow lambda_q(x.e) : \forall_{x:\tau} G} \Rightarrow\forall \quad x \notin \Sigma$$

$$\frac{\vdash \Sigma;\Delta_1, x : A, \Delta_2\ basis}{\Sigma;\Delta_1, x : A, \Delta_2 \Rightarrow x : A}\ axiom$$

$$\frac{\Sigma;\Delta_1, x : H_1 \wedge H_2, x_1 : H_1, \Delta_2 \Rightarrow e : G}{\Sigma;\Delta_1, x : H_1 \wedge H_2, \Delta_2 \Rightarrow splitl(x, x_1.e) : G} \wedge_l\Rightarrow$$

$$\frac{\Sigma;\Delta_1, x : H_1 \wedge H_2, x_1 : H_2, \Delta_2 \Rightarrow e : G}{\Sigma;\Delta_1, x : H_1 \wedge H_2, \Delta_2 \Rightarrow splitr(x, x_1.e) : G} \wedge_r\Rightarrow$$

$$\frac{\Sigma;\Delta_1, x : G_1 \supset H_1, \Delta_2 \Rightarrow e : G_1 \quad \Sigma;\Delta_1, x : G_1 \supset H_1, x_1 : H_1, \Delta_2 \Rightarrow e_1 : G}{\Sigma;\Delta_1, x : G_1 \supset H_1, \Delta_2 \Rightarrow apply(x, e, x_1.e_1) : G} \supset\Rightarrow$$

$$\frac{\Sigma;\Delta_1, x_1 : \forall_{x:\tau} H, x_2 : [\Lambda/x]H, \Delta_2 \Rightarrow e : G \quad \Sigma;\Delta_1 \vdash \Lambda : \tau}{\Sigma;\Delta_1, x_1 : \forall_{x:\tau} H, \Delta_2 \Rightarrow apply_q(x_1, \Lambda, x_2.e) : G} \forall\Rightarrow$$

$$\frac{\Sigma;\Delta_1, x : H_1, \Delta_2 \Rightarrow e : G \quad \Sigma;\Delta_1 \vdash H_1 \equiv H}{\Sigma;\Delta_1, x : H, \Delta_2 \Rightarrow e : G} \equiv_l \qquad \frac{\Sigma;\Delta \Rightarrow e : G_1 \quad \Sigma;\Delta \vdash G_1 \equiv G}{\Sigma;\Delta \Rightarrow e : G} \equiv_r$$

Figure C.4: Rules for deriving HH'-sequents.

Bibliography

[AKN89] H. Ait-Kaci and R. Nasr. Integrating logic and functional programming. *Lisp and Symbolic Computation*, 2:51–89, 1989.

[And92a] J. Andreoli. Logic programming with focusing proofs in linear logic. *Journal of Logic and Computation*, 2:297–347, 1992.

[And92b] J. Andrews. *Logic Programming: Operational Semantics and Proof Theory.* Cambridge University Press, 1992.

[AP91] J. Andreoli and R. Pareschi. Linear objects: Logical processes with built-in inheritance. *New Generation Computing*, 9:445–473, 1991.

[Bar81] H. Barendregt. *The Lambda Calculus: its Syntax and Semantics.* North-Holland, 1981.

[Bar93] H. Barendregt. Lambda calculi with types. In S. Abramsky and D. Gabbay, editors, *Handbook of Logic in Computer Science*, volume 2. Oxford Unversity Press, 1993.

[Bee89] M. Beeson. Some applications of Gentzen's proof theory in automated deduction. In P. Schroeder-Heister, editor, *1st Workshop on Extensions of Logic Programming (proceedings)*, volume 475 of *LNCS*, pages 101–156. Springer-Verlag, 1989.

[BL86] M. Bellia and G. Levi. The relation between logic and functional programming: A survey. *Journal of Logic Programming*, 3:217–236, 1986.

[Boo84] G. Boolos. Don't eliminate cut. *Journal of Philosophical Logic*, 13:373–378, 1984.

[BS94] F. Baader and J. Siekmann. Unification theory. In D. Gabbay, C. Hogger, and J. Robinson, editors, *Handbook of Logic in Artificial Intelligence and Logic Programming*, volume 2. Oxford Unversity Press, 1994.

[CF58] H. Curry and R. Feys. *Combinatory logic vol. I.* North-Holland, 1958.

[CH88] T. Coquand and G. Huet. The calculus of constructions. *Information and Computation*, 76:95–120, 1988.

[CHS72] H. Curry, J. Hindley, and J. Seldin. *Combinatory logic vol. II.* North-Holland, 1972.

[Chu32] A. Church. A set of postulates for the foundation of logic I. *Annals of Mathematics*, 32:346–366, 1932.

[Chu33] A. Church. A set of postulates for the foundation of logic II. *Annals of Mathematics*, 33:839–864, 1933.

[Chu40] A. Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.

[Cla78] K. Clark. Negation as failure. In H. Gallaire and J. Minker, editors, *Logic and Data Bases*, pages 293–322. Plenum New York, 1978.

[Coh88] J. Cohen. A view of the origins and development of Prolog. *Communications of the ACM*, 31:26–36, 1988.

[Coq90] T. Coquand. On the analogy between propositions and types. In G. Huet, editor, *Logical Foundations of Functional Programmming*. Addison-Wesley, 1990.

[dB72] N. de Bruijn. Lambda-calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. *Indagationes Mathematicae*, 34:381–392, 1972.

[dB80] N. de Bruijn. A survey of the project AUTOMATH. In [SH80], 1980.

[DL86] D. DeGroot and G. Lindstrom, editors. *Logic Programming, Functions, Relations and Equations*. Prentice-Hall, 1986.

[Dow93] G. Dowek. A complete proof synthesis method for the cube of tyepe systems. *Journal of Logic and Computation*, 3:287–315, 1993.

[DP94] R. Dyckhoff and L. Pinto. Uniform proofs and natural deductions. In Proc. of CADE-12 Workshop on "Proof search on type-theoretic languages" (edited by D. Galmiche & L. Wallen), Nancy, 1994.

[DP96a] R. Dyckhoff and L. Pinto. Permutability of inferences in cut-free LJ. In preparation. Abstract appeared in the Abstracts Volume of LMPS' 1995, 1996.

[DP96b] R. Dyckhoff and L. Pinto. A permutation-free sequent calculus for intuitionistic logic. In preparation, 1996.

[Dra88] A. Dragalin. *Mathematical Intuitionism, Introduction to Proof Theory*, volume 67 of *Translations of Mathematical Monographs*. American Mathematical Society, 1988.

[Fel91] A. Felty. Logic programming in the LF logical framework. In G. Huet and G. Plotkin, editors, *Logical Frameworks*, pages 215–249. Cambridge University Press, 1991.

[Gal93] J. Gallier. Constructive logics Part I: A tutorial on proof systems and typed λ-calculi. *Theoretical Computer Science*, 110:249–239, 1993.

[Gen35] G. Gentzen. Untersuchungen über das logische schliessen. *Mathematische Zeitschrift*, 39:176–210, 405–431, 1935. Translated as [Gen69].

[Gen69] G. Gentzen. Investigations into logical deduction. In M. Szabo, editor, *The Collected Papers of Gerhard Gentzen*, pages 68–131. North-Holland, 1969.

[Gir72] J. Girard. Interprétation fonctionnelle et élimination des coupures dans l'arithmétique d'ordre supérieur. thesis, University of Paris VII, 1972.

[Gir87] J. Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.

[GL90] M. Gelfond and V. Lifschitz. Logic programs with classical negation. In *7th Intern. Conf. on Logic Programming*, pages 579–597. MIT Press, 1990.

[GLT89] J.Y. Girard, Y. Lafont, and P. Taylor. *Proofs and Types.* Cambridge University Press, 1989.

[GP94] D. Galmiche and G. Perrier. Foundations of proof-search strategies design in linear logic. In *Symposium on Logical Foundations of Computer Science*, volume 813 of *LNCS*, pages 101–113. Springer, 1994.

[GR84] D. Gabbay and U. Reyle. N-Prolog: an extension of Prolog with hypothetical implications. *Journal of Logic Programmming*, 1:319–355, 1984.

[Han90] M. Hanus. Compiling logic programs with equality. In *Proc. of the 2nd Intern. Workshop on Programming Language Implemantation and Logic Programming*, volume 456 of *LNCS*, pages 387–401. Springer, 1990.

[Han92] M. Hanus. Improving control of logic programs by using functional logic languages. In *Proc. 4th Intern. Symp. Programming Language Implementation and Logic Programming*, volume 632 of *LNCS*, pages 1–23. Springer, 1992.

[Han94] M. Hanus. The integration of functions into logic programming: from theory to practice. *Journal of Logic Programming*, 20:583–628, 1994.

[Han95] M. Hanus. Curry: a truly functional logic language. In *Proc. ILPS'95 Workshop on Visions for the Future of Logic Programming*, pages 95–107, 1995.

[Har60] R. Harrop. Concerning formulae of the types $A \rightarrow B \vee C$, $A \rightarrow (Ex)B(x)$ in intuitionistic formal systems. *Journal of Symbolic Logic*, 25:27–32, 1960.

[Har94] J. Harland. A proof-theoretic analysis of goal-directed provability. *Journal of Logic and Computation*, 4:69–88, 1994.

[Her67] J. Herbrand. Investigations in proof theory. In Van Heijenoort, editor, *From Frege to Gödel*, pages 529–581. Harvard University Press, 1967. Translation.

[Her95] H. Herbelin. A λ-calculus structure isomorphic to sequent calculus structure. In L. Pacholsky and J. Tiuryn, editors, *Proc. of the 1994 Conference on Computer Science Logic*, volume 933 of *LNCS*, pages 61–75. Springer-Verlag, 1995.

[HHP87] R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. In *Proc. Second Annual Symposium on Logic in Computer Science*, pages 194–204. IEEE, 1987.

[HHP93] R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. *Journal of the Association for Computing Machinery*, 40:143–184, 1993.

[HL94] P. Hill and J. Lloyd. *The Gödel Programming Language.* MIT Press, 1994.

[HM94] J. Hodas and D. Miller. Logic programming in a fragment of intuitionistic linear logic. *Journal of Information and Computation*, 110:327–365, 1994.

[Hod93] W. Hodges. Logical features of Horn clauses. In D. Gabbay, C. Hogger, and J. Robinson, editors, *Handbook of Logic in Artificial Intelligence and Logic Programming*, volume 1, pages 449–503. Oxford Unversity Press, 1993.

[How69] W. Howard. The formulae-as-types notion of construction. Unpublished manuscript. Reprinted in [SH80], 1969.

[HP91] R. Harper and F. Pfenning. Modularity in the LF logical framework. Draft for the Proc. of the 2nd Workshop on Logical Frameworks, November 1991.

[HS86] J. Hindley and J. Seldin. *Introduction to combinators and λ-calculus*. Cambridge University Press, 1986.

[HSH90] L. Hallnäs and P. Schroeder-Heister. A proof-theoretical approach to logic programming. *Journal of Logic and Computation*, 1:261–283, 1990.

[Hue75] G. Huet. A unification algorithm for typed λ-calculus. *Theoretical Computer Science*, 1:27:57, 1975.

[Hue90] G. Huet, editor. *Logical Foundations of Functional Programmming*. Addison-Wesley, 1990.

[HW90] P. Hudak and P. Wadler. Report on the functional programming language Haskell. Draft, University of Glasgow, 1990.

[HW95] J. Harland and M. Winikoff. Implementation and development issues for the linear logic programming language Lygon. In *Proc. of the Eighteenth Australian Computer Science Conference*, pages 563–572, 1995.

[Jon87] S. Peyton Jones. *Implementation of Functional Programming Languages*. Prentice-Hall, 1987.

[Ker92] S. Keronen. Natural deduction proof theory for logic programming. In E. Lamma and P. Mello, editors, *3rd Workshop on Extensions of Logic Programming (proceedings)*, volume 660 of *LNCS*, pages 265–281. Springer-Verlag, 1992.

[Kle52] S. Kleene. Permutability of inferences in Gentzen's LK and LJ. *Memoirs of the American Mathematical Society*, 10, 1952.

[Kow88] R. Kowalski. The early years of logic programming. *Communications of the ACM*, 31:38–43, 1988.

[KR36] S. Kleene and J. Rosser. The inconsistency of certain formal logics. *Annals of Mathematics*, 36:630–636, 1936.

[Laf89] Y. Lafont. Functional programming and linear logic. Lecture Notes for the Summer School on Functional Programming and Constructive Logic, Glasgow, September 1989.

[Llo84] J. Lloyd. *Foundations of Logic Programming*. Springer Verlag, 1984.

[Llo94] J.W. Lloyd. Combining functional and logic programming languages. In *Proc. of the 1994 Intern. Logic Programming Symposium, ILPS'94*, pages 43–57, 1994.

[Lov91] D. Loveland. Near-Horn Prolog and beyond. *Journal of Automated Reasoning*, 7:1–26, 1991.

[Luo94] Z. Luo. *Computation and Reasoning: A Type Theory for Computer Science*. Oxford Science Publications, 1994.

[MB93] J. McKinna and R. Burstall. Deliverables: a categorical approach to program development in type theory. In A. Borzyszkowski and S. Sokolowski, editors, *Mathematical Foundations of Computer Science 1993*, volume 711 of *LNCS*, pages 32–67. Springer-Verlag, 1993.

[Mil89] D. Miller. A logical analysis of modules in logic programming. *Journal of Logic Programming*, 6:79–108, 1989.

[Mil90] D. Miller. Abstractions in logic programming. In P. Odifreddi, editor, *Logic and Computer Science*, pages 329–359. Academic Press, 1990.

[Mil91] Dale Miller. A logic programming language with lambda-abstraction, function variables and simple unification. *Journal of Logic and Computation*, 1:497–536, 1991.

[Mil94] D. Miller. A multiple-conclusion meta-logic. In *Proc. Ninth Symposium on Logic in Coputer Science, Paris, France*, pages 272–281. IEEE, 1994.

[Mil95] D. Miller. A survey of linear logic programming. To appear in the third issue of the Newsletter of the Network of Excellence on Computational Logic, 1995.

[Min94] G. Mints. Cut-elimination and normal forms of sequent derivations. Report No. CSLI-94-193, Stanford University, November 1994.

[ML82] P. Martin-Löf. Constructive mathematics and computer programming. In *Proc. of the Conf. on Logic, Philosophy and Methodology of Science VI, 1979*, pages 153–175. North-Holland, 1982.

[ML84] P. Martin-Löf. *Intuitionistic Type Theory*. Bibliopolis, Napoli, 1984.

[ML85] P. Martin-Löf. On the meanings of the logical constants and the justifications of the logical laws. Technical Report 2, Scuola di Specializzazione in Logica Matematica, Universita di Siena, 1985. Also published as [ML96].

[ML96] P. Martin-Löf. On the meanings of the logical constants and the justifications of the logical laws. *Nordic Journal of Philosophical Logic*, 1(1):11–60, 1996.

[MNPS91] D. Miller, G. Nadathur, F. Pfenning, and A. Scedrov. Uniform proofs as a foundation for logic programming. *Annals of Pure and Applied Logic*, 51:125–157, 1991.

[MNRA92] J. Moreno-Navarro and M. Rodriguez-Artalejo. Logic programming with functions and predicates: The language BABEL. *Journal of Logic Programming*, 12:191–223, 1992.

[Nad93] G. Nadathur. A proof procedure for the logic of hereditary Harrop formulas. *Journal of Automated Reasoning*, 11:115–145, 1993.

[NL95] G. Nadathur and D. Loveland. Uniform proofs and disjunctive logic programming. In *Proc. of the 10th IEEE Symposium on Logic in Computer Science*, pages 148–155, 1995.

[NM88] G. Nadathur and D. Miller. An overview of λProlog. In *Proc. Fifth Internat. Logic Programming Conference, Seattle*, pages 810–827. MIT Press, 1988.

[NM94] G. Nadathur and D. Miller. Higher-order logic programming. To appear in Volume 5 of *Handbook of Logic in Artificial Intelligence and Logic Programming*, 1994.

[NPS90] B. Nordström, K. Petersson, and J. Smith. *Programming in Martin-Löf's Type Theory.* Oxford University Press, 1990.

[Pau91] L. Paulson. *ML for the Working Programmer.* Cambridge University Press, 1991.

[PD94] L. Pinto and R. Dyckhoff. A constructive type system to integrate logic and functional programming. In Proc. of CADE-12 Workshop on "Proof search on type-theoretic languages" (edited by D. Galmiche & L. Wallen), Nancy, 1994.

[Pfe89] F. Pfenning. Elf: A language for logic definition and verified metaprogramming. In *Proc. of the Fourth Annual Symposium on Logic in Computer Science*, 1989.

[Pfe91] F. Pfenning. Logic programming in the LF logical framework. In G. Huet and G. Plotkin, editors, *Logical Frameworks*, pages 149–181. Cambridge University Press, 1991.

[Pfe92] F. Pfenning. Dependent types in logic programming. In F. Pfenning, editor, *Types in Logic Programming*, chapter 10, pages 285–311. MIT, 1992.

[Pfe94] F. Pfenning. Notes on deductive systems. Carnegie Mellon University, June 1994.

[PH94] D. Pym and J. Harland. A uniform proof-theoretic investigation of linear logic programming. *Journal of Logic and Computation*, 4:175–207, 1994.

[Pin94] L. Pinto. Cut formulae and logic programming. In R. Dyckhoff, editor, *4th Workshop on Extensions of Logic Programming (proceedings)*, volume 798 of *LNCS*, pages 281–300. Springer-Verlag, 1994.

[Pla93] D. Plaisted. Equational reasoning and term rewriting systems. In D. Gabbay, C. Hogger, and J. Robinson, editors, *Handbook of Logic in Artificial Intelligence and Logic Programming,* volume 1, pages 274–367. Oxford Unversity Press, 1993.

[Pot77] G. Pottinger. Normalization as a homomorphic inage of cut-elimination. *Annals of Mathematical Logic,* 12:323–357, 1977.

[Pra65] D. Prawitz. *Natural Deduction - A Proof-Theoretical Study.* Almqvist & Wiksell, 1965.

[Pra70] D. Prawitz. Ideas and results in proof theory. In *Proc. of the Second Scandinavian Logic Symposium.* North-Holland, 1970.

[PW91] D. Pym and L. Wallen. Proof-search in the $\lambda\Pi$-calculus. In G. Huet and G. Plotkin, editors, *Logical Frameworks*, pages 309–340. Cambridge University Press, 1991.

[PW92] D. Pym and L. Wallen. Logic programming via proof-valued computations. In K. Broda, editor, *Proc. 4th UK Conf. on Logic Programming, London, 1992.* Springer, 1992.

[Pym90] D. Pym. *Proofs, search and computation in general logic.* PhD thesis, University of Edinburgh, 1990.

[Rey74] J. Reynolds. Towards a theory of type structure. In B. Robinet, editor, *Programming Symposium, Paris,* volume 19 of *LNCS*, pages 408–425. Springer-Verlag, 1974.

[Rob65] J. Robinson. A machine-oriented logic based on resolution. *Journal of the Association for Computing Machinery,* 12:23–41, 1965.

[RW10] B. Russell and A. Whitehead. *Principia Mathematica.* Cambridge University Press, 1910.

[Sch75] H. Schwichtenberg. Definierbare funktionen im λ-kalkül mit typen. *Arch. Math. Logik Grunlagenforsh*, 17:113–114, 1975.

[SH80] J. Seldin and J. Hindley, editors. *To H. B. Curry, essays in Combinatory Logic, λ-calculus and Formalism.* Academic Press, 1980.

[Sha92] N. Shankar. Proof- search in the intuitionistic sequent calculus. In D. Kapur, editor, *Automated Deduction, CADE-11 (proceedings)*, volume 607 of *LNCS*, pages 522–536. Springer-Verlag, 1992.

[SS86] L. Sterling and E. Shapiro. *The Art of Prolog: Advanced Programming Techniques.* MIT Press, 1986.

[Tai67] W. Tait. Intensional interpretation of functionals of type I. *Journal of Symbolic Logic*, 32, 1967.

[Tho91] S. Thompson. *Type Theory and Functional Programming.* Addison-Wesley, 1991.

[TS96] T. Tammet and J. Smith. Optimised encodings of fragments of type theory in first order logic. In *Proc. of the BRA TYPES workshop held in 1995 in Torino*, LNCS. Springer, 1996. To appear.

[Tur86] D. Turner. An overview of Miranda. *ACM SIGPLAN Notices*, 21:156–166, 1986.

[TvD88] A. Troelstra and D. van Dalen. *Constructivism in mathematics: an introduction.* North-Holland, 1988.

[vEK76] M. van Emden and R. Kowalski. The semantics of predicate logic as a programming language. *Journal of the Association for Computing Machinery*, 23:733–742, 1976.

[Wad93] P. Wadler. A Curry-Howard isomorphism for sequent calculus. Preprint, University of Glasgow, December 1993.

[Wol93] S. Wolfram. *The Clausal Theory of Types*, volume 21 of *Cambridge Tracts in Theoretical Computer Science.* Cambridge University Press, 1993.

[WW92] S. Wainer and L. Wallen. Basic proof theory. In P. Aczel, H. Simmons, and S. Wainer, editors, *Proof Theory*, pages 3–26. Cambridge, 1992.

List of Figures