Proof-Relevant Resolution

The Foundations of Constructive Proof Automation

František Farka

Submitted for the degree of
Doctor of Philosophy

Heriot-Watt University

Department of Computer Science,
School of Mathematical and Computer Sciences.

May, 2019

Abstract

Dependent type theory is an expressive programming language. This language allows to write programs that carry proofs of their properties. This in turn gives high confidence in such programs, making the software trustworthy. Yet, the trustworthiness comes for a price: type inference involves an increasing number of proof obligations.

Automation of this process becomes necessary for any system with dependent types that aims to be usable in practice. At the same time, implementation of automation in a verified manner is prohibitively complex. Sometimes, external solvers are used to aid the automation. These solvers may be based on classical logic and may not be themselves verified, thus compromising the guarantees provided by constructive nature of type theory. In this thesis, we explore the idea of proof relevant resolution that allows automation of type inference in type theory in a verifiable and constructive manner, hence to restore the confidence in programs and the trustworthiness of software.

Technical content of this thesis is threefold. First, we propose a novel framework for proof-relevant resolution. We take two constructive logics, Horn-clause and hereditary Harrop formulae logics as a starting point. We formulate the standard big-step operational semantics of these logics. We expose their Curry-Howard nature by treating formulae of these logics as types and proofs as terms thus developing a theory of proof-relevant resolution. We develop small-step operational semantics of proof-relevant resolution and prove it sound with respect to the big-step operational semantics.

Secondly, we demonstrate our approach on an example of type inference in Logical Framework (LF). We translate a type-inference problem in LF into resolution in proof-relevant Horn-clause logic. Such resolution provides, besides an answer substitution to logic variables, a proof term that captures the resolution tree. We interpret the proof term as a derivation of well-formedness judgement of the object in the original problem. This allows for a straightforward implementation of type checking of the resolved solution since type checking is reduced to verifying the

derivation captured by the proof term. The theoretical development is substantiated by an implementation.

Finally, we demonstrate that our approach allows to reason about semantic properties of code. Type class resolution has been well-known to be a proof-relevant fragment of Horn-clause logic, and recently its coinductive extensions were introduced. In this thesis, we show that all of these extensions amalgamate with the theoretical framework we introduce. Our novel result here is exposing that the coinductive extensions are actually based on hereditary Harrop logic, rather than Horn-clause logic. We establish a number of soundness and completeness results for them. We also discuss soundness of program transformation that are allowed by proof-relevant presentation of type class resolution.

Index

Rodičům, dědovi a Lucce

In Edinburgh, Friday 17$^{\text{th}}$ May, 2019

Xi, H. and Pfenning, F. (1999). Dependent types in practical programming. In Appel, A. W. and Aiken, A., editors, *POPL '99, Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Antonio, TX, USA, January 20-22, 1999*, pages 214–227. ACM.

Stuckey, P. J. and Sulzmann, M. (2005). A theory of overloading. *ACM Trans. Program. Lang. Syst.*, 27(6):1216–1269.

Stump, A. (2009). Proof checking technology for satisfiability modulo theories. *Electr. Notes Theor. Comput. Sci.*, 228:121–133.

Sulzmann, M., Duck, G. J., Peyton Jones, S. L., and Stuckey, P. J. (2007). Understanding functional dependencies via constraint handling rules. *J. Funct. Program.*, 17(1):83–129.

Sulzmann, M. and Stuckey, P. J. (2008). HM(X) type inference is CLP(X) solving. *J. Funct. Program.*, 18(2):251–283.

Tait, W. W. (1967). Intensional interpretations of functionals of finite type i. *The Journal of Symbolic Logic*, 32(2):198–212.

Troelstra, A. S. (1991). History of constructivism in the 20th century. *ITLI Publ. Ser.*

Urban, C., Cheney, J., and Berghofer, S. (2011). Mechanizing the metatheory of LF. *ACM Trans. Comput. Log.*, 12(2):15:1–15:42.

Vazou, N., Tondwalkar, A., Choudhury, V., Scott, R. G., Newton, R. R., Wadler, P., and Jhala, R. (2018). Refinement reflection: complete verification with SMT. *PACMPL*, 2(POPL):53:1–53:31.

Vytiniotis, D., Jones, S. L. P., Schrijvers, T., and Sulzmann, M. (2011). Outsidein(x) modular type inference with local assumptions. *J. Funct. Program.*, 21(4-5):333–412.

Wadler, P. (2015). Propositions as types. *Commun. ACM*, 58(12):75–84.

Wadler, P. and Blott, S. (1989). How to make ad-hoc polymorphism less ad-hoc. In *Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages, Austin, Texas, USA, January 11-13, 1989*, pages 60–76. ACM Press.

Weirich, S., Voizard, A., de Amorim, P. H. A., and Eisenberg, R. A. (2017). A specification for dependent types in haskell. *PACMPL*, 1(ICFP):31:1–31:29.

HERIOT WATT UNIVERSITY

ACADEMIC REGISTRY
Research Thesis Submission

Name:	*František Farka*		
School:	School of Mathematical and Computer Sciences		
Version:	First	Degree Sought:	*Doctor of Philosophy*

Declaration

In accordance with the appropriate regulations I hereby submit my thesis and I declare that:

1. The thesis embodies the results of my own work and has been composed by myself
2. Where appropriate, I have made acknowledgement of the work of others
3. Where the thesis contains published outputs under Regulation 6 (9.1.2) these are accompanied by a critical re which accurately describes my contribution to the research and, for multi-author outputs, a signed declaration indicating the contribution of each author (complete Inclusion of Published Works Form – see below)
4. The thesis is the correct version for submission and is the same version as any electronic versions submitted*
5. My thesis for the award referred to, deposited in the Heriot-Watt University Library, should be made available f loan or photocopying and be available via the Institutional Repository, subject to such conditions as the Librari may require
6. I understand that as a student of the University I am required to abide by the Regulations of the University and conform to its discipline.
7. Inclusion of published outputs under Regulation 6 (9.1.2) shall not constitute plagiarism.
8. I confirm that the thesis has been verified against plagiarism via an approved plagiarism detection application Turnitin.

* *Please note that it is the responsibility of the candidate to ensure that the correct version of the thesis is subm*

Signature of Candidate:		Date:	

Submission

Submitted By *(name in capitals):*	
Signature of Individual Submitting:	
Date Submitted:	

For Completion in the Student Service Centre (SSC)

Received in the SSC by *(name in capitals):*			
Method of Submission *(Handed in to SSC; posted through internal/external mail):*			
E-thesis Submitted ***(mandatory for final theses)***			
Signature:		Date:	

Pientka, B. (2013). An insider's look at LF type reconstruction: everything you (n)ever wanted to know. *J. Funct. Program.*, 23(1):1–37.

Pientka, B. and Dunfield, J. (2010). Beluga: A framework for programming and reasoning with deductive systems (system description). In *Proc. IJCAR 2010*, pages 15–21.

Pitts, A. M. (2000). Parametric polymorphism and operational equivalence. *Mathematical Structures in Computer Science*, 10(3):321–359.

Ritter, E., Pym, D. J., and Wallen, L. A. (2000a). On the intuitionistic force of classical search. *Theor. Comput. Sci.*, 232(1-2):299–333.

Ritter, E., Pym, D. J., and Wallen, L. A. (2000b). Proof-terms for classical and intuitionistic resolution. *J. Log. Comput.*, 10(2):173–207.

Sangiorgi, D. (2009). On the origins of bisimulation and coinduction. *ACM Trans. Program. Lang. Syst.*, 31(4):15:1–15:41.

Schubert, A. and Urzyczyn, P. (2018). First-order answer set programming as constructive proof search. *TPLP*, 18(3-4):673–690.

Sewell, P., Nardelli, F. Z., Owens, S., Peskine, G., Ridge, T., Sarkar, S., and Strnisa, R. (2010). Ott: Effective tool support for the working semanticist. *J. Funct. Program.*, 20(1):71–122.

Simon, L., Bansal, A., Mallya, A., and Gupta, G. (2007). Co-logic programming: Extending logic programming with coinduction. In Arge, L., Cachin, C., Jurdzinski, T., and Tarlecki, A., editors, *Automata, Languages and Programming, 34th International Colloquium, ICALP 2007, Wroclaw, Poland, July 9-13, 2007, Proceedings*, volume 4596 of *Lecture Notes in Computer Science*, pages 472–483. Springer.

Slama, F. and Brady, E. (2017). Automatically proving equivalence by type-safe reflection. In *Proc. CICM 2017*, pages 40–55.

Stuckey, P. J. and Sulzmann, M. (2002). A theory of overloading. In *Proc. ICFP 2002*, pages 167–178.

Miller, D., Nadathur, G., Pfenning, F., and Scedrov, A. (1991). Uniform proofs as a foundation for logic programming. *Ann. Pure Appl. Logic*, 51(1-2):125–157.

Milner, R. (1978). A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375.

Negri, S. (2016). Glivenko sequent classes in the light of structural proof theory. *Arch. Math. Log.*, 55(3-4):461–473.

Norell, U. (2007). *Towards a practical programming language based on dependent type theory*. PhD thesis.

Odersky, M., Sulzmann, M., and Wehr, M. (1999). Type inference with constrained types. *TAPOS*, 5(1):35–55.

Ong, C. L. and Wagner, D. (2019). Hochc: a refutationally-complete and semantically-invariant system of higher-order logic modulo theories. *CoRR*, abs/1902.10396.

Orevkov, V. P. (1968). On glivenko sequent classes. In *Logical and logical-mathematical calculus. Part I*, volume 98 of *Trudy Mat. Inst. Steklov.*, pages 131–154.

Orevkov, V. P. (2006). A new decidable Horn fragment of predicate calculus. *J. Math. Sci.*, 134(5):2403–2410.

Peyton Jones, S., Jones, M., and Meijer, E. (1997). Type classes: an exploration of the design space. In *Haskell workshop*.

Pfenning, F. (1991). Logical frameworks. chapter Logic Programming in the LF Logical Framework, pages 149–181. Cambridge University Press, New York, NY, USA.

Pfenning, F. and Schürmann, C. (1999). System description: Twelf - A meta-logical framework for deductive systems. In Ganzinger, H., editor, *Automated Deduction - CADE-16, 16th International Conference on Automated Deduction, Trento, Italy, July 7-10, 1999, Proceedings*, volume 1632 of *Lecture Notes in Computer Science*, pages 202–206. Springer.

Contents

Jaffar, J. and Stuckey, P. J. (1986). Semantics of infinite tree logic programming. *Theor. Comput. Sci.*, 46(3):141–158.

Johann, P., Komendantskaya, E., and Komendantskiy, V. (2015). Structural resolution for logic programming. In Vos, M. D., Eiter, T., Lierler, Y., and Toni, F., editors, *Proceedings of the Technical Communications of the 31st International Conference on Logic Programming (ICLP 2015), Cork, Ireland, August 31 - September 4, 2015.*, volume 1433 of *CEUR Workshop Proceedings*. CEUR-WS.org.

Jones, M. P. (1994). A theory of qualified types. *Sci. Comput. Program.*, 22(3):231–256.

Kolmogorov, A. (1932). Zur deutung der intuitionistischen logik. *Mathematische Zeitschrift*, (35):58–65.

Komendantskaya, E. and Johann, P. (2015). Structural resolution: a framework for coinductive proof search and proof construction in horn clause logic. *Submitted to ACM Transcations in Computational Logic*.

Lämmel, R. and Peyton Jones, S. L. (2005). Scrap your boilerplate with class: extensible generic functions. In Danvy, O. and Pierce, B. C., editors, *Proceedings of the 10th ACM SIGPLAN International Conference on Functional Programming, ICFP 2005, Tallinn, Estonia, September 26-28, 2005*, pages 204–215. ACM.

Lloyd, J. W. (1987). *Foundations of Logic Programming, 2nd Edition*. Springer.

Martin-Löf, P. (1972). An intuitionistic theory of types. preprint.

Martin-Löf, P. (1982). Constructive mathematics and computer programming. In Cohen, L. J., Łoś, J., Pfeiffer, H., and Podewski, K.-P., editors, *Logic, Methodology and Philosophy of Science VI*, volume 104 of *Studies in Logic and the Foundations of Mathematics*, pages 153 – 175. Elsevier.

Miller, D. and Nadathur, G. (2012). *Programming with Higher-Order Logic*. Cambridge University Press.

Girard, J.-Y. (1972). Interpretation fonctionnelle et elimination des coupures de l'arithmetique d'ordre superieur. *PhD thesis, University of Paris VII*.

Glivenko, V. I. (1929). Sur quelques points de la logique de m. brouwer. *Bulletins de la classe des sciences*, 5(15):183–188.

Gonthier, G. and Mahboubi, A. (2010). An introduction to small scale reflection in Coq. *J. Formalized Reasoning*, 3(2):95–152.

Gonthier, G., Ziliani, B., Nanevski, A., and Dreyer, D. (2011). How to make ad hoc proof automation less ad hoc. In Chakravarty et al. (2011), pages 163–175.

Gregor, D. P., Järvi, J., Siek, J. G., Stroustrup, B., Reis, G. D., and Lumsdaine, A. (2006). Concepts: linguistic support for generic programming in C++. In Tarr, P. L. and Cook, W. R., editors, *Proceedings of the 21th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2006, October 22-26, 2006, Portland, Oregon, USA*, pages 291–310. ACM.

Hall, C. V., Hammond, K., Jones, S. L. P., and Wadler, P. (1996). Type classes in haskell. *ACM Trans. Program. Lang. Syst.*, 18(2):109–138.

Harper, R., Honsell, F., and Plotkin, G. D. (1993). A framework for defining logics. *J. ACM*, 40(1):143–184.

Harper, R. and Pfenning, F. (2005). On equivalence and canonical forms in the LF type theory. *ACM T. Comp. Log.*, 6(1):61–101.

Hemann, J., Friedman, D. P., Byrd, W. E., and Might, M. (2016). A small embedding of logic programming with a simple complete search. In *Proc. DLS 2016*, pages 96–107.

Heyting, A. (1934). *Mathematische Grundlagenforschung Intuitionismus Beweistheorie*. Springer-Verlag Berlin Heidelberg, 1 edition.

Howard, W. (1980). The formulae-as-types notion of construction. In Seldin, J. P. and Hindley, J. R., editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda-Calculus, and Formalism*, pages 479–490, NY, USA. Academic Press.

Proceedings, volume 9450 of *Lecture Notes in Computer Science*, pages 460–468. Springer.

Dyckhoff, R. and Negri, S. (2015). Geometrisation of first-order logic. *Bulletin of Symbolic Logic*, 21(2):123–163.

Farka, F. (2018). Proof-relevant resolution for elaboration of programming languages. In Palù, A. D., Tarau, P., Saeedloei, N., and Fodor, P., editors, *Technical Communications of the 34th International Conference on Logic Programming, ICLP 2018, July 14-17, 2018, Oxford, United Kingdom*, volume 64 of *OASICS*, pages 18:1–18:9. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik.

Farka, F., Komendantskaya, E., and Hammond, K. (2016). Coinductive soundness of corecursive type class resolution. In *Proc. LOPSTR 2016*.

Farka, F., Komendantskaya, E., and Hammond, K. (2018). Proof-relevant horn clauses for dependent type inference and term synthesis. *TPLP*, 18(3-4):484–501.

Fu, P. and Komendantskaya, E. (2017). Operational semantics of resolution and productivity in horn clause logic. *Formal Asp. Comput.*, 29(3):453–474.

Fu, P., Komendantskaya, E., Schrijvers, T., and Pond, A. (2016). Proof relevant corecursive resolution. In Kiselyov, O. and King, A., editors, *Functional and Logic Programming - 13th International Symposium, FLOPS 2016, Kochi, Japan, March 4-6, 2016, Proceedings*, volume 9613 of *Lecture Notes in Computer Science*, pages 126–143. Springer.

Geuvers, H. (1994). A short and flexible proof of strong normalization for the calculus of constructions. In Dybjer, P., Nordström, B., and Smith, J. M., editors, *Types for Proofs and Programs, International Workshop TYPES'94, Båstad, Sweden, June 6-10, 1994, Selected Papers*, volume 996 of *Lecture Notes in Computer Science*, pages 14–38. Springer.

Geuvers, H. and Barendsen, E. (1999). Some logical and syntactical observations concerning the first-order dependent type system lambda-p. *Mathematical Structures in Computer Science*, 9(4):335–359.

Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2010, October 17-21, 2010, Reno/Tahoe, Nevada, USA, pages 341–360. ACM.

De Angelis, E., Fioravanti, F., Pettorossi, A., and Proietti, M. (2015). Proving correctness of imperative programs by linearizing constrained horn clauses. *TPLP*, 15(4-5):635–650.

de Moura, L. M., Avigad, J., Kong, S., and Roux, C. (2015). Elaboration in dependent type theory. *CoRR*, abs/1505.04324.

de Moura, L. M. and Bjørner, N. (2008). Z3: an efficient SMT solver. In Ramakrishnan, C. R. and Rehof, J., editors, *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer.

Devriese, D. and Piessens, F. (2011). On the bright side of type classes: instance arguments in agda. In Chakravarty et al. (2011), pages 143–155.

Dowek, G. (1993). The undecidability of typability in the lambda-pi-calculus. In Bezem, M. and Groote, J. F., editors, *Typed Lambda Calculi and Applications, International Conference on Typed Lambda Calculi and Applications, TLCA '93, Utrecht, The Netherlands, March 16-18, 1993, Proceedings*, volume 664 of *Lecture Notes in Computer Science*, pages 139–145. Springer.

Dreyer, D., Ahmed, A., and Birkedal, L. (2009). Logical step-indexed logical relations. In *Proceedings of the 24th Annual IEEE Symposium on Logic in Computer Science, LICS 2009, 11-14 August 2009, Los Angeles, CA, USA*, pages 71–80. IEEE Computer Society.

Dunchev, C., Guidi, F., Coen, C. S., and Tassi, E. (2015). ELPI: fast, embeddable, \lambda prolog interpreter. In Davis, M., Fehnker, A., McIver, A., and Voronkov, A., editors, *Logic for Programming, Artificial Intelligence, and Reasoning - 20th International Conference, LPAR-20 2015, Suva, Fiji, November 24-28, 2015,*

1 | Introduction

Qual vaghezza di Lauro? qual di Mirto?
Povera, e nuda vai, Filosofia,
Dice la turba al vil guadagno intesa.

Pochi compagni avrai per l'altra via;
Tanto ti prego più, gentile spirto,
Non lassar la magnanima tua impresa.

— Francesco Petrarca, *Sonetto VII.*

Dependent type theory is an expressive programming language. This language allows to write programs that carry proofs of their properties. This in turn gives high confidence in such programs, making the software trustworthy. Yet, the trustworthiness comes for a price. Typing rules raise a number of proof obligations.

Automation of this process, which, for the sake of simplicity, we refer to as *type inference*, becomes necessary for any system with dependent types that aims to be usable in practice. At the same time, implementation of type inference in a verified manner is prohibitively complex. Sometimes, external solvers are used to aid it. These solvers may be based on classical logic and may not be themselves verified, thus compromising the guarantees provided by the constructive nature of type theory. In this thesis, we explore the idea of proof relevant resolution that allows both to carry out type inference in a verifiable manner and reason about semantics, hence to restore the confidence in programs and the trustworthiness of software.

1.1 Constructive Logic and Type Theory

First, we briefly mention the development of thought that leads to the general area in which lies the subject of this thesis starting with mathematical and philosophical origins. In the course of the 20th century, a new, normative point of view on what constitutes acceptable methods and objects of mathematics emerged—constructivism. This point of view originated as an opposing reaction to the use of highly abstract proof methods in works of, *e.g.*, Cantor and Dedekind. The original characterisation of constructivism was the appeal to proof methods that construct the objects of concern (hence the name). Alternatively, constructivism can be characterised by insisting on proof methods that *compute* the objects of concern (Troelstra, 1991). Theorems that state properties of certain objects give us means to construct, or compute, properties of these objects. Constructivist agenda in the form of *Brouwer's programme* led to development of *intuitionistic logic*. Heyting (1934) and Kolmogorov (1932) formalised intuitionistic logic and developed *Brouwer-Heyting-Kolmogorov* (BHK) interpretation of intuitionistic logic—a proof of implication is interpreted as a construction that transforms a proof of implicant into a proof of conclusion, negation is treated as an abbreviation for a construction that from a proposition absurdity follows.

The intuitionistic reading of proof in BHK interpretation is closely related to the notion of *propositions–as–types* (*cf.* Wadler, 2015). Curry was the first to suggest that a proposition in implicational form can be understood as a type of functions. Howard (1980) refined this idea with the observation that proof simplification can be understood as function evaluation. This is now referred to as *Curry-Howard interpretation* of proofs. Since early 70's, the idea of types has been a driving force behind an important part of computer science and propositions–as–types were providing a tight coupling between constructive mathematics and computer science. Martin-Löf (1972), directly inspired by Howard's ideas, introduced the *Intuitionistic theory of types* as a precise symbolism for constructive mathematics, and the notion of a *dependent type*, a type of objects that depend on proofs. However, he also explicitly linked constructive mathematics to computer science by regarding his intuitionistic theory of types as a programming language (Martin-Löf, 1982). In the

Basold, H., Komendantskaya, E., and Li, Y. (2018). Coinduction in uniform: Foundations for corecursive proof search with horn clauses.

Birkedal, L. and Harper, R. (1999). Relational interpretations of recursive types in an operational setting. *Inf. Comput.*, 155(1-2):3–63.

Bjørner, N., Gurfinkel, A., McMillan, K. L., and Rybalchenko, A. (2015). Horn clause solvers for program verification. In Beklemishev, L. D., Blass, A., Dershowitz, N., Finkbeiner, B., and Schulte, W., editors, *Fields of Logic and Computation II - Essays Dedicated to Yuri Gurevich on the Occasion of His 75th Birthday*, volume 9300 of *Lecture Notes in Computer Science*, pages 24–51. Springer.

Bottu, G., Karachalias, G., Schrijvers, T., d. S. Oliveira, B. C., and Wadler, P. (2017). Quantified class constraints. In *Haskell 2017*, pages 148–161.

Brady, E. (2013). Idris, a general-purpose dependently typed programming language: Design and implementation. *J. Funct. Program.*, 23(5):552–593.

Burn, T. C., Ong, C. L., and Ramsay, S. J. (2018). Higher-order constrained horn clauses for verification. *PACMPL*, 2(POPL):11:1–11:28.

Castro, D., Hu, R., Jongmans, S., Ng, N., and Yoshida, N. (2019). Distributed programming using role-parametric session types in go: statically-typed endpoint apis for dynamically-instantiated communication structures. *PACMPL*, 3(POPL):29:1–29:30.

Cave, A. and Pientka, B. (2018). Mechanizing proofs with logical relations - kripke-style. *Mathematical Structures in Computer Science*, 28(9):1606–1638.

Chakravarty, M. M. T., Hu, Z., and Danvy, O., editors (2011). *Proceeding of the 16th ACM SIGPLAN international conference on Functional Programming, ICFP 2011, Tokyo, Japan, September 19-21, 2011*. ACM.

Colmerauer, A. (1984). Equations and inequations on finite and infinite trees. In *FGCS*, pages 85–99.

d. S. Oliveira, B. C., Moors, A., and Odersky, M. (2010). Type classes as objects and implicits. In Cook, W. R., Clarke, S., and Rinard, M. C., editors,

Bibliography

Abadi, M., Cardelli, L., Curien, P., and Lévy, J. (1990). Explicit substitutions. In Allen, F. E., editor, *Conference Record of the Seventeenth Annual ACM Symposium on Principles of Programming Languages, San Francisco, California, USA, January 1990*, pages 31–46. ACM Press.

Abel, A., Öhman, J., and Vezzosi, A. (2018). Decidability of conversion for type theory in type theory. *PACMPL*, 2(POPL):23:1–23:29.

Ahn, K. Y. and Vezzosi, A. (2016). Executable relational specifications of polymorphic type systems using Prolog. In *FLOPS 2016*, pages 109–125.

Altenkirch, T. and Kaposi, A. (2017). Normalisation by evaluation for type theory, in type theory. *Logical Methods in Computer Science*, 13(4).

Appel, A. W., Michael, N. G., Stump, A., and Virga, R. (2003). A trustworthy proof checker. *J. Autom. Reasoning*, 31(3-4):231–260.

Armand, M., Faure, G., Grégoire, B., Keller, C., Théry, L., and Werner, B. (2011). A modular integration of SAT/SMT solvers to coq through proof witnesses. In Jouannaud, J. and Shao, Z., editors, *Certified Programs and Proofs - First International Conference, CPP 2011, Kenting, Taiwan, December 7-9, 2011. Proceedings*, volume 7086 of *Lecture Notes in Computer Science*, pages 135–150. Springer.

Asperti, A., Ricciotti, W., Coen, C. S., and Tassi, E. (2012). A bi-directional refinement algorithm for the calculus of (co)inductive constructions. *Logical Methods in Computer Science*, 8(1).

Barendregt, H. and Barendsen, E. (2002). Autarkic computations in formal proofs. *J. Autom. Reasoning*, 28(3):321–336.

following two decades, intuitionistic type theory found applications in interactive theorem provers, or proof assistants, like Coq or Agda (Norell, 2007), or the general purpose programming language Idris (Brady, 2013).

Around the same time as Martin-Löf was working on his theoretical developments, Milner was utilising the idea of types for a very practical purpose in the form of the theory of type polymorphism in programming languages (Milner, 1978). He coined the slogan that "well-typed programs cannot 'go wrong'". He connected his work to Hindley's principal type schemes in combinatory logic and observed that a language that is to be practically useful requires certain amount of automated reasoning. Resulting Hindley-Milner type inference algorithm for lambda calculus with parametric polymorphism was used in the ML programming language and strongly influenced the area of functional programming. ML's successors include commercially successful languages like OCaml and Haskell. The idea that programs should not "go wrong" gave rise to languages with expressive and powerful type systems. Such type systems allow to precisely encode invariants of programs in types and specify what it means not to "go wrong". An example of languages that originate in Hindley-Milner tradition and feature a powerful type system are Dependent ML (Xi and Pfenning, 1999) and Dependent Haskell (Weirich et al., 2017).

1.2 Trustworthiness of Automation

Since the initial steps taken by Milner in the form of automation of type inference for parametric polymorphism, automated reasoning has found a plethora of applications in type systems. First-order resolution is an example of automated reasoning that can be traced to Hindley-Milner type inference. Type inference in simply typed lambda calculus ($\lambda_{\to}$) can be expressed as a first-order unification problem. A general framework for Hindley-Milner type inference $HM(X)$ was developed by Odersky et al. (1999) and later formulated in terms of logic programming (Sulzmann and Stuckey, 2008). For example, the rule for term application in $\lambda_{\to}$

$$\frac{\Gamma \vdash M : A \to B \qquad \Gamma \vdash N : A}{\Gamma \vdash MN : B}\ \textsc{App}$$

gives rise to a type inference problem that can be encoded by the following Horn clause:

$$type(\Gamma, \mathrm{app}(M, N), B) \leftarrow type(\Gamma, M, A \to B) \wedge type(\Gamma, N, A)$$

Given a term E, the query $type(\Gamma, E, T)$ infers a type T in a context Γ such that the typing judgement $\Gamma \vdash E : T$ holds. In recent years, the idea of relational type inference has been taken further by a relational, embedded domain specific language (DSL) miniKanren (Hemann et al., 2016). The DSL supports a range of functional languages like ML, Rust, Haskell and many other non-functional languages. As Ahn and Vezzosi (2016) point out, a relational language is a very convenient device for encoding of type inference problems. However, automation of type inference in dependently typed languages represents a more substantial challenge. Most dependently typed languages incorporate a range of algorithms that automate various aspects of type inference (*cf.* Pientka, 2013). One approach is using reflection into underlying abstract syntax tree representation of the language (*cf.* Slama and Brady, 2017) to employ automation there. In some cases, the algorithms are similar to first-order resolution (Gonthier and Mahboubi, 2010), in other cases, *e.g.* Liquid Haskell (Vazou et al., 2018) and F*, languages incorporate external Satisfiability Modulo Theory (SMT) solvers like the Z3 Theorem Prover (de Moura and Bjørner, 2008).

The use of external solvers constitutes a dissent from constructivist ideas that initiated the interest in expressive type systems. As an example, consider that an external SMT solver is not verified and a bug may result in a wrong answer. Moreover the solver uses classical logic and the computed results need not be valid intuitionistically. In either of these two situations, soundness of type inference is compromised. That is, there are programs that are accepted by the type checker despite the fact that these programs cannot be shown well typed in the metatheory. The issue of trustworthiness of a computer system is well-recognised in the community (Barendregt and Barendsen, 2002). A general approach, called *autarkic* or *skeptical*, is for such a system to provide a machine checkable witness, a *proof term*, of correctness of the result (Appel et al., 2003). Stump (2009) did an early study of such proof-checking for SMT solvers and autarkic approach is a basis for SMT solving in, *e.g.*, Coq (Armand et al., 2011). Despite these results, there is no firm consensus in the community on the rigour of implementation of the language. This research area still remains very active and presents challenging problems (Schubert

malised proof of normalisation for dependent type theory using a proof-relevant logical predicate, a merge of presheaf model with logical relation. Moreover, the terms are presented as well-formed, using inductive-inductive types. We believe that the methodology of our framework can be successfully applied to proof search in the theory of the said logical predicate while obtaining proof terms witnessing well-formed terms.

basis for efficient proof search.

7.2.2 Elaboration of programming languages

A natural extension of our work on type inference and term synthesis in LF encompasses extending the language with more language constructs. To some extent, we already did this in our description of type class resolution. The proof-relevant treatment of type class resolution and the resulting proof term, or dictionary, represents a rudimentary form of elaboration. Combining the two systems is straightforward and results in a first order language with dependent types and type classes. A system that is to address realistic languages needs to support elaboration of features like Σ (record) types or a module system, and higher order term language. The former, namely Σ types, can be already supported in our framework by extending the internal language. A module system can be incorporated using an appropriate representation (*cf.* Miller and Nadathur, 2012). Higher-order elaboration was explored by de Moura et al. (2015) in the Lean theorem prover. We believe that refinement for higher order type theory can be presented in a similar way using our framework.

7.2.3 Coinductive semantics

Coinductive semantics in general admits potentially infinite data structures, *e.g.* streams. The framework in this thesis allows for proof-search in coinductive settings and provides coinductive proof terms. We believe that our framework can be effectively applied in the formal treatment of concurrent and distributed programs. A promising application of this technique is session types for concurrency and distribution (*e.g.* Castro et al., 2019). Further, coinductive reasoning admits mutual interleaving of inductive and coinductive structures. We would like to investigate proof-relevant inductive-coinductive reasoning for modelling of interleaving sequential and concurrent computation, and local and distributed computation using dependently typed language.

The use of proof-relevant methods in dependently typed languages promises to maintain close correspondence between transformations of syntax and language semantics. Recently, Altenkirch and Kaposi (2017) carried out a (partially) for-

and Urzyczyn, 2018, Vazou et al., 2018). Next we discuss an application of proof terms in type inference in detail. We refer to the approach that employs proof terms as *proof-relevant*.

1.2.1 Type-class resolution

Type classes are used to implement ad-hoc polymorphism and overloading in programming languages. The approach originated in Haskell (Hall et al., 1996, Wadler and Blott, 1989) and has been further developed in dependently typed functional languages (Devriese and Piessens, 2011, Gonthier et al., 2011) as well as in object-oriented languages (d. S. Oliveira et al., 2010, Gregor et al., 2006). *Type classes* introduce syntax that allows to specify a new class of types, equip it with certain *methods*, and provide implementations of these methods for particular types, hence making these types members of the class, in a compositional way. The implementations are called *instances*. We illustrate type class mechanism using Haskell.

Example 1.1 (Farka et al. (2016), Fu et al. (2016), Hall et al. (1996))
It is convenient to define equality for all data structures in a uniform way. In Haskell, this is achieved by introducing the class `Eq`*:*

```
class Eq a where
    eq :: a → a → Bool
```

and then declaring any necessary instances of the class, e.g. *for pairs and integers:*

```
instance (Eq x, Eq y) ⇒ Eq (x, y) where
    eq (x1, y1) (x2, y2) = eq x1 x2 && eq y1 y2


instance Eq Int where
    eq x y = primitiveIntEq x y
```

The distinguishing feature is that instances are defined separately from use sites of methods. *Type-class resolution*[1] is then used to infer a proper composition of instances, while hiding the technical details. The automated mechanism creates an internal object, a *dictionary*, that describes how to compose instances in order to

[1] Properly, the name should be type class instance resolution as it is instances that are being resolved. We will follow the common practice and omit the reference to instances.

execute the type class method in the use site.

Example 1.2 (Farka et al. (2016))

For example, it is required that `Eq (Int, Int)` *is a valid instance of type class* `Eq` *in order to type check the following function:*

```
test :: Eq (Int, Int) ⇒ Bool
test = eq (1,2) (1,2)
```

The function `test` *type checks since a comparison of pairs of integers can be simplified into a comparison of integers using the first instance in Example 1.1. Comparisons of integers are than carried out using the other instance.*

The initial work (Hall et al., 1996, Jones, 1994, Peyton Jones et al., 1997) on type classes focused on practical design of the language feature. This work did not make it explicit that type class resolution resembles SLD-resolution that is known from logic programming although it had been a long-standing folklore (*cf.* Farka et al., 2016). Fu and Komendantskaya (2017) extended the connection further: the constructed dictionary is an instance of a proof term and type-class resolution can be treated as an employment of *proof-relevant Horn-clause resolution.*

Example 1.3 (Farka et al. (2016), Fu et al. (2016))

The type class instance declarations in Example 1.1 can be viewed as the following two Horn clauses that are annotated with atomic symbols $\kappa_{\texttt{pair}}$ *and* $\kappa_{\texttt{int}}$*:*

$$\kappa_{\texttt{pair}} : \ \texttt{eq}(x),\ \texttt{eq}(y) \ \Rightarrow \texttt{eq}(\texttt{pair}(x, y))$$

$$\kappa_{\texttt{int}} : \ \Rightarrow \texttt{eq}(\texttt{int})$$

Then, given the query $\texttt{eq}(\texttt{pair}(\texttt{int}, \texttt{int}))$ *that corresponds to requirement* `Eq (Int , Int)` *in Example 1.2 SLD-resolution terminates successfully with the following sequence of resolution steps:*

7.1.3 Type class resolution

In our syntactical analysis of type class resolution, we addressed three research questions. First, we provided a uniform analysis of type class resolution in both inductive and coinductive settings and proved it sound relative to (standard) least and greatest Herbrand models. Secondly, we demonstrated, through several examples, that coinductive resolution is indeed coinductive—that is, it is not sound relative to least Herbrand models. Thirdly, we showed completeness relative to least Herbrand models in the inductive case and a lack thereof relative to greatest Herbrand models in the coinductive case. Finally, we asserted that the methods listed in this thesis can be reapplied to coinductive dialects of logic programming *via* soundness preserving program transformations.

A feature of our analysis is the choice of greatest Herbrand models instead of greatest complete models for coinductive analysis that is allowed by properties of type class resolution. We discussed how constrains that are laid upon type class instances allow such choice.

7.2 Future Work

7.2.1 Foundations of proof search

The underlying mechanism of proof search in our work, the uniform proofs, originates, via Curry-Howard isomorphism, in sequent calculus for Horn-clause and hereditary Harrop formulae logics. There are several other well-behaved classes of sequents (*cf.* Negri, 2016) with the advantage that sequents in these classes can be identified syntactically. A Curry-Howard interpretation of these classes has yet not been given and such interpretation is of an interest as it allows embedding of search based automation into verified programs. Further, Orevkov (2006) identified complexity characteristics of sequents in these classes that separate the logic into polynomially decidable subclasses. Application of Orevkov's results to proof-relevant search methodology would allow optimisation of the search in form of decomposition of the search space of the algorithm into subspaces of polynomial size. Since these classes of sequents are identified syntactically, this approach provides a promising

extended Horn clauses to obtain the logic of hereditary Harrop formulae. In parallel to the language, we gave a big-step operational semantics and a small-step operational semantics of proof-relevant resolution. The big-step semantics was obtained by instrumenting the standard semantics of resolution in logic programming, the *uniform proofs* semantics, with proof terms. The small-step operational semantics has not been presented in literature before. We showed that the small-step semantics is sound w.r.t. the big-step semantics. The small-step semantics involves reasoning about free variables and thus allows for a richer class of sequents than the big-step semantics. Thus, we employed logical relation and carried out the proof of soundness in two steps. we showed embedding of the small-step semantics into the logical relation and we showed *escape* from the logical relation to the big-step semantics.

The introduction of our framework in two, compositional steps has a practical motivation. In our case studies, we demonstrated that some applications allow to use the simpler, Horn-clause fragment. This was the case with type-inference and term-synthesis in LF. In the case study of type-class resolution we illustrated how increased requirements on the proof strength of the system force the use of a richer logic, moving form Horn clauses to hereditary Harrop formulae.

7.1.2 Type inference and term synthesis

Programming in languages with dependent types such as Agda, Coq or Idris is a complex task. The usability of such languages critically depends on the amount of automation that is provided to a programmer. Current automation is implementation dependent and hard to understand. This complicates the reuse of existing approaches in the development of tools for new languages or sharing between the existing implementations.

We presented a description of type inference and term synthesis in LF, first order dependent type theory that is significantly simpler than the existing approaches. We showed a translation of an incomplete term with metavariables to - goal and a program in Horn-clause logic by a syntactic traversal of the term. The inference is then performed by proof-relevant resolution. Moreover, the generated goal and program have a straightforward interpretation as judgements of type theory and inference rules and hence are easier to understand and to work with.

$$\texttt{eq}(\texttt{pair}(\texttt{int},\texttt{int})) \rightarrow$$

by the clause $\kappa_{\texttt{pair}}$

$$\texttt{eq}(\texttt{int}),\ \texttt{eq}(\texttt{int}) \rightarrow$$

by the clause $\kappa_{\texttt{int}}$

$$\texttt{eq}(\texttt{int}) \rightarrow$$

by the clause $\kappa_{\texttt{int}}$

$$\emptyset$$

The proof term $\kappa_{\texttt{pair}}\kappa_{\texttt{int}}\kappa_{\texttt{int}}$ *corresponds to a dictionary constructed by the compiler. It is treated internally as an executable function.*

Moreover, the explicit treatment of type-class resolution as Horn-clause resolution gives a firm basis for semantical analysis. The models of Horn-clause resolution serve as a semantics of the type-class mechanism.

1.3 Constructive Approach to Automation

A primary goal of this thesis is to establish a simple, conceptual framework for proof-relevant and constructive automated theorem proving in type inference. The automated theorem proving we consider is resolution in extensions of first-order Horn-clause logic. First order Horn-clause logic has been long understood as an expressive and constructive language (Dyckhoff and Negri, 2015). It has a wide use in program verification (*cf.* a survey by Bjørner et al., 2015, Burn et al., 2018, Ong and Wagner, 2019). Miller and Nadathur (2012) have shown that its semantics extends seamlessly to higher order terms and to hereditary Harrop formulae in a way that maintains the constructive nature of the logic. The importance of these extensions is demonstrated by the continuing work on type classes (Bottu et al.,

2017, Fu et al., 2016) and by applications in coinductive settings (Basold et al., 2018). We thus chose to build our framework on the higher-order hereditary Harrop logic by its instrumentation with proof terms. The complementary goal of this thesis is to demonstrate assets of such framework by its application to examples we described in previous sections.

1.4 Contributions

The technical contributions in this thesis span several areas.

Proof-relevant resolution This thesis develops a systematic and generic approach to proof relevant resolution. In particular:

- We identify higher-order Horn-clause (*hohc*) and hereditary Harrop logics (*hohh*) as the appropriate languages for the framework for proof-relevant resolution. We instrument the uniform proof (Miller and Nadathur, 2012) semantics of *hohc* and *hohh* with proof terms.
- We develop a small-step operational semantics of proof-relevant *hohc* and *hohh*.
- We show soundness of the small-step operational semantics w.r.t. the uniform proof semantics.

We show applications of the general framework in two settings.

Type inference and term synthesis in dependent type theory.

- We present a novel approach to type inference and term synthesis for a first-order type theory with dependent types that is simpler than existing methods (*e.g.* Pientka and Dunfield, 2010).
- We prove that generation of goals and logic programs from the extended language is decidable.
- We show that proof-relevant first-order Horn-clause resolution gives an appropriate inference mechanism for dependently typed languages: first, it is sound with respect to type checking in LF; secondly, the proof term construction alongside the resolution trace allows to reconstruct derivations of well-typedness judgements.

7 | Conclusions and Future Work

Die Herren wollen leben und zwar von der Philosophie *leben: […] trotz dem* povera e nuda vai filosofia *des Petrarka, es darauf gewagt.*

— Arthur Schopenhauer, *Die Welt als Wille und Vorstellung*

Dependent type theory is an expressive programming language for writing verified programs. Technical obligations of the type theory require a level of automation of proof obligations for any system with dependent types that aims to be usable in practice. In this thesis, we developed a simple, conceptual framework for such automation that is based in proof-relevant, constructive resolution in Horn-clause logic and its extension, the logic of hereditary Harrop formulae. We demonstrated applicability of our framework using two case studies. First, we used our framework for a syntactical manipulation of a programming language in form of type inference and term synthesis. Secondly, we used our framework in a semantical analysis by employing it for the purpose of a study of soundness and completeness, or rather a lack thereof, of type class construct. The use of the framework in both syntactical and semantical applications shows its generality.

In this chapter, we briefly conclude on the framework and on each of the applications. Next, we discuss some directions of future work.

7.1 Conclusions

7.1.1 Proof-relevant resolution

We introduced the language of our resolution framework and the corresponding semantics in two steps. First, we introduced the Horn-clause logic. Secondly, we

(Colmerauer, 1984, Jaffar and Stuckey, 1986) was studied from both an operational semantics and a declarative semantics point of view. Simon et al. (2007) introduced *co-logic programming* (co-LP) that also allows for terms that are rational infinite trees and hence that have infinite proofs. However, corecursive resolution, as studied in this paper, is more expressive than co-LP: while also allowing infinite proofs, and closing of coinductive hypotheses is less constrained in our approach.

We raised a question whether the context update technique given in Fu et al. (2016) can be reapplied to logic programming and can be re-used in its corecursive dialects such as CoLP Simon et al. (2007) and CoALP Komendantskaya and Johann (2015) or, even broader, whether it can be incorporated into program transformation techniques (De Angelis et al., 2015). The answer to the question is less straightforward. The way the implicative coinductive lemmata are used in proofs alongside all other Horn clauses in Fu et al. (2016) indeed resembles a program transformation method when considered from the logic programming point of view. In reality, however, different fragments of the calculus given in Fu et al. (2016) allow proofs for Horn clauses which, when added to the initial program, may lead to inductively or coinductively unsound extensions. We analysed this situation and highlight which program transformation methods can be soundly borrowed from existing work on corecursive resolution.

The formulation of corecursive type class resolution we used was given by Fu et al. (2016) and Fu and Komendantskaya (2017). They extended Howard's simply-typed λ-calculus (Howard, 1980) with a resolution rule and a ν-rule. The resulting calculus is general and accounts for all previously suggested kinds of type class resolution. We embedded the general framework into the calculus of proof-relevant resolution we gave in Chapter 3 in the inductive case. We have shown that the coinductive case of type class resolution is a proper extension of our calculus.

Type classes The semantical analysis of proof-relevant type class resolution provides the following contributions:

- We establish that type class resolution and its two recent corecursive extensions (Fu et al., 2016, Lämmel and Peyton Jones, 2005) are sound relative to the standard (Herbrand model) semantics of logic programming.
- We show that these extensions are indeed corecursive, *i.e.* that they are more accurately modelled by the greatest Herbrand model semantics rather than by the least Herbrand model semantics.
- We discuss whether the context update technique given by Fu et al. (2016) can be reapplied to logic programming and can be re-used in its corecursive dialects such as CoLP (Simon et al., 2007) and CoALP (Komendantskaya and Johann, 2015) or, even broader, whether it can be incorporated into program transformation techniques (De Angelis et al., 2015).

1.5 Structure of the Thesis

This chapter provides a general motivation for a proof-relevant, constructive framework for automated theorem proving.

Chapter 2 gives an overview of preliminaries. First, we give a language of Horn-clause logic that is used throughout the thesis. Secondly, we recall the notion of Herbrand models for logic programming that is used for semantical analysis of type classes. Finally, we give a nameless formulation of LF, which is the language that is subject to type inference and term synthesis in Chapter 5.

Chapter 3 introduces the general framework of proof-relevant resolution. We give a big-step operational semantics that is based on the uniform proof semantics and a small-step operational semantics of proof-relevant resolution in Horn-clause logic. We generalise the language of the framework to hereditary Harrop formulae and extend the semantics accordingly.

Chapter 4 show soundness of the small-step semantics w.r.t. the big step-semantics. The result is achieved by introducing a logical relation.

Chapter 5 show an application of our framework to type inference and term synthesis in nameless LF, a first-order type theory with dependent types.

Chapter 6 carries out a semantical analysis of soundness of proof-relevant type class resolution. We show soundness and completeness, or the lack of it, for different notions of inductive and coinductive interpretation of type-class resolution.

Chapter 7 concludes the thesis and discusses related and future work.

1.6 Declaration of Authorship

Chapter 2 contains background information. The definitions and results can be found in cited literature but the presentation has been adjusted to fit the scope of this thesis.

The contents of Chapters 3 and 4 are original work of the author. Chapters 5 and 6 are based on join work with Ekaterina Komendantskaya and Kevin Hammond, who were author's supervisors. Both the type inference and term synthesis approach (Farka et al., 2018) and the semantical analysis of type class resolution (Farka et al., 2016) has been published before. An initial exposition of applications of proof relevant resolution in a single framework that proceeds the ideas behind this thesis has also been published (Farka, 2018).

to use with any coinductive dialect of logic programming, *e.g.* with CoLP (Simon et al., 2007).

Example 6.43

Recall the Example 6.25 and the fact that, for any atomic formula A:

$$\cdot \longrightarrow \lambda\alpha.\alpha : A \Rightarrow A$$

Assume a program $\mathcal{P}$ consisting of a single formula $\kappa : A \Rightarrow B$. Both the least and the greatest Herbrand model of this program are empty. However, adding the formula $A \Rightarrow A$ to the program results in the greatest Herbrand model $\mathcal{M}'_{\kappa:A\Rightarrow B} = \{A, B\}$. Thus, $\mathcal{M}'_{\kappa:A\Rightarrow B} \neq \mathcal{M}'_{\kappa:A\Rightarrow B,\lambda\alpha.\alpha:A\Rightarrow A}$.

The Example 6.43 demonstrates that extending a program with a formula $A \Rightarrow A$ is not a coinductively sound transformation. However, calculus consisting of rules Lp-m and Lam as can be observed inductively sound by inspecting the proof of Theorem 6.37—rules of the calculus do not allow unguarded use of such Horn clauses in further entailment. In fact, rules of the calculus do not allow any use of such clauses in further entailment at all. On the other hand, both corecursive type class resolution and its extended version need to impose guardedness conditions on the proof term in order to ensure that any use of a Horn clause that was previously entailed is guarded in order to avoid unsound derivations. The side conditions of the rules Nu' and Nu requiring the proof term to be in the head normal form are exactly these conditions.

6.5 Related Work

The standard approach to type inference for type classes, corresponding to type class resolution as studied in this chapter, was described by Stuckey and Sulzmann (2005). Type class resolution was further studied by Lämmel and Peyton Jones (2005), who described what we here call *corecursive type class resolution*. The description of the extended calculus of Section 6.4 was first presented by Fu et al. (2016). In general, there is a rich body of work that focuses on allowing for infinite data structures in logic programming. Logic programming with rational trees

Let us denote a term $(\mathtt{s}\,(\mathtt{s}\,(\ldots(\mathtt{s}\;x)\ldots)))$ *where the symbol* s *is applied* i*-times as* $(\mathtt{s}^i\;x)$*. By observing the construction of* $\mathcal{M}'_{\mathcal{P}}$ *we can see that, for all* i*, if* $\mathtt{D}\;\mathtt{z}\;(\mathtt{s}^i\;x)$ *then* $\mathtt{D}\;(\mathtt{s}^i\;x)\;\mathtt{z} \in \mathcal{M}'_{\mathcal{P}}$ *and also* $\mathtt{D}\;\mathtt{z}\;(\mathtt{s}^{i-1}\;\mathtt{x}) \in \mathcal{M}'_{\mathcal{P}}$*. Therefore* $\mathtt{D}\;\mathtt{z}\;\mathtt{z} \in \mathcal{M}'_{\mathcal{P}}$*. However, there is no proof of* $\mathtt{D}\;\mathtt{z}\;\mathtt{z}$ *since any number of proof steps resulting from the use of* Lp-m *generates yet another ground premise that is different from all previous premises. Consequently, the proof cannot be closed by* Nu*. Also, no lemma that would allow for a proof can be formulated; an example of such a lemma would be the above* $\mathtt{D}\;\mathtt{z}\;(\mathtt{s}^i\;x) \Rightarrow \mathtt{D}\;\mathtt{z}\;(\mathtt{s}^{i-1}\;\mathtt{x})$*. This is a higher order formula and cannot be expressed in the first order Horn-clause logic we consider in this Chapter.*

6.4.2 Program transformation methods

We conclude this section with a discussion of program transformation with Horn clauses that are entailed by the rules Lam and Nu. From the fact that the Nu' rule is inductively unsound, it is clear that using program transformation techniques based on the lemmata that were proved by the Lam and Nu rules would also be inductively unsound. However, a more interesting result is that adding such program clauses will not change the coinductive soundness of the initial program:

Theorem 6.42
Let $\mathcal{P}$ *be a logic program, let* G *be a formula and let* e *be a proof term such that* $\mathrm{gHNF}(e)$*. Let* $\mathcal{S};\mathcal{P} \longrightarrow e : G$ *by the* Lp-m*,* Lam *and* Nu *rules. Given a formula* G'*,* $\mathcal{P} \vDash_{coind} G'$ *iff* $(\mathcal{P}, G) \vDash_{coind} G'$*.*

Proof. By the Theorem 6.37, $\mathcal{P} \vDash_{coind} G$. Therefore, $\mathcal{M}'_{\mathcal{P}}$ is a model of G and $\mathcal{M}'_{\mathcal{P}} = \mathcal{M}'_{\mathcal{P},G}$. Hence $\mathcal{P} \vDash_{coind} G'$ iff $\mathcal{P}, G \vDash_{coind} G'$. □

The above result is possible thanks to the guarded head normal form condition, since it is then impossible to use a clause $A \Rightarrow A$ that was derived from an empty context by the rule Lam. It is also impossible to make such a derivation within the proof term e itself and then derive A by the Nu rule from $A \Rightarrow A$. The resulting proof term will fail to satisfy the guarded head normal form condition that is required by Nu. Since this condition guards against any such cases, we can be sure that this program transformation method is coinductively sound and hence that it is safe

2 | Preliminaries

In this chapter, we discuss preliminaries that are needed in our development in the rest of the thesis. First, we introduce term language and Horn-clause logic that is studied and extended in this thesis. Secondly, we describe Herbrand models as a simple and convenient tool for the analysis of inductive and coinductive soundness of type class resolution we carry out in Chapter 6. Next, we describe a nameless variant of Logical Framework (LF) that is suitable for automated type inference and term synthesis that we introduce in Chapter 5.

2.1 Term Language

In this section, we introduce the language of terms that is used and extended in this thesis. The language is based on LF (Harper et al., 1993, Harper and Pfenning, 2005).

2.1.1 Syntax

The syntax of our language features separate terms, types, and kinds. *Terms* of our language consist of *term constants*, *variables*, abstraction and application and are classified by *types*. We let term constants to range over the set $\mathcal{C}$ and use identifiers c, d to denote individual constants. We let variables to range over the set $\mathcal{V}$ and use identifiers x, y for variables in general and identifiers X, Y for variables that are subject to unification. Types consist of *type constants*, type application and formation of dependent type families. Types are classified by *kinds*. We let type constants to range over the set $\mathcal{A}$. We use identifiers a, p, and q to denote individual constants in $\mathcal{A}$ unless stated otherwise. Kinds consists of two sorts, `o` and `type`, and formation of kind that classifies dependent type families. The intended meaning of

the sorts is to distinguish between types that stand in the position of formulae and in the position of types in the proof-relevant resolution. Formally, the language is given as follows.

Definition 2.1 (Syntax)

$\mathcal{C} \ni c, d$		*term constants*
$\mathcal{A} \ni a, p, q$		*type constants*
$\mathcal{V} \ni x, y, X, Y$		*variables*
$t \ni M, N$	$:= c \mid x \mid \lambda x : A.N \mid M\ N$	*terms*
$T \ni A, B$	$:= a \mid \Pi x : A.B \mid A\ M$	*types*
$K \ni L$	$:= \mathtt{type} \mid \mathtt{o} \mid \Pi x : A.L$	*kinds*

Terms in t are denoted using identifiers M, N, types in T are denoted using identifiers A, B and kinds in K are denoted using the identifier L. We use $A \to B$ as an abbreviation for the type $\Pi x : A.B$ when x does not occur in B and similarly for kinds.

Example 2.2

Let `zero`, `pair` *be term constants in* $\mathcal{C}$. *Let* `Eq`, `Pair`, `int` *be type constants in* $\mathcal{A}$. *Then* `zero` *and* `pair` $x\ y$ *are terms and* `Pair int int` *and* `Eq` x *are types.*

In order to state well-formedness of terms, types, and kinds we define signatures and contexts. We say that variable x is *bound* in a syntactic object O if there is a subterm $\lambda x : A.t$ of O. In order to avoid excessive technical details regarding renaming and freshness, we assume that constants and variable names are always unique. A variable that is not bound in a syntactic object is *free*. We define a function var$(-)$ that acts on syntactic objects and extracts the set of free variables. We say that a syntactic object is *ground* if it contains no free variables.

Definition 2.3 (Signatures and contexts)

Example 6.39

The greatest Herbrand model of the program $\mathcal{P}$ *in Example 6.22 is* $\mathcal{M}'_{\mathcal{P}} = \mathcal{M}_{\mathcal{P}} = \{A\ g, A\ (f\ g), A\ (f\ (f\ g)), \dots\}$. *Therefore, for an atomic formula* `A` x, $\mathcal{P} \vDash_{coind}$ `A` x. *However, it is impossible to construct a proof of*

$$\frac{\vdots}{\mathcal{P} \longrightarrow e : \mathtt{A}\ x}$$

The rules Lp-m *and* Lam *are not applicable for the same reasons as in the inductive case. The rule* Nu *results in assumption of the inference rule being the same as the conclusion since* `A` x *is an atom and not a Horn clause with a non-empty body and the proof state does not change.*

A similar argument can be carried out for the Example 6.23 by observing the inductive structure of a proof when we notice that the rule Nu does not instantiate the clause that is being proven. The notion of completeness for valid formulae fails similar to the inductive case.

Moreover, a more restricted formulation in the traditional style of Lloyd (1987) does not improve the situation:

Definition 6.40 (Coinductive Completeness à la Lloyd)

If a ground atomic formula G *is in* $\mathcal{M}'_{\mathcal{P}}$, *then* $\mathcal{P} \longrightarrow e : G$ *in the* Lp-m + Lam + Nu *proof system.*

Such a result does not hold, since there exist logic programs that define corecursive schemes that cannot be captured in this proof system. We demonstrate this on an example that was already used in literature (Fu et al., 2016) and we analyse its model.

Example 6.41

Let Σ *be a signature with a binary predicate symbol* D, *a unary function symbol* s *and a constant function symbol* z. *Consider a program* $\mathcal{P}$ *with the signature* Σ *given by the following axiom environment:*

$$\kappa_1 :\ \mathtt{D}\ x\ (\mathtt{s}\ y) \Rightarrow \mathtt{D}\ (\mathtt{s}\ x)\ y$$

$$\kappa_2 :\ \mathtt{D}\ (\mathtt{s}\ x)\ \mathtt{z} \Rightarrow \mathtt{D}\ \mathtt{z}\ x$$

$$\frac{\mathcal{P}, (\alpha : B_1 \wedge \cdots \wedge B_n \Rightarrow A) \longrightarrow e : B_1 \wedge \cdots \wedge B_n \Rightarrow A}{\mathcal{P} \longrightarrow \nu\alpha.e : B_1 \wedge \cdots \wedge B_n \Rightarrow A}\ \textsc{Nu}$$

for atomic formulae A, B_1 to B_n, a variable α and a proof term e in the guarded head normal form. W.l.o.g. let $e = \lambda\beta_1 \ldots \beta_n.\kappa\, e_1 \ldots e_m$. Therefore there is inference step of the form

$$\frac{\begin{array}{cccc}\mathcal{P}, (\beta_1 : \ \Rightarrow B_1), & & \mathcal{P}, (\beta_1 : \ \Rightarrow B_1), & \\ \ldots, (\beta_n : \ \Rightarrow B_n), & & \ldots, (\beta_n : \ \Rightarrow B_n), & \\ (\alpha : B_1 \wedge \cdots \wedge B_n \Rightarrow A) & & (\alpha : B_1 \wedge \cdots \wedge B_n \Rightarrow A) & \\ \longrightarrow e_1 : \sigma C_1' & \ldots & \longrightarrow e_m : \sigma C_m' & \kappa : C_1' \wedge \cdots \wedge C_m' \Rightarrow A'\end{array}}{\mathcal{P}, (\beta_1 : \ \Rightarrow B_1), \ldots, (\beta_n : \ \Rightarrow B_n), (\alpha : B_1 \wedge \cdots \wedge B_n \Rightarrow A) \longrightarrow \kappa\, e_1 \ldots e_n : \sigma A'}\ \textsc{Lp-m}$$

for $\sigma A' = A$. By the induction assumption, for all i, $\mathcal{P}, (\beta_1 : B_1), \ldots, (\beta_n : B_n), (\alpha : B_1 \wedge \cdots \wedge B_n \Rightarrow A) \vDash C_i$. By Lemma 6.35, $\mathcal{P} \vDash_{coind} B_1 \wedge \cdots \wedge B_n \Rightarrow A$. □

Note that we discussed the correspondence of the system Lp-m + Lam to the big step semantics of hereditary Harrop formulae. We also discussed that the system Lp-m + Nu' can be seen as the coinductive counterpart of the semantics of Horn clauses. In the same light we can see the system Lp-m + Lam + Nu' as a coinductive big-step operational semantics of the logic of hereditary Harrop formulae.

Coinductive incompleteness of system Lp-m + Lam + Nu

In Section 6.2, we considered two ways of stating inductive completeness of type class resolution. We state the corresponding result for the coinductive case here. As both the notions of completeness are shown not to hold we discuss them in the reversed order than the inductive completeness, first the more general case and then the more restricted one:

Definition 6.38 (Coinductive Completeness w.r.t. a model)
If $\mathcal{M}'_P \vDash_{coind} G$ then $\mathcal{S}; \mathcal{P} \longrightarrow e : G$ in the Lp-m + Lam + Nu *proof system.*

Recall programs in Examples 6.22 and 6.23 that we used to show incompleteness in the inductive case. We demonstrated that, in general, there are formulae that are valid in $\mathcal{M}_\mathcal{P}$ but do not have a proof in $\mathcal{P}$. The same two examples will serve our purpose here.

$$\begin{array}{lll}\text{Sig} \ni \mathcal{S} & := \cdot \mid \mathcal{S}, p : L \mid \mathcal{S}, c : A & \textit{signatures} \\ \text{Ctx} \ni \Gamma & := \cdot \mid \Gamma, x : A & \textit{contexts}\end{array}$$

Signatures assign types to term constants and kinds to type constants. Contexts assign types to variables. We use notation $\mathcal{S}_1, \mathcal{S}_2$ for a signature $\mathcal{S}_1, p_1 : L_1, \ldots, L_n$ where $\mathcal{S}_2 = p_1 : L_1, \ldots, L_n$ and similarly for contexts.

Example 2.4
Consider constants in Example 2.2. Then

$$\cdot, \texttt{int} : \texttt{type}, \texttt{zero} : \texttt{int}, \texttt{Pair} : \texttt{type}, \texttt{pair} : \texttt{int} \to \texttt{int} \to \texttt{type}, \texttt{eq} : \texttt{o}$$

is a signature. We denote this signature $\mathcal{S}_{Pair}$. Similarly,

$$\cdot, x : \texttt{int}, y : \texttt{int}$$

is a context.

When the signature is non-empty, *e.g.* $\mathcal{S} = \cdot, \texttt{int} : \texttt{type}$, we write $\mathcal{S} = \texttt{int} : \texttt{type}$. Similarly for contexts.

Substitution

Next we define substitution of a variable with a term.

Definition 2.5 (Substitution)

$$\texttt{type}[M/x] = \texttt{type}$$

$$\texttt{o}[M/x] = \texttt{o}$$

$$(\Pi y : A.L)[M/x] = \Pi y : A[M/x].L[M/x]$$

$$
\begin{aligned}
a[M/x] &= a \\
(\Pi y : A.B)[M/x] &= \Pi y : A[M/x].B[M/x] \\
(A\,N)[M/x] &= A[M/x]\,N[M/x] \\
c[M/x] &= c \\
y[M/x] &= M && \text{if } x = y \\
&= y && \text{otherwise} \\
(A\,N)[M/x] &= (A[M/x])\,(N[M/x]) \\
(\lambda x.A : N)[M/x] &= \lambda x : A[M/x].N[M/x]
\end{aligned}
$$

We define a *simultaneous substitution* on a set of distinct variables x_1 to x_n:

Definition 2.6

$$Subst \ni \sigma, \tau, \theta \qquad ::= \{M_1/x_1, \ldots, M_n/x_n\} \qquad \textit{simultaneous substitution}$$

We use σ, τ and θ to denote simultaneous substitutions. A simultaneous substitution $\{M_1/x_1, \ldots, M_n/x_n\}$ is called *ground* if all terms M_1, ..., M_n are ground. We refer to a simultaneous substitution as a substitution where there is no risk of confusion. Since we assume that all variable names are unique, application of simultaneous substitution to a term is a straightforward extension of Definition 2.5.

Definition 2.7

The application *of a simultaneous substitution* $\{M_1/x_1, \ldots M_n/x_n\}$ *to a term* N *or a type* A *is defined as substituting each variable* x_i *in* N *or* A *respectively with the term* M_i.

We denote application of a substitution σ to a term M or to a type A by σM and σA respectively. A substitution σ is called *grounding* for a term M if σM is a ground term, and similarly for a type. A substitution is grounding if it is grounding for

σA is valid in $\mathcal{M}'_{\mathcal{P}}$. The substitution σ is chosen arbitrary whence, for any σ, if, for all i, σB_i are valid in $\mathcal{P}$ then also σA is valid in $\mathcal{P}$. From the definition of validity it follows that $\mathcal{P} \vDash_{coind} B_1 \wedge \cdots \wedge B_n \Rightarrow A$. □

Now, the universal coinductive soundness of extended corecursive type class resolution follows straightforwardly:

Theorem 6.37

Let $\mathcal{P}$ *be a logic program, and let be* $\mathcal{S}; \mathcal{P} \longrightarrow e : G$ *for a formula* G *by the* Lp-m, Lam, *and* Nu *rules. Then* $\mathcal{P} \vDash_{coind} G$.

Proof. By structural induction on the derivation tree.

Base case: Let the derivation be in one step. Then it is by the rule Lp-m and of the form

$$\frac{(\kappa : \ \Rightarrow A) \in \mathcal{P}}{\mathcal{P} \longrightarrow \kappa : \sigma A}\ \text{Lp-m}$$

for an atomic formula A, a constant symbol κ, and a substitution σ. By Lemma 2.37 c), $\mathcal{P} \vDash_{coind} \sigma A$.

Inductive case, subcase Lp-m*:* Let the last step be by the rule Lp-m and of the form

$$\frac{\mathcal{P} \longrightarrow e_1 : \sigma B_1 \qquad \cdots \qquad \mathcal{P} \longrightarrow e_n : \sigma B_n \qquad (\kappa : B_1 \wedge \cdots \wedge B_n \Rightarrow A) \in \mathcal{P}}{\mathcal{P} \longrightarrow \kappa\, e_1 \ldots e_n : \sigma A}\ \text{Lp-m}$$

for an atomic formulae A, B_1, ..., B_n a constant symbol κ, a substitution σ and proof terms e_1, ..., e_n. By the induction assumption, for $i \in \{1, \ldots, n\}$, $\mathcal{P} \vDash_{coind} B_i$ and by Lemma 2.37 d), $\mathcal{P} \vDash_{coind} \sigma A$.

Subcase Lam*:* Let the last step of the derivation be by the rule Lam. Then it is of the form

$$\frac{\mathcal{P}, (\beta_1 : \ \Rightarrow B_1), \ldots, (\beta_n : \ \Rightarrow B_n) \longrightarrow e : A}{\mathcal{P} \longrightarrow \lambda\beta_1, \ldots, \beta_n.e : B_1 \wedge \cdots \wedge B_n \Rightarrow A}\ \text{Lam}$$

for atomic formulae A, B_1 to B_n, a proof term e, and variables b_1, ..., b_n. By the induction assumption, $\mathcal{P}, (\beta_1 : \ \Rightarrow B_1), \ldots, (\beta_n : \ \Rightarrow B_n) \vDash_{coind} A$ and by Lemma 6.36 also $\mathcal{P} \vDash_{coind} B_1 \wedge \cdots \wedge B_n \Rightarrow A$.

Subcase Nu*:* Let the last step be by the rule Nu and of the form

Coinductive soundness of system LP-M + LAM + NU

Before proceeding with the proof of soundness of extended type class resolution we need to show two intermediate lemmata. The first lemma states that inference by the NU rule preserves coinductive soundness:

Lemma 6.35

Let $\mathcal{P}$ be a logic program, let σ be a substitution, and let A, B_1, …, B_n, C_1, …, C_m be atomic formulae. If, for all i, $\mathcal{P}, B_1, \ldots, B_n, (B_1 \wedge \cdots \wedge B_n \Rightarrow \sigma A) \vDash_{coind} \sigma C_i$ and $(C_1, \ldots, C_m \Rightarrow A) \in P$ then $\mathcal{P} \vDash_{coind} B_1 \wedge \cdots \wedge B_n \Rightarrow \sigma A$.

Proof. Consider the construction of the greatest Herbrand model of the program $\mathcal{P}$ and proceed by induction with hypothesis: for all n, $B_1 \wedge \cdots \wedge B_n \Rightarrow \sigma A$ is valid in $\mathcal{T}_\mathcal{P} \downarrow n$. The base case is the same as in the proof of Lemma 6.29.

Assume that, for a grounding substitution τ, for all i, $\tau B_i \in \mathcal{T}_\mathcal{P} \downarrow n$. Then also $(\tau \circ \sigma)A \in \mathcal{T}_\mathcal{P} \downarrow n$. For the definition of the semantic operator, it follows from the monotonicity of the operator itself, and from the assumptions made by the lemma that $(\tau \circ \sigma)C_i \in \mathcal{T}_\mathcal{P} \downarrow n$. Since $C_1, \ldots, C_n \Rightarrow A \in P$ also $(\tau \circ \sigma)A \in \mathcal{T}_\mathcal{P} \downarrow (n+1)$. If the assumption does not hold then from the monotonicity of $\mathcal{T}_\mathcal{P}$ it follows that, for all i, $\tau B_i \notin \mathcal{T}_\mathcal{P} \downarrow (n+1)$. Therefore, $B_1 \wedge \cdots \wedge B_n \Rightarrow \sigma A$ is valid in $\mathcal{T}_\mathcal{P} \downarrow (n+1)$. From induction we conclude that the same holds for $\mathcal{T}_\mathcal{P} \downarrow \omega$ and from Proposition 2.35 for $\mathcal{M}'_\mathcal{P}$. Hence whenever, for a substitution τ, all instances of τB_1 to τB_n are in the greatest Herbrand model then also all instances of $(\tau \circ \sigma)A$ are in the greatest Herbrand model. Hence $\mathcal{P} \vDash_{coind} B_1 \wedge \cdots \wedge B_n \Rightarrow \sigma A$. □

The other lemma that we need in order to prove coinductive soundness of extended type class resolution states that inference using LAM preserves coinductive soundness, *i.e.* we need to show the coinductive counterpart to Lemma 6.18:

Lemma 6.36

Let $\mathcal{P}$ be a logic program and A, B_1, …, B_n atomic formulae. If $\mathcal{P}, (\ \Rightarrow B_1), \ldots (\ \Rightarrow B_n) \vDash_{coind} A$ then $\mathcal{P} \vDash_{coind} B_1 \wedge \cdots \wedge B_n \Rightarrow A$.

Proof. Assume that, for an arbitrary substitution σ, for all i, σB_i is valid in $\mathcal{M}'_\mathcal{P}$. Then, for any grounding substitution τ, from the definition of the semantic operator and from the assumption of the lemma it follows that $(\tau \circ \sigma)A \in \mathcal{M}'_\mathcal{P}$. Therefore,

any term. A simultaneous substitution $\{M_1/x_1, \ldots, M_n/x_n\}$, as a syntactic object, gives rise to a (partial) mapping that, for each i, assigns M_i to x_i. We will use the substitution and the assignment interchangeably.

Definition 2.8

A composition *of a substitution $\sigma = \{M_1/x_1, \ldots M_n/x_n\}$ with a substitution $\tau = \{N_1/y_1, \ldots, N_m/y_m\}$ is defined as*

$$\{M_1/x_1, \ldots M_n/x_n, \sigma N_1/y_1, \ldots, \sigma N_m/y_m\}$$

Note that the usual condition on variables x_1 to x_n being distinct is subsumed by our implicit assumption of uniqueness of variable names. We denote composition of substitutions σ and τ by $\sigma \circ \tau$. Composition of substitutions is clearly a substitution.

Example 2.9

Consider constants in Example 2.2. Then $\sigma = \{\mathtt{zero}/x, \mathtt{pair}\, z\, z/y\}$ and $\tau = \{\mathtt{zero}/z\}$ are substitutions. The term $\sigma(\mathtt{pair}\, x\, y) = \mathtt{pair}\, \mathtt{zero}\, (\mathtt{pair}\, z\, z)$ is application of the substitution σ to the term $\mathtt{pair}\, x\, y$. The composition of substitutions τ and σ is the substitution $\tau \circ \sigma = \{\mathtt{zero}/z, \mathtt{zero}/x, \mathtt{pair}\, \mathtt{zero}\, \mathtt{zero}/y\}$. The substitution $\tau \circ \sigma$ is grounding for the term $\mathtt{pair}\, x\, y$ since $\sigma \circ \tau(\mathtt{pair}\, x\, y) = \mathtt{pair}\, \mathtt{zero}\, (\mathtt{pair}\, \mathtt{zero}\, \mathtt{zero})$.

2.1.2 Typing and equality

Well-formedness of syntactic objects is given by the following judgements:

- $\vdash \mathcal{S}$, for $\mathcal{S}$ a well-formed signature,
- $\mathcal{S} \vdash \Gamma$, for Γ a well-formed context in a signature $\mathcal{S}$,
- $\mathcal{S}, \Gamma \vdash L : \mathtt{kind}$, for L a well-formed kind in a signature $\mathcal{S}$ and a context Γ,
- $\mathcal{S}, \Gamma \vdash A : L$, for A a well-formed type of a kind L in a signature $\mathcal{S}$ and a context Γ, and
- $\mathcal{S}, \Gamma \vdash M : A$, for M a well-formed term of a type A in a signature $\mathcal{S}$ and a context Γ.

Definition 2.10

The well-formedness judgements for signatures and contexts are given in Figure 2.1.

$\boxed{\vdash \mathcal{S}}$

$$\frac{}{\vdash \cdot} \qquad \frac{\vdash \mathcal{S} \qquad \mathcal{S}; \cdot \vdash L : \mathtt{kind}}{\vdash \mathcal{S}, a : L} \qquad \frac{\vdash \mathcal{S} \qquad \mathcal{S}; \cdot \vdash A : \mathtt{type}}{\vdash \mathcal{S}, c : A}$$

$\boxed{\mathcal{S} \vdash \Gamma}$

$$\frac{}{\mathcal{S} \vdash \cdot} \qquad \frac{\mathcal{S} \vdash \Gamma \qquad \mathcal{S}; \Gamma \vdash x : \mathtt{type}}{\mathcal{S} \vdash \Gamma, x : \mathtt{type}}$$

Figure 2.1: Well-formedness of signatures and contexts

$\boxed{\mathcal{S}; \Gamma \vdash M : A}$

$$\frac{c : A \in \mathcal{S} \qquad \mathcal{S} \vdash \Gamma}{\mathcal{S}; \Gamma \vdash c : A}$$

$$\frac{x : A \in \Gamma \qquad \mathcal{S} \vdash \Gamma}{\mathcal{S}; \Gamma \vdash x : A}$$

$$\frac{\mathcal{S}; \Gamma \vdash M : \Pi x : A.B \qquad \mathcal{S}; \Gamma \vdash N : A}{\mathcal{S}; \Gamma \vdash M\, N : B[M/x]}$$

$$\frac{\mathcal{S}; \Gamma \vdash A : \mathtt{type} \qquad \Gamma, x : A \vdash M : B}{\mathcal{S}; \Gamma \vdash \lambda x : A.M : \Pi x : A.B}$$

Figure 2.2: Well-formedness of terms

$\boxed{\mathcal{S}; \Gamma \vdash A : L}$

$$\frac{p : L \in \mathcal{S} \qquad \mathcal{S} \vdash \Gamma}{\mathcal{S}; \Gamma \vdash c : L}$$

$$\frac{\mathcal{S}; \Gamma \vdash A : \mathtt{type} \qquad \mathcal{S}; \Gamma, A \vdash B : L}{\mathcal{S}; \Gamma \vdash \Pi x : A.B : \Pi x : A.L}$$

$$\frac{\mathcal{S}; \Gamma \vdash M : \Pi x : A.B \qquad \mathcal{S}; \Gamma \vdash N : A}{\mathcal{S}; \Gamma \vdash MN : B[M/x]}$$

Figure 2.3: Well-formedness of types

$\boxed{\mathcal{S}; \Gamma \vdash A : L}$

$$\frac{\mathcal{S} \vdash \Gamma}{\mathcal{S}; \Gamma \vdash \mathtt{type} : \mathtt{kind}}$$

$$\frac{\mathcal{S} \vdash \Gamma}{\mathcal{S}; \Gamma \vdash \mathtt{o} : \mathtt{kind}}$$

$$\frac{\mathcal{S}; \Gamma \vdash A : \mathtt{type} \qquad \mathcal{S}; \Gamma, A \vdash L : \mathtt{kind}}{\mathcal{S}; \Gamma \vdash \Pi x : A.L : \mathtt{kind}}$$

Figure 2.4: Well-formedness of kinds

The side condition of the Nu rule requires the proof term to be in guarded head normal form. However, unlike corecursive type class resolution, extended corecursive type class resolution also uses the Lam rule and a guarded head normal term is not restricted as in the previous section but is in a general form $\lambda\underline{\alpha}.\kappa\,\underline{e}$ for a possibly non-empty sequence of proof-term variables $\underline{\alpha}$. First, let us note that extended corecursive type class resolution indeed extends the calculus of Section 6.3:

Proposition 6.33

The inference rule Nu' *is admissible in the extended corecursive type class resolution.*

Proof. Let $\mathcal{P}$ be a program, let A be an atomic formula and let $\mathcal{S}; \mathcal{P}, (\alpha : \ \Rightarrow A) \longrightarrow e : A$ where e is in gHNF. Then by the Lam rule $\mathcal{S}; \mathcal{P}, (\alpha : \ \Rightarrow A) \longrightarrow \lambda\underline{\beta}.e : A$ where $\underline{\beta}$ is an empty sequence of variables. Therefore $\mathcal{S}; \mathcal{P}, (\alpha : \ \Rightarrow A) \longrightarrow e : \ \Rightarrow A$. Since e is in guarded head normal form by the Nu rule also $\mathcal{S}; \mathcal{P} \longrightarrow \nu\alpha.e : A$. □

Furthermore, this is a proper extension. The Nu rule allows queries to be entailed that were beyond the scope of corecursive type class resolution.

Example 6.34

Recall Example 6.8 where no cycles arise for query $\mathtt{eq}(\mathtt{bush}(\mathtt{int}))$ *and thus the query cannot be resolved by corecursive type class resolution. Using the extended query the calculus is resolved as follows:*

$$\dfrac{\dfrac{\kappa_{\mathtt{int}} : \mathtt{eq}\,\mathtt{int} \in \mathcal{P}_{Bush}}{\mathcal{P}_{Bush} \longrightarrow \kappa_{\mathtt{int}} : \mathtt{eq}\,\mathtt{int}} \qquad \dfrac{\dfrac{\dfrac{\dfrac{(\beta : \ \Rightarrow \mathtt{eq}\,x) \in \mathcal{P}_{Bush,\alpha,\beta}}{\mathcal{P}_{Bush,\alpha,\beta} \longrightarrow \beta : \mathtt{eq}\,x} \qquad \dfrac{\alpha : \mathtt{eq}\,x \Rightarrow \mathtt{eq}\,(\mathtt{bush}\,x) \in \mathcal{P}_{\mathtt{bush},\alpha,\beta}}{\mathcal{P}_{Bush,\alpha,\beta} \longrightarrow \alpha\,\beta : \mathtt{eq}\,(\mathtt{bush}\,x)}}{\mathcal{P}_{Bush,\alpha,\beta} \longrightarrow \alpha\,(\alpha\,\beta) : \mathtt{eq}\,(\mathtt{bush}\,(\mathtt{bush}\,x))} \qquad \dfrac{\dfrac{(\beta : \ \Rightarrow \mathtt{eq}\,x) \in \mathcal{P}_{Bush,\alpha,\beta}}{\mathcal{P}_{Bush,\alpha,\beta} \longrightarrow \beta : \mathtt{eq}\,x} \qquad \kappa_{\mathtt{bush}} : \mathtt{eq}\,x \wedge \mathtt{eq}\,(\mathtt{bush}\,(\mathtt{bush}\,x)) \Rightarrow \mathtt{eq}\,(\mathtt{bush}\,x) \in \mathcal{P}_{Bush,\alpha,\beta}}{\mathcal{P}_{Bush,\alpha,\beta} \longrightarrow \kappa_2\,\beta\,(\alpha\,(\alpha\,\beta)) : \mathtt{eq}\,(\mathtt{bush}\,x)}\,\textsc{Lam}}{\mathcal{P}_{Bush,\alpha} \longrightarrow \lambda\beta.\kappa_{\mathtt{bush}}\beta(\alpha(\alpha\beta)) : \mathtt{eq}\,x \Rightarrow \mathtt{eq}\,(\mathtt{bush}\,x)}\,\textsc{Lam}}{\mathcal{P}_{Bush} \longrightarrow \nu\alpha.\lambda\beta.\kappa_{\mathtt{bush}}\beta(\alpha(\alpha\beta)) : \mathtt{eq}\,x \Rightarrow \mathtt{eq}\,(\mathtt{bush}\,x)}\,\textsc{Nu}}{\mathcal{P}_{Bush} \longrightarrow (\nu\alpha.\lambda\beta.\kappa_{\mathtt{bush}}\beta(\alpha(\alpha\beta)))\kappa_{\mathtt{int}} : \mathtt{eq}\,(\mathtt{bush}\,\mathtt{int})}$$

In the derivation, we use $\mathcal{P}_\beta$ to abbreviate the program $\mathcal{P}$ extended with the clause $\beta : \ \Rightarrow \mathtt{eq}\,x \wedge$ and $\mathcal{P}_\alpha$ to abbreviate the program $\mathcal{P}$ extended with the clause $\alpha : \mathtt{eq}\,x \wedge \Rightarrow \mathtt{eq}\,(\mathtt{bush}\,x)$.

Example 6.31

Consider a program $\mathcal{P}_{Inf}$ given by a single clause $\kappa_{inf} : \mathtt{p}\, x \Rightarrow \mathtt{p}\,(\mathtt{f}\, x)$. The greatest Herbrand model of that program is empty, i.e. *$\mathcal{M}_{\mathcal{P}_{Inf}} = \emptyset$. However, its greatest complete Herbrand model $\mathcal{M}'_{\mathcal{P}_{Inf}} = \{\mathtt{p}\,(\mathtt{f}\,(\mathtt{f}\,(...))\}$ contains the infinite formula $\mathtt{p}\,(\mathtt{f}\,(\mathtt{f}\,(...))$.*

Restrictions of Definition 6.3, imposed by type class resolution, mean that the greatest Herbrand models regain those same advantages as complete Herbrand models. It was noticed by Lloyd (1987) that restriction 2 implies that the semantic operator converges in at most ω steps. Restriction 1 and and the resolution by matching imply that proofs by type class resolution have a universal interpretation, *i.e.* that they hold for all finite instances of goals. Therefore, we never need to talk about programs for which only one infinite instance of a goal is valid. To cohere with the fact that the discussed restrictions are distinguishing features of type class resolution, we prove all our soundness results relative to greatest Herbrand models. Extensions to complete Herbrand models hold trivially and we omit their explicit formulation.

6.4 Extended Coinductive Type Class Resolution

The class of problems that can be resolved by coinductive type class resolution is limited to problems where a coinductive hypothesis is in atomic form. Fu et al. (2016) extended coinductive type class resolution with implicative reasoning and adjusted the rule Nu' such that this restriction of coinductive type class resolution is relaxed.

6.4.1 Proof system Lp-m + Lam + Nu

Definition 6.32 (Extended corecursive type class resolution)

Let $\mathcal{P}$ be a program, A, B_1 to B_n atoms, e a proof term, and α a proof-term variable. The calculus of extended corecursive type class resolution *consists of the inference rules* Lp-m, Lam *and the following inference rule:*

$$\textit{if } \mathrm{gHNF}(e)\ \frac{\mathcal{P}, (\alpha : B_1 \wedge \cdots \wedge B_n \Rightarrow A) \longrightarrow e : B_1 \wedge \cdots \wedge B_n \Rightarrow A}{\mathcal{P} \longrightarrow \nu\alpha.e : B_1 \wedge \cdots \wedge B_n \Rightarrow A} \qquad (\textsc{Nu})$$

Well-formedness of terms, types and kinds is given in Figures 2.2, 2.3, and 2.4.

We will also consider well-formedness of simultaneous substitutions that preserves well-formedness of objects under substitution:

Definition 2.11

A simultaneous substitution $\{M_1/x_1, \cdots, M_n/x_n\}$ is well-formed in a a signature $\mathcal{S}$ and a context $\Gamma = x_1 : A_1, \ldots, x_n : A_n$ and of shape Γ', for a context Γ', if, for each i, $\mathcal{S}; \Gamma \vdash M_i : A_i$.

We use $\mathcal{S}; \Gamma \vdash \sigma : \Gamma'$ to denote that σ is a well-formed substitution in a signature $\mathcal{S}$ and a context Γ of shape Γ'.

Example 2.12

Consider the signature $\mathcal{S}_{Pair}$ in Example 2.4. It is easy to show that $\cdot; \mathtt{int} : \mathtt{type}, \mathtt{zero} : \mathtt{int}, \mathtt{Pair} : \mathtt{type} \vdash \mathtt{pair} : \mathtt{int} \to \mathtt{int} \to \mathtt{type}$. Hence the signature $\mathcal{S}_{Pair}$ is well-formed. Similarly, the term $\mathtt{pair}\, x\, y$ is well-formed in signature $\mathcal{S}_{Pair}$ and context $x : \mathtt{int}, y : \mathtt{int}$.

Further, there is a notion of definitional equality of terms, types and kinds. The equality is given by the following judgements:

- $\mathcal{S}; \Gamma \vdash L \equiv L' : \mathtt{kind}$ for a kind L equal to a kind L',
- $\mathcal{S}; \Gamma \vdash A \equiv B : L$ for a type A equal to a type B at a kind L, and
- $\mathcal{S}; \Gamma \vdash M \equiv N : A$ for a term A equal to a term N at a type B.

The notion of equality we consider is the $\beta\eta$-conversion. Since this notion of equality is standard in literature, we no not provide a definition of the appropriate judgements (*cf.* Harper and Pfenning, 2005).

We state some metatheoretic properties of the calculus that are used in the rest of this chapter. Proofs of these properties for standard LF can be found in the literature (*cf.* Harper and Pfenning (2005)). Due to the large number of well-formedness judgements of LF and due to the fact that these judgements are mutually defined, proofs of the following properties are rather large and require a substantial development of an apparatus of auxiliary lemmata. Our language differs only in presence of an additional sort `o` that for the purpose of meta-theoretical properties bellow behaves like the sort `type` and does not change the nature of the proofs. Therefore, we omit the proofs here as these can be easily recovered from the corresponding

proofs for LF.

Theorem 2.13

1. *(Unicity of Types) If* $\mathcal{S};\Gamma \vdash M : A_1$ *and* $\mathcal{S};\Gamma \vdash M : A_2$ *then* $\mathcal{S};\Gamma \vdash A_1 \equiv A_2 : L$.
2. *(Substitutivity) If* $\mathcal{S};\Gamma, x : A \vdash \mathcal{I}$ *and* $\mathcal{S};\Gamma \vdash M : A$ *then* $\mathcal{S};\Gamma \vdash \mathcal{I}[M/x]$ *where* $\mathcal{I}$ *is any right side of a judgement that admits substitution.*

Proposition 2.14

1. *If* $\mathcal{S}_1, \mathcal{S}_2;\Gamma \vdash M : B$ *and* $\vdash \mathcal{S}_1, c : A, \mathcal{S}_2$ *then* $\mathcal{S}_1, c : A, \mathcal{S}_2;\Gamma \vdash M : B$.
2. *If* $\mathcal{S};\Gamma_1, \Gamma_2 \vdash M : B$ *and* $\mathcal{S} \vdash \Gamma_1, x : A, \Gamma_2$ *then* $\mathcal{S};\Gamma_1, x : A, \Gamma_2 \vdash M : B$.

Proposition 2.15

1. *If* $\mathcal{S};\Gamma \vdash A : L$ *and* $x \notin \Gamma$ *then* $\mathcal{S};\Gamma \vdash A[M/x] : L$.

Judgements of LF, and consecutively of our language, admit several properties that are generally referred to as *implicit syntactic validity*. For the purpose of this thesis, we require the following theorem:

Theorem 2.16 (Implicit syntactic validity)

- *If* $\mathcal{S} \vdash \Gamma$ *then* $\vdash \mathcal{S}$, *and*
- *if* $\mathcal{S};\Gamma \vdash A \equiv B : L$ *then* $\mathcal{S} \vdash \Gamma$.

Let us note that we set up well-formedness in such a way that we can recover notions familiar from (typed) logic programming. First, type constants in a signature that are of kind $\Pi x_1 : A_1.(\cdots(\Pi x_n : A_n.\mathtt{o})\cdots)$ where each A_i is of kind $\mathtt{type}$ can be regarded as predicates. Similarly, term constants in the signature can be regarded as function symbols. *Atomic formulae*, or atoms then are the expressions in the syntactic class of types that are well-formed and of kind $\mathtt{o}$. This intuition is formalised using the following lemma:

Lemma 2.17

If $\mathcal{S};\Gamma \vdash A : (\Pi x_1 : A_1 \ldots (\Pi x_n : A_n.\mathtt{o})\ldots)$ *then* A *is equal to* $((c\, N_{n+1})\ldots N_m)$

for an atomic formula A, a constant symbol κ, and a substitution σ. By Lemma 2.37 c), $\mathcal{P} \vDash_{coind} \sigma A$.

Inductive case, subcase LP-M: Let the last step be by the rule LP-M and of the form

$$\frac{\mathcal{P} \longrightarrow e_1 : \sigma B_1 \quad \cdots \quad \mathcal{P} \longrightarrow e_n : \sigma B_n \quad (\kappa : B_1 \wedge \cdots \wedge B_n \Rightarrow A) \in \mathcal{P}}{\mathcal{S};\mathcal{P} \longrightarrow \kappa\, e_1 \ldots e_n : \sigma A}$$

for an atomic formulae A, B_1, to B_n, a constant symbol κ, a substitution σ and proof term e_1, ..., e_n. By the induction assumption, for $i \in \{1, \ldots, n\}$, $\mathcal{P} \vDash_{coind} B_i$ and by Lemma 2.37 d), $\mathcal{P} \vDash_{coind} \sigma A$.

Subcase NU': Let the last step be by the rule NU' and of the form

$$\frac{\mathcal{P}, (\alpha : \ \Rightarrow A) \longrightarrow e : A}{\mathcal{P} \longrightarrow \nu\alpha.e : A}\ \text{NU'}$$

for an atomic formula A, a proof-term variable α and a proof term e in the guarded head normal form. W.l.o.g. let $e = \kappa e_1 \ldots e_n$. Therefore there is an inference step of the form

$$\frac{\mathcal{P} \longrightarrow e_1 : \sigma B'_1 \quad \ldots \quad \mathcal{P} \longrightarrow e_n : \sigma B'_n \quad (\kappa : B'_1 \wedge \cdots \wedge B'_n \Rightarrow A') \in P}{\mathcal{P} \longrightarrow \kappa\, e_1 \ldots e_n : \sigma A'}$$

for $\sigma A' = A$. By the induction assumption, for all i, $\mathcal{P}, (\alpha : \ \Rightarrow A) \vDash B_i$. By Lemma 6.29, $\mathcal{P} \vDash_{coind} A$. □

6.3.2 Choice of coinductive models

Perhaps the most unusual feature of the semantics given in this chapter is the use of the greatest Herbrand models rather than the greatest *complete* Herbrand models. The latter is more common in the literature on coinduction in logic programming (Johann et al., 2015, Lloyd, 1987, Simon et al., 2007). *The greatest complete Herbrand models* are obtained as the greatest fixed point of the semantic operator $\mathcal{T}'_{\mathcal{P}}$ on the *complete Herbrand base*, *i.e.* the set of all finite and *infinite* ground atomic formulae formed by the signature of the given program. This construction is preferred in the literature for two reasons. First, $\mathcal{T}'_{\mathcal{P}}$ reaches its greatest fixed point in at most ω steps, whereas $\mathcal{T}_{\mathcal{P}}$ may take more than ω steps in the general case. This is due to compactness of the complete Herbrand base. Moreover, greatest complete Herbrand models give a more natural characterisation for some programs.

operator $\mathcal{T}_\mathcal{P}$. In order for induction to be applicable in our proof, we must ensure that the construction of the greatest Herbrand model is completed within ω steps of iteration of $\mathcal{T}_\mathcal{P}$. This holds by virtue of Proposition 2.35 since we consider only Horn clauses without existential variables. The essence of the coinductive soundness of NU' is captured by the following lemma:

Lemma 6.29

Let $\mathcal{P}$ be a logic program, let σ be a substitution, and let A, B_1, …, B_n be atomic formulae. If, $\forall i \in \{1, \ldots, n\}$, $\mathcal{P}, (\ \Rightarrow \sigma A) \vDash_{coind} \sigma B_i$ and $(B_1 \wedge \cdots \wedge B_n \Rightarrow A) \in \mathcal{P}$ then $\mathcal{P} \vDash_{coind} \sigma A$.

Proof. Consider construction of the greatest Herbrand model for the program $\mathcal{P}$ and proceed by induction with hypothesis: for all n, for any grounding substitution τ, $(\tau \circ \sigma)A \in \mathcal{T}_\mathcal{P} \downarrow n$. By Definition of $\mathcal{T}_\mathcal{P}$, $\mathcal{T}_\mathcal{P} \downarrow 0$ is the Herbrand base $\mathbf{B}_\Sigma$ and, for any grounding τ, $(\tau \circ \sigma)A \in \mathbf{B}_\Sigma$.

Assume that, for any grounding τ, $(\tau \circ \sigma)A \in \mathcal{T}_\mathcal{P} \downarrow n$. The set $\mathcal{T}_\mathcal{P} \downarrow n$ is by definition of the operator $\mathcal{T}_\mathcal{P}$ the same as the set $\mathcal{T}_{\mathcal{P},(\ \Rightarrow \sigma A)}$ and from the assumptions of the lemma and monotonicity of $\mathcal{T}_\mathcal{P}$ also, for all i, for any grounding substitution τ, $(\tau \circ \sigma)B_i \in \mathcal{T}_\mathcal{P} \downarrow n$. Since $B_1 \wedge \cdots \wedge B_n \Rightarrow A \in \mathcal{P}$ also $(\tau \circ \sigma)A \in \mathcal{T}_\mathcal{P} \downarrow (n+1)$. From induction follows that the same will be true for all subsequent iterations of $\mathcal{T}_\mathcal{P}$ and all instances of σA will be in $\mathcal{T}_\mathcal{P} \downarrow \omega$ and, by Proposition 2.35 in $\mathcal{M}'_\mathcal{P}$. Hence $\mathcal{P} \vDash_{coind} \sigma A$ □

Finally, Theorem 6.30 states universal coinductive soundness of the corecursive type class resolution:

Theorem 6.30

Let $\mathcal{P}$ be a logic program and G a formula. Let there be a derivation of $\mathcal{S}; \mathcal{P} \longrightarrow e : G$ by the rules LP-M *and* NU'. *Then $\mathcal{P} \vDash_{coind} G$.*

Proof. By structural induction on the derivation tree.

Base case: Let the derivation be in one step. Then it is by the rule LP-M and of the form

$$\frac{(\kappa :\ \Rightarrow A) \in \mathcal{P}}{\mathcal{P} \longrightarrow \kappa : \sigma A}\ \text{LP-M}$$

and c is of a kind $(\Pi A_1. \ldots (\Pi A_m.\, \mathtt{o}) \ldots)$.

Proof. By induction on the derivation of the judgement.

- Let the derivation be $\dfrac{a : L \in \mathcal{S} \qquad \mathcal{S} \vdash \Gamma}{\mathcal{S}; \Gamma \vdash a : L}$. Then the lemma holds trivially.
- Let the derivation be $\dfrac{\mathcal{S}; \Gamma \vdash A : (\Pi x_{i+1} : A_{i+1}.L) \qquad \mathcal{S}; \Gamma \vdash N_{i+1} : A_{i+1}}{\mathcal{S}; \Gamma \vdash A\, N_{i+1} : L[N_{i+1}/x_{i+1}]}$. From the induction assumption A is equal to $((c\, N_{n+1}) \ldots N_i)$ and c is of a kind $(\Pi x_1 : A_1. \ldots (\Pi x_{i+1} : A_{i+1}.(\ldots \mathtt{o})) \ldots)$. Hence $A\, N_{i+1}$ is equal to $((c\, N_{n+1}) \ldots N_i)\, N_{i+1}$ and c is of the required kind.

□

A well-formed type of kind $\mathtt{o}$ then corresponds to the intuitive understanding of an atomic formula, that is a predicate symbol that is applied to a number of terms.

Corollary 2.18

If $\mathcal{S}; \Gamma \vdash A : \mathtt{o}$ then A is of shape $((p\, N_1) \ldots N_n)$ and p is of kind $(\Pi x_1 : A_1. \ldots (\Pi x_n : A_n.\, \mathtt{o}) \ldots)$.

2.2 Horn-Clause Logic

We now move on to definition of expressions that constitute valid programs and goals. The syntactic classes of *clauses* and *goals* are mutually defined as follows:

Definition 2.19 (Syntax of goals and clauses)

$\mathcal{D} \ni D$	$:= A \mid A \Rightarrow D \mid \forall X : A.D$	*clauses*
$\mathcal{G} \ni G$	$:= A \mid \exists X : A.G$	*goals*

The clauses in $\mathcal{D}$ are denoted by identifier D and consist of atomic formulae, implication $\Rightarrow$, and universal quantification over a clause. The goals in $\mathcal{G}$ are denoted by identifier G and consist of atomic formulae and existential quantification. Implication and quantification have the usual meaning. For a clause, an *existential variable* is a variable that does not occur in the right-most atomic formula of the

$\boxed{\mathcal{S};\Gamma \vdash D : \mathtt{o}}$

$$\frac{\mathcal{S};\Gamma \vdash A : \mathtt{o} \qquad \mathcal{S};\Gamma \vdash D : \mathtt{o}}{\mathcal{S};\Gamma \vdash A \Rightarrow D : \mathtt{o}} \qquad \frac{\mathcal{S};\Gamma, X : A \vdash D : \mathtt{o}}{\mathcal{S};\Gamma \vdash \forall X : A.D : \mathtt{o}}$$

$\boxed{\mathcal{S};\Gamma \vdash G : \mathtt{o}}$

$$\frac{\mathcal{S};\Gamma, X : A \vdash M : \mathtt{o}}{\mathcal{S};\Gamma \vdash \exists X : A.M : \mathtt{o}}$$

Figure 2.5: Well formedness of clauses and goals

clause. We use notation $G \Leftarrow D$ for a Horn clause $D \Rightarrow G$ where such notation facilitates reading of the clause or a logic program containing such clauses.

Example 2.20

Consider constants in Example 2.2. Then $\forall x : \mathtt{int}.\forall y : \mathtt{int}.\mathtt{eq}\, x \Rightarrow \mathtt{eq}\, y \Rightarrow \mathtt{eq}\,(\mathtt{pair}\, x\, y)$ *and* $\mathtt{eq}\,\mathtt{zero}$ *are Horn clauses.*

To ensure that clauses and goals indeed consist of atomic formulae in positions of types we introduce further well-formedness judgements:

- $\mathcal{S};\Gamma \vdash D : \mathtt{o}$, for D a well-formed clause in signature $\mathcal{S}$ and context Γ, and
- $\mathcal{S};\Gamma \vdash G : \mathtt{o}$, for G a well-formed goal in signature $\mathcal{S}$ and context Γ.

These are intended to be read as extension of well-formedness of types and terms to formulae. The judgements are given in Figure 2.5.

Definition 2.21 (Well formed clauses and goals)

A clause D is well formed in $\mathcal{S}$ if $\mathcal{S};\cdot \vdash D : \mathtt{o}$ can be derived. A goal G is well formed in $\mathcal{S}$ if $\mathcal{S};\cdot \vdash G : \mathtt{o}$ can be derived.

Our choice of syntax of Horn-clause logic is one of several possible definitions. Our motivation for choosing this definition is to minimise the number of logical connectives without compromising expressivity of the system. Thus we omit logical conjunctions and disjunctions. Reducing the number of logical connectives simplifies our exposition of its semantics and reduces the number of cases that are necessary to consider in the proof of its soundness. However, it is convenient to allow at least logical conjunctions in goals and Horn clauses to simplify presentation in the rest of this thesis. Different program transformation methods that preserve logical equivalence and their impact on size of programs and derivations are studied in

$$\dfrac{\dfrac{\kappa_{\mathtt{int}} : \mathtt{eq}\,\mathtt{int} \in \mathcal{P}_{EvenOdd}}{\mathcal{P}_{EvenOdd} \longrightarrow \kappa_{\mathtt{int}} : \mathtt{eq}\,\mathtt{int}} \qquad \dfrac{\dfrac{\kappa_{\mathtt{int}} : \mathtt{eq}\,\mathtt{int} \in \mathcal{P}_{EvenOdd}}{\mathcal{P}_{EvenOdd} \longrightarrow \kappa_{\mathtt{int}} : \mathtt{eq}\,\mathtt{int}} \qquad \dfrac{\alpha : \ \Rightarrow \mathtt{eq}\,(\mathtt{evenList}\,\mathtt{int}) \in \mathcal{P}_{EvenOdd}, \ \alpha : \ \Rightarrow \mathtt{eq}\,(\mathtt{evenList}\,\mathtt{int})}{\mathcal{P}_{EvenOdd}, \ \alpha : \ \Rightarrow \mathtt{eq}\,(\mathtt{evenList}\,\mathtt{int}) \longrightarrow \alpha : \mathtt{eq}\,(\mathtt{evenList}\,\mathtt{int})}}{\mathcal{P}_{EvenOdd}, \alpha : \ \Rightarrow \mathtt{eq}\,(\mathtt{evenList}\,\mathtt{int}) \longrightarrow \kappa_{\mathtt{oddList}}\kappa_{\mathtt{int}}\alpha : \mathtt{eq}\,(\mathtt{oddList}\,\mathtt{int})}}{\mathcal{P}_{EvenOdd}, \alpha : \ \Rightarrow \mathtt{eq}\,(\mathtt{evenList}\,\mathtt{int}) \longrightarrow \kappa_{\mathtt{evenList}}\kappa_{\mathtt{int}}(\kappa_{\mathtt{oddList}}\kappa_{\mathtt{int}}\alpha) : \mathtt{eq}\,(\mathtt{evenList}\,\mathtt{int})}}{\mathcal{P}_{EvenOdd} \longrightarrow \nu\alpha.\kappa_{\mathtt{evenList}}\kappa_{\mathtt{int}}(\kappa_{\mathtt{oddList}}\kappa_{\mathtt{int}}\alpha) : \mathtt{eq}\,(\mathtt{evenList}\,\mathtt{int})}\ \textsc{Nu'}$$

Recall that when the index is omitted the inference proceeds by the Lp-m *rule.*

Coinductive soundness of system Lp-m + Nu'

We can now discuss the coinductive soundness of the Nu' rule, *i.e.* its soundness relative to the greatest Herbrand models. We note that, not surprisingly (*cf.* Sangiorgi, 2009), the rule Nu' is inductively unsound.

Example 6.28

Consider a program $\mathcal{P}$ consisting of just one clause $\kappa : A \Rightarrow A$. The rule Nu' *allows us to entail A:*

$$\dfrac{\dfrac{(\alpha : \ \Rightarrow A) \in \mathcal{P}, (\alpha : \ \Rightarrow A)}{\mathcal{P}, (\alpha : \ \Rightarrow A) \longrightarrow \alpha : A} \qquad (\kappa : A \Rightarrow A) \in \mathcal{P}, (\alpha : \ \Rightarrow A)}{\mathcal{P}, (\alpha : \ \Rightarrow A) \longrightarrow \kappa\alpha : A}\ \textsc{Nu'}$$

However, the least Herbrand model $\mathcal{M}_{\mathcal{P}_s} = \emptyset$ of the program does not contain (any ground instance of) A.

This example also shows that the system Lp-m + Nu is a proper extension of the semantics of Horn-clause logic. We can see the system as a *coinductive* big-step operational semantics of Horn-clause logic.

Similarly, the formula $\mathtt{eq}\,(\mathtt{oddList}\,\mathtt{int})$ proven in Example 6.27 is not inductively sound, either. Thus, the coinductive fragment of the extended corecursive resolution is only coinductively sound. When proving the coinductive soundness of the Nu' rule, we carefully choose the proof method by which we proceed. Inductive soundness of the Lp-m rule was proven by induction on the derivation tree and the construction of the least Herbrand models by iterations of $\mathcal{T}_{\mathcal{P}}$. Here, we give an analogous result, where coinductive soundness is proven by induction on the iterations of the semantic

6.3 Coinductive Type Class Resolution

Resolution using the LP-M rule may not terminate as demonstrated by Example 6.5 in Section 6.2. Lämmel and Peyton Jones (2005) observed that in such cases there may be a cycle in the inference that can be detected. Such treatment of cycles amounts to coinductive reasoning and results in building a corecursive proof term—*i.e.* a *(co-)recursive dictionary* in Haskell terminology.

6.3.1 Proof system LP-M + NU'

A (restricted) proof system that captures treatment of type classes such as in Example 6.5 is given in the following definition.

Definition 6.26 (Corecursive type class resolution)
Let $\mathcal{P}$ be a program, A an atom, e a proof term, and α a proof-term variable. The calculus of corecursive type class resolution *consists of the inference rule* LP-M *and the following inference rule:*

$$\textit{if } \mathrm{gHNF}(e)\ \frac{\mathcal{P}, (\alpha : \ \Rightarrow A) \longrightarrow e : A}{\mathcal{P} \longrightarrow \nu\alpha.e : A} \qquad \text{(NU')}$$

The side condition of NU' requires the proof term to be in guarded head normal form. Since, in this section, we are working with a calculus consisting of the rules LP-M and NU', there is no way to introduce a λ-abstraction into a proof term. Therefore, in this section, we restrict ourselves to guarded head normal form terms of the form $\kappa\,\underline{e}$.

Example 6.27
Recall the program $\mathcal{P}_{EvenOdd}$ in Example 6.5. The originally non-terminating resolution trace for the query `eq (evenList int)` *is resolved using the* NU' *rule as follows:*

literature (*cf.* Miller and Nadathur, 2012, Section 2.6.2). For the sake of simplicity, in the remainder of this thesis we employ the following syntactic abbreviation for Horn clauses:

$$\Rightarrow A \quad = A$$

$$A_1 \wedge \cdots \wedge A_n \Rightarrow A \quad = A_1 \Rightarrow (A_2 \wedge \cdots \wedge A_n \Rightarrow A)$$

In such a case, the atom A is called a *head* of the clause and the atoms A_1, ..., A_n are called a *body* of the clause. With this notation, we follow the standard practice and we routinely understand that the clause is implicitly universally quantified. When we use a conjunctive goal $p\,M_1 \cdots M_n \wedge q\,N_1 \cdots N_m$ we understand that the signature is implicitly extended with a new predicate symbol r of the appropriate kind. The program is implicitly extended with the following clause:

$$\forall x_1.\cdots\forall x_n.\forall y_1.\cdots.\forall y_m.(p\,x_1 \cdots x_n \wedge q\,y_1 \cdots y_m \Rightarrow r\,x_1 \cdots x_n\,y_1 \cdots y_m)$$

We then understand the conjunctive goal to stand for $r\,M_1 \cdots M_n\,N_1 \cdots N_m$.

The properties stated in Proposition 2.14 can be extended to well-formed clauses and goals.

Proposition 2.22

1. *If $\mathcal{S}_1, \mathcal{S}_2; \Gamma \vdash D : \mathtt{o}$ and $\vdash \mathcal{S}_1, c : A, \mathcal{S}_2$ then $\mathcal{S}_1, c : A, \mathcal{S}_2; \Gamma \vdash D : \mathtt{o}$.*
2. *If $\mathcal{S}; \Gamma_1, \Gamma_2 \vdash D : \mathtt{o}$ and $\mathcal{S} \vdash \Gamma_1, x : A, \Gamma_2$ then $\mathcal{S}; \Gamma_1, x : A, \Gamma_2 \vdash D : \mathtt{o}$.*
3. *If $\mathcal{S}_1, \mathcal{S}_2; \Gamma \vdash G : \mathtt{o}$ and $\vdash \mathcal{S}_1, c : A, \mathcal{S}_2$ then $\mathcal{S}_1, c : A, \mathcal{S}_2; \Gamma \vdash G : \mathtt{o}$.*
4. *If $\mathcal{S}; \Gamma_1, \Gamma_2 \vdash G : \mathtt{o}$ and $\mathcal{S} \vdash \Gamma_1, x : A, \Gamma_2$ then $\mathcal{S}; \Gamma_1, x : A, \Gamma_2 \vdash G : \mathtt{o}$.*

Similarly, the property of Proposition 2.15 can be extended to clauses and goals:

Proposition 2.23

1. *If $\mathcal{S}; \Gamma \vdash D : \mathtt{o}$ and $x \notin \Gamma$ then $\mathcal{S}; \Gamma \vdash D[M/x] : \mathtt{o}$.*

2. *If* $\mathcal{S}; \Gamma \vdash G : \mathtt{o}$ *and* $x \notin \Gamma$ *then* $\mathcal{S}; \Gamma \vdash G[M/x] : \mathtt{o}$.
3. *If* $\mathcal{S}; \Gamma \vdash D : \mathtt{o}$ *and* $x \notin \Gamma$ *then* $\mathcal{S}; \Gamma \vdash D[M/x] \equiv D : \mathtt{o}$.

Finally, we define logic programs as collections of clauses.

Definition 2.24 (Programs)

$$\mathcal{P} \ni \mathcal{P} \qquad := \cdot \mid \mathcal{P}, D \qquad \textit{programs}$$

For the purpose of this section, we implicitly assume that programs consists only of well-formed clauses.

Example 2.25

Returning to Example 1.3 and ignoring the annotating symbols,

$$\mathcal{P}_{\mathtt{pair}} = \cdot, \forall x : \mathtt{int}.\forall y : \mathtt{int}.\mathtt{eq}\, x \Rightarrow \mathtt{eq}\, y \Rightarrow \mathtt{eq}\,(\mathtt{pair}\, x\, y), \mathtt{eq}(\mathtt{int})$$

is a logic program. $\mathcal{P}_{\mathtt{Pair}}$ *consists of clauses that are well-formed in signature* $\mathcal{S}_{\mathtt{Pair}}$.

When the program is non-empty, we omit the leading empty program, similarly to notation for signatures and contexts.

2.3 Models of Logic Programs

In our analysis of soundness of type class resolution in Chapter 6, we make use of the least and the greatest Herbrand models. The models are defined in the standard way, that is for the first-order, untyped language.

In this section, we restrict terms of the language that we introduced in the previous section:

Definition 2.26 (First-order syntax)

$$t \ni M, N \qquad := c \mid x \mid M\, N \qquad \textit{terms}$$

Other syntactic objects of the language remain the same as in the previous section.

The least Herbrand model is $\mathcal{M}_{\mathcal{P}} = \{A\ f, B\ f\}$. *Therefore* $\mathcal{P} \vDash_{ind} \mathtt{B}\ x \Rightarrow \mathtt{A}\ x$. *However, any proof of* $\mathtt{B}\ x \Rightarrow \mathtt{A}\ x$ *needs to show that:*

$$\dfrac{\dfrac{\cdots}{\mathcal{P}, \alpha : \ \Rightarrow \mathtt{B}\ x \longrightarrow e : \mathtt{A}\ x}}{\mathcal{P} \longrightarrow \lambda\alpha.e : \mathtt{B}\ x \Rightarrow \mathtt{A}\ x}\ \textsc{Lam}$$

where e *is a proof term. This proof will not succeed since no axiom or hypothesis matches* $\mathtt{A}\, x$.

Program transformation methods

The main purpose of introducing the rule Lam in literature was to increase expressivity of the proof system. In particular, obtaining an entailment $\mathcal{P} \longrightarrow e : H$ of a Horn clause H enables the program $\mathcal{P}$ to be extended with Horn clause $e : H$, which can be used in further proofs. We show that transforming (the standard, untyped) logic programs in this way is inductively sound.

Theorem 6.24

Let $\mathcal{P}$ *be a logic program, and let* $\mathcal{P} \longrightarrow e : G$ *for a formula* G *by the* Lp-m *and* Lam *rules. Given a formula* G', $\mathcal{P} \vDash_{ind} G'$ *iff* $\mathcal{P}, G \vDash_{ind} G'$.

Proof. By the Theorem 6.14, $\mathcal{P} \vDash_{ind} G$. Therefore, $\mathcal{M}_{\mathcal{P}}$ is a model of G and $\mathcal{M}_{\mathcal{P}} = \mathcal{M}_{\mathcal{P},G}$. Hence $\mathcal{P} \vDash_{ind} G'$ iff $\mathcal{P}, G \vDash_{ind} G'$. □

Note, however, that the above theorem is not as trivial as it looks, in particular, it does not hold coinductively, *i.e.* if we replace $\vDash_{ind}$ with $\vDash_{coind}$ in the statement above. Consider the following example.

Example 6.25

Using the Lam *rule, one can prove* $\cdot \longrightarrow \lambda\alpha.\alpha : A \Rightarrow A$:

$$\dfrac{\dfrac{}{\alpha : \ \Rightarrow A \longrightarrow \alpha : A}}{\cdot \longrightarrow \lambda\alpha.\alpha : A \Rightarrow A}\ \textsc{Lam}$$

The greatest Herbrand models of the extended program $\cdot, A \Rightarrow A$ *then contains all ground instances of* A *and hence* $\cdot, A \Rightarrow A \vDash_{coind} A$. *However, clearly* $\cdot \nvDash_{coind} A$.

Example 6.25 concludes our discussion of program transformation methods in the inductive case.

of the rule Lp-m and on the properties of the semantic operator $\mathcal{T}_\mathcal{P}$ that is used to construct the least Herbrand models.

An alternative formulation of the completeness result, this time involving implicative formulae and hence the rule Lam in the proof, is:

Definition 6.21 (Inductive completeness w.r.t. a model)
If $\mathcal{M}_\mathcal{P} \vDash_{ind} G$ then there is a derivation of $\mathcal{P} \longrightarrow e : G$ in the Lp-m + Lam *proof system.*

However, neither of the systems Lp-m or Lp-m + Lam is complete in the sense of Definition 6.21. We illustrate this by a means of an example. First, we consider the proof system consisting solely of the rule Lp-m.

Example 6.22
Let Σ be a signature consisting of a unary predicate symbol `A`*, a unary function symbol* `f` *and a constant function symbol* `g`*. Let $\mathcal{P}$ be the following program:*

$$\kappa_1 : \quad \Rightarrow \texttt{A}\ (\texttt{f}\ x)$$

$$\kappa_2 : \quad \Rightarrow \texttt{A}\ \texttt{g}$$

The least Herbrand model of $\mathcal{P}$ is $\mathcal{M}_\mathcal{P} = \{A\ g, A\ (f\ g), A\ f(f\ g), \dots\}$. Therefore, $\mathcal{P} \vDash_{ind} \texttt{A}\ x$. However, neither κ_1 nor κ_2 matches $\texttt{A}\ x$. Thus there is no way to construct a proof term e satisfying:

$$\frac{\dots}{\mathcal{P} \longrightarrow e : \texttt{A}\ x}\ \text{Lp-m}$$

We demonstrate the incompleteness of the proof system Lp-m + Lam through the following example:

Example 6.23
Let Σ be a signature consisting of the unary predicate symbols `A` *and* `B`*, and a constant function symbol* `f`*. Consider a program $\mathcal{P}$ given as follows:*

$$\kappa_1 : \quad \Rightarrow \texttt{A}\ \texttt{f}$$

$$\kappa_2 : \quad \Rightarrow \texttt{B}\ \texttt{f}$$

Note that the grammar that gives syntax of first-order terms is a sub-grammar of the grammar that gives terms in the previous section. Hence the the well-formedness and judgemental equality is preserved. To stay close to the usual presentation of untyped logic programming, we will also employ *implicit quantification*. We consider all free variables in a goal to be bound by an implicit existential quantifier and all free variables in a definite clause to be bound by an implicit universal quantifier.

We reconstruct the notion of an untyped language by considering only signatures in Sig that contain a single type constant α of kind `type`. We use Σ to denote such signatures. Term constants in Σ of type $(\Pi x_1 : \alpha. \dots (\Pi x_n : \alpha.\, \texttt{type}) \dots)$ represent function symbols of arity n, and similarly term constants of type $(\Pi x_1 : \alpha. \dots (\Pi x_n : \alpha.\, \texttt{o}) \dots)$ represent predicate symbols of arity n. Note that, for an untyped language, the order of implicit quantifiers is unimportant. Since the types in context cannot depend on the previous variables, it is easy to derive admissibility of the structural rule for swapping. Hence, the implicit quantifiers can be arbitrarily reordered.

Definition 2.27
Given a signature Σ, the Herbrand universe *is the set of all ground terms over Σ.*

We use $\mathbf{U}_\Sigma$ to denote the Herbrand universe over signature Σ.

Definition 2.28
Let $\mathbf{U}_\Sigma$ be a Herbrand universe. The Herbrand base *is the set of all atoms consisting of predicate symbols in Σ and ground terms in $\mathbf{U}_\Sigma$.*

We use $\mathbf{B}_\Sigma$ to denote a Herbrand base over signature Σ.

Example 2.29
The Herbrand universe $\mathbf{U}_{\Sigma_{\texttt{pair}}}$ is the set $\{\texttt{int}, \texttt{pair}(\texttt{int}, \texttt{int}),$
$\texttt{pair}(\texttt{pair}(\texttt{int}, \texttt{int}), \texttt{int}), \texttt{pair}(\texttt{int}, \texttt{pair}(\texttt{int}, \texttt{int})), \dots\}$.

The Herbrand Base $\mathbf{B}_{\Sigma_{\texttt{pair}}}$ is the set $\{\texttt{eq}(\texttt{int}), \texttt{eq}(\texttt{pair}(\texttt{int}, \texttt{int}), \dots\}$.

Definition 2.30 (Semantic operator)
Let $\mathcal{P}$ be a logic program over signature Σ. The mapping $\mathcal{T}_\mathcal{P} : 2^{\mathbf{B}_\Sigma} \to 2^{\mathbf{B}_\Sigma}$ is defined

as follows. Let I be a subset of $\mathbf{B}_\Sigma$.

$$\mathcal{T}_\mathcal{P}(I) = \{A \in \mathbf{B}_\Sigma \mid B_1 \wedge \cdots \wedge B_n \Rightarrow A \textit{ is a ground instance of a clause in } \mathcal{P} \textit{ ,}$$
$$\textit{and } \{B_1, \ldots, B_n\} \subseteq I\}$$

We call $\mathcal{T}_\mathcal{P}$ the *semantic operator*. The operator gives inductive and coinductive interpretation to the logic program $\mathcal{P}$.

Definition 2.31 (Least and greatest Herbrand models)
Let $\mathcal{P}$ be a logic program.

- *The* least Herbrand model *is the least set $\mathcal{M}_\mathcal{P} \in \mathbf{B}_\Sigma$ such that $\mathcal{T}_\mathcal{P}(\mathcal{M}_\mathcal{P}) = \mathcal{M}_\mathcal{P}$, and*
- *the* greatest Herbrand model *is the greatest set $\mathcal{M}'_\mathcal{P} \in \mathbf{B}_\Sigma$ such that $\mathcal{T}_\mathcal{P}(\mathcal{M}'_\mathcal{P}) = \mathcal{M}'_\mathcal{P}$.*

That is, the least Herbrand model of $\mathcal{P}$ is the least fixed point of $\mathcal{T}_\mathcal{P}$ and the greatest Herbrand model of $\mathcal{P}$ is the greatest fixed point. In general, fixed points of the semantic operator $\mathcal{T}_\mathcal{P}$ are stable under formation of logical consequences of $\mathcal{P}$ and models of $\mathcal{P}$. By the virtue of $\mathcal{T}_\mathcal{P}$ being monotonous and as a consequence of Knaster-Tarski theorem fixed points of $\mathcal{T}_\mathcal{P}$ form a complete lattice and both the greatest fixed point and the least fixed point exist.

Definition 2.32
Let $\mathcal{P}$ be a logic program with signature Σ.

$$\mathcal{T}_\mathcal{P} \uparrow 0 = \emptyset$$

$$\mathcal{T}_\mathcal{P} \uparrow \alpha = \begin{cases} \mathcal{T}_\mathcal{P}(\mathcal{T}_\mathcal{P}(\alpha - 1)) & , \alpha \textit{ is a successor ordinal} \\ \mathrm{lub}\{\mathcal{T}_\mathcal{P} \uparrow \beta \mid \beta < \alpha\} & , \textit{otherwise} \end{cases}$$

Proof. By structural induction on the derivation tree.

Base case: Let the derivation be

$$\frac{\kappa : A \in P}{\mathcal{P} \longrightarrow \kappa : \sigma A}$$

for an atomic formula A, a constant symbol κ, and a substitution σ. From the Lemma 2.37 part a) follows that $\mathcal{P} \vDash_{ind} \sigma A$.

Inductive case, subcase LP-M: Let the last step in the derivation tree be by the rule LP-M thus of the form

$$\frac{\mathcal{P} \longrightarrow e_1 : \sigma B_1 \quad \ldots \quad \mathcal{P} \longrightarrow e_n : \sigma B_n \quad (\kappa : B_1 \wedge \ldots B_n \wedge \Rightarrow A) \in \mathcal{P}}{\mathcal{S}; \mathcal{P} \longrightarrow \kappa\, e_1 \ldots e_n : \sigma A}$$

for atomic formulae A, B_1, ..., B_n, a proof-term symbol κ, a substitution σ and proof term e_1, ..., e_n. From the induction assumption, for $i \in \{1, \ldots, n\}$, $\mathcal{P} \vDash_{ind} \sigma B_i$ and by the Lemma 2.37 part b), $\mathcal{P} \vDash_{ind} \sigma A$.

Subcase LAM: Let the last step of the derivation be by the rule LAM thus of the form

$$\frac{\mathcal{P}, (\beta_1 : \ \Rightarrow B_1), \ldots, (\beta_n : \ \Rightarrow B_n) \longrightarrow e : A}{\mathcal{P} \longrightarrow \lambda\beta_1, \ldots, \beta_n.e : B_1 \wedge \cdots \wedge B_n \Rightarrow A}$$

for atomic formulae A, B_1, ..., B_n, proof term e, and variables b_1, ..., b_n. From the induction assumption, $\mathcal{P}, (\beta_1 : \ \Rightarrow B_1), \ldots, (\beta_n : \ \Rightarrow B_n) \vDash A$ and from the Lemma 6.18 also $\mathcal{P} \vDash_{ind} A$. □

Inductive completeness of system LP-M + LAM

Let us comment on completeness of the calculus of LP-M and the calculus of LP-M and LAM. In principle, one can consider two different variants of completeness results for LP-M + LAM. Recalling the standard results of Lloyd (1987), the first formulation is:

Definition 6.20 (Inductive completeness à la Lloyd)
If a ground atomic formula A is in $\mathcal{M}_\mathcal{P}$, then $\mathcal{P} \longrightarrow e : A$ is in the LP-M + LAM *proof system.*

Such a result can be found in (Lloyd, 1987, pp. 47-49) and follows by straightforward induction on the construction of $\mathcal{M}_\mathcal{P}$. The proof is based solely on the properties

Proposition 6.17

The inference rule Lam *is admissible in big-step operational semantics of the logic of hereditary Harrop formulae.*

Proof. By induction on the number of clauses $\beta_1 : \ \Rightarrow B_1$, ..., $\beta_n : \ \Rightarrow B_n$ using proposition 6.13 and the fact the semantics of the logic of hereditary Harrop formulae is given as an extension of the semantics of Horn-clause logic. □

By inspecting the proof we can observe that, similarly to the system Lp-m, derivations of the big-step semantics can be regarded as derivations in the system Lp-m+Lam. Thus the system Lp-m+Lam corresponds to a fragment of the logic of hereditary Harrop formulae. In this fragment, programs consists of Horn clauses and goals are those of Horn-clause logic and goals in the shape of universally quantified Horn clauses.

Inductive soundness of system Lp-m + Lam

We show that the calculus comprising the rules Lp-m and Lam is (universally) inductively sound.

Lemma 6.18

Let $\mathcal{P}$ be a logic program, let A, B_1 to B_n be atomic formulae and let κ_1 to κ_n be proof-term symbols. If $\mathcal{P}, (\kappa_1 : \ \Rightarrow B_1), \ldots, (\kappa_n : \ \Rightarrow B_n) \vDash_{ind} A$ then $\mathcal{P} \vDash_{ind} B_1 \wedge \cdots \wedge B_n \Rightarrow A$.

Proof. Assume that $\mathcal{P}, (\kappa_1 : \ \Rightarrow B_1), \ldots, (\kappa_n : \ \Rightarrow B_n) \vDash_{ind} A$. From Definition 2.30 there is the least n such that for any grounding substitution τ, $(\tau \circ \sigma)A \in \mathcal{T}_{\mathcal{P},(\kappa_1: \Rightarrow B_1),\ldots,(\kappa_n: \Rightarrow B_n)} \uparrow n$. Consider any substitution σ and suppose that for all i, $\mathcal{P} \vDash_{ind} \sigma B_i$. From the definition of validity for any grounding τ for all i, $(\tau \circ \sigma)B_i \in \mathcal{M}_{\mathcal{P}}$ hence there is the least m such that $(\tau \circ \sigma)B_i \in \mathcal{T}_{\mathcal{P}} \uparrow m$. From the assumption, for any grounding substitution τ also $(\tau \circ \sigma)A \in \mathcal{T}_{\mathcal{P}} \uparrow (n + m)$ and $\mathcal{P} \vDash_{ind} \sigma A$. Hence $\mathcal{P} \vDash_{ind} B_1 \wedge \cdots \wedge B_n \Rightarrow A$. □

Theorem 6.19

Let $\mathcal{P}$ be a logic program, G a formula, and e a proof term. Let $\mathcal{S}; \mathcal{P} \longrightarrow e : G$ be derived using the rules Lp-m *and* Lam*. Then $\mathcal{P} \vDash_{ind} G$.*

$$\mathcal{T}_{\mathcal{P}} \downarrow 0 = \mathbf{B}_{\Sigma}$$

$$\mathcal{T}_{\mathcal{P}} \downarrow \alpha = \begin{cases} \mathcal{T}_{\mathcal{P}}(\mathcal{T}_{\mathcal{P}}(\alpha - 1)) & , \ \alpha \text{ is a successor ordinal} \\ \text{glb}\{\mathcal{T}_{\mathcal{P}} \downarrow \beta \mid \beta < \alpha\} & , \ \text{otherwise} \end{cases}$$

Where lub *is the least upper bound of a set and* glb *is the greatest lower bound of a set.*

We call these operators ordinal powers of $\mathcal{T}_{\mathcal{P}}$. Ordinal powers can be used to give standard characterisation of Herbrand models.

Proposition 2.33

Let $\mathcal{P}$ be a logic program. Then $\mathcal{M}_{\mathcal{P}} = \mathcal{T}_{\mathcal{P}} \uparrow \omega$.

Proof of the proposition can be found in literature (Lloyd, 1987, Theorem 6.5, p.38). We emphasise that the characterisation of least Herbrand models holds in general. However, a converse characterisation does not hold.

Example 2.34

Consider a signature consisting of a unary function symbol f, a constant a, a unary predicate symbol P and a nullary predicate symbol Q. Let $\mathcal{P} = P(x) \Rightarrow P(f(x)), P(y) \Rightarrow Q$ be a program that consists of two clauses. One of the clauses contains existential variable. Then $\mathcal{T}_{\mathcal{P}} \downarrow \omega = \text{glb}\{\mathcal{T}_{\mathcal{P}} \downarrow \beta \mid \beta < \omega\} = \{Q\}$. However, this set is not a fixed point of $\mathcal{T}_{\mathcal{P}}$ and there is necessary one more application of $\mathcal{T}_{\mathcal{P}}$. Indeed, $\mathcal{T}_{\mathcal{P}}(\{Q\}) = \emptyset$ is the greatest fixed point of $\mathcal{T}_{\mathcal{P}}$, that is, $\mathcal{M}'_{\mathcal{P}} = \mathcal{T}_{\mathcal{P}} \downarrow (\omega + 1)$.

In general, the corresponding property does not hold for the greatest Herbrand model construction (Lloyd, 1987, p. 38). However, it does hold when we restrict Horn-clause logic to a fragment that does not contain existential variables. We assume the restriction in the remainder of this section. Lloyd (1987) observed that this restriction implies that the $\mathcal{T}_{\mathcal{P}}$ operator converges in at most ω steps although he did not provide a proof. We state and prove the property here.

Proposition 2.35

Let $\mathcal{P}$ be a logic program without existential variables. Then $\mathcal{M}'_{\mathcal{P}} = \mathcal{T}_{\mathcal{P}} \downarrow \omega$.

Proof. By contradiction. Consider a program $\mathcal{P}$ and the set $I = \mathcal{T}_\mathcal{P} \uparrow \omega$. Assume that $\mathcal{T}_\mathcal{P}(I) \neq I$. Then there is a ground atom A such that $A \in I$ and $A \notin \mathcal{T}_\mathcal{P}(I)$. Consider all clauses in $\mathcal{P}$ such that A is an instance of a head of such clause. Since there are no existential variables each instance of a head uniquely identifies instances of atoms in the body of the clause and these instances are ground. Call the set of all such identified instances of atoms in the bodies of the clauses a support S. Since $A \notin \mathcal{T}_\mathcal{P}(I)$ then $S \not\subseteq I$ and there is $n < \omega$ such that $S \not\subseteq \mathcal{T}_\mathcal{P} \downarrow n$. Hence $A \notin \mathcal{T}_\mathcal{P} \downarrow (n+1)$ and $A \notin \mathcal{T}_\mathcal{P} \downarrow \omega$ which is a contradiction and I is a fixed point. For any fixed point J, $J \subseteq \mathbf{B}_\Sigma$ and from monotonicity of $\mathcal{T}_\mathcal{P}$ follows that $J \subseteq I$. Hence I is the greatest fixed point. □

The above theorem provides a characterisation of greatest Herbrand models for the class of Horn clauses without existential variables that we consider here.

The *validity* of a formula in a model is defined as usual.

Definition 2.36

An atomic formula is valid *in a model I if and only if for any grounding substitution σ, we have $\sigma F \in I$. A Horn clause $B_1 \wedge \cdots \wedge B_n \Rightarrow A$ is valid in I if for any substitution σ, if σB_1, …, σB_n are valid in I then σA is valid in I.*

We use the notation $\mathcal{P} \vDash_{ind} F$ to denote that a formula F is valid in $\mathcal{M}_\mathcal{P}$ and $\mathcal{P} \vDash_{coind} F$ to denote that a formula F is valid in $\mathcal{M}'_\mathcal{P}$.

Lemma 2.37

Let $\mathcal{P}$ be a logic program and let σ be a substitution. The following holds:

1. *If $(\ \Rightarrow A) \in \mathcal{P}$ then both $\mathcal{P} \vDash_{ind} \sigma A$ and $\mathcal{P} \vDash_{coind} \sigma A$.*
2. *If, for all i, $\mathcal{P} \vDash_{ind} \sigma B_i$ and $(B_1 \wedge \cdots \wedge B_n \Rightarrow A) \in \mathcal{P}$ then $\mathcal{P} \ \vDash_{ind} \ \sigma A$.*
3. *If, for all i, $\mathcal{P} \vDash_{coind} \sigma B_i$ and $(B_1 \wedge \cdots \wedge B_n \Rightarrow A) \in \mathcal{P}$ then $\mathcal{P} \vDash_{coind} \sigma A$.*

Proof. a) Let $\mathcal{P}$ be a logic program such that $(\ \Rightarrow A) \in \mathcal{P}$. By Definition 2.30 of the semantic operator, for any grounding substitution τ, $\tau A \in \mathcal{T}_\mathcal{P}(\mathcal{M}_\mathcal{P})$. Since $\mathcal{M}_\mathcal{P}$ is a fixed point of $\mathcal{T}_\mathcal{P}$ also $\tau A \in \mathcal{M}_\mathcal{P}$ and by definition of validity of a formula, $P \vDash_{ind} A$ and also, for any substitution σ, $\mathcal{P} \vDash_{ind} \sigma A$. Since we do not use the fact that $\mathcal{M}_\mathcal{P}$ is the least fixed point the proof of the coinductive case is identical.

b) Let $\mathcal{P}$, A, B_1, …, B_n be as above. Assume, for all i, $\mathcal{P} \vDash_{ind} B_i$ whence, for all i, for any grounding substitution σ, $\sigma B_i \in \mathcal{M}_\mathcal{P}$. By Definition 2.30 of semantic

for atomic formulae A, B_1, …, B_n, a proof term symbol κ, a substitution σ and proof term e_1, …, e_n. From the induction assumption, for $i \in \{1, \ldots, n\}$, $\mathcal{P} \vDash_{ind} \sigma B_i$ and by the Lemma 2.37 part b), $\mathcal{P} \vDash_{ind} \sigma A$. □

This is a standard result that can be found in literature (Lloyd, 1987). We include a proof since the rule Lp-m also plays a crucial role in the coinductive fragment of type class resolution, as will be discussed in Sections 6.3 and 6.4. We believe that it is illustrative to compare structure of this proof with and the proofs of the appropriate lemmata in those sections.

6.2.2 Proof system Lp-m + Lam

A natural extension of the proof system Lp-m is the extension with a rule that allows us to prove implicative goals.

Definition 6.15

Let $\mathcal{P}$ be a program, A, B_1 to B_n atoms, e a proof term and β_1 to β_n proof variables. The calculus of extended type class resolution *is given by rule* Lp-m *and the following rule:*

$$\frac{\mathcal{P}, (\beta_1 : \ \Rightarrow B_1), \ldots, (\beta_n : \ \Rightarrow B_n) \longrightarrow e : A}{\mathcal{P} \ \longrightarrow \lambda\beta_1, \ldots, \beta_n.e : B_1 \wedge \cdots \wedge B_n \Rightarrow A} \quad \text{(Lam)}$$

We illustrate the use of the Lam rule by an example.

Example 6.16

Let $\mathcal{P} = (\kappa_1 : A \Rightarrow B), (\kappa_2 : B \Rightarrow C)$. Both the least and the greatest Herbrand model of $\mathcal{P}$ are empty. Equally, no formulae can be derived from the program by the Lp-m *rule. However, we can derive $A \Rightarrow C$ by using a combination of the* Lam *and* Lp-m *rules:*

$$\dfrac{\dfrac{\dfrac{\dfrac{\alpha : A \in \mathcal{P}, (\alpha : \ \Rightarrow A)}{\mathcal{P}, (\alpha : \ \Rightarrow A) \longrightarrow \alpha : A}}{\mathcal{P}, (\alpha : \ \Rightarrow A) \longrightarrow \kappa_1\, \alpha : B}}{\mathcal{P}, (\alpha : \ \Rightarrow A) \longrightarrow \kappa_1\, \alpha : C}}{\mathcal{P} \ \longrightarrow \lambda\alpha.\kappa_2\, (\kappa_1\, \alpha) : A \Rightarrow C}\ \text{Lam}$$

When there is no label on the right-hand side of an inference step, inference proceeds by Lp-m *rule. We follow this convention throughout the rest of this chapter.*

Again, the we relate the proof system to the big-step semantics:

$$\frac{\kappa_{\mathtt{int}} : \mathtt{eq\,int} \in \mathcal{P}_{Pair} \qquad \mathcal{P}_{Pair} \longrightarrow \kappa_{\mathtt{int}} : \mathtt{eq\,int}}{\mathcal{P}_{Pair} \longrightarrow \kappa_{\mathtt{pair}}\kappa_{\mathtt{int}}\kappa_{\mathtt{int}} : \mathtt{eq}\,(\mathtt{pair\,int\,int})} \quad \frac{\kappa_{\mathtt{int}} : \mathtt{eq\,int} \in \mathcal{P}_{Pair} \qquad \mathcal{P}_{Pair} \longrightarrow \kappa_{\mathtt{int}} : \mathtt{eq\,int}}{\mathcal{P}_{Pair} \longrightarrow \kappa_{\mathtt{pair}}\kappa_{\mathtt{int}}\kappa_{\mathtt{int}} : \mathtt{eq}\,(\mathtt{pair\,int\,int})} \quad \kappa_{\mathtt{int}} : \mathtt{eq\,int} \in \mathcal{P}_{Pair}$$

Derivations of type class resolution can be reproduced in the semantics of Horn-clause logic we gave in Section 3.1:

Proposition 6.13

The inference rule Lp-m *is admissible in big-step operational semantics of Horn-clause logic.*

Proof. By induction on length of the body of the clause $B_1 \wedge \ldots \wedge B_n \Rightarrow A$. □

Moreover, proof terms that are entailed in the big-step semantics of Horn-clause logic can be regarded as derivations in the system Lp-m as can be observed by inspecting the proof of Proposition 6.13. That is, the system given by the inference rule Lp-m corresponds to the proof-relevant big-step operational semantics of Horn-clause resolution we gave in Section 3.1. For the purpose of discussion of soundness of the system Lp-m we understand that the judgement of big-step operational semantics is restricted to Horn-clause logic.

Inductive soundness of system Lp-m

The entailment in Example 6.12 is inductively sound, *i.e.* it is sound with respect to the least Herbrand model of $\mathcal{P}_{Pair}$.

Theorem 6.14

Let $\mathcal{P}$ be a program, e a proof term, and A an atom. Let $\mathcal{P} \longrightarrow e : A$ hold. Then $\mathcal{P} \vDash_{ind} A$.

Proof. By structural induction on the derivation of the entailment.

Base case: Let the derivation be

$$\frac{\kappa : A \in \mathcal{P}}{\mathcal{P} \longrightarrow \kappa : \sigma A}$$

for an atomic formula A, a proof term symbol κ, and a substitution σ. From Lemma 2.37 part a) follows that $\mathcal{P} \vDash_{ind} \sigma A'$.

Inductive case: Let the last step in the derivation of the entailment be

$$\frac{\mathcal{P} \longrightarrow e_1 : \sigma B_1 \quad \ldots \quad \mathcal{P} \longrightarrow e_n : \sigma B_n \quad \kappa : B_1 \wedge \cdots \wedge B_n \Rightarrow A \in \mathcal{P}}{\mathcal{P} \longrightarrow \kappa\, e_1 \ldots e_n : \sigma A'}$$

operator, $\sigma A \in \mathcal{T}_{\mathcal{P}}(\mathcal{M}_{\mathcal{P}})$. Since $\mathcal{M}_{\mathcal{P}}$ is a fixed point also $\sigma A \in \mathcal{M}_{\mathcal{P}}$ and $\mathcal{P} \vDash_{ind} \sigma A$.

c) Note that the proof of b) does not make any use of the fact that $\mathcal{M}_{\mathcal{P}}$ is the least fixed point. Therefore use the proofs of b) *mutatis mutandis*. □

Discussion

Let us make a note on some properties of greatest Herbrand models. The properties will drive our choice of coinductive models in our analysis in Chapter 6. The literature (Lloyd, 1987) offers two kinds of greatest Herbrand model construction for logic programs. The greatest Herbrand model of a program $\mathcal{P}$ is obtained as the greatest fixed point of the semantic operator $\mathcal{T}_{\mathcal{P}}$ on the Herbrand base of $\mathcal{P}$, *i.e.* on the set of all finite ground atomic formulae formed in the signature of the program $\mathcal{P}$. *The greatest complete Herbrand model* of a program $\mathcal{P}$ is obtained as the greatest fixed point of the semantic operator $\mathcal{T}'_{\mathcal{P}}$ on the *complete Herbrand base*. The complete Herbrand base is defined as the set of all *finite and infinite* ground atomic formulae formed in the signature of the program $\mathcal{P}$. Usually, greatest complete Herbrand models are preferred in the literature on coinduction in logic programming (Komendantskaya and Johann, 2015, Lloyd, 1987, Simon et al., 2007). There are two reasons for such bias: first, $\mathcal{T}'_{\mathcal{P}}$ reaches its greatest fixed point in at most ω steps due to compactness of the complete Herbrand base. $\mathcal{T}_{\mathcal{P}}$ does not posses this property in general as we demonstrated in Example 2.34. However, the prohibition of existential variables we impose on Horn clauses means that the greatest Herbrand models regain the same advantage. This is the subject of Proposition 2.35.

2.4 Nameless Logical Framework

Standard expositions of a type theory use variable names. However, variable names carry a burden when implementing such a type theory. For example, types need to be checked up to α-equivalence of bound variables and fresh names need to be introduced in order to expand terms to η-long form. In Chapter 5, we commit to a version of Logical Framework (LF) (Harper et al., 1993) as our choice of first-order dependent type theory that uses de Bruijn indices instead of explicit names; we call such LF *nameless*. The use de Bruijn indices allows us to avoid the above problems

when checking the equality of terms and types and when synthesising new terms and types. In this section, we present syntax and typing judgements of nameless LF. Our presentation follows Harper and Pfenning (2005) but employs de Bruijn indices and explicit substitutions (Abadi et al., 1990) instead of names.

2.4.1 Syntax

The LF is a first-order dependent type theory. The syntax is separated into three levels of objects. There are separate levels of kinds, of types and of terms.

We use natural numbers in $\mathbb{N}$ for de Bruijn indices $\iota, \iota_1, \ldots$, and we denote successor by $\sigma(-)$. We assume countably infinite disjoint sets $\mathcal{C}$ of *term constants*, and $\mathcal{A}$ of *type constants*. We denote elements of $\mathcal{C}$ by c, c', *etc.*, and elements of $\mathcal{A}$ by α, β, *etc.* We define *terms*, *types*, and *kinds* as well as *signatures* and *contexts* of LF.

Definition 2.38

$t \ni M, N$	$::= c \mid \mathbb{N} \mid A\,M \mid \lambda A.M$	*terms*
$T \ni A, B$	$::= \alpha \mid A\,M \mid \Pi A.B$	*types*
$K \ni L$	$::= \mathtt{type} \mid \Pi A.L$	*kinds*
$\mathrm{Sig} \ni \mathcal{S}$	$::= \cdot \mid \mathcal{S}, c : A \mid \mathcal{S}, \alpha : L$	*signatures*
$\mathrm{Ctx} \ni \Gamma$	$::= \cdot \mid \Gamma, A$	*contexts*

Terms consist of term constants, de Bruijn indices, function application and function abstraction. We use identifiers M, N to denote terms in t. Types consists of type constants, type application, and formation of dependent type family. We do not consider type level abstraction. Note that this does not decrease expressive power of the calculus (Geuvers and Barendsen, 1999). We use identifiers A, B to denote

6.2 Inductive Type Class Resolution

In this section, we describe the inductive fragment of the calculus for the extended type class resolution that was introduced by Fu et al. (2016). We show that inference rules of this calculus are admissible in the framework of Chapter 3. We reconstruct the standard theorem of universal inductive soundness for the resolution rule. We consider an extended version of type class resolution, working also with implicative goals rather than working just with atomic formulae. We show that the resulting proof system is inductively sound, but coinductively unsound; we also show that it is incomplete. Based on these results, we discuss the program transformation methods that arise.

6.2.1 Proof system LP-M

First, we give semantics of *type class resolution* using the syntax of proof-relevant Horn-clause resolution.

Definition 6.11 (Type class resolution)
Let $\mathcal{P}$ be a program, A, B_1 to B_n atoms, σ a substitution, and e, e_1 to e_n proof terms. The calculus of type class resolution *is given by the following single rule:*

$$\frac{\mathcal{P} \longrightarrow e_1 : \sigma B_1 \qquad \cdots \qquad \mathcal{P} \longrightarrow e_n : \sigma B_n \qquad (e : B_1 \wedge \cdots \wedge B_n \Rightarrow A) \in \mathcal{P}}{\mathcal{P} \longrightarrow e\ e_1 \ldots e_n : \sigma A} \tag{LP-M}$$

If, for a given atomic formula A, a given proof term e, and a given program $\mathcal{P}$, $\mathcal{P} \longrightarrow e : A$ is derived using the LP-M rule we say that A is entailed by $\mathcal{P}$ and that the proof term e witnesses this entailment. The signature $\mathcal{S}$ of the logic program $\mathcal{P}$ does not play a role in the inference rule and we keep it implicit. We will do so for signatures in the rest of this chapter.

Example 6.12
Recall the logic program $\mathcal{P}_{Pair}$ in Example 1.3. The inference steps for resolution of the goal $\mathtt{eq}\,(\mathtt{pair}\,\mathtt{int}\,\mathtt{int})$ *correspond to the following derivation tree in the calculus of Definition 6.11.*

$\lambda\alpha_1.\ldots.\lambda\alpha_n$ and proof term applications $(e_1\,(\ldots(e_m)\ldots))$ respectively where n and m are known from the context or are unimportant.

Restriction 1 of Definition 6.3 requires that Horn clauses in a logic program do not overlap, *i.e.* heads of the Horn clauses in the program do not unify. However, an auxiliary goal in an implicative shape may be proven in the course of corecursive type class resolution and added to the program. Such formula may overlap with other clauses in the program—only Horn clauses in the original program, that is Horn clauses that originate as type class instances, are subject to restriction 1.

Example 6.10

In the program in Example 6.8 the Horn formula $\mathtt{eq}\,x\wedge\Rightarrow\mathtt{eq}\,(\mathtt{bush}\,x)$ *can be (coinductively) proven with the recursive proof term* $\kappa_{\mathtt{bush}'}=\nu\alpha.\lambda\beta.\kappa_2\beta(\alpha(\alpha\beta))$*. If we add this Horn clause to the program* $\mathcal{P}_{Bush}$ *we obtain a proof of* $\mathtt{eq}\,(\mathtt{bush}\,\mathtt{int})$ *by applying* $\kappa_{\mathtt{bush}'}$ *to* $\kappa_{\mathtt{int}}$*.*

In the case of implicative queries it is even more challenging to understand whether the obtained proof is indeed sound: whether inductively, coinductively or in any other sense. In Section 6.4, we establish *coinductive soundness* for proofs of such implicative queries relative to the greatest Herbrand models of logic programs. Namely, we determine that proofs that are obtained by extending the proof context with coinductively proven Horn clauses are coinductively sound but inductively unsound. This result completes our study of the semantic properties of corecursive type class resolution. Sections 6.2 and 6.4 summarise our arguments concerning the inductive and coinductive incompleteness of corecursive type class resolution.

In the following sections, we will gradually introduce inference rules for proof-relevant corecursive resolution that was given in Fu et al. (2016) as admissible in the calculus we exposed in Chapter 3 in the inductive case and as a proper extension of the calculus in the coinductive case. We start with its inductive fragment, *i.e.* the fragment that is sound relative to the least Herbrand models, and then in subsequent sections consider its two coinductive extensions (which are both sound with respect to the greatest Herbrand models).

types in T. Kinds are a technical device to classify types and include a distinguished kind `type` and the kind of dependent type families. We use the identifier L to denote kinds in K. Signatures store information about types and kinds assigned to term and type constants respectively. Contexts store information about types of variables. Since we use de Bruijn indices for variables, variable name is not stored in a context. We use $\mathcal{S}$ for signatures and Γ for contexts. We use parenthesis for the sake of readability as is standard.

Example 2.39

Let `bool` *and* $\equiv_{\mathtt{bool}}$ *be type constants. Let* `tt`*,* `ff`*, and* `refl` *be term constants. Then* $\Pi\,\mathtt{bool}.(\Pi\,\mathtt{bool}\,.\,\mathtt{type})$ *is a kind,* $\Pi\,\mathtt{bool}.((\equiv_{\mathtt{bool}}0)0)$ *is a type, and* $(\lambda\,\mathtt{bool}\,.\,\mathtt{refl}\;0)$ *and* `refl tt` *are terms.*

Also, $\cdot,\mathtt{bool}:\mathtt{type},\mathtt{tt}:\mathtt{bool},\mathtt{ff}:\mathtt{bool},\equiv_{\mathtt{bool}}:\Pi\,\mathtt{bool}.(\Pi\,\mathtt{bool}\,.\,\mathtt{type})$ *is a signature and* $\cdot,\mathtt{bool}$ *is a context.*

Explicit substitutions (Abadi et al., 1990) are manipulated using two operations. *Shifting* recursively traverses a term, a type, or a kind and increases all indices greater then ι by one.

Definition 2.40 (Shifting)

Term and type shifting, denoted by $(-)\uparrow^\iota$ *is defined as follows:*

$$c\uparrow^\iota=c$$

$$(\lambda A.M)\uparrow^\iota=\lambda A\uparrow^\iota.M\uparrow^{\sigma\iota}$$

$$(MN)\uparrow^\iota=(M\uparrow^\iota)(N\uparrow^\iota)$$

$$\iota\uparrow^0=\sigma\iota$$

$$0\uparrow^{\sigma\iota}=0$$

$$\sigma\iota\uparrow^{\sigma\iota'}=\sigma(\iota\uparrow^{\iota'})$$

$$\alpha\uparrow^{\iota} = \alpha$$

$$(\Pi A.B)\uparrow^{\iota} = \lambda A\uparrow^{\iota} .B\uparrow^{\sigma\iota}$$

$$(AM)\uparrow^{\iota} = (A\uparrow^{\iota})(M\uparrow^{\iota})$$

Substitution with a term N and index ι replaces indices that are bound by the ι-th binder while updating remaining indices. The index ι is increased when traversing under a binder.

Definition 2.41 (Substitution)

Term and type substitution, denoted by $(-)[N/\iota]$ is defined as follows:

$$c[N/\iota] = c$$

$$(\lambda A.M)[N/\iota] = \lambda A[N/\iota].M[N\uparrow^0 /\sigma\iota]$$

$$(M_1M_2)[N/\iota] = (M_1[N/\iota])(M_2[N/\iota])$$

$$0[N/0] = N$$

$$0[N/\sigma\iota] = 0$$

$$\sigma\iota[N/0] = \sigma\iota$$

$$\sigma\iota[N/\sigma\iota'] = \sigma(\iota[N/\iota'])$$

$$\alpha[N/\iota] = \alpha$$

$$(\Pi A.B)[N/\iota] = \lambda A[N/\iota].B\uparrow^0 [N/\sigma\iota]$$

$$(AM)[N/\iota] = (A[N/\iota])(M[N/\iota])$$

Shifting with a greater index than zero and substitution for other indices than zero will not be needed in many cases. For the sake of readability we introduce the

$$\mathtt{eq}\,(\mathtt{bush}\,\mathtt{int}) \rightsquigarrow (\mathtt{eq}\,(\mathtt{bush}\,\mathtt{int}))^{\kappa_{\mathtt{bush}:}} - \rightsquigarrow^* \mathtt{eq}\,\mathtt{int} \wedge \mathtt{eq}\,(\mathtt{bush}\,(\mathtt{bush}\,\mathtt{int})) \rightsquigarrow$$

$$(\mathtt{eq}\,\mathtt{int})^{\kappa_{\mathtt{int}:}} - \wedge \mathtt{eq}\,(\mathtt{bush}\,(\mathtt{bush}\,\mathtt{int})) \rightsquigarrow^* \mathtt{eq}\,(\mathtt{bush}\,(\mathtt{bush}\,\mathtt{int})) \rightsquigarrow$$

$$(\mathtt{eq}\,(\mathtt{bush}\,(\mathtt{bush}\,\mathtt{int})))^{\kappa_{\mathtt{bush}:}} - \rightsquigarrow^* \mathtt{eq}\,\mathtt{int} \wedge \mathtt{eq}\,(\mathtt{bush}\,(\mathtt{bush}\,(\mathtt{bush}\,\mathtt{int})) \rightsquigarrow$$

$$(\mathtt{eq}\,\mathtt{int})^{\kappa_{\mathtt{int}:}} - \wedge \mathtt{eq}\,(\mathtt{bush}\,(\mathtt{bush}\,(\mathtt{bush}\,\mathtt{int})) \rightsquigarrow^*$$

$$\mathtt{eq}\,(\mathtt{bush}\,(\mathtt{bush}\,(\mathtt{bush}\,\mathtt{int})) \rightsquigarrow \ldots$$

Fu et al. (2016) have recently introduced an extension to corecursive type class resolution that allows implicative goals to be proved by corecursion and uses the recursive proof term construction. Implicative goals require that we extend the language we use for representing logic programs. The shape of these goals is always that of Horn clauses, as will be stated formally by the inference rule Lam below. We could define a proper syntactic class to exactly capture these extended goals but we will opt out for the syntax of the logic of hereditary Harrop formulae we introduced in Section 3.2 of Chapter 3. Consecutively, proof terms then contain λ-abstraction. However, in order to study corecursive resolution, we need to extend the syntax of proof terms to allow for recursive proof terms.

Definition 6.9 (Recursive proof terms)

$$\mathrm{PT} \ni e \qquad := \cdots \mid \nu\kappa.e \qquad \textit{proof terms}$$

Proof terms are extended with a new syntactic construct, ν abstraction, that represents recursion. The ellipsis in the definition are to be understood as the appropriate syntactic constructs of Definition 3.23 in Chapter 3. In this chapter, we refer to recursive proof terms as proof terms. We keep the use of the identifier e for proof terms. We further use identifiers α, β for proof-term symbols that are subject to ν abstraction. A proof term e is in *guarded head normal form* (denoted gHNF(e)), if $e = \lambda\underline{\alpha}.\kappa\ \underline{e}$ where $\underline{\alpha}$ and $\underline{e}$ denote (possibly empty) sequences of abstraction

Example 6.7

The formula $\texttt{eq}\,(\texttt{evenList}\,\texttt{int})$ *is not in the least Herbrand model of the logic program* $\mathcal{P}_{EvenOdd}$ *in Example 6.5 but it is in the greatest Herbrand model of the program.*

Similarly to the inductive case, corecursive type class resolution is coinductively incomplete. Consider the clause $\kappa_{inf} : \texttt{p}\,x \Rightarrow \texttt{p}\,(\texttt{f}\,x)$. This clause may be given an interpretation by the greatest (complete) Herbrand models. However, corecursive type class resolution does not yield infinite proofs.

Unfortunately, the simple method of cycle detection does not work for all non-terminating programs. In some cases, type class resolution does not terminate but does not exhibit cycles either. We illustrate this behaviour using an example that originates in work of Fu et al. (2016), the adaptation is by Farka et al. (2016).

Example 6.8 (Farka et al. (2016))

Consider a data structure `Bush` *and its corresponding instance for type class* `Eq`*:*

```
data Bush a = Nil
            | Cons a (Bush (Bush a))

instance Eq a, Eq (Bush (Bush a)) ⇒ Eq (Bush a) where
    eq Nil          Nil          = True
    eq (Cons x xs) (Cons y ys) = eq x y && eq xs ys
```

Horn-clause presentation of type class declarations for data structure `Bush` *is given by the program* $\mathcal{P}_{Bush}$*:*

$$\kappa_{\texttt{bush}} : \ \texttt{eq}\,x \wedge \texttt{eq}\,(\texttt{bush}\,(\texttt{bush}\,x)) \Rightarrow \texttt{eq}\,(\texttt{bush}\,x)$$

$$\kappa_{\texttt{int}} : \ \Rightarrow \texttt{eq}\,\texttt{int}$$

The derivation below shows that no cycles arise when we resolve the goal $\texttt{eq}\,(\texttt{bush}\,\texttt{int})$ *against the program* $\mathcal{P}_{Bush}$*:*

following abbreviations:

Definition 2.42

$$A{\uparrow} \stackrel{def}{=} A{\uparrow^0}$$

$$M{\uparrow} \stackrel{def}{=} M{\uparrow^0}$$

$$A[N] \stackrel{def}{=} A[N/0]$$

$$M[N] \stackrel{def}{=} M[N/0]$$

We demonstrate shifting and substitution on an example.

Example 2.43

Consider the term $(\texttt{refl}\ 0)$*. Shifting of this term with index zero* $(\texttt{refl}\ 0){\uparrow^0}$ *is the term* $(\texttt{refl}\ 1)$*. A substitution of the term* `tt` *for variable* 0 *in this term that is* $(\texttt{refl}\ 0)[\texttt{tt}/0]$ *is the term* $\texttt{refl}\ \texttt{tt}$*.*

Well-formedness of objects introduced by Definition 2.38 is stated by a means of several judgements. In particular, we give equality in nameless LF as algorithmic, following Harper and Pfenning (2005). In order to do so we define *simple kinds*, *simple types*, *simple signatures*, and *simple contexts*.

Definition 2.44

$K^- \ni \kappa$	$::= \kappa \mid \tau \to \kappa$	*simple kinds*
$T^- \ni \tau$	$::= a \mid \tau \to \tau$	*simple types*
$\mathrm{Sig}^- \ni \mathcal{S}^- ::=$	$\cdot \mid \mathcal{S}^-, c : \tau \mid \mathcal{S}^-, a : \kappa$	*simple signatures*
$\mathrm{Ctx}^- \ni \Delta ::=$	$\cdot \mid \Delta, \tau$	*simple contexts*

Algorithmic statement of equality uses simple types and simple kinds rather than types and kinds as there are no dependencies on terms. We use identifiers κ for simple kinds, τ for simple types, $\mathcal{S}^-$ for simple signatures and Δ for simple contexts. The erasure from objects to corresponding simple objects, denoted $(-)^-$ is defined as follows:

Definition 2.45 (Erasure)

$$(\mathtt{type})^- = \mathtt{type}$$

$$(\Pi A.L)^- = (A)^- \to (L)^-$$

$$(\alpha)^- = \alpha$$

$$(\Pi A.B)^- = (A)^- \to (B)^-$$

$$(AM)^- = (A)^-$$

We conclude exposition of syntax of nameless LF with an example of simple kinds, simple types and simple signatures and contexts.

Example 2.46

Consider constants given in Example 2.39. Then $\mathtt{bool} \to (\mathtt{bool} \to \mathtt{type})$ *is a simple kind and* $\mathtt{bool} \to \equiv_{\mathtt{bool}}$ *is a simple type. These are results of erasure on kinds and types given in Example 2.39.*

Also, $\cdot, \mathtt{bool} : \mathtt{type}, \mathtt{tt} : \mathtt{bool}, \mathtt{ff} : \mathtt{bool}, \equiv_{\mathtt{bool}} : \mathtt{bool} \to (\mathtt{bool} \to \mathtt{type})$ *is a simple signature and* $\cdot; \mathtt{bool}$ *is a simple context.*

2.4.2 Typing and equality

Typing judgements of nameless LF and equality of objects are defined mutually. We call these judgements commonly well-formedness judgements. The notion of equality we consider is weak algorithmic equality (we refer to Harper and Pfenning (2005) for details).

A non-terminating small-step resolution trace is given by:

$$\underline{\mathtt{eq}\,(\mathtt{evenList}\,\mathtt{int})} \rightsquigarrow (\mathtt{eq}\,(\mathtt{evenList}\,\mathtt{int}))^{\kappa_{\mathtt{evenList}}\,\_} \rightsquigarrow^*$$

$$\mathtt{eq}\,\mathtt{int} \wedge \mathtt{eq}\,(\mathtt{oddList}\,\mathtt{int}) \rightsquigarrow (\mathtt{eq}\,\mathtt{int})^{\kappa_{\mathtt{int}}\,\_} \wedge \mathtt{eq}\,(\mathtt{oddList}\,\mathtt{int}) \rightsquigarrow^*$$

$$\mathtt{eq}\,(\mathtt{oddList}\,\mathtt{int}) \rightsquigarrow (\mathtt{eq}\,(\mathtt{oddList}\,\mathtt{int}))^{\kappa_{\mathtt{oddList}}\,\_} \rightsquigarrow^*$$

$$\mathtt{eq}\,\mathtt{int} \wedge \mathtt{eq}\,(\mathtt{evenList}\,\mathtt{int}) \rightsquigarrow (\mathtt{eq}\,\mathtt{int})^{\kappa_{\mathtt{int}}\,\_} \wedge \mathtt{eq}\,(\mathtt{evenList}\,\mathtt{int}) \rightsquigarrow^*$$

$$\underline{\mathtt{eq}\,(\mathtt{evenList}\,\mathtt{int})} \rightsquigarrow \ldots$$

The goal $\mathtt{eq}\,(\mathtt{evenList}\,\mathtt{int})$ *is simplified using the clause* $\kappa_{\mathtt{evenList}}$ *to goals* $\mathtt{eq}\,\mathtt{int}$ *and* $\mathtt{eq}\,(\mathtt{oddList}\,\mathtt{int})$*. The first of these is discarded using the clause* $\kappa_{\mathtt{int}}$*. Resolution continues using the clauses* $\kappa_{\mathtt{oddList}}$ *and* $\kappa_{\mathtt{int}}$*, resulting in the original goal* $\mathtt{eq}\,(\mathtt{evenList}\,\mathtt{int})$*. It is easy to see that such process could continue infinitely and that this goal constitutes a* cycle *(underlined above).*

As suggested by Lämmel and Peyton Jones (2005), the compiler can terminate an infinite inference process as soon as it detects all cycles. Moreover, it can also construct the corresponding proof term in a form of a recursive function.

Example 6.6 (Fu et al. (2016))

The infinite resolution trace in Example 6.5 is captured by a proof term

$$\nu\alpha.\kappa_{\mathtt{evenList}}\kappa_{\mathtt{int}}(\kappa_{\mathtt{oddList}}\kappa_{\mathtt{int}}\alpha)$$

where ν *is a fixed point operator that binds the variable* α*, which will be formally defined below. The intuitive reading of such proof term is that an infinite proof of the goal* $\mathtt{eq}\,(\mathtt{evenList}\,\mathtt{int})$ *exists, and that its shape is fully specified by the recursive function given by the term above.*

We say that the proof is given by *corecursive type class resolution*. Corecursive type class resolution is not inductively sound. However, as we prove in Section 6.3, it is *(universally) coinductively sound*, *i.e.* it is sound relative to the greatest Herbrand models.

Example 6.4 (Farka et al. (2016), Fu et al. (2016))

Consider the following mutually defined data structures that represent lists of odd and even length:

```
data OddList a   =  OCons a (EvenList a)
data EvenList a  =  Nil | ECons a (OddList a)
```

The lists give rise to the following instance declarations for the `Eq` *class:*

```
instance (Eq a, Eq (EvenList a)) ⇒  Eq (OddList a) where
    eq (OCons x xs) (OCons y ys) =  eq x y && eq xs ys


instance (Eq a, Eq (OddList a)) ⇒ Eq (EvenList a) where
    eq Nil          Nil          =  True
    eq (ECons x xs) (ECons y ys) =  eq x y && eq xs ys
    eq _            _            =  False
```

The following function triggers type class resolution in the Haskell compiler with goal `eq (evenList int)`*:*

```
    test :: Eq (EvenList Int) ⇒ Bool
    test = eq Nil Nil
```

For some data structures, resolving a type class instance that is necessary to type-check a function leads to a cycle. The goal that represents the type class instance is simplified, possibly in several steps, using instance declarations into subgoals such that one of the subgoals is identical with the original goal.

Example 6.5 (Logic program $\mathcal{P}_{EvenOdd}$, Farka et al. (2016))

Consider the Horn-clause representation of the type class instance declarations in Example 6.4:

$$\begin{aligned}
\kappa_{\texttt{oddList}}&: \ \texttt{eq}\,x \wedge \texttt{eq}\,(\texttt{evenList}\,x) \ \Rightarrow \texttt{eq}\,(\texttt{oddList}\,x)\\
\kappa_{\texttt{evenList}}&: \ \texttt{eq}\,x \wedge \texttt{eq}\,(\texttt{oddList}\,x) \ \Rightarrow \texttt{eq}\,(\texttt{evenList}\,x)\\
\kappa_{\texttt{int}}&: \ \Rightarrow \texttt{eq}\,\texttt{int}
\end{aligned}$$

$\boxed{\mathcal{S};\Gamma \vdash L : \texttt{kind}}$

$$\frac{\mathcal{S} \vdash \Gamma\ \text{ctx}}{\mathcal{S};\Gamma \vdash \texttt{type} : \texttt{kind}}\ \text{K-TY}$$

$$\frac{\mathcal{S};\Gamma \vdash A : \texttt{type} \qquad \mathcal{S};\Gamma, A \vdash L : \texttt{kind}}{\mathcal{S};\Gamma \vdash \Pi A.L : \texttt{kind}}\ \text{K-}\Pi\text{-INTRO}$$

Figure 2.6: Well-formedness of nameless kinds

$\boxed{\mathcal{S};\Gamma \vdash A : L}$

$$\frac{\mathcal{S} \vdash \Gamma\ \text{ctx} \qquad \alpha : L \in \mathcal{S}}{\mathcal{S};\Gamma \vdash \alpha : L}\ \text{T-CON}$$

$$\frac{\mathcal{S};\Gamma \vdash A : \texttt{type} \qquad \mathcal{S};\Gamma, A \vdash B : \texttt{type}}{\mathcal{S};\Gamma \vdash \Pi A.B : \texttt{type}}\ \text{T-}\Pi\text{-INTRO}$$

$$\frac{\mathcal{S};\Gamma \vdash A : \Pi B.L \qquad \mathcal{S};\Gamma \vdash M : B' \qquad \mathcal{S}^{-};\Gamma^{-} \vdash B \rightleftharpoons B' : \texttt{type}^{-}}{\mathcal{S};\Gamma \vdash AM : L[M]}\ \text{T-}\Pi\text{-ELIM}$$

Figure 2.7: Well-formedness of nameless types

The well-formedness of judgements are:

- $\mathcal{S};\Gamma \vdash L : \texttt{kind}$ for L a well-formed kind,
- $\mathcal{S};\Gamma \vdash A : L$ for A a well-formed type of a kind L,
- $\mathcal{S};\Gamma \vdash M : A$ for M a well-formed term of a type A,
- $\mathcal{S}^{-};\Delta \vdash A_1 \rightleftharpoons A_2 : \kappa$ for A_1 and A_2 being equal types of a simple kind κ,
- $\mathcal{S}^{-};\Delta \vdash M_1 \rightleftharpoons M_2 : \tau$ for M_1 and M_2 being equal terms of a simple kind τ, and
- $M \xrightarrow{\text{whr}} M'$ for a term M weak head reduces to term M'.

Definition 2.47

The well-formedness judgements for kinds, types, and terms are given by inference rules in Figures 2.6, 2.7, and 2.8.

The judgements defined in the above definition depend on the following two judgements:

- $\vdash \mathcal{S}$ sig for $\mathcal{S}$ well-formed signature, and
- $\mathcal{S} \vdash \Gamma$ ctx for Γ well-formed context in signature $\mathcal{S}$.

Definition 2.48

The well-formedness of signatures and contexts is given by inference rules in Fig-

$\boxed{\mathcal{S}; \Gamma \vdash M : A}$

$$\frac{\mathcal{S} \vdash \Gamma \text{ ctx} \qquad c : A \in \mathcal{S}}{\mathcal{S}; \Gamma \vdash c : A} \text{ CON}$$

$$\frac{\mathcal{S} \vdash \Gamma, A \text{ ctx}}{\mathcal{S}; \Gamma, A \vdash 0 : A\uparrow} \text{ ZERO}$$

$$\frac{\mathcal{S}; \Gamma \vdash \iota : A}{\mathcal{S}; \Gamma, B \vdash \sigma\iota : A\uparrow} \text{ SUCC}$$

$$\frac{\mathcal{S}; \Gamma \vdash A : \mathtt{type} \qquad \mathcal{S}; \Gamma, A \vdash M : B}{\mathcal{S}; \Gamma \vdash \lambda A.M : \Pi A.B} \Pi\text{-INTRO}$$

$$\frac{\mathcal{S}; \Gamma \vdash M : \Pi A.B \qquad \mathcal{S}; \Gamma \vdash N : A' \qquad \mathcal{S}^{-}; \Gamma^{-} \vdash A \rightleftharpoons A' : \mathtt{type}}{\mathcal{S}; \Gamma \vdash MN : B[N]} \Pi\text{-ELIM}$$

Figure 2.8: Well-formedness of nameless terms

ure 2.9.

Example 2.49

Let $\mathcal{S}$ be the signature we introduced in Example 2.39. Then $\lambda\,\mathtt{bool}\,.\,\mathtt{refl}\,0$ is a well-formed term of type $\Pi\,\mathtt{bool}\,.\,\equiv_{\mathtt{bool}}\,0$ in the signature $\mathcal{S}$ and an empty context. We show a part of a derivation of the judgement.

$$\dfrac{\dfrac{\mathcal{S} \vdash \cdot \ ctx}{\mathcal{S}; \cdot \vdash \mathtt{bool} : \mathtt{type}} \quad \dfrac{\dfrac{\dots}{\mathcal{S}; \cdot, \mathtt{bool} \vdash \mathtt{refl} : \Pi\,\mathtt{bool}\,.(\equiv_{\mathtt{bool}}\,0)\,0} \quad \dfrac{\dots}{\mathcal{S}; \cdot, \mathtt{bool} \vdash \mathtt{tt} : \mathtt{bool}} \quad \mathcal{S}; \cdot, \mathtt{bool} \vdash \mathtt{bool} \rightleftharpoons \mathtt{bool} : \mathtt{type}}{\mathcal{S}; \cdot, \mathtt{bool} \vdash \mathtt{refl}\,0 : (\equiv_{\mathtt{bool}}\,0)\,0}}{\mathcal{S}; \cdot \vdash \lambda\,\mathtt{bool}\,.\,\mathtt{refl}\,0 : \Pi\,\mathtt{bool}\,.\,\equiv_{\mathtt{bool}}\,0}$$

Ellipsis stand for omitted parts of the judgement, which can be constructed in a straightforward manner.

The above example demonstrates the fact that the well-formedness judgements of terms, types and kinds, of signatures and contexts, and the equality judgements are mutually recursively defined. In the next part we discuss judgements defining equality of objects in nameless LF.

Equality

We consider algorithmic equality as the notion of equality for its convenience in formalisation in Chapter 5. Equality of terms is informally decided as follows:

- two terms of function type are equal if their η-expansions are equal,

In literature, there restrictions are known as Paterson Conditions (Sulzmann et al., 2007). We include a formulation of Paterson Conditions on instance declarations as restrictions of Horn-clause programs for the purpose of referring to particular restrictions in the remainder of this chapter:

Definition 6.3 (Instance restrictions)

A logic program $\mathcal{P} = D_1, \dots, D_n$ adheres to Paterson Conditions if

1. *for all $i \neq j$, D_i does not unify with D_j, and*
2. *for all i, D_i does not contain existential variables.*

These restrictions guarantee that type class inference computes the *principal* (most general) type. Restrictions 1 and 2 of Definition 6.3 amount to deterministic inference by resolution, in which only one derivation is possible for every goal. Note that our characterisation of greatest Herbrand model (Proposition 2.35) employed the restriction 2. Restriction of SLD resolution to term matching means that no substitution is applied to a goal during inference, *i.e.* we prove the goal in an implicitly universally quantified form. In order to accompany for this restriction, we treat any variables in Haskell type class goals as Skolem constants in the calculus of proof-relevant resolution, *i.e.* as fresh constant symbols of the appropriate type. Such treatment allows us to stay within the model theory of Horn-clause logic we defined in Chapter 2.

It is a standard result that (as with SLD resolution) type class resolution is *inductively sound*, *i.e.* that it is sound relative to the least Herbrand models of logic programs (Lloyd, 1987). Moreover, in Section 6.2 we establish that it is also *universally inductively sound*, *i.e.* that if a formula A is proved by type class resolution, every ground instance of A is in the least Herbrand model of the given program. In contrast to SLD resolution, however, type class resolution is *inductively incomplete*, *i.e.* it is incomplete relative to least Herbrand models, even for the class of Horn clauses that is subject to restrictions 1 and 2 of Definition 6.3. For example, given a clause $\Rightarrow \mathtt{q}(\mathtt{f}(x))$ and a goal $\mathtt{q}(x)$, SLD resolution is able to find a proof (by instantiating x with $\mathtt{f}(x)$), but type class resolution fails.

Lämmel and Peyton Jones (2005) have suggested an extension to type class resolution that accounts for some non-terminating cases of type class resolution.

```
instance (Eq x, Eq y) ⇒ Eq (x, y) where
    eq (x1, y1) (x2, y2) = eq x1 x2 && eq y1 y2

instance Eq Int where
    eq x y = primitiveIntEq x y
```

In the Introduction, we observed that the instance declarations resemble Horn clauses in the following logic program:

Example 6.2 (Fu et al. (2016))

$$\kappa_{\texttt{pair}} : \ \texttt{eq}\, x \wedge \texttt{eq}\, y \ \Rightarrow \texttt{eq}\,(\texttt{pair}\, x\, y)$$

$$\kappa_{\texttt{int}} : \qquad\qquad \Rightarrow \texttt{eq}\,\texttt{int}$$

Resolving type class instance for type `( Int , Int)` then resembles SLD resolution of the goal `pair(int, int)`. Despite the apparent similarity of type class syntax and type class resolution to Horn clauses and SLD resolution they are not, however, identical. Type class and instance declarations are subject to certain restrictions. At the syntactic level, type class instance declarations correspond to a restricted form of Horn clauses, namely ones that do not *overlap* (*i.e.* whose heads do not unify); and that do not contain existential variables (*viz.* Definition 2.19). At the algorithmic level type class resolution corresponds to SLD resolution in which unification is restricted to term-matching; assuming there is a clause

$$B_1 \wedge \ldots \wedge B_n \Rightarrow A'$$

then a goal A' can be resolved with this clause only if A can be matched against A', *i.e.* if a substitution σ exists such that $A = \sigma A'$. In comparison, SLD resolution incorporates *unifiers*, as well as *matchers*, *i.e.* it also proceeds to resolve the above goal and clause in all the cases where $\sigma A = \sigma A'$ holds. Let us note at this point that, similar to the previous chapter, we understand that all program clauses are implicitly universally quantified.

$\boxed{\vdash \mathcal{S}\ \text{sig}}$

$$\frac{}{\vdash \cdot\ \text{sig}}$$

$$\frac{\vdash \mathcal{S}\ \text{sig} \qquad \mathcal{S}; \cdot \vdash L : \texttt{kind} \qquad a \notin \mathcal{S}}{\vdash \mathcal{S}, a : L\ \text{sig}}$$

$$\frac{\vdash \mathcal{S}\ \text{sig} \qquad \mathcal{S}; \cdot \vdash A : \texttt{type} \qquad c \notin \mathcal{S}}{\vdash \mathcal{S}, c : A\ \text{sig}}$$

$\boxed{\mathcal{S} \vdash \Gamma\ \text{ctx}}$

$$\frac{\vdash \mathcal{S}\ \text{sig}}{\mathcal{S} \vdash \cdot\ \text{ctx}}$$

$$\frac{\mathcal{S} \vdash \Gamma\ \text{ctx} \qquad \mathcal{S}; \Gamma \vdash A : \texttt{type}}{\vdash \mathcal{S}; \Gamma, A\ \text{ctx}}$$

Figure 2.9: Well-formedness of signatures and contexts

- two terms of base type are equal if their weak head-normal forms are equal, and
- two terms of base type in weak head-normal form are equal if their heads are equal and the corresponding arguments are equal.

This definition of equality required discriminating between objects of function type and objects of base type. However, information about terms is not necessary and the equality can be defined using simple types. We proceed with definition of *weak head-reduction.*

Definition 2.50

Weak head reduction is given by inference rules in Figure 2.10.

The definition of equality follows the structure we gave in the informal account. The type-directed phase is given as a judgement called *algorithmic equality.* It carries out reduction of either of the terms that are subject to equality judgement, η-expansion of terms of function type, and reduction to equality of weak head-normal forms. Equality of weak head-normal forms is given as a judgement called *structural equality.*

Definition 2.51

Algorithmic equality of terms and structural equality of terms are defined by inference rules in Figures 2.11 and 2.12 respectively.

$\boxed{M \xrightarrow{\text{whr}} M'}$

$$\frac{}{(\lambda A.M)N \xrightarrow{\text{whr}} M[N]} \qquad \frac{M \xrightarrow{\text{whr}} M'}{MN \xrightarrow{\text{whr}} M'N'}$$

Figure 2.10: Weak head reduction of terms

$\boxed{\mathcal{S}^{-}; \Delta \vdash M \Leftrightarrow M' : \tau}$

$$\frac{M \xrightarrow{\text{whr}} M' \qquad \mathcal{S}^{-}; \Delta \vdash M' \Leftrightarrow N : \tau}{\mathcal{S}^{-}; \Delta \vdash M \Leftrightarrow N : \tau}$$

$$\frac{N \xrightarrow{\text{whr}} N' \qquad \mathcal{S}^{-}; \Delta \vdash M \Leftrightarrow N' : \tau}{\mathcal{S}^{-}; \Delta \vdash M \Leftrightarrow N : \tau}$$

$$\frac{\mathcal{S}^{-}; \Delta \vdash M \leftrightarrow N : \tau}{\mathcal{S}^{-}; \Delta \vdash M \Leftrightarrow N : \tau}$$

$$\frac{\mathcal{S}^{-}; \Delta, \tau_1 \vdash (M\uparrow)\,0 \Leftrightarrow (N\uparrow)\,0 : \tau_2}{\mathcal{S}^{-}; \Delta \vdash M \Leftrightarrow N : \tau_1 \to \tau_2}$$

Figure 2.11: Algorithmic equality of terms

The notion of equality of types is simplified due to the fact that we do not consider abstraction on the level of types. The absence of abstraction means there is no need for weak head reduction on the level of types and equality comprises decomposing of function type into equality of types and decomposing of type application into equality of types and equality of term arguments. We refer to the equality as *weak algorithmic equality.*

Definition 2.52

Weak algorithmic equality of types is defined by inference rules in Figure 2.13.

We conclude this section with an example concerning equality.

Example 2.53

Consider the signature $\mathcal{S}$ we introduced in Example 2.39. Then the term $(\lambda\,\texttt{bool}\,.\,\texttt{refl}\;0)\;\texttt{tt}$ *is equal to term* `refl tt` *in the simple signature $\mathcal{S}^{-}$ and an empty simple context. The following is a derivation of the equality judgement.*

6 | Type Class Resolution

In this chapter we demonstrate a use of proof-relevant resolution for the purpose of semantical analysis of programming languages. Our use case is type class resolution. Type class resolution is commonly understood to correspond to first-order Horn-clause resolution. Recently, several corecursive extensions to type classes have been proposed (Fu and Komendantskaya, 2017, Fu et al., 2016, Lämmel and Peyton Jones, 2005). The corecursive type-class resolution calculus of Fu and Komendantskaya (2017) falls outside of Horn-clause logic as it in fact uses implicational shape of goals to handle coinductive assumptions. Hence, in this chapter we employ both Horn-clause logic and the logic of hereditary Harrop formulae to capture type-class resolution. We expose, in a compositional manner, the calculus of type class resolution and, as its extensions, two calculi of corecursive type class resolution. We show that type class resolution is inductively sound with respect to least Herbrand models; that the corecursive extensions are coinductively sound with respect to greatest Herbrand models of logic programs; and that the corecursive extensions are inductively unsound. Further, we establish incompleteness results for fragments of the proof system.

6.1 Type Class Mechanism

In this section we summarise the type class mechanism. Recall our running example that we used in the Introduction.

Example 6.1 (Farka et al. (2016), Fu et al. (2016), Hall et al. (1996))

The the class `Eq` *and its instances for pairs and integers are defined as follows:*

```
class Eq a where
    eq :: a → a → Bool
```

$\boxed{\mathcal{S}^-; \Delta \vdash M \leftrightarrow N : \tau}$

$$\frac{\vdash \mathcal{S}\ \text{sig}}{\mathcal{S}^-; \Delta, \tau \vdash 0 \leftrightarrow 0 : \tau}$$

$$\frac{\mathcal{S}^-; \Delta \vdash \iota \leftrightarrow \iota' : \tau}{\mathcal{S}^-; \Delta, \tau' \vdash \sigma\iota \leftrightarrow \sigma\iota' : \tau}$$

$$\frac{\vdash \mathcal{S}\ \text{sig} \qquad c : \tau \in \mathcal{S}^-}{\mathcal{S}^-; \Delta \vdash c \leftrightarrow c : \tau}$$

$$\frac{\mathcal{S}^-; \Delta \vdash M_1 \leftrightarrow N_1 : \tau_2 \to \tau_1 \qquad \mathcal{S}^-; \Delta \vdash M_2 \Leftrightarrow N_2 : \tau_2}{\mathcal{S}^-; \Delta \vdash M_1 M_2 \leftrightarrow N_1 N_2 : \tau_1}$$

Figure 2.12: Structural equality of terms

$\boxed{\mathcal{S}^-; \Delta \vdash A \rightleftharpoons A' : \kappa}$

$$\frac{\vdash \mathcal{S}\ \text{sig} \qquad \alpha : \kappa \in \mathcal{S}^-}{\mathcal{S}^-; \Delta \vdash \alpha \rightleftharpoons \alpha : \kappa}$$

$$\frac{\mathcal{S}^-; \Delta \vdash A \rightleftharpoons B : \tau \to \kappa \qquad \mathcal{S}^-; \Delta \vdash M \Leftrightarrow N : \tau}{\mathcal{S}^-; \Delta \vdash AM \rightleftharpoons BN : \kappa}$$

$$\frac{\mathcal{S}^-; \Delta \vdash A_1 \rightleftharpoons B_1 : \mathtt{type} \qquad \mathcal{S}^-; \Delta, (A_1)^- \vdash (A_2\uparrow) \rightleftharpoons (B_2\uparrow) : \mathtt{type}}{\mathcal{S}^-; \Delta \vdash (\Pi A_1.A_2) \rightleftharpoons (\Pi B_1.B_2) : \mathtt{type}}$$

Figure 2.13: Weak algorithmic equality of types

$$\dfrac{\dfrac{}{(\lambda\,\mathtt{bool}\,.\,\mathtt{refl}\ 0)\ \mathtt{tt}} \xrightarrow{whr} \mathtt{refl}\ \mathtt{tt} \qquad \dfrac{\dfrac{\dfrac{\dfrac{\dots}{\vdash \mathcal{S}\ \mathit{sig}} \qquad \mathtt{refl} : \equiv_{\mathtt{bool}} \in \mathcal{S}^-}{\mathcal{S}^-;\cdot \vdash \mathtt{refl} \leftrightarrow \mathtt{refl} : \mathtt{bool} \to \equiv_{\mathtt{bool}}} \qquad \dfrac{\dfrac{\dots}{\vdash \mathcal{S}\ \mathit{sig}} \qquad \mathtt{tt} : \mathtt{bool} \in \mathcal{S}^-}{\mathcal{S}^-;\cdot \vdash \mathtt{tt} \leftrightarrow \mathtt{tt} : \mathtt{bool}}}{\mathcal{S}^-;\cdot \vdash \mathtt{refl}\ \mathtt{tt} \leftrightarrow \mathtt{refl}\ \mathtt{tt} : \equiv_{\mathtt{bool}}}}{\mathcal{S}^-;\cdot \vdash \mathtt{refl}\ \mathtt{tt} \Leftrightarrow \mathtt{refl}\ \mathtt{tt} : \equiv_{\mathtt{bool}}}}{\mathcal{S}^-;\cdot \vdash (\lambda\,\mathtt{bool}\,.\,\mathtt{refl}\ 0)\ \mathtt{tt} \Leftrightarrow \mathtt{refl}\ \mathtt{tt} : \equiv_{\mathtt{bool}}}$$

We omit derivations of well-formedness of the signature for the sake of brevity. This is denoted by ellipsis.

of this interpretation of generated goals and programs makes it feasible to adjust the refinement calculus for different type theories.

Type inference in type theory with dependent types is an undecidable problem (Dowek, 1993). However, a relaxation thereof, type refinement, is common in existing languages based in type theory with dependent types. A bi-directional type inference algorithm that depends on constraint solving has been implemented for the Agda interactive prover (Norell, 2007). More recent work by Asperti et al. (2012) on type inference in type theory for the Matita theorem prover also employs a bi-directional approach. However, this algorithm is based on rewriting rather than constraint solving. A similar approach to refinement has been taken by Brady (2013) in the dependently typed programming language Idris. Pientka (2013) presented a type reconstruction algorithm for LF and Beluga.

Currently implemented systems (*cf.* Pientka, 2013) make use of a bidirectional approach to type checking. That is, there are separate type checking and type synthesis phases. The key difference between these systems and our own work is that we do not explicitly discuss bidirectionality. Combining this with a clear identification of atomic formulae with judgements, and Horn clauses with inference rules, in our opinion, makes the presentation significantly more accessible. However, bidirectionality in our system is still implicitly present, albeit postponed to the resolution phase. As future work, we intend to analyse structural resolution (Fu and Komendantskaya, 2017) for the generated goals. We intend to show that the matching steps in the resolution correspond to type checking in the bidirectional approach whereas resolution steps by unification correspond to type synthesis.

(Farka et al., 2018) and the source code can be found online[1]. Most importantly, we formalised definitions of nameless LF. The exported Coq definitions were then used in formalisation of decidability of the refinement calculus. An implementation of translation from the extended language to logic programs and goals and an implementation of a function interpreting proof terms was obtained from formal proofs via code extraction into OCaml. A parser was extracted from the formalisation of the grammar. The translation outputs logic programs and goals suitable for an external resolution engine. We used a thin layer of OCaml code to connect the generated code with ELPI (Dunchev et al., 2015), a λProlog interpreter that we use to simulate proof-relevant resolution in the universal fragment of Horn-clause logic.

5.5 Related Work

Although we specifically work with LF (Harper et al., 1993, Harper and Pfenning, 2005), our work relates in general to type inference in typed λ-calculi. A standard approach to type inference in the simply typed lambda calculus is the HM(X) algorithm (Odersky et al., 1999). Essentially, this algorithm traverses the abstract syntax tree and generates constraints in a specific constraint domain X. Then, a solver for X is employed. Stuckey and Sulzmann (2002) presented the type inference algorithm HM(X) in terms of constraint logic programming (Sulzmann and Stuckey, 2008). Another modification of the constraint solving approach to HM type inference is the inference algorithm OutsideIn(X) by (Vytiniotis et al., 2011), which has been used for type inference in the Glasgow Haskell Compiler (GHC). Ideas underlying our work originate in the work of Stuckey and Sulzmann (2002) on HM(X) type inference as (constraint) logic programming. There are two key differences. First, in our work we consider dependent types. Other approaches, such as that of Sulzmann and Stuckey do not give a motivation for the shape of generated logic goals and programs.

We make explicit that atomic formulae represent judgements of the type theory and that the program originates on one hand from inference rules of the type theory and on the other from a signature of a term. We believe that a clear identification

[1] https://github.com/frantisekfarka/slepice

3 | Proof-Relevant Resolution

In this chapter, we introduce the theory of proof relevant resolution. We develop the theory in several steps. First, we give big-step (uniform proof-relevant) operational semantics and a small-step operational semantics of Horn-clause logic. We state soundness of the small-step semantics relative to the big-step semantics. Then we introduce the language of hereditary Harrop formulae by extending goals and definite clauses of Horn-clause logic. We extend the big-step and the small-step semantics accordingly.

3.1 Horn-Clause Logic

First, we extend the notion of programs. Programs are collections of clauses that are annotated with atomic proof-term symbols in a set $\mathcal{K}$. We use κ to denote symbols in $\mathcal{K}$.

Definition 3.1 (Programs)

$$\mathcal{P} \ni \mathcal{P} \qquad := \cdot \mid \mathcal{P}, \kappa : D \qquad \textit{programs}$$

We use notation $\mathcal{P}_1, \mathcal{P}_2$ for a program $\mathcal{P}_1, \kappa_1 : D_1, \ldots, \kappa_n : D_n$ where $\mathcal{P}_2 = \cdot, \kappa_1 : D_1, \ldots, \kappa_n : D_n$. Intuitively, we assume that programs consists only of well-formed definite clauses. This formally translates into a well-formedness judgement for programs.

Definition 3.2

Well-formedness of programs $\mathcal{S} \vdash \mathcal{P}$ is given by inference rules in Figure 3.1.

We implicitly assume that all proof-term symbols in a program are unique.

$\boxed{\mathcal{S} \vdash \mathcal{P}}$

$$\frac{\vdash \mathcal{S}}{\mathcal{S} \vdash \cdot} \qquad \frac{\mathcal{S} \vdash \mathcal{P} \qquad \mathcal{S}; \cdot \vdash D : \mathtt{o}}{\mathcal{S} \vdash \mathcal{P}, \kappa : D}$$

Figure 3.1: Well formedness of programs

Example 3.3

Recall Example 1.3. The program

$$\mathcal{P}_{\mathtt{Pair}} = \cdot, \kappa_{\mathtt{pair}} : \forall x : \mathtt{int}. \forall y : \mathtt{int}. \mathtt{eq}\, x \Rightarrow \mathtt{eq}\, y \Rightarrow \mathtt{eq}\, (\mathtt{pair}\, x\, y), \kappa_{\mathtt{int}} : \mathtt{eq}(\mathtt{int})$$

is a program . $\mathcal{P}_{\mathtt{Pair}}$ consists of clauses that are well-formed in signature $\mathcal{S}_{\mathtt{Pair}}$ and is well-formed, or $\mathcal{S}_{\mathtt{Pair}} \vdash \mathcal{P}_{\mathtt{Pair}}$.

Note that the well-formedness judgement for programs admits implicit syntactic validity property:

Proposition 3.4

If $\mathcal{S} \vdash \mathcal{P}$ then $\vdash \mathcal{S}$.

Proof. By induction on derivation of the judgement. □

Further, the properties of Propositions 2.14 and 2.22 concerning weakening of signature can be extended to programs.

Proposition 3.5

If $\mathcal{S}_1, \mathcal{S}_2; \Gamma \vdash \mathcal{P}$ and $\vdash \mathcal{S}_1, c : A, \mathcal{S}_2$ then $\mathcal{S}_1, c : A, \mathcal{S}_2; \Gamma \vdash \mathcal{P}$.

Proof. By induction on the program using Proposition 2.22, Part 1. □

Since programs consists of definite clauses that are well-formed in an empty context, programs and program clauses are stable under substitution:

Proposition 3.6

1. *If $\mathcal{S} \vdash \mathcal{P}_1, \kappa : D, \mathcal{P}_2$ then $\mathcal{S} \vdash \mathcal{P}_1, \kappa : D[M/x], \mathcal{P}_2$.*

Proof. By induction on the program using Proposition 2.23. □

Theorem 5.25 (Soundness of interpretation)

Let M be a term in the extended syntax with signature $\mathcal{S}$. Let $\mathcal{P}$ and G_M be a program and a goal such that $\mathcal{S}, \cdot \vdash (G_M | A)$ and $\mathcal{S} \vdash_{Prog} \mathcal{P}$ respectively. Let ρ, R be a substitution and a proof term assignment for proof term e computed by proof-relevant resolution such that $\mathcal{S}; \mathcal{P} \vdash \cdot \mid G_M \rightsquigarrow \cdot \mid e$. Then if there is a solution for a well-formed term, then there are solutions (ρ', R') and (ρ'', R'') such that $(\rho', R')M$ is a well-formed term and

$$(\rho'', R'')((\rho, \ulcorner R \urcorner)M) = (\rho', R')M$$

Proof. Generalise the statement of the theorem for an arbitrary well-formed context Γ. By simultaneous induction on derivation of the well-formedness judgement of $(\rho', R')M$ and derivation of $\mathcal{S}; \mathcal{P} \vdash \cdot \mid G \rightsquigarrow \cdot \mid e$. The theorem follows from the generalisation. □

Theorem 5.25 guarantees that the refinement computed in Examples 5.18–5.24 is well typed in the internal language. That is, there is a derivation of the following judgement:

$$\mathcal{S}; \cdot, \mathtt{maybe}_{\mathtt{A}}\, \mathtt{tt} \vdash \mathtt{elim}_{\mathtt{maybe}_{\mathtt{A}}}\ \mathtt{tt}\ 0\ (\lambda\, \mathtt{tt} \equiv_{\mathtt{bool}} \mathtt{ff}\, .\, \mathtt{elim}_{\equiv_{\mathtt{bool}}}\, 0)$$

$$: (\mathtt{tt} \equiv_{\mathtt{bool}} \mathtt{tt}) \to \mathtt{A} \to \mathtt{A}) \to \mathtt{A}$$

We omit the actual derivation of the judgement. However, note that it can be easily reconstructed in a similar way as the intended interpretation of proof terms is computed in Definition 5.23. For example, in case of our running example, the subterm δ_{eqT} of the proof term gives derivation of the definitional equality that is necessary to verify application of $\mathtt{elim}_{\equiv_{\mathtt{bool}}}$ to index 0.

5.4 Implementation

We formalised the results in this chapter using the Ott tool (Sewell et al., 2010) and the Coq theorem prover. The formalisation was reported in a published paper

the computed substitution can be used directly. The interpretation of the computed assignment of proof terms depends on assignment of atomic proof term symbols in the program $\mathcal{P}_e$. We define a mapping that gives the intended interpretation:

Definition 5.23 (Interpretation of proof terms)
We define interpretation of proof terms $\ulcorner - \urcorner : \mathrm{PT} \to T$ *as follows:*

$$\ulcorner \kappa_\sigma \delta \delta_\iota \delta' \urcorner = \sigma \ulcorner \delta_\iota \urcorner$$

$$\ulcorner \kappa_{proj} \delta_\iota \urcorner = \ulcorner \delta_\iota \urcorner$$

$$\ulcorner \kappa_{T\text{-}elim} \delta_M \delta_N \delta \delta' \urcorner = \ulcorner \delta_M \urcorner \ulcorner \delta_N \urcorner$$

$$\ulcorner \kappa_{T\text{-}intro} \delta_A \delta \delta_B \urcorner = \Pi \ulcorner \delta_A \urcorner . \ulcorner \delta_B \urcorner$$

$$\ulcorner \kappa_{t\text{-}elim} \delta_A \delta_M \delta \delta' \urcorner = \ulcorner \delta_A \urcorner \ulcorner \delta_M \urcorner$$

$$\ulcorner \kappa_{t\text{-}intro} \delta_A \delta \delta_M \delta' \urcorner = \lambda \ulcorner \delta_A \urcorner . \overrightarrow{\ulcorner \delta_M \urcorner}$$

$$\ulcorner \kappa_0 \urcorner = 0$$

$$\ulcorner \kappa_c \urcorner = c$$

$$\ulcorner \kappa_a \urcorner = a.$$

We extend $\ulcorner - \urcorner$ to assignments of identifiers to subterms of a proof term that are bound by the identifiers. We use $\ulcorner e \urcorner$ to denote this assignment.

Example 5.24
In Example 5.22, the computed proof term bound to δ_b *is interpreted as follows:*

$$\ulcorner \kappa_{\mathtt{elim}} \kappa_{\mathtt{elim}_{\equiv_{\mathtt{bool}}}} \kappa_0 \delta_{eq_T} \urcorner = \mathtt{elim}_{\equiv_{\mathtt{bool}}}\, 0$$

Hence, the original problem is refined to $\mathtt{elim}_{\mathtt{maybe}_{\mathtt{A}}}\ \mathtt{tt}\ 0\ (\lambda\, \mathtt{A}\, .\, \mathtt{elim}_{\equiv_{\mathtt{bool}}}\, 0)$ *while the computed type is* $((\mathtt{tt} \equiv_{\mathtt{bool}} \mathtt{tt}) \to \mathtt{A} \to \mathtt{A}) \to \mathtt{A}$.

Finally, the above interpretation allows us to state the soundness of our system:

$\boxed{\mathcal{S}; \mathcal{P} \longrightarrow e : G}$

$$\frac{\mathcal{S}; \mathcal{P} \longrightarrow e : G[M/x] \qquad \mathcal{S}; \emptyset \vdash M : A}{\mathcal{S}; \mathcal{P} \longrightarrow \langle M, e \rangle : \exists x : A.G}$$

Figure 3.2: Right introduction rule

3.1.1 Big-step operational semantics

Now we come to definition of the big-step semantics. Since our semantics is proof-relevant, we need to provide a definition of proof terms:

Definition 3.7 (Proof terms)

$$\mathrm{PT} \ni e \qquad := \kappa \mid e\ e \mid \langle M, e \rangle \qquad \textit{proof terms}$$

Proof terms consist of a proof-term symbol in $\mathcal{K}$, an application, and of an existential witness $\langle -, - \rangle$. We use the identifier e to denote proof terms in PT

The semantics we give is essentially the semantics of uniform proofs (Miller and Nadathur, 2012) that is instrumented with proof terms. That is, there are two judgements, $\mathcal{S}; \mathcal{P} \longrightarrow e : G$, and $\mathcal{S}; \mathcal{P} \overset{e_1 : D_1}{\longrightarrow} e : A$. These judgements utilise the idea of "logic-formulae as search". The first judgement correspond to right-introduction rules of the logical connective $\exists$ in sequent calculus for intuitionistic logic and decomposes the goal that is the subject of the judgement. When the goal cannot be further decomposed (*i.e.* it is an atomic formula), a program clause is selected and the second judgement is used to decompose the selected clause to subgoals. The second judgement, called *backchaining*, corresponds to left-introduction rules of connectives $\Rightarrow$ and $\forall$ and decomposes the selected program clause into subgoals. We say that the proof term e_1 and the clause D_1 *annotate* the judgement $\mathcal{S}; \mathcal{P} \overset{e_1 : D_1}{\longrightarrow} e : G$. Conversely, e_1 is the annotating proof term and D_1 is the annotating clause.

Definition 3.8 (Operational semantics, big-step)
The judgements $\mathcal{S}; \mathcal{P} \longrightarrow e : G$ *and* $\mathcal{S}; \mathcal{P} \overset{e_1 : D_1}{\longrightarrow} e : A$ *are given in Figures 3.2 and 3.3 respectively.*

Let us illustrate the big-step semantics using a simple example. We introduce

$\boxed{\mathcal{S};\mathcal{P} \xrightarrow{e':D} e : A}$

$$\frac{}{\mathcal{S};\mathcal{P} \xrightarrow{e:A} e : A}$$

$$\frac{\mathcal{S};\mathcal{P} \longrightarrow e_1 : A_1 \qquad \mathcal{S};\mathcal{P} \xrightarrow{e\,e_1:D} e_2 : A_2}{\mathcal{S};\mathcal{P} \xrightarrow{e:A_1 \Rightarrow D} e_2 : A_2}$$

$$\frac{\mathcal{S};\mathcal{P} \xrightarrow{\kappa:D} e : A \qquad \kappa : D \in \mathcal{P}}{\mathcal{S};\mathcal{P} \longrightarrow e : A}$$

$$\frac{\mathcal{S};\mathcal{P} \xrightarrow{e:D[M/x]} e_2 : A_2 \qquad \mathcal{S};\cdot \vdash M : A_1}{\mathcal{S};\mathcal{P} \xrightarrow{e:\forall x:A_1.D} e_2 : A_2}$$

Figure 3.3: Backchaining rules

a signature that allows us to encode facts about natural numbers. The signature contains function symbols z and s that denote zero and successor respectively. The signature further contains a predicate *nat* that has one argument and denotes that its argument is a natural number. We discuss several goals that are formed in this signature and show their big-step resolution derivations.

Example 3.9

Let $\mathcal{S}$ be the following signature:

$$\mathcal{S} = a : \mathtt{type}, z : a, s : a \to a, nat : a \to \mathtt{o}$$

The predicate nat is given an interpretation by the following program:

$$\mathcal{P} = \kappa_z : nat\, z,$$

$$\kappa_s : \forall x : a.nat\, x \Rightarrow nat\, (s\, x)$$

First, consider a well-formed goal $nat\, z$. The goal is resolved with the proof term κ_z:

$$\frac{\dfrac{}{\mathcal{S};\mathcal{P} \xrightarrow{\kappa_z:nat\, z} \kappa_z : nat\, z} \qquad \kappa_z : nat\ z \in \mathcal{P}}{\mathcal{S};\mathcal{P} \longrightarrow \kappa_z : nat\, z}$$

Similarly, a well-formed goal $nat\, (s\, z)$ is resolved with the proof term $\kappa_s\, \kappa_z$.

resolves as follows:

$$\cdot \mid G \rightsquigarrow^* \quad \cdot \mid \delta_b : term(?_b, \mathtt{A}, (\Gamma_1, \mathtt{tt} \equiv_{\mathtt{bool}} \mathtt{ff})$$

The computed substitution assigns $(\Pi(\Pi(\mathtt{tt} \equiv_{\mathtt{bool}} \mathtt{tt}).(\Pi\, \mathtt{A}\, .\, \mathtt{A})).\, \mathtt{A}$ to the logic variable $?_{B_7}$, which occurs in G. We now show the trace for the remaining goal $?_b : term(?_b, \mathtt{A}, \mathtt{tt} \equiv_{\mathtt{bool}} \mathtt{ff} : \Gamma_1)$. Given the clauses of Example 5.20, a resolution trace that computes a proof term that is bound to identifier δ_b can be given as follows:

$$?_a : t \mid term\, ?_a\ \mathtt{A}\, (\Gamma_1, \mathtt{tt} \equiv_{\mathtt{bool}} \mathtt{ff}) \rightsquigarrow$$

$$?_a : t \mid (term\, ?_a\ \mathtt{A}\, (\Gamma_1, \mathtt{tt} \equiv_{\mathtt{bool}} \mathtt{ff}))^{\kappa_{\mathtt{elim}}:\_} \rightsquigarrow$$

$$?_{a_1} : t, ?_{a_2} : t, ?_{a_2'} : t, ?_A : T, ?_{B'} : T \mid \kappa_{\mathtt{elim}}\, (term\, (?_{a_1}\, ?_{a_2})\, (\Pi ?_A.?_{B'})\, (\Gamma_1, \mathtt{tt} \equiv_{\mathtt{bool}} \mathtt{ff}) \wedge$$
$$term\, ?_{a_2'}\, ?_A\, \Gamma_1 \wedge eq_T\, ?_{B_4}\, ?_{B'}\ \mathtt{type}\, (\Gamma_1, \mathtt{tt} \equiv_{\mathtt{bool}} \mathtt{ff})) \rightsquigarrow$$

$$?_{a_1} : t, ?_{a_2} : t, ?_{a_2'} : t, ?_A : T, ?_{B'} : T \mid \kappa_{\mathtt{elim}}\, ((term\, (?_{a_1}\, ?_{a_2})\, (\Pi ?_A.?_{B'})\, (\Gamma_1, \mathtt{tt} \equiv_{\mathtt{bool}} \mathtt{ff}))^{\kappa_{\mathtt{elim}_{\equiv_{\mathtt{bool}}}}} \wedge$$
$$term\, ?_{a_2'}\, ?_A\, \Gamma_1 \wedge eq_T\, ?_{B_4}\, ?_{B'}\ \mathtt{type}\, (\Gamma_1, \mathtt{tt} \equiv_{\mathtt{bool}} \mathtt{ff})) \rightsquigarrow$$

$$?_{a_2'} : t \mid \kappa_{\mathtt{elim}}\, \kappa_{\mathtt{elim}_{\equiv_{\mathtt{bool}}}}\, ((term\, ?_{a_2'}\, (\mathtt{tt} \equiv_{\mathtt{bool}} \mathtt{ff})\, (\Gamma_1, \mathtt{tt} \equiv_{\mathtt{bool}} \mathtt{ff}) \wedge$$
$$eq_T\, (\mathtt{tt} \equiv_{\mathtt{bool}} \mathtt{ff})\, (\mathtt{tt} \equiv_{\mathtt{bool}} \mathtt{ff})\, \mathtt{type}\, (\Gamma_1, \mathtt{tt} \equiv_{\mathtt{bool}} \mathtt{ff}))) \rightsquigarrow$$

$$?_{a_2'} : t \mid \kappa_{\mathtt{elim}}\, \kappa_{\mathtt{elim}_{\equiv_{\mathtt{bool}}}}\, (((term\, ?_{a_2'}\, (\mathtt{tt} \equiv_{\mathtt{bool}} \mathtt{ff})\, (\Gamma_1, \mathtt{tt} \equiv_{\mathtt{bool}} \mathtt{ff}))^{\kappa_0} \wedge$$
$$eq_T\, (\mathtt{tt} \equiv_{\mathtt{bool}} \mathtt{ff})\, (\mathtt{tt} \equiv_{\mathtt{bool}} \mathtt{ff})\, \mathtt{type}\, (\Gamma_1, \mathtt{tt} \equiv_{\mathtt{bool}} \mathtt{ff}))) \rightsquigarrow$$

$$\cdot \mid \kappa_{\mathtt{elim}}\, \kappa_{\mathtt{elim}_{\equiv_{\mathtt{bool}}}}\, \kappa_0$$
$$(eq_T\, (\mathtt{tt} \equiv_{\mathtt{bool}} \mathtt{ff})\, (\mathtt{tt} \equiv_{\mathtt{bool}} \mathtt{ff})\ \mathtt{type}\, (\Gamma_1, \mathtt{tt} \equiv_{\mathtt{bool}} \mathtt{ff})) \rightsquigarrow^*$$

$$\cdot \mid \kappa_{\mathtt{elim}}\, \kappa_{\mathtt{elim}_{\equiv_{\mathtt{bool}}}}\, \kappa_0\, \delta_{\mathtt{eq}_T}$$

Above, we omit writing full derivation of the last goal but denote the result as δ_{eq_T}. The assignment to the logic variable $?_A$ is $\mathtt{A}$ and the subterm of the computed proof term that is bound to δ_b is $\kappa_{\mathtt{elim}}\kappa_{\mathtt{elim}_{\equiv_{\mathtt{bool}}}}\kappa_0\delta_{eq_T}$ where the subterm δ_{eq_T} is a witness of the appropriate type equality.

Since we have used types and terms of nameless LF to define our atomic formulae,

into play.

5.3 Proof-Relevant Resolution and Soundness

As we have shown in Example 5.8, we utilise a proof-relevant resolution we described in Chapter 3 as the inference engine for solving refinement problems. However, for the purpose of this chapter we extend the syntax of goals in such a way as to allow us identify subterms of the computed proof term that correspond to atomic goals. This will allow us to refer to these subterms for the purpose of interpretation of proof terms as well-formedness judgements of the internal language. We assume an infinite set Δ of proof term identifiers. We use identifiers δ, δ_1 *etc.* to denote identifiers in Δ. We alter definition of goals such that an atomic goal is assigned with an identifier in Δ.

Definition 5.21

$$\mathcal{G} \ni G \qquad := \delta : A \mid \ldots \qquad \textit{goals}$$

In the course of resolution, when an atomic subgoal $\delta : A$ is resolved with a subterm e of the proof term, we use δ to refer to e and we say that δ is bound to e. We omit δ in notation of goals where this identifier is not used later for referring to the computed subterm.

Assume that G and $\mathcal{P}$ are a goal and a program that originate from a refinement problem M in signature $\mathcal{S}$. An answer substitution for G computed by $\mathcal{P}$ provides a solution to the type-level metavariables in M. Similarly the computed assignment of proof terms to proof variables provides a solution to the term-level metavariables in M.

We continue with our running example, building upon Examples 5.11–5.20.

Example 5.22 (Proof-relevant Resolution Trace)

The resolution trace of our example is rather long, and we show only a fragment. Suppose that in several small steps, denoted by $\rightsquigarrow^*$, the goal G given in Example 5.18

$$\dfrac{\dfrac{\vdots}{\mathcal{S};\mathcal{P} \longrightarrow \kappa_z : nat\, z} \qquad \dfrac{}{\mathcal{S};\mathcal{P} \overset{\kappa_s\, \kappa_z : nat\,(s\,z)}{\longrightarrow} \kappa_s\, \kappa_z : nat\,(s\,z)}}{\dfrac{\mathcal{S};\mathcal{P} \overset{\kappa_s : -}{\longrightarrow} \kappa_s\, \kappa_z : nat\,(s\,z) \qquad \mathcal{S};\cdot \vdash z : a}{\dfrac{\mathcal{S};\mathcal{P} \overset{\kappa_s : -}{\longrightarrow} \kappa_s\, \kappa_z : nat\,(s\,z) \qquad \kappa_s : \forall x : a.nat\, x \Rightarrow nat\,(s\,x) \in \mathcal{P}}{\mathcal{S};\mathcal{P} \longrightarrow \kappa_s\, \kappa_z : nat\,(s\,z)}}}$$

Note that we omit resolution of the goal $nat\, z$ as it was given above. We abbreviate the annotating clause $\kappa_s : \forall x : a.nat\, x \Rightarrow nat\,(s\,x)$ to $\kappa_s : -$. Finally, let us consider a goal $\exists x : a.nat\,(s\,x)$ that contains an existentially quantified variable. Using the previous two derivations, the big-step resolution of the goal, that is a derivation of judgement $\mathcal{S};\mathcal{P} \longrightarrow e : \exists x : a.nat\,(s\,x)$ is carried out as follows:

$$\dfrac{\dfrac{\vdots}{\mathcal{S};\mathcal{P} \longrightarrow \kappa_s\, \kappa_z : nat\,(s\,z)} \qquad \dfrac{z : a \in \mathcal{S}}{\mathcal{S};\cdot \vdash z : a}}{\mathcal{S};\mathcal{P} \longrightarrow \langle z, \kappa_s\, \kappa_z \rangle : \exists x : a.nat\,(s\,x)}$$

The proof term e that witnesses resolution of the goal $\exists x : a.nat\,(s\,x)$ in signature $\mathcal{S}$ and program $\mathcal{P}$ is $e = \langle z, \kappa_s\, \kappa_z \rangle$.

Let us discuss an example with a program clause that contains a nested universally quantified variable. In terminology of Miller and Nadathur (2012), in this particular case this is an "essentially existentially quantified variable", that is a universally quantified that gets instantiated in the course of resolution.

Example 3.10 (Essentially existential)

Consider a signature $\mathcal{S}$:

$$\mathcal{S} = a : \mathtt{type}, p : \mathtt{o}, q : a \to \mathtt{o}, c : a$$

A program $\mathcal{P}$ consists of two clauses:

$$\mathcal{P} = \kappa_p : \forall x : a.q\, x \Rightarrow p,$$

$$\kappa_q : q\, c$$

Consider resolution of the goal p in the big-step semantics. A derivation of the judgement $\mathcal{S};\mathcal{P} \longrightarrow e : p$ for a proof term e is constructed as follows:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\mathcal{S};\mathcal{P} \xrightarrow{\kappa_q : q\,c} \kappa_q : q\,c} \qquad \kappa_q : q\,c \in \mathcal{P}}{\mathcal{S};\mathcal{P} \longrightarrow \kappa_q : q\,c} \qquad \dfrac{}{\mathcal{S};\mathcal{P} \xrightarrow{\kappa_p\,\kappa_q : p} \kappa_p\,\kappa_q : p}}{\mathcal{S};\mathcal{P} \xrightarrow{\kappa_p : q\,c \Rightarrow p} \kappa_p\,\kappa_q : p} \qquad \mathcal{S};\cdot \vdash c : a}{\mathcal{S};\mathcal{P} \xrightarrow{\kappa_p : \forall x : a.\, q\,x \Rightarrow p} \kappa_p\,\kappa_q : p} \qquad \kappa_p : \forall x : a.\, q\,x \Rightarrow p \in \mathcal{P}}{\mathcal{S};\mathcal{P} \longrightarrow \kappa_p\,\kappa_q : p}$$

Example 3.10 illustrates an essential feature of the big-step semantics. Namely, instances of unification variables need to be given beforehand and moreover, these instances need to be terms that are well-formed in an empty context. This effectively means that goals resolved in the big-step semantics need to be well-formed and ground. We state this result formally as the following proposition:

Proposition 3.11

1. *If* $\mathcal{S};\mathcal{P} \longrightarrow e : G$ *and* G *is well-formed,* $\mathcal{S};\mathcal{P} \vdash G : \mathtt{o}$*, then* e *is ground,* i.e., $\mathrm{var}(e) = \emptyset$.
2. *If* $\mathcal{S};\mathcal{P} \xrightarrow{e' : D} e : A$ *and* $\mathrm{var}(e') = \emptyset$ *and* A *is well-formed,* $\mathcal{S};\mathcal{P} \vdash A : \mathtt{o}$*, then* e *is ground,* i.e., $\mathrm{var}(e) = \emptyset$.

Proof. By simultaneous structural induction on derivations.

Part 1

- Let the derivation step be $\dfrac{\mathcal{S};\mathcal{P} \longrightarrow e : B[M/x] \qquad \mathcal{S};\emptyset \vdash M : A}{\mathcal{S};\mathcal{P} \longrightarrow \langle M, e\rangle : \exists x : A.B}$. Since $\mathcal{S};\emptyset \vdash M : A$ then also $\mathrm{var}(M) = \emptyset$ and from Part 2 of the proposition follows that $\mathrm{var}(e) = \emptyset$. Hence, $\mathrm{var}(\langle M, e\rangle) = \emptyset$.
- Let the derivation step be $\dfrac{\mathcal{S};\mathcal{P} \xrightarrow{\kappa : D} e : A \qquad \kappa : D \in \mathcal{P}}{\mathcal{S};\mathcal{P} \longrightarrow e : A}$. The $\mathrm{var}(e) = \emptyset$ follows from Part 2 of the proposition and the fact that $\mathrm{var}(\kappa) = \emptyset$.

Part 2

- Let the derivation step be $\dfrac{}{\mathcal{S};\mathcal{P} \xrightarrow{e : A} e : A}$. From the assumption, $\mathrm{var}(e) = \emptyset$.
- Let the derivation step be $\dfrac{\mathcal{S};\mathcal{P} \longrightarrow e_1 : A_1 \qquad \mathcal{S};\mathcal{P} \xrightarrow{e\,e_1 : D} e_2 : A_2}{\mathcal{S};\mathcal{P} \xrightarrow{e : A_1 \Rightarrow D} e_2 : A_2}$. From Part 1 of the proposition follows that $\mathrm{var}(e) = \emptyset$. From this fact and from the assumption $\mathrm{var}(e\,e_1) = \emptyset$, using the induction hypothesis, we conclude that $\mathrm{var}(e_2) = \emptyset$.
- Let the derivation step be $\dfrac{\mathcal{S};\mathcal{P} \xrightarrow{e : D[X/M]} e_2 : A_2 \qquad \mathcal{S};\cdot \vdash M : A_1}{\mathcal{S};\mathcal{P} \xrightarrow{e : \forall X : A_1.\, D} e_2 : A_2}$. From Part 1 of the proposition, $\mathrm{var}(e) = \emptyset$.

□

that the judgement $\mathcal{S};\Gamma_1;M' \vdash (G \mid ?_{B_7})$ *holds:*

$$G = \top \wedge \top \wedge eq_T\,(\Pi\,\mathtt{bool}\,.(\Pi(\mathtt{maybe_A}\,0_T).(\Pi(\Pi(2_T \equiv_{\mathtt{bool}} \mathtt{ff}).\,\mathtt{A}).(\Pi(\Pi(3_T \equiv_{\mathtt{bool}} \mathtt{tt}).$$

$$(\Pi\,\mathtt{A}\,.\,\mathtt{A})).\,\mathtt{A}))))(\Pi\,\mathtt{bool}\,.?_{B_1})\,(\Pi\,\mathtt{type}\,.?_{L_1})\,\Gamma_1 \wedge (?_{B_1}[\mathtt{tt}\,/0_T] \equiv\ ?_{B_2}) \wedge \top \wedge$$

$$eq_T\,?_{B_2}\,(\Pi(\mathtt{maybe_A}\,\mathtt{tt}).?_{B_3})\,(\Pi\,\mathtt{type}\,.?_{L_2})\Gamma_1 \wedge (?_{B_3}[0_\Gamma/0_T] \equiv\ ?_{B_4}) \wedge$$

$$type\,?_A\,?_{L_3}\,\Gamma_1 \wedge term\,?_b\,?_{A_1}\,(\Gamma_1, ?_A) \wedge eq_K\,?_{L_3}\,\mathtt{type}\,\Gamma_1 \wedge (?_{A_1}[0_T/0_\Gamma] \equiv\ ?_{B_5}) \wedge$$

$$eq_T\,?_{B_4}\,(\Pi(\Pi ?_A.?_{B_5}).?_{B_6}, \Pi type.?_{L_5})\,\Gamma \wedge (?_{B_6}[(\lambda ?_A.?_b)/0_T] \equiv\ ?_{B_7})$$

That is, the type of M' *will be computed as a substitution for logic variable* $?_{B_7}$ *and resolving the goal in small steps also computes assignments to* $?_A$ *and* $?_b$.

Proposition 5.19 (Decidability of program construction)
Let $\mathcal{S}$ *be a signature. Then inference rules in Figure 5.4 construct the program* $\mathcal{P}$ *such that* $\mathcal{S} \vdash_{Prog} \mathcal{P}$.

We develop our running example further to illustrate the proposition:

Example 5.20 (From a signature to a program)
The signature $\mathcal{S}$ *contains the constant* $\mathtt{elim}_{\equiv_{\mathtt{bool}}}$ *hence the generated program contains the clause:*

$$\kappa_{\mathtt{elim}_{\equiv_{\mathtt{bool}}}} : term\ \mathtt{elim}_{\equiv_{\mathtt{bool}}}\,(\Pi\,\mathtt{tt} \equiv_{\mathtt{bool}} \mathtt{ff}\,.\,\mathtt{A})\ \mathtt{type}\ ?_\Gamma \Leftarrow$$

The following clauses come from the program $\mathcal{P}_e$ *and represent inference rules of the internal language:*

$$\kappa_0 : term\,0\,?_A\,(?_\Gamma, ?_{A'}) \quad \Leftarrow\ ?_{A'}\!\uparrow \equiv\ ?_A$$

$$\kappa_{\mathtt{elim}} : term\,(?_a ?_b)\,?_B\,?_\Gamma \quad \Leftarrow term\,?_a\,(\Pi ?_A.?_{B'})\,?_\Gamma \wedge term\,?_b\,?_A\,?_\Gamma \wedge$$

$$eq_T\,?_B\,?_{B'}\ \mathtt{type}\ ?_\Gamma \wedge (?_{B'}[?_b] \equiv ?_B)$$

Example 5.18 shows unresolved meta-variables in the goal, and Example 5.20 gives a program against which to resolve the goal. Now the proof-relevant resolution comes

$\boxed{\mathcal{S} \vdash_{\text{Prog}} \mathcal{P}}$

$$\frac{}{\cdot \vdash_{\text{Prog}} \mathcal{P}_{\text{e}}}$$

$$\frac{\mathcal{S} \vdash_{\text{Prog}} \mathcal{P}}{\mathcal{S}, c : A \vdash_{\text{Prog}} \mathcal{P}, \quad \kappa_c : term\, c\, A\, ?_\Gamma \leftarrow, \quad \kappa_{\text{shift}_c} : (c{\uparrow}^0 \equiv\ c) \leftarrow, \quad \kappa_{\text{subst}_c} : c[?_M/0] \equiv c \leftarrow, \quad \kappa_{eq^s_c} : eq^s(c, c, A, ?_\Gamma) \leftarrow}$$

$$\frac{\mathcal{S} \vdash_{\text{Prog}} \mathcal{P}}{\mathcal{S}, a : L \vdash_{\text{Prog}} \mathcal{P}, \quad \kappa_{\text{shift}_\alpha} : (\alpha{\uparrow}^0 \equiv\ \alpha) \leftarrow, \kappa_{\text{subst}_\alpha} : \alpha[?_M/0] \equiv \alpha \leftarrow, \quad \kappa_{eq_T} : eq_T\, \alpha\, \alpha\, L\, ?_\Gamma \leftarrow, \kappa_{eq^a{}_\alpha} : eq^a\, ?_N\, ?_M \alpha\, ?_\Gamma \leftarrow eq^s\, ?_M\ N\, \alpha\, \Gamma}$$

Figure 5.4: Refinement of signatures

Definition 5.16 (Refinement program)

The judgement $\mathcal{S} \vdash_{Prog} P$ is given by the inference rules of Figure 5.4.

The Figure 5.4 gives definition of signature refinement. The refinement judgement of a signature into a program concludes our transformation of refinement problem into a goal and a program.

Theorem 5.17 (Decidability of goal construction)

Let M be a refinement problem in a well-formed signature $\mathcal{S}$ and a well-formed context Γ such that a solution (ρ, R) exists. Then inference rules in Figures 5.2 and 5.3 construct the goal G and the extended type A such that $\mathcal{S}; \Gamma; M \vdash (G \mid A)$.

Proof. By induction on the derivation of the well-formedness judgement of $(\rho, R)(M)$. □

The next example illustrates the construction of a refinement goal.

Example 5.18 (From an extended nameless term to a goal)

Let us take the refinement problem $M' = (\texttt{elim}_{\texttt{maybe}_{\mathbb{A}}}\ \texttt{tt}\ 0)(\lambda ?_A.?_b)$ and the implicit context and signature from Example 5.11. By Theorem 5.17 we can generate G such

Although providing only ground answer substitutions is sufficient from the point of view of traditional logic programming, it is not sufficient for our intended application. In the traditional logic programming domains are considered to be inhabited (*cf.* Lloyd, 1987) whereas we seek applications in type theory where empty domains often play important role. Also, the big-step semantics does not provide a computational device, it does not provide any insight into how to implement a resolution engine that adheres to such semantics. We address these shortcomings by introducing a small-step operational semantics. This semantic will be both more general, allowing for non-ground answer substitutions, and detailed enough to allow for a direct implementation.

3.1.2 Small-step operational semantics

Our exposition of the small-step operational semantics of resolution in Horn-clause logic generalises the original presentation of proof-relevant resolution given by Fu and Komendantskaya (2017). We incorporate unification into resolution whereas Fu and Komendantskaya were working only with matching. Small-step semantics is expressed in the form of mixed terms and rewriting contexts. In the small-step semantics, mixed terms, which consist of both proof terms and goals that have not been resolved yet, allow to express an intermediate state of computation. Rewriting contexts allow to formally identify a particular goal in the intermediate state of the computation that is subject to a computation step.

Definition 3.12 (Mixed terms and rewriting contexts)

$$\begin{array}{lll} \text{MT} \ni \hat{e}, \hat{e}', \hat{e}_1, \hat{e}_2 & := \kappa \mid G \mid \hat{e}\ \hat{e} \mid \langle M, \hat{e} \rangle & \textit{mixed terms} \\ \mathcal{R} \ni C, C' & := \kappa \mid \bullet \mid e\ C \mid \langle M, C \rangle & \textit{rewriting contexts} \end{array}$$

We use identifiers $\hat{e}$, $\hat{e}'$, $\hat{e}_1$, and $\hat{e}_2$ for mixed terms in MT and identifiers C and C' for rewriting contexts in $\mathcal{R}$. Clearly, every proof term is a mixed term. We extend substitution to mixed terms.

Definition 3.13

$$\kappa[M/x] = \kappa$$

$$G[M/x] = G[M/x]$$

$$\hat{e}_1\,\hat{e}_2[M/x] = (\hat{e}_1[M/x])\,(\hat{e}_2[M/x])$$

$$\langle N, \hat{e}\rangle[M/x] = \langle N[M/x], \hat{e}[M/x]\rangle$$

Rewriting contexts are used in the definition of the small-step semantics as a device to identify a subterm of a mixed term where the computational step happens. More precisely, a mixed term that is subject of a judgement of the small-step semantics is decomposed into a rewriting context with a hole $\bullet$ in the position of such subterm and the subterm itself. We introduce an operation of *hole replacement*, denoted $-\{-\}$. Hole replacement replaces a hole in a rewriting context with a mixed term. Hole replacement allows us manipulating rewriting contexts in definition of the small-step semantics.

Definition 3.14

$$\kappa\{\hat{e}\} = \kappa$$

$$\bullet\{\hat{e}\} = \hat{e}$$

$$(\hat{e}_1\,C)\{\hat{e}\} = \hat{e}_1\,(C\{\hat{e}\})$$

$$\langle M, C\rangle\{\hat{e}\} = \langle M, C\{\hat{e}\}\rangle$$

A result of hole replacement with a mixed term in a rewriting context is a mixed term. We say that a mixed term $\hat{e}'$ *identifies* a rewriting context C in a mixed term $\hat{e}$ if $C\{\hat{e}'\} = \hat{e}$. Conversely, $\hat{e}'$ is the identifying mixed term for C. We state the following property about identification of rewriting contexts:

Proposition 3.15

If $C_1\{G\} = C_2\{\hat{e}\}$ then there is a unique C' such that $\hat{e} = C'[G]$.

Proof. By induction on C_1 and C_2. The compatible cases are:

- $C_1 = \bullet$ and $C_2 = \bullet$. Then $\hat{e} = G$ and $C' = \bullet$.

Finally, there are clauses that represent shifting and substitution on terms and types:

$$\kappa_{shiftTtintro}: \quad (\Pi ?_A.?_M)\uparrow^\iota \equiv\ (\Pi ?_{A'}.?_{M'}) \Leftarrow ?_A\uparrow^\iota \equiv\ ?_{A'} \wedge ?_M\uparrow^{\sigma\iota} \equiv\ ?_{M'}$$

$$\kappa_{shiftTtintro}: \quad (\lambda ?_A.?_M)\uparrow^\iota \equiv\ (\lambda ?_{A'}.?_{M'}) \Leftarrow ?_A\uparrow^\iota \equiv\ ?_{A'} \wedge ?_M\uparrow^{\sigma\iota} \equiv\ ?_{M'}$$

$$\kappa_{shifttelim}: \quad (?_M ?_N)\uparrow^\iota \equiv\ (?_{M'} ?_{N'}) \Leftarrow ?_M\uparrow^\iota \equiv\ ?_{M'} \wedge ?_N\uparrow^\iota \equiv\ ?_{N'}$$

$$\kappa_{shifttgt}: \quad \iota\uparrow^0 \equiv\ \sigma\iota \Leftarrow$$

$$\kappa_{shifttpred}: \quad 0\uparrow^{\sigma\iota} \equiv\ 0 \Leftarrow$$

$$\kappa_{shifttstep}: \quad \sigma\iota\uparrow^{\sigma\iota'} \equiv\ \sigma\iota'' \Leftarrow \iota\uparrow^{\iota'} \equiv\ \iota''$$

$$\kappa_{substTintro}: \quad (\Pi ?_A.?_M)[?_N/\iota] \equiv (\Pi ?_{A'}.?_{M'}) \Leftarrow (?_A[?_N/\iota] \equiv ?_{A'}) \wedge (?_N\uparrow^0 \equiv\ ?'_N)$$
$$\wedge\ ?_M[?'_N/\sigma\iota] \equiv ?_{M'}$$

$$\kappa_{substintro}: \quad (\lambda ?_A.?_M)[N/\iota] \equiv (\lambda ?_{A'}.?_{M'}) \Leftarrow (?_A[\iota/?_{A'}] \equiv) \wedge (?_N\uparrow^0 \equiv\ ?'_N)$$
$$\wedge\ ?_M[?'_N/\sigma\iota] \equiv ?_{M'}$$

$$\kappa_{substtelim}: \quad (?_{M_1} ?_{M_2})[?_N/\iota] \equiv (?_{M'_1} ?_{M'_2}) \Leftarrow ?_{M_1}[?_N/\iota] \equiv ?_{M'_1} \wedge ?_{M_2}[?_N/\iota] \equiv ?_{M'_2}$$

$$\kappa_{substz}: \quad 0[?_N/0] \equiv ?_N \Leftarrow$$

$$\kappa_{substs}: \quad 0[?_N/\sigma\iota] \equiv 0 \Leftarrow$$

$$\kappa_{substgt}: \quad \sigma\iota[?_N/0] \equiv \sigma\iota \Leftarrow$$

$$\kappa_{substpred}: \quad \sigma\iota[?_N/\sigma\iota'] \equiv \sigma\iota'' \Leftarrow \iota[?_N/\iota'] \equiv \iota''$$

The clauses in Definition 5.15 correspond to judgements in Figures 2.7–2.9 and Figure 2.10. They are direct translations of the inference rules of nameless LF in these figures. The judgement $\mathcal{S} \vdash_{\text{Prog}} \mathcal{P}$ extends $\mathcal{P}_e$ with a clause for each type and term constant in $\mathcal{S}$ and initialises shifting and substitution with term and type-level constants as constant under the operation.

mic and structural equality of terms, and weak head reduction of terms:

$$\kappa_{eqTintro}: \quad eq_T\,(\Pi ?_{A_1}.?_{A_2})\,(\Pi ?_{B_1}.?_{B_2})\ \texttt{type}\ ?_\Gamma \Leftarrow eq_T\,?_{A_1}\,?_{B_1}\ \texttt{type}\ ?_\Gamma) \wedge$$

$$eq_T\,?_{A_2}\,?_{B_2}\ \texttt{type}\,(?_\Gamma, ?_{A_1})$$

$$\kappa_{eqTelim}: \quad eq_T\,(?_A\,?_M)\,(?_B\,?_N)\,?_L\,?_\Gamma \Leftarrow eq_T ?_A\,?_B\,(\Pi ?_C.?_L)\,?_\Gamma \wedge eq^a_t\,?_M\,?_N\,?_C\,?_\Gamma$$

$$\kappa_{eqtzero}: \quad eq^s_t\,0_\Gamma\,0_\Gamma, ?_A\,(?_\Gamma, ?_A) \Leftarrow$$

$$\kappa_{eqtsucc}: \quad eq^s_t\,(\sigma\,?_{\iota_\Gamma})\,(\sigma\,?_{\iota'_\Gamma})\,?_A\,(?_\Gamma, ?_B) \Leftarrow eq^s_t\,?_{\iota_\Gamma}\,?_{\iota'_\Gamma}\,?_A\,?_\Gamma$$

$$\kappa_{eqtrefl}: \quad eq^s_t\,?_a\,?_a\,?_A\,?_\Gamma \Leftarrow$$

$$\kappa_{eqtelim}: \quad eq^s_t\,(?_{M_1}\,?_{M_2})\,(?_{N_1}\,?_{N_2})?_B\,?_\Gamma \Leftarrow eq^s_t\,?_{M_1}\,?_{N_1}\,(\Pi ?_A.?_B)?_\Gamma \wedge eq^a_t\,?_{M_2}\,?_{N_2}\,?_B\,?_\Gamma$$

$$\kappa_{eqtwhrl}: \quad eq^a_t\,?_M\,?_N\,?_A\,?_\Gamma \Leftarrow whr\,?_M\,?_{M'} \wedge eq_t\,?_{M'}\,?_N\,?_A\,?_\Gamma$$

$$\kappa_{eqtwhrr}: \quad eq^a_t\,?_M\,?_N\,?_A\,?_\Gamma \Leftarrow whr?_N\,?_{N'} \wedge eq^a_t\,?_M\,?_{N'}\,?_A\,?_\Gamma$$

$$\kappa_{eqtstr}: \quad eq^a_t\,?_M\,?_N\,?_A\,?_\Gamma \Leftarrow eq^s_t\,?_M\,?_N\,?_A\,?_\Gamma$$

$$\kappa_{eqtexp}: \quad eq^a_t\,?_M\,?_N\,(\Pi ?_A.?_B)\,?_\Gamma \Leftarrow (?_M\uparrow \equiv\ ?_{M'}) \wedge (?_N\uparrow\,?_{N'}) \wedge$$

$$eq^a_t\,(?_{M'}\,0)\,(?_{N'}\,0)\,?_B\,(?_\Gamma, ?_A)$$

$$\kappa_{whrs}: \quad whr\,(\lambda ?_A.?_M)\,?_N\,?_{M'} \Leftarrow ?_M[?_N/0] \equiv ?_{M'}$$

$$\kappa_{whrh}: \quad whr\,(?_M\,?_N)\,(?_{M'}\,?_N) \Leftarrow whr\,?_M\,?_{M'}$$

$\boxed{\mathcal{S}; \mathcal{P} \vdash \Gamma \mid \hat{e} \rightsquigarrow \Gamma' \mid \hat{e}'}$

$$\frac{\mathcal{S}; \mathcal{P} \vdash \Gamma \mid C\{A\} \overset{\kappa:D}{\rightsquigarrow} \Gamma' \mid \hat{e}}{\mathcal{S}; \mathcal{P} \vdash \Gamma \mid C\{A\} \rightsquigarrow \Gamma' \mid \hat{e}}$$

$$\frac{\mathcal{S}; \mathcal{P} \vdash \Gamma, Y : A \mid C\{\langle Y, G[Y/x]\rangle\} \rightsquigarrow \Gamma' \mid \hat{e}}{\mathcal{S}; \mathcal{P} \vdash \Gamma \mid C\{\exists x : A.G\} \rightsquigarrow \Gamma' : A \mid \hat{e}'}$$

Figure 3.4: Right introduction rules, small-step

- $C_1 = \hat{e}'\,C'_1$ and $C_2 = \bullet$. Then $\hat{e} = \hat{e}'\,C'_1\{G\}$ and $C' = \hat{e}\,C_1$.
- $C_1 = \hat{e}'\,C'_1$ and $C_2 = \hat{e}'\,C'_2$. Then $C'_1\{G\} = C'_2\{\hat{e}\}$ and from induction hypothesis there is unique C' such that $\hat{e} = C'\{G\}$.
- $C_1 = \langle M, C'_1\rangle$ and $C_2 = \langle M, C_2\rangle$. Then $C'_1\{G\} = C'_2\{\hat{e}\}$ and from induction hypothesis there is unique C' such that $\hat{e} = C'\{G\}$.

□

We proceed with definition of the actual small-step semantics. Similarly to the big-step semantics, the small-step semantics is defined using two judgements,

- $\mathcal{S}; \mathcal{P} \vdash \Gamma \mid \hat{e} \rightsquigarrow \Gamma' \mid \hat{e}'$, and
- $\mathcal{S}; \mathcal{P} \vdash \Gamma \mid \hat{e} \overset{\hat{e}'':D}{\rightsquigarrow} \Gamma' \mid \hat{e}'$.

The first judgement corresponds to right-introduction rules of logical connectives and proceeds on mixed terms in shapes of goals. The other judgement, which we again call *backchaining*, is annotated with a proof term and a definite clause and corresponds to left-introduction rules of logical connectives. Goals and atomic goals identify rewriting contexts in the sense we introduced above. This also motivates our statement of Proposition 3.15.

Definition 3.16 (Operational semantics, small-step)
The judgements $\mathcal{S}; \mathcal{P} \vdash \Gamma \mid \hat{e} \rightsquigarrow \Gamma' \mid \hat{e}'$, and $\mathcal{S}; \mathcal{P} \vdash \Gamma \mid \hat{e} \overset{\hat{e}'':D}{\rightsquigarrow} \Gamma' \mid \hat{e}'$ are given by inference rules in Figures 3.4 and 3.5.

Note that $\mathcal{P}$ is not changed by the inference rules. However, it will change later when we extend the logic. Thus we keep $\mathcal{P}$ explicit to maintain the same shape of judgements throughout the thesis.

Let us now show how the goal in Example 3.9 resolves in the small-step semantics. Note that we do not provide a proper derivation in small-step semantics as it is

$\boxed{\mathcal{S};\mathcal{P} \vdash \Gamma \mid \hat{e} \overset{\hat{e}'':D}{\rightsquigarrow} \Gamma' \mid \hat{e}'}$

$$\frac{\mathcal{S};\Gamma \vdash \sigma : \Gamma' \qquad \mathcal{S};\Gamma' \vdash \sigma A \equiv \sigma A' : \mathtt{o}}{\mathcal{S};\mathcal{P} \vdash \Gamma \mid C\{A\} \overset{\hat{e}:A'}{\rightsquigarrow} \Gamma' \mid \sigma(C\{\hat{e}\})}$$

$$\frac{\mathcal{S};\mathcal{P} \vdash \Gamma \mid C\{A\} \overset{\hat{e}_1\, A_1:D}{\rightsquigarrow} \Gamma' \mid \hat{e}}{\mathcal{S};\mathcal{P} \vdash \Gamma \mid C\{A\} \overset{\hat{e}_1:A_1 \Rightarrow D}{\rightsquigarrow} \Gamma' \mid \hat{e}}$$

$$\frac{\mathcal{S};\mathcal{P} \vdash \Gamma, Y : A_1 \mid C\{A_2\} \overset{\hat{e}_1:D[Y/x]}{\rightsquigarrow} \Gamma' \mid \hat{e}}{\mathcal{S};\mathcal{P} \vdash \Gamma \mid C\{A_2\} \overset{\hat{e}_1:\forall x:A_1.D}{\rightsquigarrow} \Gamma' \mid \hat{e}}$$

Figure 3.5: Backchaining rules, small-step

rather lengthy but indicate only rewriting of the identified goals in the course of computation. We superscript the identified goals with the annotating mixed term and the annotating definite clause, that is we will write, *e.g.*, $\Gamma \mid C\{A^{e:A}\} \rightsquigarrow \Gamma \mid C\{e\}$ for $\dfrac{\mathcal{S};\Gamma \vdash \{\} : \Gamma \qquad \mathcal{S};\Gamma \vdash \{\}A \equiv \{\}A : \mathtt{o}}{\mathcal{S};\mathcal{P} \vdash \Gamma \mid C\{A\} \overset{e:A}{\rightsquigarrow} \Gamma \mid C\{e\}}$. Occasionally, when several resolutions steps are straightforward, we will omit them and write $\Gamma \mid \hat{e} \rightsquigarrow^* \Gamma' \mid \hat{e}'$ for $\Gamma \mid \hat{e} \rightsquigarrow \Gamma_1 \mid \hat{e}_1 \rightsquigarrow \ldots \rightsquigarrow \Gamma_n \mid \hat{e}_n \rightsquigarrow \Gamma' \mid \hat{e}'$ in order to simplify presentation.

Example 3.17

Resolving the goal $\exists x : a.nat\,(s\,X)$ *in* $\mathcal{S}$ *and* $\mathcal{P}$*:*

$$\cdot \mid \exists x : a.nat\,(s\,x) \rightsquigarrow Z : a \mid \langle Z, nat\,(s\,Z)\rangle \rightsquigarrow$$

$$Z : a \mid \langle Z, (nat\,(s\,Z))^{\kappa_s:\forall x:a.nat\,x \Rightarrow nat\,(s\,x)}\rangle \rightsquigarrow$$

$$Z : a, Y : a \mid \langle Z, (nat\,(s\,Z))^{\kappa_s:nat\,Y \Rightarrow nat\,(s\,Y)}\rangle \rightsquigarrow$$

$$Z : a, Y : a \mid \langle Z, (nat\,(s\,Z))^{\kappa_s\,(nat\,Y):nat\,(s\,Y)}\rangle \rightsquigarrow$$

$$Z : a \mid \langle Z, \kappa_s\,(nat\,Z)\rangle \rightsquigarrow$$

$$Z : a \mid \langle Z, \kappa_s\,(nat\,Z)^{\kappa_z:nat\ z}\rangle \rightsquigarrow$$

$$\cdot \mid \langle z, \kappa_s\kappa_z\rangle$$

Similarly, the goal in Example 3.10 can be resolved using the small-step semantics as well.

that are generated from a signature $\mathcal{S}$. The clauses that represent inference rules of LF are the same for all programs and Definition 5.15 gives a minimal program $\mathcal{P}_e$ that contains only these clauses.

Definition 5.15

Let P_e *be a program with clauses that represent inference rules for well-formedness of terms and types:*

$$\kappa_{true}: \quad \top \Leftarrow$$

$$\kappa_0: \quad proj\,0\,?_A\,(?_\Gamma, ?_{A'}) \Leftarrow (?_{A'}\uparrow \equiv\ ?_A)$$

$$\kappa_\sigma: \quad proj\,(\sigma\,?_\iota)\,?_A\,(?_\Gamma, ?_B) \Leftarrow proj\,?_\iota\,?_{A'}\,?_\Gamma \wedge (?_{A'}\uparrow \equiv\ ?_A)$$

$$\kappa_{proj}: \quad type\,?_\iota\,?_A\ \mathtt{type}\ ?_\Gamma \Leftarrow proj\,?_\iota\,?_A\,?_\Gamma$$

$$\kappa_{T\text{-}elim}: \quad type\,(?_A\,?_M)\,?_L\,?_\Gamma \Leftarrow type\,?_A\,(\Pi ?_{A_1}.?_{L'})\,?_\Gamma \wedge term\,?_M\,?_{A_2}\,?_\Gamma \wedge eq_T\,?_{A_1}\,?_{A_2}\ \mathtt{type}\ ?_\Gamma \wedge (?_{L'}[?_M] \equiv ?_L)$$

$$\kappa_{T\text{-}intro}: \quad type\,(\Pi ?_A.?_B)\ \mathtt{type}\ ?_\Gamma \Leftarrow type\,?_A\ \mathtt{type}\ ?_\Gamma \wedge type\,?_B\ \mathtt{type}\,(?_\Gamma, ?_B)$$

$$\kappa_{t\text{-}elim}: \quad term\,(?_M\,?_N)\,?_B\,?_\Gamma \Leftarrow term\,?_M\,(\Pi ?_{A_1}.?_{B'})\,?_\Gamma \wedge term\,?_N\,?_{A_2}\,?_\Gamma \wedge eq_T\,?_{A_1}\,?_{A_2}, \mathtt{type}, ?_\Gamma \wedge (?_{B'}[?_N] \equiv ?_B)$$

$$\kappa_{t\text{-}intro}: \quad term\,(\lambda ?_A.?_M)(\Pi ?_A.?_B)?_\Gamma \Leftarrow type\,?_A\ \mathtt{type}\ ?_\Gamma \wedge term\,?_M\,?_B\,?_\Gamma)$$

Further, there are clauses that represent weak algorithmic equality of types, algorith-

$\boxed{\mathcal{S}; \Gamma; M \vdash (G \mid A)}$

$$\frac{c : A \in \mathcal{S}}{\mathcal{S}; \Gamma; c \vdash (\top \mid A)}\ \text{R-CON}$$

$$\frac{}{\mathcal{S}; \Gamma; ?_a \vdash (?_a : term\ ?_{a'}\ ?_A\ \Gamma \mid ?_A)}\ \text{R-T-META}$$

$$\frac{}{\mathcal{S}; \Gamma, A; 0 \vdash (A{\uparrow} \equiv\ ?_A \mid ?_A)}\ \text{R-ZERO}$$

$$\frac{\mathcal{S}; \Gamma; \iota \vdash (G \mid A)}{\mathcal{S}; \Gamma, B; \sigma\,\iota \vdash (G \wedge (A{\uparrow} \equiv\ ?_A) \mid\ ?_A)}\ \text{R-SUCC}$$

$$\frac{\mathcal{S}; \Gamma; A \vdash (G_A \mid L) \qquad \mathcal{S}; \Gamma, A; M \vdash (G_M \mid B)}{\mathcal{S}; \Gamma; \lambda A.M \vdash (G_A \wedge G_M \wedge (eq_K\ L\ \texttt{type}\ \Gamma) \mid\ \Pi A.B)}\ \text{R-}\lambda\text{-INTRO}$$

$$\frac{\mathcal{S}; \Gamma; M \vdash (G_M \mid A) \qquad \mathcal{S}; \Gamma; N \vdash (G_N \mid A_2)}{\begin{array}{c}\mathcal{S}; \Gamma; MN \vdash (G_M \wedge G_N \wedge (eq_T\ A\ (\Pi A_2.?_B)\ \texttt{type}\ \Gamma) \\ \wedge (?_B[N] \equiv ?_{B'}) \mid\ ?_{B'})\end{array}}\ \text{R-}\lambda\text{-ELIM}$$

Figure 5.2: Refinement of terms

$\boxed{\mathcal{S}; \Gamma; A \vdash (G \mid L)}$

$$\frac{a : L \in \mathcal{S}}{\mathcal{S}; \Gamma; a \vdash (\top \mid L)}\ \text{R-TCON}$$

$$\frac{}{\mathcal{S}; \Gamma; ?_A \vdash (type\ ?_A\ ?_L\ \Gamma \mid ?_L)}\ \text{R-T-META}$$

$$\frac{\mathcal{S}; \Gamma; A \vdash (G_A \mid L_1) \qquad \mathcal{S}; \Gamma, A; B \vdash (G_B \mid L_2)}{\mathcal{S}; \Gamma; \Pi A.B \vdash (G_A \wedge G_B \wedge (eq_K\ L_1\ \texttt{type}\ \Gamma) \wedge (eq_K\ L_2\ \texttt{type}\ \Gamma) \mid \texttt{type})}\ \text{R-}\Pi\text{-INTRO}$$

$$\frac{\mathcal{S}; \Gamma; A \vdash (G_A \mid L) \qquad \mathcal{S}; \Gamma; M \vdash (G_M \mid B)}{\mathcal{S}; \Gamma; AM \vdash (G_A \wedge G_M \wedge (eq_K\ L\ (\Pi B.?_L)\ \Gamma) \wedge (?_L[M] \equiv ?_{L'}) \mid\ ?_{L'})}\ \text{R-}\Pi\text{-ELIM}$$

Figure 5.3: Refinement of types

Example 3.18

Consider signature $\mathcal{S}$ and program $\mathcal{P}$ in Example 3.10. The goal p is resolved in small-step semantics with proof term $\kappa_p\kappa_q$:

$$\cdot \mid p \rightsquigarrow \cdot \mid p^{\kappa_p:(\forall x:a,q\,x)\Rightarrow p} \rightsquigarrow X : a \mid p^{\kappa_p:q\,X\Rightarrow p} \rightsquigarrow X : a \mid p^{\kappa_p(q\,X):p} \rightsquigarrow X : a \mid \kappa_p(q\,X) \rightsquigarrow$$

$$X : a \mid \kappa_p(q\,X)^{\kappa_q:q\,c} \rightsquigarrow \cdot \mid \kappa_p\,\kappa_q$$

However, assume that we include a new clause, $\kappa_{q'} : \forall y : a.q\,y$. The following is a valid small-step resolution:

$$\cdot \mid p \rightsquigarrow \cdot \mid p^{\kappa_p:(\forall x:a,q\,x)\Rightarrow p} \rightsquigarrow X : a \mid p^{\kappa_p:q\,X\Rightarrow p} \rightsquigarrow X : a \mid p^{\kappa_p(q\,X):p} \rightsquigarrow X : a \mid \kappa_p(q\,X) \rightsquigarrow$$

$$X : a \mid \kappa_p(q\,X)^{\kappa_{q'}:\forall y:a.q\,y} \rightsquigarrow X : a, Y : a \mid \kappa_p(q\,X)^{\kappa_{q'}:q\,Y} \rightsquigarrow X : a \mid \kappa_p\,\kappa_{q'}$$

That is, the goal is not resolved in an empty context but in a context that consists of a single variable X.

We have introduced the big-step and the small-step semantics of proof-relevant resolution. Before we move to a discussion of soundness of the small-step semantics, we state a lemma that will be required in the following development.

Lemma 3.19 (Subderivations)

1. *If $\mathcal{S}; \mathcal{P} \vdash \Gamma \mid C\{\hat{e}\} \rightsquigarrow \Gamma' \mid \hat{e}'$ then there is a mixed term $\hat{e}''$ and a substitution θ such that $\hat{e}' = (\theta C)\{\hat{e}''\}$ and $\mathcal{S}; \mathcal{P} \vdash \Gamma \mid \hat{e} \rightsquigarrow \Gamma \mid \hat{e}''$*
2. *If $\mathcal{S}; \mathcal{P} \vdash \Gamma \mid C\{\hat{e}\} \overset{\hat{e}_1:D_1}{\rightsquigarrow} \Gamma' \mid \hat{e}'$ then there is a mixed term $\hat{e}''$ and a substitution θ such that $\hat{e}' = (\theta C)\{\hat{e}''\}$ and $\mathcal{S}; \mathcal{P} \vdash \Gamma \mid \hat{e} \overset{\hat{e}_1:D_1}{\rightsquigarrow} \Gamma \mid \hat{e}''$*

Proof. By simultaneous structural induction on the derivation and the rewriting context.

Part 1 The compatible cases are:

- Let the derivation be of the shape $\dfrac{\mathcal{S}; \mathcal{P} \vdash \Gamma \mid C_1\{A\} \overset{\kappa:\mathcal{D}}{\rightsquigarrow} \Gamma' \mid \hat{e}'}{\mathcal{S}; \mathcal{P} \vdash \Gamma \mid C_1\{A\} \rightsquigarrow \Gamma' \mid \hat{e}'}$ and the rewriting context of the shape $C = e\,C_2$. By Proposition 3.15, there is a

unique C' such that $\hat{e} = C'\{A\}$. By the induction assumption, there is a mixed term $\hat{e}''$, a substitution θ such that $\hat{e}' = \theta(e\,C_2\{C'\{A\}\})\{\hat{e}''\}$, and a derivation of $\mathcal{S}; \mathcal{P} \vdash \Gamma \mid C'\{A\} \overset{\kappa:\mathcal{D}}{\rightsquigarrow} \Gamma' \mid \hat{e}''$. Then there is a derivation $\dfrac{\mathcal{S}; \mathcal{P} \vdash \Gamma \mid C'\{A\} \overset{\kappa:\mathcal{D}}{\rightsquigarrow} \Gamma' \mid \hat{e}''}{\mathcal{S}; \mathcal{P} \vdash \Gamma \mid C'\{A\} \rightsquigarrow \Gamma' \mid \hat{e}''}$.
We use Proposition 3.15 in the rest of the proof implicitly.

- Let the derivation be $\dfrac{\mathcal{S}; \mathcal{P} \vdash \Gamma, Y : A \mid e\,C_1\{\langle Y, G[Y/x]\rangle\} \rightsquigarrow \Gamma' \mid \hat{e}'}{\mathcal{S}; \mathcal{P} \vdash \Gamma \mid e\,C_1\{\exists x : A.G\} \rightsquigarrow \Gamma' \mid \hat{e}'}$. By the induction assumption, there is a mixed term $\hat{e}''$, a substitution θ such that $\hat{e}' = \theta(e\,C_2\{C'\{A\}\})\{\hat{e}''\}$, and a derivation of $\mathcal{S}; \mathcal{P} \vdash \Gamma, Y : A \mid C'\{\langle Y, G[Y/x]\rangle\} \rightsquigarrow \Gamma' \mid \hat{e}''$. Then there is a derivation $\dfrac{\mathcal{S}; \mathcal{P} \vdash \Gamma, Y : A \mid C'\{\langle Y, G[Y/x]\rangle\} \rightsquigarrow \Gamma' \mid \hat{e}''}{\mathcal{S}; \mathcal{P} \vdash \Gamma \mid C'\{\exists x : A.G\} \rightsquigarrow \Gamma' \mid \hat{e}''}$.
- Let the derivation be of the shape $\dfrac{\mathcal{S}; \mathcal{P} \vdash \Gamma \mid \langle M, C_1\rangle\{A\} \overset{\kappa:\mathcal{D}}{\rightsquigarrow} \Gamma' \mid \hat{e}'}{\mathcal{S}; \mathcal{P} \vdash \Gamma \mid \langle M, C_1\rangle\{A\} \rightsquigarrow \Gamma' \mid \hat{e}'}$ By the induction assumption, there is a mixed term $\hat{e}''$, a substitution θ such that $\hat{e}' = \theta(\langle M, C_2\rangle\{C'\{A\}\})\{\hat{e}''\}$, and a derivation of $\mathcal{S}; \mathcal{P} \vdash \Gamma \mid C'\{A\} \rightsquigarrow \Gamma' \mid \hat{e}''$. Then there is a derivation $\dfrac{\mathcal{S}; \mathcal{P} \vdash \Gamma \mid C'\{A\} \overset{\kappa:\mathcal{D}}{\rightsquigarrow} \Gamma' \mid \hat{e}''}{\mathcal{S}; \mathcal{P} \vdash \Gamma \mid C'\{A\} \rightsquigarrow \Gamma' \mid \hat{e}''}$.
- Let the derivation be $\dfrac{\mathcal{S}; \mathcal{P} \vdash \Gamma, Y : A \mid \langle M, C_1\rangle\{\langle Y, G[Y/x]\rangle\} \rightsquigarrow \Gamma' \mid \hat{e}'}{\mathcal{S}; \mathcal{P} \vdash \Gamma \mid \langle M, C_1\rangle\{\exists x : A.G\} \rightsquigarrow \Gamma' \mid \hat{e}'}$. By the induction assumption, there is a mixed term $\hat{e}''$, a substitution θ such that $\hat{e}' = \theta(\langle M, C_2\rangle\{C'\{A\}\})\{\hat{e}''\}$, and a derivation of $\mathcal{S}; \mathcal{P} \vdash \Gamma, Y : A \mid C'\{\langle Y, G[Y/x]\rangle\} \rightsquigarrow \Gamma' \mid \hat{e}''$. Then there is a derivation $\dfrac{\mathcal{S}; \mathcal{P} \vdash \Gamma, Y : A \mid C'\{\langle Y, G[Y/x]\rangle\} \rightsquigarrow \Gamma' \mid \hat{e}''}{\mathcal{S}; \mathcal{P} \vdash \Gamma \mid C'\{\exists x : A.G\} \rightsquigarrow \Gamma' \mid \hat{e}''}$.

Part 2 The compatible cases are:

- Let the derivation be of the shape $\dfrac{\mathcal{S}; \Gamma \vdash \theta : \Gamma' \qquad \mathcal{S}; \Gamma' \vdash \theta A \equiv \theta A' : \mathtt{o}}{\mathcal{S}; \mathcal{P} \vdash \Gamma \mid C\{A\} \overset{\hat{e}:A'}{\rightsquigarrow} \Gamma' \mid \theta C\{\hat{e}\}}$ and the rewriting context of the shape $C = \bullet$. Then $\hat{e}'' = \hat{e}$ and $\mathcal{S}; \mathcal{P} \vdash \Gamma \mid A \overset{\hat{e}:A'}{\rightsquigarrow} \Gamma \mid \hat{e}$.
- Let the derivation be of the shape $\dfrac{\mathcal{S}; \mathcal{P} \vdash \Gamma \mid \bullet\{A\} \overset{\hat{e}_1 A':D}{\rightsquigarrow} \Gamma' \mid \hat{e}}{\mathcal{S}; \mathcal{P} \vdash \Gamma \mid \bullet\{A\} \overset{\hat{e}_1:A' \Rightarrow D}{\rightsquigarrow} \Gamma' \mid \hat{e}}$. Then $\hat{e}'' = \hat{e}$ and $\mathcal{S}; \mathcal{P} \vdash \Gamma \mid A \overset{\hat{e}:A'}{\rightsquigarrow} \Gamma' \mid \hat{e}$.
- Let the derivation be of the shape $\dfrac{\mathcal{S}; \mathcal{P} \vdash \Gamma, Y : A_1 \mid \bullet\{A\} \overset{\hat{e}_1:D[Y/x]}{\rightsquigarrow} \Gamma' \mid \hat{e}}{\mathcal{S}; \mathcal{P} \vdash \Gamma \mid \bullet\{A\} \overset{\hat{e}_1:\forall x:A_1.D}{\rightsquigarrow} \Gamma' \mid \hat{e}}$. Then $\hat{e}'' = \hat{e}$ and $\mathcal{S}; \mathcal{P} \vdash \Gamma \mid A \overset{\hat{e}_1:\forall x:A_1.D}{\rightsquigarrow} \Gamma' \mid \hat{e}$.
- Let the derivation be of the shape $\dfrac{\mathcal{S}; \Gamma \vdash \theta : \Gamma' \qquad \mathcal{S}; \Gamma' \vdash \theta A \equiv \theta A' : \mathtt{o}}{\mathcal{S}; \mathcal{P} \vdash \Gamma \mid e\,C_1\{A\} \overset{\hat{e}':A'}{\rightsquigarrow} \Gamma \mid \hat{e}'}$. By the induction assumption, there is a mixed term $\hat{e}''$ and a substitution θ such

$$\lambda - .- : T \to t \to t, -_T- : T \to T \to T, \Pi_T - .- : T \to T \to T,$$

$$type_K : K, \Pi_K - .- : T \to T \to T,$$

$$\mathrm{Ctx} : \mathtt{type}, -_{,\mathrm{Ctx}} - : \mathrm{Ctx} \to T \to \mathrm{Ctx},$$

$$eq_t^a : t \to t \to \mathrm{Ctx} \to \mathtt{o}, eq_t^s : t \to t \to \mathrm{Ctx} \to \mathtt{o},$$

$$eq_T : T \to T \to \mathrm{Ctx} \to \mathtt{o}, eq_K : K \to K \to \mathrm{Ctx} \to \mathtt{o},$$

$$term : t \to T \to \mathrm{Ctx} \to \mathtt{o}, type : T \to K \to \mathrm{Ctx} \to \mathtt{o},$$

$$-\uparrow \equiv\ - : t \to t \to \mathtt{o}, -[-] \equiv - : t \to t \to t \to \mathtt{o}$$

$$whr : t \to t \to \mathtt{o}, proj : t \to T \to \mathrm{Ctx} \to \mathtt{o}$$

We use dashes $-f-$ to denote that the function symbol f is used in infix notation. Formally, we define different symbols, *e.g.*, $-_t-$ and $-_T-$ for application of terms and types respectively. In the rest of this chapter, we will drop the subscript where the notation is unambiguous. Since the signature of nameless LF is fixed, we keep it implicit in the encoded representation.

We define a calculus with two kinds of judgements, one for transforming refinement problems into goals and the other for transforming signatures into logic programs. These judgements are defined mutually in a similar way to the well-formedness judgements of nameless LF in Figures 2.7 and 2.8. We use $\mathcal{S}; \Gamma; M \vdash (G \mid A)$ to denote the transformation of a term M in a signature $\mathcal{S}$ and a context Γ to a goal G. The judgement also synthesises a type A of the term M. Similarly, $\mathcal{S}; \Gamma; A \vdash (G \mid L)$ denotes a transformation of a type A in $\mathcal{S}$ and Γ to a goal G while synthesising a kind L.

Definition 5.14

The judgements $\mathcal{S}; \Gamma; M \vdash (G \mid A)$ and $\mathcal{S}; \Gamma; A \vdash (G \mid L)$ are given by inference rules in Figures 5.2 and 5.3. Metavariables that do not occur among assumptions have an implicit freshness condition.

The inference judgement for a logic program generation is denoted by $\mathcal{S} \vdash_{\mathrm{Prog}} \mathcal{P}$ where $\mathcal{S}$ is a signature and $\mathcal{P}$ is a generated logic program. A generated logic program contains clauses that represent inference rules of type theory and clauses

5.2.2 From a refinement problem to a logic program

In this section, we explain how a term with metavariables is transformed into a goal, and a signature into a logic program. At the end of the section we state that, for a refinement problem, either a goal and a program exist or else the problem cannot be refined to a well-formed term. Our representation of nameless LF in the language of proof-relevant Horn-clause logic that we introduced in Chapter 3 requires that there are constants in the signature of the logic that encode judgements of nameless LF. In particular, we require:

- constant $\top$ for a trivially satisfied formula,
- constants for encoding sorts of nameless LF,
- constants for encoding de Bruijn indices,
- function symbols for encoding abstraction, application, Π type and kind formation and a designated kind that classifies types,
- the predicates eq_t^a and eq_t^s denote algorithmic and structural equality respectively of terms of a certain simple type in a context,
- the predicates eq_T and eq_K denote equality of terms of a certain simple kind, and equality of kinds in a context respectively.
- the predicates *term* and *type* denote, respectively, that a term or a type is well-formed in a context,
- predicates $A{\uparrow} \equiv A'$ to denote that a type A' is the result of shifting of A; and we use $A[M] \equiv A'$ to denote that A' is the result of substitution of A with M, and
- predicate *whr* to denote weak head reduction of terms and predicate *proj* to denote that a variable is present in a context (or a *projection* of a variable from a context).

In order to avoid an unnecessary syntactic clutter we keep the same syntax for Π types, abstraction and application in the internal language and in the logic. Hence, we can define the signature that contains the necessary symbols as follows:

Definition 5.13

$$\mathcal{S}_{empty} = \top : \mathtt{o}, N : \mathtt{type}, 0 : N, \sigma : N, t : \mathtt{type}, T : \mathtt{type}, K : \mathtt{type}, -_t- : t \to t \to t,$$

that $\hat{e}' = \theta(e\, C_2\{C'\{A\}\})\{\hat{e}''\}$, and $\mathcal{S}; \Gamma \vdash \theta' A \equiv \theta' A' : \mathtt{o}$. Then there is a derivation $\dfrac{\mathcal{S}; \Gamma \vdash \theta' : \Gamma' \qquad \mathcal{S}; \Gamma' \vdash \theta' A \equiv \theta' A' : \mathtt{o}}{\mathcal{S}; \mathcal{P} \vdash \Gamma \mid C'\{A\} \rightsquigarrow \Gamma' \mid \hat{e}''}$.

- Let the derivation be of the shape $\dfrac{\mathcal{S}; \mathcal{P} \vdash \Gamma \mid e\, C_1\{A\} \overset{\hat{e}_1 : A' ; D}{\rightsquigarrow} \Gamma' \mid \hat{e}}{\mathcal{S}; \mathcal{P} \vdash \Gamma \mid e\, C_1\{A\} \overset{\hat{e}_1 : A' \Rightarrow D}{\rightsquigarrow} \Gamma' \mid \hat{e}}$. By the induction assumption, there is a mixed term $\hat{e}''$, a substitution θ such that $\hat{e}' = \theta(e\, C_2\{C'\{A\}\})\{\hat{e}''\}$, and a derivation of $\mathcal{S}; \mathcal{P} \vdash \Gamma \mid C'\{A\} \overset{\hat{e}_1 : A' ; D}{\rightsquigarrow} \Gamma' \mid \hat{e}''$. Then there is a derivation $\dfrac{\mathcal{S}; \mathcal{P} \vdash \Gamma \mid C'\{A\} \overset{\hat{e}_1 : A' ; D}{\rightsquigarrow} \Gamma' \mid \hat{e}''}{\mathcal{S}; \mathcal{P} \vdash \Gamma \mid C'\{A\} \overset{\hat{e}_1 : A' \Rightarrow D}{\rightsquigarrow} \Gamma' \mid \hat{e}}$.
- Let the derivation be $\dfrac{\mathcal{S}; \mathcal{P} \vdash \Gamma, Y : A_1 \mid e\, C_1\{A\} \overset{\hat{e}_1 : D[Y/x]}{\rightsquigarrow} \Gamma' \mid \hat{e}}{\mathcal{S}; \mathcal{P} \vdash \Gamma \mid e\, C_1\{A\} \overset{\hat{e}_1 : \forall x A_1 . D}{\rightsquigarrow} \Gamma' \mid \hat{e}}$. By the induction assumption, there is a mixed term $\hat{e}''$, a substitution θ such that $\hat{e}' = \theta(e\, C_2\{C'\{A\}\})\{\hat{e}''\}$, and a derivation of $\mathcal{S}; \mathcal{P} \vdash \Gamma, Y : A_1 \mid e\, C_1\{A\} \overset{\hat{e}_1 : D[Y/x]}{\rightsquigarrow} \Gamma' \mid \hat{e}$. Then there is a derivation $\dfrac{\mathcal{S}; \mathcal{P} \vdash \Gamma, Y : A_1 \mid e\, C_1\{A\} \overset{\hat{e}_1 : D[Y/x]}{\rightsquigarrow} \Gamma' \mid \hat{e}}{\mathcal{S}; \mathcal{P} \vdash \Gamma \mid e\, C_1\{A\} \overset{\hat{e}_1 : \forall x A_1 . D}{\rightsquigarrow} \Gamma' \mid \hat{e}}$.
- Let the derivation be of the shape $\dfrac{\mathcal{S}; \Gamma \vdash \theta' : \Gamma' \qquad \mathcal{S}; \Gamma' \vdash \theta' A \equiv \theta' A' : \mathtt{o}}{\mathcal{S}; \mathcal{P} \vdash \Gamma \mid \langle M, C_1 \rangle\{A\} \overset{\hat{e}' : A'}{\rightsquigarrow} \Gamma \mid \hat{e}'}$. By the induction assumption, there is a mixed term $\hat{e}''$ and a substitution θ such that $\hat{e}' = \theta(\langle M, C_2 \rangle\{C'\{A\}\})\{\hat{e}''\}$, and $\mathcal{S}; \Gamma \vdash \theta' A \equiv \theta' A' : \mathtt{o}$. Then there is a derivation $\dfrac{\mathcal{S}; \Gamma \vdash \theta' : \Gamma' \qquad \mathcal{S}; \Gamma' \vdash \theta' A \equiv \theta' A' : \mathtt{o}}{\mathcal{S}; \mathcal{P} \vdash \Gamma \mid C'\{A\} \rightsquigarrow \Gamma' \mid \hat{e}''}$.
- Let the derivation be of the shape $\dfrac{\mathcal{S}; \mathcal{P} \vdash \Gamma \mid \langle M, C_1 \rangle\{A\} \overset{\hat{e}_1 : A' ; D}{\rightsquigarrow} \Gamma' \mid \hat{e}}{\mathcal{S}; \mathcal{P} \vdash \Gamma \mid \langle M, C_1 \rangle\{A\} \overset{\hat{e}_1 : A' \Rightarrow D}{\rightsquigarrow} \Gamma' \mid \hat{e}}$. By the induction assumption, there is a mixed term $\hat{e}''$, a substitution θ such that $\hat{e}' = \theta(\langle M, C_2 \rangle\{C'\{A\}\})\{\hat{e}''\}$, and a derivation of $\mathcal{S}; \mathcal{P} \vdash \Gamma \mid C'\{A\} \overset{\hat{e}_1 : A' ; D}{\rightsquigarrow} \Gamma' \mid \hat{e}''$. Then there is a derivation $\dfrac{\mathcal{S}; \mathcal{P} \vdash \Gamma \mid C'\{A\} \overset{\hat{e}_1 : A' \Rightarrow D}{\rightsquigarrow} \Gamma' \mid \hat{e}}{\mathcal{S}; \mathcal{P} \vdash \Gamma \mid C'\{A\} \overset{\hat{e}_1 : A' \Rightarrow D}{\rightsquigarrow} \Gamma' \mid \hat{e}}$.
- Let the derivation be $\dfrac{\mathcal{S}; \mathcal{P} \vdash \Gamma, Y : A_1 \mid \langle M, C_1 \rangle\{A\} \overset{\hat{e}_1 : D[Y/x]}{\rightsquigarrow} \Gamma' \mid \hat{e}}{\mathcal{S}; \mathcal{P} \vdash \Gamma \mid \langle M, C_1 \rangle\{A\} \overset{\hat{e}_1 : \forall x A_1 . D}{\rightsquigarrow} \Gamma' \mid \hat{e}}$. By the induction assumption, there is a mixed term $\hat{e}''$, a substitution θ such that $\hat{e}' = \theta(\langle M, C_2 \rangle\{C'\{A\}\})\{\hat{e}''\}$, and a derivation of $\mathcal{S}; \mathcal{P} \vdash \Gamma, Y : A_1 \mid \langle M, C_1 \rangle\{A\} \overset{\hat{e}_1 : D[Y/x]}{\rightsquigarrow} \Gamma' \mid \hat{e}$. Then there is a derivation $\dfrac{\mathcal{S}; \mathcal{P} \vdash \Gamma, Y : A_1 \mid \langle M, C_1 \rangle\{A\} \overset{\hat{e}_1 : D[Y/x]}{\rightsquigarrow} \Gamma' \mid \hat{e}}{\mathcal{S}; \mathcal{P} \vdash \Gamma \mid e\, C_1\{A\} \overset{\hat{e}_1 : \forall x A_1 . D}{\rightsquigarrow} \Gamma' \mid \hat{e}}$.

□

The above lemma allows us to obtain small-step derivations for identifying mixed terms. We refer to the property stated by the lemma as *subderivation property*.

This property will play an important role in the proof of soundness of the small-step semantics as it allows us to proceed by induction on derivations of small-step judgements. Finally, we state the soundness property of the small-step semantics.

Theorem 3.20 (Soundness)

If $\mathcal{S}; \mathcal{P} \vdash \cdot \mid G \rightsquigarrow \cdot \mid e$ *then* $\mathcal{S}; \mathcal{P} \longrightarrow e : G$.

In the following section, we introduce an extension of Horn-clause logic. The soundness of the small-step semantics of proof-relevant resolution in Horn-clause logic is a special case of a more general statement in the following section. Moreover, a proof of the statement requires a significant development that is carried out in the next chapter. Hence, we omit the proof here.

3.2 Logic of Hereditary Harrop Formulae

In this section we present the language of hereditary Harrop formulae. The language is obtained by extending the syntax of definite clauses and goals of Horn-clause logic. The extended syntax is given in the following definition.

Definition 3.21 (Syntax of goals and clauses)

$$\mathcal{D} \ni D \qquad := A \mid G \Rightarrow D \mid \forall x : A.D \qquad \textit{clauses}$$

$$\mathcal{G} \ni G \qquad := A \mid \exists x : A.G \mid D \Rightarrow G \mid \forall x : A.G \qquad \textit{goals}$$

Since we see hereditary Harrop formulae as an extension of Horn clauses we maintain the convention that the clauses in $\mathcal{D}$ are denoted by the identifier D and the goals in $\mathcal{G}$ are denoted by the identifier G. Clauses consist of atomic formulae, implication $\Rightarrow$, and universal quantification $\forall$ over a clause as in in the case of Horn-clause syntax. However, a goal instead of an atom is allowed on the left side of an implication. Goals consists of atomic formulae and existential quantification, as in the case of Horn clauses, and implication and universal quantification over a goal. In contrast with Horn clauses this definition allows nesting of implications in clauses and goals.

A *refinement* of a term is a pair of assignments (ρ, R) such that $\rho : ?_{\mathcal{V}} \to t$ is an assignment of (extended) terms to term-level metavariables and $R : ?_{\mathcal{B}} \to T$ is an assignment of (extended) types to type-level metavariables. We define application of refinement $(\rho, R)(-)$ to terms, types and kinds by induction on definition of the syntactic object.

Definition 5.12 (Refinement application)

Let $\rho : ?_{\mathcal{V}} \to t$ *be an assignment of terms and* $R : ?_{\mathcal{B}} \to T$ *be an assignment of types. Application of the refinement* (ρ, R) *to kinds, types and terms is defined by:*

$$(\rho, R)(\texttt{type}) = \texttt{type}$$

$$(\rho, R)(\Pi A.L) = \Pi(\rho, R)(A).(\rho, R)(L)$$

$$(\rho, R)(\alpha) = \alpha$$

$$(\rho, R)(?_A) = R(?_A)$$

$$(\rho, R)(\Pi A.B) = \Pi(\rho, R)(A).(\rho, R)(B)$$

$$(\rho, R)(AN) = (\rho, R)(A)(\rho, R)(N)$$

$$(\rho, R)(c) = c$$

$$(\rho, R)(\iota) = \iota$$

$$(\rho, R)(?_a) = \rho(?_a)$$

$$(\rho, R)(\lambda x : A.M) = \lambda x : (\rho, R)(A).(\rho, R)(M)$$

$$(\rho, R)(MN) = (\rho, R)(M)(\rho, R)(N)$$

A *solution* to a refinement problem t is a refinement (ρ, R) such that $(\rho, R)(t)$ is a well-formed term of nameless LF. That is, by Lemma 5.10, $(\rho, R)(t)$ does not contain neither term-level nor type-level metavariables.

Definition 5.9 (Extended nameless LF)

We define extended nameless types, terms and contexts *as follows:*

$t \ni M, N$	$:= \cdots \mid ?_a$	*terms*
$T \ni A, B$	$:= \cdots \mid ?_A$	*types*
$\mathrm{Ctx} \ni \Gamma$	$:= \cdots \mid \Gamma, ?_a : A$	*contexts*

The ellipsis in the definition are to be understood as the appropriate syntactic constructs of Definition 2.38 in Chapter 2. Note that we do not define an extended signature. We assume that the signature is always fixed and does not contain any metavariables. This does not pose any problem since well-typedness of signature does not depend on the term that is being refined.

We use $\mathrm{mtvar}(-)$ and $\mathrm{mvar}(-)$ to denote the sets of type-level and term-level metavariables respectively. The well-formedness judgements of the nameless LF remain the same as in Chapter 2 but now they are seen as defined on a subset of extended objects. These are the ground extended objects, as we show by the following lemma:

Lemma 5.10

Let L be an extended nameless kind, A an extended nameless type and M an extended nameless term. Let $\mathcal{S}$ be a signature and Γ a context.

- *If* $\mathcal{S}; \Gamma \vdash L : \mathtt{kind}$ *then* $\mathrm{mvar}(L) = \emptyset$ *and* $\mathrm{mtvar}(L) = \emptyset$,
- *if* $\mathcal{S}; \Gamma \vdash A : L$ *then* $\mathrm{mvar}(A) = \emptyset$ *and* $\mathrm{mtvar}(A) = \emptyset$, *and*
- *if* $\mathcal{S}; \Gamma \vdash M : A$ *then* $\mathrm{mvar}(M) = \emptyset$ *and* $\mathrm{mtvar}(M) = \emptyset$.

Proof. By induction on the derivation of judgements. □

A *refinement problem* is defined as a term in the extended syntax. A signature and a context of the term are kept implicit.

Example 5.11 (Refinement problem)

Taking our leading example, the term M' given by $(\mathtt{elim}_{\mathtt{maybe_A}}\ \mathtt{tt}\ 0)(\lambda ?_A.?_b)$ *is a refinement problem. The appropriate context is* $\Gamma_1 = \cdot, \mathtt{maybe_A}\ \mathtt{tt}$. *The signature in Figure 5.1 is adjusted to a nameless signature $\mathcal{S}$.*

$\boxed{\mathcal{S}; \Gamma \vdash G : \mathtt{o}}$

$$\frac{\mathcal{S}; \Gamma, X \vdash D : \mathtt{o} \qquad \mathcal{S}; \Gamma, X \vdash G : \mathtt{o}}{\mathcal{S}; \Gamma \vdash D \Rightarrow G : \mathtt{o}} \qquad \frac{\mathcal{S}; \Gamma, X \vdash M : \mathtt{o}}{\mathcal{S}; \Gamma \vdash \forall X.M : \mathtt{o}}$$

Figure 3.6: Well formedness of goals

To ensure that clauses and goals indeed consists of atomic formulae in positions of types we extend well-formedness judgements. However, since the syntactic constructs of clauses are the same as in the case of Horn clauses we only need to extend well-formedness judgement $\mathcal{S}; \Gamma \vdash G : \mathtt{o}$ of goals.

Definition 3.22

The judgement $\mathcal{S}; \Gamma \vdash G : \mathtt{o}$ is given by inference rules in Figure 3.6.

We list only inference rules for the new syntactic constructs. Other inference rules are the same as in Figure 2.5.

The structure of programs stays the same, with respect to the extended definition of clauses in $\mathcal{D}$. The well-formedness judgement $\mathcal{S} \vdash \mathcal{P}$ remains the same up to the extended definition of clauses in $\mathcal{D}$ and the judgement $\mathcal{S}; \Gamma \vdash D : \mathtt{o}$.

The presence of nested implications that is allowed by the extended syntax of definite clauses requires an extension of the syntax of proof terms:

Definition 3.23 (Proof terms)

$\mathrm{PT} \ni e$	$:= \kappa \mid e\ e \mid \langle M, e \rangle \mid \lambda\kappa.e$	*proof terms*

We extend proof terms with abstraction over atomic proof-term symbols in $\mathcal{K}$.

3.2.1 Big-step operational semantics

We extend big-step operational semantics of proof relevant resolution for Horn-clause logic that we introduced in Section 3.1.1 to the language of hereditary Harrop formulae.

Definition 3.24 (Operational semantics, big-step)

The judgements $\mathcal{S}; \mathcal{P} \longrightarrow e : G$ and $\mathcal{S}; \mathcal{P} \xrightarrow{e':D} e : G$ for logic of hereditary Harrop

$\boxed{\mathcal{S};\mathcal{P} \longrightarrow e : G}$

$$\frac{\mathcal{S};\mathcal{P},\kappa : D \longrightarrow e : G}{\mathcal{S};\mathcal{P},\kappa : D \longrightarrow \lambda\kappa.e : D \Rightarrow G}$$

$$\frac{\mathcal{S},c : A;\mathcal{P} \longrightarrow e[c/x] : G[c/x]}{\mathcal{S};\mathcal{P} \longrightarrow e : \forall x : A.G}$$

Figure 3.7: Right introduction rules

formulae are given by inference rules in figures 3.2, 3.3, and 3.7.

There are no new backchaining inference rules with respect to Horn-clause logic as the syntactic forms of definite clauses remain the same. New right-introduction rules that correspond to new syntactic forms of goals are listed in Figure 3.7. Note that the program is no longer static in the course of resolution but gets extended with new clauses in the case of a goal in an implicational form. This justifies having program as a parameter of the judgement and, since we aim to treat different fragments uniformly, to keep it as a part of the judgement even in the previous section.

We proceed with a demonstration of the use of hereditary Harrop formulae. We further develop our running example that utilises encoding of natural numbers.

Example 3.25

Let $\mathcal{S}$ be the following signature:

$$\mathcal{S} = a : \mathtt{type}, z : a, s : a \to a, even : a \to \mathtt{o}, odd : a \to \mathtt{o}$$

The predicates even and odd are given interpretation by the following program:

$$\mathcal{P} = \kappa_z : odd(z),$$

$$\kappa_e : \forall x : a.odd\, x \Rightarrow even\,(s\, x)$$

$$\kappa_o : \forall x : a.even\, x \Rightarrow odd\,(s\, x)$$

We can resolve atomic goals similar to Example 3.9 but we can also resolve hypothetical goals in implicational form. For example, the goal $\forall x : a.even\, x \Rightarrow even\,(s\,(s\, x))$ is resolved as follows:

variable w in particular. A clause κ_{proj_w} is used to project the variable w from the context. Such lenience allows us to avoid excessive technical detail and to postpone further discussion of the exact shape of the clauses until the next section since it depends on the de Bruijn representation of variables. We omit clause bodies. denoted by _, as we did in previous chapters.

For the moment, we are just interested in the computed proof term:

$$\kappa_{\mathtt{elim}}\ \kappa_{\mathtt{elim}_{\equiv_{\mathtt{bool}}}}\ \kappa_{proj_w} \kappa_{subst_{\mathtt{A}}}$$

Note that by resolving goal II in Example 5.5, we obtain a substitution θ that assigns the type A to the logic variable $?_B$, *i.e.* $\theta(?_B) = A$. At the same time, the proof term computed by the derivation in Example 5.8 is interpreted as a solution $(\mathtt{elim}_{\equiv_{\mathtt{bool}}}\ w)$ for the term-level metavariable $?_b$. However, the proof term can be used to reconstruct the derivation of well-typedness of the judgement $m : \mathtt{maybe}_{\mathtt{A}}\ \mathtt{tt}, w : \mathtt{tt} \equiv_{\mathtt{bool}} \mathtt{ff} \vdash \mathtt{elim}_{\equiv_{\mathtt{bool}}}\ w : \mathtt{A}$ as well. In general, a substitution is interpreted as a solution to a type-level metavariable and a proof term as a solution to a term-level metavariable. The remaining solution for $?_A$ is computed using similar methodology, and we omit the details here.

5.2 Refinement in Nameless LF

Following the ideas we described in the previous section, we present a translation of a refinement problem into Horn-clause logic with explicit proof terms. First, we extend the language of nameless LF with metavariables, which allows us to capture incomplete terms. Next, we give a calculus for transformation of an incomplete term to a goal and a program.

5.2.1 Refinement problem

We capture missing information in nameless LF terms by metavariables. We assume infinitely countable disjoint sets $?_{\mathcal{B}}$ and $?_{\mathcal{V}}$ that stand for omitted types and terms and we call elements of these sets type-level and term-level metavariables respectively. We use identifiers $?_a$, $?_b$, *etc.* to denote elements of $?_{\mathcal{V}}$ and identifiers $?_A$, $?_B$, *etc.* to denote elements of $?_{\mathcal{B}}$. The extended syntax is defined as follows:

terms of the logic programs. This is possible since we assume that the internal language is represented using *de Bruijn indices* for variables. Finally, in order to avoid unnecessary syntactic clutter, in this chapter we omit explicit quantifiers. We assume that clauses are implicitly universally quantified and that goals are implicitly existentially quantified.

Example 5.8

The presence of $w : \mathtt{tt} \equiv_{\mathtt{bool}} \mathtt{ff}$ *in the context allows us to use the clause* $\mathtt{elim}_{\equiv_{\mathtt{bool}}}$ *to resolve the goal term* $(?_M\, ?_N)\ \mathtt{A}\,[m : \mathtt{maybe}_{\mathtt{A}}\ \ \mathtt{tt}, w : \mathtt{tt} \equiv_{\mathtt{bool}} \mathtt{ff}]$. *The implicit quantification of variables* $?_M$ *and* $?_N$ *is made explicit by the context* $?_M : t, ?_N : t$

$$?_M : t, ?_N : t \mid term\,(?_M\, ?_N)\ \mathtt{A}\,[m : \mathtt{maybe}_{\mathtt{A}}, w : \mathtt{tt} \equiv_{\mathtt{bool}} \mathtt{ff}] \rightsquigarrow$$

$$?_M : t, ?_N : t \mid (term\,(?_M\, ?_N)\ \mathtt{A}\,[m : \mathtt{maybe}_{\mathtt{A}}, w : \mathtt{tt} \equiv_{\mathtt{bool}} \mathtt{ff}])^{\kappa_{\mathtt{elim}} :\!-} \rightsquigarrow^*$$

$$?_M : t, ?_N : t, ?_A : T, ?_B : T \mid \kappa_{\mathtt{elim}}\,(term\,?_M\,(\Pi x : ?_A.\,\mathtt{A})\,[\ldots] \wedge$$
$$term\,?_N\,?_A\,[\ldots, w : \mathtt{tt} \equiv_{\mathtt{bool}} \mathtt{ff}] \wedge \mathtt{A}[?_N/x] \equiv ?_B) \rightsquigarrow$$

$$?_M : t, ?_N : t, ?_A : T, ?_B : T \mid \kappa_{\mathtt{elim}}\,((term\,?_M\,(\Pi x : ?_A.\,\mathtt{A})\,[\ldots])^{\kappa_{\mathtt{elim}_{\equiv_{\mathtt{bool}}}} :\!-} \wedge$$
$$term\,?_N\,?_A\,[\ldots, w : \mathtt{tt} \equiv_{\mathtt{bool}} \mathtt{ff}] \wedge \mathtt{A}[?_N/x] \equiv ?_B) \rightsquigarrow^*$$

$$?_N : t, ?_B : T \mid \kappa_{\mathtt{elim}}\,\kappa_{\mathtt{elim}_{\equiv_{\mathtt{bool}}}}\,(term\,?_N\ \mathtt{tt} \equiv_{\mathtt{bool}} \mathtt{ff}\,[\ldots, w : \mathtt{tt} \equiv_{\mathtt{bool}} \mathtt{ff}] \wedge$$
$$\mathtt{A}[?_N/x] \equiv ?_B) \rightsquigarrow$$

$$?_N : t, ?_B : T \mid \kappa_{\mathtt{elim}}\,\kappa_{\mathtt{elim}_{\equiv_{\mathtt{bool}}}}\,((term\,?_N\ \mathtt{tt} \equiv_{\mathtt{bool}} \mathtt{ff}\,[\ldots, w : \mathtt{tt} \equiv_{\mathtt{bool}} \mathtt{ff}])^{\kappa_{prog_w}}$$
$$\wedge\, \mathtt{A}[?_N/x] \equiv ?_B) \rightsquigarrow^*$$

$$?_B : T \mid \kappa_{\mathtt{elim}}\,\kappa_{\mathtt{elim}_{\equiv_{\mathtt{bool}}}}\,\kappa_{proj_w}\,(\mathtt{A}[w/x] \equiv ?_B)^{\kappa_{subst_{\mathtt{A}}}} \rightsquigarrow^*$$

$$\cdot \mid \kappa_{\mathtt{elim}}\,\kappa_{\mathtt{elim}_{\equiv_{\mathtt{bool}}}}\,\kappa_{proj_w}\,\kappa_{subst_{\mathtt{A}}}$$

The resolution here is a trace of the small-step semantics we introduced in Chapter 3 up to certain lenience we allow ourselves with treatment of variable names, the

$$\dfrac{\dfrac{}{\mathcal{S}, c : a; \mathcal{P}, \kappa_x : even\,c \xrightarrow{\kappa_x : even\,c} \kappa_x : even\,c} \qquad \kappa_x : even\,c \in \mathcal{P}, \kappa_x : even\,c}{\mathcal{S}, c : a; \mathcal{P}, \kappa_x : even\,c \longrightarrow \kappa_x : even\,c}$$

$$\dfrac{\dfrac{\vdots}{\mathcal{S}, c : a; \mathcal{P}, \kappa_x : even\,c \longrightarrow \kappa_x : even\,c} \qquad \dfrac{}{\mathcal{S}, c : a; \mathcal{P}, \kappa_x : even\,c \xrightarrow{\kappa_o\,\kappa_x : odd\,(s\,c)} \kappa_o\,\kappa_x : odd\,(s\,c)}}{\dfrac{\mathcal{S}, c : a; \mathcal{P}, \kappa_x : even\,c \xrightarrow{\kappa_o : even\,c \Rightarrow odd\,(s\,c)} \kappa_o\,\kappa_x : odd\,(s\,c)}{\dfrac{\mathcal{S}, c : a; \mathcal{P}, \kappa_x : even\,c \xrightarrow{\kappa_o : \forall x : a.\,even\,x \Rightarrow odd\,(s\,x)} \kappa_o\,\kappa_x : odd\,(s\,c)}{\mathcal{S}, c : a; \mathcal{P}, \kappa_x : even\,c \longrightarrow \kappa_o\,\kappa_x : odd\,(s\,c)}}} \qquad \kappa_o : \_ \in \mathcal{P}$$

$$\dfrac{\dfrac{\vdots}{\dfrac{\mathcal{S}, c : a; \mathcal{P}, \kappa_x : even\,c \xrightarrow{\kappa_e\,(\kappa_o\,\kappa_x) : even\,(s\,(s\,c))} \kappa_e\,(\kappa_o\,\kappa_x) : even\,(s\,(s\,c))}{\mathcal{S}, c : a; \mathcal{P}, \kappa_x : even\,c \xrightarrow{\kappa_e : odd\,(s\,c) \Rightarrow even\,(s\,(s\,c))} \kappa_e\,(\kappa_o\,\kappa_x) : even\,(s\,(s\,c))} \qquad \mathcal{S}; \cdot \vdash s\,(s\,c) : a}{\mathcal{S}, c : a; \mathcal{P}, \kappa_x : even\,c \xrightarrow{\kappa_e : \forall x : a.\,odd\,x \Rightarrow even\,(s\,x)} \kappa_e\,(\kappa_o\,\kappa_x) : even\,(s\,(s\,c))}$$

$$\dfrac{\dfrac{\dfrac{\vdots}{\mathcal{S}, c : a; \mathcal{P}, \kappa_x : even\,c \xrightarrow{\kappa_e : \forall x : a.\,odd\,x \Rightarrow even\,(s\,x)} \kappa_e\,(\kappa_o\,\kappa_x) : even\,(s\,(s\,c))} \qquad \kappa_e : \forall x : a.\,odd\,x \Rightarrow even\,(s\,x) \in \mathcal{P}}{\mathcal{S}, c : a; \mathcal{P}, \kappa_x : even\,c \longrightarrow \kappa_e\,(\kappa_o\,\kappa_x) : even\,(s\,(s\,c))}}{\mathcal{S}, c : a; \mathcal{P} \longrightarrow \lambda\kappa_x.\kappa_e\,(\kappa_o\,\kappa_x) : even\,c \Rightarrow even\,(s\,(s\,c))}}{\mathcal{S}; \mathcal{P} \longrightarrow \lambda\kappa_x.\kappa_e(\kappa_o\kappa_x) : \forall x : a.even\,x \Rightarrow even\,(s\,(s\,x))}$$

We separate the derivation into four partial derivations that compose in the obvious way. We abbreviate the clause $\forall x : a.even\,x \Rightarrow odd\,(s\,x)$ *by* $\_$ *and we use* $\kappa : D \in \mathcal{P}$ *instead* $\kappa : D \in \mathcal{P}, \kappa_x : even\,c$ *where the clause* D *is in* $\mathcal{P}$.

3.2.2 Small-step operational semantics

In this section, we extend small-step operational semantics to the language of hereditary Harrop formulae. First, we need to adjust the definition of mixed terms and rewriting contexts to accommodate for new syntactic constructs.

Definition 3.26 (Mixed terms and rewriting contexts)

$$\mathrm{MT} \ni \hat{e} \qquad := \kappa \mid G \mid \hat{e}\ \hat{e} \mid \langle M, \hat{e} \rangle \mid \lambda\kappa.\hat{e} \qquad \textit{mixed terms}$$

$$\mathcal{R} \ni C \qquad := \bullet \mid \hat{e}\ C \mid \langle M, C \rangle \mid \lambda\kappa.C \qquad \textit{rewriting contexts}$$

We keep using the identifier $\hat{e}$ for mixed terms. Extended contexts posses a property that corresponds to Proposition 3.15.

Proposition 3.27

Let C_1 *and* C_2 *be rewriting contexts,* G *a goal and* $\hat{e}$ *a mixed term. If* $C_1\{G\} = C_2\{\hat{e}\}$ *then there is a unique* C' *such that* $\hat{e} = C'[G]$.

$\boxed{\mathcal{S};\mathcal{P} \vdash \Gamma \mid \hat{e} \rightsquigarrow \Gamma' \mid \hat{e}'}$

$$\frac{\mathcal{S};\mathcal{P},\kappa : D \vdash \Gamma \mid C\{\lambda\kappa.G\} \rightsquigarrow \Gamma' \mid \hat{e}}{\mathcal{S};\mathcal{P} \vdash \Gamma \mid C\{D \Rightarrow G\} \rightsquigarrow \Gamma' \mid \hat{e}}$$

$$\frac{\mathcal{S},c : A;\mathcal{P} \vdash \Gamma \mid C\{G[c/x]\} \rightsquigarrow \Gamma' \mid \hat{e}}{\mathcal{S};\mathcal{P} \vdash \Gamma \mid C\{\forall x : A.G\} \rightsquigarrow \Gamma' \mid \hat{e}}$$

Figure 3.8: Right introduction rules, small-step

Proof. By induction on C_1 and C_2. The new compatible cases w.r.t. the proof of Proposition 3.15 are:

- $C_1 = \lambda\kappa.C_1'$ and $C_2 = \bullet$. Then $\hat{e} = \lambda\kappa.C_1'\{G\}$ and $C' = \lambda\kappa.C_1$.
- $C_1 = \lambda\kappa C_1'$ and $C_2 = \lambda\kappa.C_2'$. Then $C_1'\{G\} = C_2'\{\hat{e}\}$ and from induction hypothesis there is unique C' such that $\hat{e} = C'\{G\}$.

□

The small-step semantics is, as was the case with the big-step semantics, given by extending right-introduction rules. Since we do not extend syntax of clauses, the backchaining judgement does not change.

Definition 3.28 (Operational semantics, small-step)
The judgements $\mathcal{S};\mathcal{P} \vdash \Gamma \mid \hat{e} \rightsquigarrow \Gamma' \mid \hat{e}'$, and $\mathcal{S};\mathcal{P} \vdash \Gamma \mid \hat{e} \overset{\hat{e}'':D}{\rightsquigarrow} \Gamma' \mid \hat{e}'$ are given by inference rules in Figures 3.4, 3.5, and 3.8.

The small-step semantics posses subderivation property (Lemma 3.19).

Lemma 3.29 (Subderivations)

1. *If $\mathcal{S};\mathcal{P} \vdash \Gamma \mid C\{\hat{e}\} \rightsquigarrow \Gamma' \mid \hat{e}'$ then there is a mixed term $\hat{e}''$ and a substitution θ such that $\hat{e}' = (\theta C)\{\hat{e}''\}$ and $\mathcal{S};\mathcal{P} \vdash \Gamma \mid \hat{e} \rightsquigarrow \Gamma \mid \hat{e}''$*
2. *If $\mathcal{S};\mathcal{P} \vdash \Gamma \mid C\{\hat{e}\} \overset{\hat{e}_1:D_1}{\rightsquigarrow} \Gamma' \mid \hat{e}'$ then there is a mixed term $\hat{e}''$ and a substitution θ such that $\hat{e}' = (\theta C)\{\hat{e}''\}$ and $\mathcal{S};\mathcal{P} \vdash \Gamma \mid \hat{e} \overset{\hat{e}_1:D_1}{\rightsquigarrow} \Gamma \mid \hat{e}''$*

Proof. By simultaneous structural induction on the derivation and the rewriting context. We list only new cases w.r.t. Lemma 3.19.

Part 1

- Let the derivation be $\dfrac{\mathcal{S};\mathcal{P},\kappa' : D \vdash \Gamma \mid \lambda\kappa.C_1\{\lambda\kappa'.G\} \rightsquigarrow \Gamma' \mid \hat{e}'}{\mathcal{S};\mathcal{P} \vdash \Gamma \mid \lambda\kappa.C_1\{D \Rightarrow G\} \rightsquigarrow \Gamma' \mid \hat{e}'}$. By the induction assumption, there is a mixed term $\hat{e}''$, a substitution θ such that

The unifiers that are computed by proof-relevant resolution give an assignment of types to type-level metavariables. At the same time, the computed proof terms are interpreted as an assignment of terms to term-level metavariables.

Example 5.6
Assuming the term $\lambda(w : ?_A).?_b$ is of type $(\mathtt{tt} \equiv_{\mathtt{bool}} \mathtt{ff}) \to \mathtt{A}$, type checking places restrictions on the term $?_b$:

$$\frac{m : \mathtt{maybe}_{\mathtt{A}}\ \ \mathtt{tt} \vdash \mathtt{tt} \equiv_{\mathtt{bool}} \mathtt{ff} : \mathtt{type} \qquad m : \mathtt{maybe}_{\mathtt{A}}\ \ \mathtt{tt}, w : \mathtt{tt} \equiv_{\mathtt{bool}} \mathtt{ff} \vdash ?_b : \mathtt{A}}{m : \mathtt{maybe}_{\mathtt{A}}\ \ \mathtt{tt} \vdash \lambda(w : \mathtt{tt} \equiv_{\mathtt{bool}} \mathtt{ff}).?_b : \mathtt{tt} \equiv_{\mathtt{bool}} \mathtt{ff} \to \mathtt{A}}$$

That is, $?_b$ needs to be a well-typed term of type $\mathtt{A}$ in a context consisting of m and w. When resolving the computed goal, $?_b$ will be bound to a proof term that we use to extract the required term.

Our translation will turn this constant into a clause in the generated logic program. Additionally, our translation will include clauses that describe inference rules of the type theory of the internal language.

Example 5.7
Recall that in the signature there is a constant $\mathtt{elim}_{\equiv_{\mathtt{bool}}}$ of type $\mathtt{tt} \equiv_{\mathtt{bool}} \mathtt{ff} \to \mathtt{A}$. There will be a clause that corresponds to the inference rule for elimination of a Π type as well:

$$\kappa_{\mathtt{elim}_{\equiv_{\mathtt{bool}}}} :\ term\ \mathtt{elim}_{\equiv_{\mathtt{bool}}}\ (\Pi x : \mathtt{tt} \equiv_{\mathtt{bool}} \mathtt{ff}\ .\ \mathtt{A})\ ?_\Gamma \Leftarrow$$

$$\kappa_{\mathtt{elim}} :\ term\ ?_M\ ?_N\ ?_B\ ?_\Gamma \Leftarrow term\ ?_M\ (\Pi x : ?_A.?_{B'})\ ?_\Gamma$$

$$\wedge\, term\ ?_N\ ?_A\ ?_\Gamma \wedge\, ?_{B'}[?_N/x] \equiv ?_B$$

In these clauses, $?_M$, $?_N$, $?_A$, $?_B$, $?_{B'}$ and $?_\Gamma$ are logic variables, *i.e. variables of the Horn-clause logic.*

By an abuse of notation, we use the same symbols for metavariables of the internal language and logic variables in the logic programs generated by the refinement algorithm. We also use the same notation for objects of the internal language and

The missing information comprises the two types $?_A$ and $?_B$ and the term $?_b$ for the constructor `nothing`*. Obtaining types $?_A$, $?_B$ amounts to type inference (in the internal language, as opposed to checking in the surface language), whereas obtaining the term $?_b$ amounts to term synthesis.*

In this thesis, we use the notion *refinement* to refer to the combined problem of type inference and term synthesis. We make use of *proof-relevant* Horn-clause logic to solve *refinement problems*. We translate refinement problems into the syntax of logic programs. The *refinement algorithm* that we propose takes a signature and a term with metavariables in the extended internal language to a logic program and a goal in proof-relevant Horn-clause logic.

Example 5.5

Consider the inference rule Π-T-Elim *in LF. This inference rule generalises the inference rule* App *that we used to motivate Horn clauses in type inference in the Introduction (Chapter 1).*

$$\frac{\Gamma \vdash M : \Pi x : A.B \qquad \Gamma \vdash N : A}{\Gamma \vdash MN : B[N/x]}\ \Pi\text{-T-Elim}$$

When type checking the term $t_{\texttt{fromJust}}$ an application of $\texttt{elim}_{\texttt{maybe}_\texttt{A}}\ \texttt{tt}\ m$ to the term $\lambda(w : ?_A).?_b$ in the context $m : \texttt{maybe}_\texttt{A}\ \texttt{tt}$ needs to be type checked. This amounts to providing a derivation of the typing judgement that contains the following instance of the rule Π-T-Elim*:*

$$\frac{\begin{array}{c} m : \texttt{maybe}_\texttt{A}\ \texttt{tt} \vdash \texttt{elim}_{\texttt{maybe}_\texttt{A}}\ \texttt{tt}\ m \\ : (\texttt{tt} \equiv_{\texttt{bool}} \texttt{ff} \to \texttt{A}) \to \cdots \to \texttt{A} \end{array} \qquad m : \texttt{maybe}_\texttt{A}\ \texttt{tt} \vdash \lambda(w : ?_A).?_b : ?_A \to ?_B}{m : \texttt{maybe}_\texttt{A}\ \texttt{tt} \vdash (\texttt{elim}_{\texttt{maybe}_\texttt{A}}\ \texttt{tt}\ m)\ (\lambda(w : ?_A).?_b) : (\texttt{tt} \equiv_{\texttt{bool}} \texttt{tt} \to \texttt{A} \to \texttt{A}) \to \texttt{A}}$$

For the above inference step to be a valid instance of the inference rule Π-T-Elim*, it is necessary that $(\texttt{tt} \equiv_{\texttt{bool}} \texttt{ff}) = ?_A$ and $\texttt{A} = ?_B$. This is reflected in the goal:*

$$((\texttt{tt} \equiv_{\texttt{bool}} \texttt{ff}) = ?_A) \wedge (\texttt{A} = ?_B) \wedge G_{(\texttt{elim}_{\texttt{maybe}_\texttt{A}}\ \texttt{tt}\ m)} \wedge G_{\lambda(w:?_A).?_b} \qquad \text{(II)}$$

The additional goals $G_{(\texttt{elim}_{\texttt{maybe}_\texttt{A}}\ \texttt{tt}\ m)}$ and $G_{\lambda(w:?_A).?_b}$ are recursively generated for the terms $\texttt{elim}_{\texttt{maybe}_\texttt{A}}\ \texttt{tt}\ m$ and $\lambda(w : ?_A).?_b$, respectively.

$\hat{e}' = \theta(\lambda\kappa.C_2\{C'\{A\}\})\{\hat{e}''\}$, and a derivation of $\mathcal{S}; \mathcal{P} \vdash \Gamma \mid C'\{D \Rightarrow G\} \rightsquigarrow \Gamma' \mid \hat{e}''$. Then there is a derivation $\dfrac{\mathcal{S}; \mathcal{P}, \kappa : D \vdash \Gamma \mid C'\{\lambda\kappa.G\} \rightsquigarrow \Gamma' \mid \hat{e}''}{\mathcal{S}; \mathcal{P} \vdash \Gamma \mid C'\{D \Rightarrow G\} \rightsquigarrow \Gamma' \mid \hat{e}''}$.

- Let the derivation be $\dfrac{\mathcal{S}; \mathcal{P} \vdash \Gamma, x : A \mid \lambda\kappa.C_1\{G\} \rightsquigarrow \Gamma' \mid \hat{e}'}{\mathcal{S}; \mathcal{P} \vdash \Gamma \mid \lambda\kappa.C_1\{\forall x : A.G\} \rightsquigarrow \Gamma' \mid \hat{e}'}$. By the induction assumption, there is a mixed term $\hat{e}''$, a substitution θ such that $\hat{e}' = \theta(\lambda\kappa.C_2\{C'\{A\}\})\{\hat{e}''\}$, and a derivation of $\mathcal{S}; \mathcal{P} \vdash \Gamma, x : A \mid C'\{G\} \rightsquigarrow \Gamma' \mid \hat{e}''$. Then there is a derivation $\dfrac{\mathcal{S}; \mathcal{P}, \kappa : D \vdash \Gamma, x : A \mid C'\{G\} \rightsquigarrow \Gamma' \mid \hat{e}''}{\mathcal{S}; \mathcal{P} \vdash \Gamma \mid C'\{\forall x : A.G\} \rightsquigarrow \Gamma' \mid \hat{e}''}$.

Part 2

- Let the derivation be of the shape $\dfrac{\mathcal{S}; \Gamma \vdash \theta' : \Gamma' \qquad \mathcal{S}; \Gamma' \vdash \theta' A \equiv \theta' A' : \texttt{o}}{\mathcal{S}; \mathcal{P} \vdash \Gamma \mid \lambda\kappa.C_1\{A\} \overset{\hat{e}' : A'}{\rightsquigarrow} \Gamma' \mid \hat{e}'}$. By the induction assumption, there is a mixed term $\hat{e}''$ and a substitution θ such that $\hat{e}' = \theta(\lambda\kappa.C_2\{C'\{A\}\})\{\hat{e}''\}$, and $\mathcal{S}; \Gamma \vdash \theta' A \equiv \theta' A' : \texttt{o}$. Then there is a derivation $\dfrac{\mathcal{S}; \Gamma \vdash \theta' : \Gamma' \qquad \mathcal{S}; \Gamma' \vdash \theta' A \equiv \theta' A' : \texttt{o}}{\mathcal{S}; \mathcal{P} \vdash \Gamma \mid C'\{A\} \rightsquigarrow \Gamma' \mid \hat{e}''}$.
- Let the derivation be of the shape $\dfrac{\mathcal{S}; \mathcal{P}, \kappa : D \vdash \Gamma \mid e\,\mathcal{R}_1\{A\} \overset{\hat{e}_1\, A' : D}{\rightsquigarrow} \Gamma' \mid \hat{e}}{\mathcal{S}; \mathcal{P} \vdash \Gamma \mid \lambda\kappa.\mathcal{R}_1\{A\} \overset{\hat{e}_1 : A' \Rightarrow D}{\rightsquigarrow} \Gamma' \mid \hat{e}}$. By the induction assumption, there is a mixed term $\hat{e}''$, a substitution θ such that $\hat{e}' = \theta(e\,C_2\{C'\{A\}\})\{\hat{e}''\}$, and a derivation of $\mathcal{S}; \mathcal{P} \vdash \Gamma \mid \mathcal{R}'\{A\} \overset{\hat{e}_1\, A' : D}{\rightsquigarrow} \Gamma' \mid \hat{e}''$. Then there is a derivation $\dfrac{\mathcal{S}; \mathcal{P} \vdash \Gamma \mid \mathcal{R}'\{A\} \overset{\hat{e}_1\, A' : D}{\rightsquigarrow} \Gamma' \mid \hat{e}''}{\mathcal{S}; \mathcal{P} \vdash \Gamma \mid \mathcal{R}'\{A\} \overset{\hat{e}_1 : A' \Rightarrow D}{\rightsquigarrow} \Gamma' \mid \hat{e}}$.

□

The example we used for illustration of the big-step semantics can also be resolved in small-step semantics.

Example 3.30

Consider the signature $\mathcal{S}$ and the program $\mathcal{P}$ from Example 3.25. The goal $\forall x : a.even\, x \Rightarrow even\,(s\,(s\,x))$ is resolved in small steps to a proof term $\lambda\kappa_x.\kappa_e\,(\kappa_o\kappa_x)$:

$$\cdot \mid \forall x : a.even\, x \Rightarrow even\,(s\,(s\,x)) \rightsquigarrow \cdot \mid even\, c \Rightarrow even\,(s\,(s\,c)) \rightsquigarrow$$

$$\cdot \mid \lambda\kappa_x.even\,(s\,(s\,c)) \rightsquigarrow \cdot \mid \lambda\kappa_x.even\,(s\,(s\,c))^{\kappa_e : \forall x : a.odd\, x \Rightarrow even\,(s\,x)} \rightsquigarrow$$

$$X : a \mid \lambda\kappa_x.even\,(s\,(s\,c))^{\kappa_e : odd\, X \Rightarrow even\,(s\,X)} \rightsquigarrow$$

$$X : a \mid \lambda\kappa_x.even\,(s\,(s\,c))^{\kappa_e (odd\, X) : even\,(s\,X)} \rightsquigarrow$$

$$\cdot \mid \lambda\kappa_x.\kappa_e\,(odd\,(s\,c)) \rightsquigarrow \cdot \mid \lambda\kappa_x.\kappa_e\,(odd\,(s\,c))^{\kappa_o : \forall x : a.even\, x \Rightarrow odd\,(s\,x)} \rightsquigarrow$$

$$Y : a \mid \lambda\kappa_x.\kappa_e(odd\,(s\,c))^{\kappa_o : even\, Y \Rightarrow odd\,(s\,Y)} \rightsquigarrow \cdot \mid \lambda\kappa_x.\kappa_e\,(\kappa_o\,(even\, c)) \rightsquigarrow$$

$$\cdot \mid \lambda\kappa_x.\kappa_e\,(\kappa_o\,(even\,c)^{\kappa_x:even\,c}) \rightsquigarrow \cdot \mid \lambda\kappa_x.\kappa_e\,(\kappa_o\,\kappa_x)$$

We conclude this section by statement of soundness of the small-step semantics. However, as we saw in Example 3.18, small-step semantics does not necessarily produce judgements with empty context on the right of $\rightsquigarrow$. We can relax this condition and allow an arbitrary context Γ'. It is then necessary to transform goals of the big-step semantics. In order to do so, we introduce a notion of universal quantification with a variable context.

Definition 3.31

$$\forall_{Ctx}\,\cdot.G = G$$

$$\forall_{Ctx}(\Gamma, x : A).G = \forall_{Ctx}\Gamma.(\forall x : A.G)$$

We call this transformation a *generalisation* of a goal with a context. Finally, we state the soundness property for small-step semantics of proof-relevant resolution in the logic of hereditary Harrop formulae using generalisation.

Theorem 3.32 (Generalised soundness)

If $\mathcal{S};\mathcal{P} \vdash \cdot \mid G \rightsquigarrow \Gamma \mid e$ *then* $\mathcal{S};\mathcal{P} \longrightarrow e : \forall_{Ctx}\Gamma.G$

Our proof of the theorem requires further technical development. In particular, we need to develop a notion of logical relation for mixed terms. Logical relation will allow us to reason on intermediate subderivations of the big-step and the small-step semantics by structural induction and to guarantee that such subderivations are well-formed. We devote the following chapter to development of the logical relation and a proof of the above statement will constitute the main result of the next chapter.

3.3 Related Work

The big-step semantics we present in this chapter is based on the semantics of uniform proofs (Miller et al., 1991) and λProlog (Miller and Nadathur, 2012). However, unlike our work, the work of Miller *et al.* is carried out using only simple types, which


```
A       : type
bool    : type
ff tt   : bool

(≡bool) : bool → bool → type
refl    : Π(b:bool).b ≡bool b
elim≡bool: tt ≡bool ff → A

maybeA  : bool → type
nothing : maybeA ff
just    : A → maybeA tt
elimmaybeA: Π(b:bool).maybeA b
          → (b ≡bool ff → A)
          → (b ≡bool tt → A → A)
          → A
```

Figure 5.1: Signature for encoding `fromJust`

not contain all the information required by the type theory of the internal language and that this information needs to be inferred, preferably by an automated tool and without any human intervention.

Example 5.3

The function `fromJust` *is encoded as follows:*

$$t_{fromJust} := \lambda\ (m{:}maybe_A\ tt).elim_{maybeA}\ tt\ m\ (\lambda\ (w{:}tt{\equiv}_{bool}ff).elim_{\equiv_{bool}}\ w)\ (\lambda\ (w{:}tt{\equiv}_{bool}tt).\lambda\ (x{:}A).x)$$

The missing case for `nothing` *must be accounted for (*cf. *the line* $(\lambda\ (w{:}tt{\equiv}_{bool}ff).elim_{\equiv_{bool}}w)$ *above).*

We allow for explicit working with the information that is missing in the external language by extending the internal language with term level metavariables, denoted by $?_a$, and type level metavariables, denoted by $?_A$. These stand for the parts of a term in the internal language that are not yet known.

Example 5.4

Using metavariables, the term that directly corresponds to `fromJust` *is:*

$$t_{fromJust} := \lambda\ (m{:}maybe_A\ tt).elim_{maybeA}\ ?_a\ m\ (\lambda\ (w{:}\ ?_A\ ).\ ?_b\ )\ (\lambda\ (w{:}\ ?_B\ ).\lambda\ (x{:}A).x)$$

by a Boolean:

$$\begin{array}{lll} \textbf{data}\ \mathit{maybe}_A\ (a : A) & : \mathit{bool} \to \mathtt{type}\ \textbf{where} \\ \\ \quad \mathit{nothing} & : \mathit{maybe}_A\ \mathit{ff} \\ \\ \quad \mathit{just} & : A \to \mathit{maybe}_A\ \mathit{tt} \end{array}$$

Here, `nothing` and `just` are the two constructors *of the `maybe` type. The type is indexed by `ff` when the `nothing` constructor is used, and by `tt` when the `just` constructor is used (`ff` and `tt` are constructors of `bool`). A function `fromJust` extracts the value from the `just` constructor:*

$$\mathit{fromJust}\ :\ \mathit{maybe}_A\ \mathit{tt} \to A$$

$$\mathit{fromJust}\ (\mathit{just}\ x) = x$$

Note that the value `tt` appears within the type $\mathit{maybe}_A\ \mathit{tt} \to A$ of this function (the type depends *on the value), allowing for a more precise function definition that omits the redundant case when the constructor of the type maybe_A is `nothing`. The challenge for the type checker is to determine that the missing case `fromJust nothing` above definition is contradictory (rather than being omitted by mistake). Indeed, the type of `nothing` is $\mathit{maybe}_A\ \mathit{ff}$. However, the function specifies its argument to be of type $\mathit{maybe}_A\ \mathit{tt}$.*

To type check functions in the surface language, the compiler translates them into terms in a type-theoretic calculus of nameless LF. We call this calculus the *internal language* of the compiler.

The number of objects of the internal language that are required to elaborate even a simple example such as Example 5.1 is rather large.

Example 5.2

One possible choice of objects to encode definition of `fromJust` is given by the signature in Figure 5.1. Recall that we use $A \to B$ as an abbreviation for $\Pi(a : A).B$ where a does not occur free in B.

The final goal of type checking of a function in the surface language is to obtain an encoding in the internal language. It is important to note that surface language does

limits expressive power of the resulting calculus. The case for dependent types in logic programming and proof search has been strongly advocated in Elf and Twelf programming languages (Pfenning, 1991, Pfenning and Schürmann, 1999, Xi and Pfenning, 1999). The work on Elf and Twelf is based on LF as is our work. However, there are three important differences:

- The treatment of resolution in Elf and Twelf does not utilise proof terms as we do. We present a case for proof terms in Chapter 5 where we show how to use proof terms as certificates when goal-directed search is embedded in a verifiable way into another system.
- Elf and Twelf languages are carried out directly in the syntax of LF. We distinguish between sorts `type` of types and `o` of formulae. This separation captures distinct fragments of syntax that are the term language of LF, and Horn-clause and hereditary Harrop logics that are defined atop of this term language. Logic formulae in these logics are then types with the sort `o` in head position as we discussed previously. Well-formedness judgements for these two fragments do not interact and it is possible to replace the term language without changing the semantics of resolution. This is demonstrated in Chapter 5 where we encode an external language using de Bruijn indices. The encoding effectively means that we do not need presence of binders in the term language. Hence the term language can be seen as a proper restriction of LF. A concrete advantage then is that first-order unification suffices for the purpose of the small-step resolution in Chapter 5.
- The distinction between sorts `type` and `o` has one further advantage. Predicates in our logic (that is types with head symbol `o`) can represent constructs not captured by the term language and proof relevant resolution can be used as a means of program transformation (or elaboration). We discuss this advantage on an example of type classes in Chapter 6.

Finally, let us comment on proof-theoretic aspects of logics that we discussed in this chapter. We study resolution in the logic of hereditary Harrop formulae. This logic is a constructive fragment of classical logic. The study of relation between intuitionistic and classical provability goes back to Glivenko (1929). Orevkov (1968) presented several so called Glivenko classes of sequents in classical logics. These

classes of sequents are conservative over intuitionistic or minimal logic. Recently, Negri (2016) generalised Orevkov's results using proof-theory. Proof-theoretic treatment of such results is at the basis of uniform proofs— the relation between provability in classical, intuitionistic and minimal logic for uniform proofs was studied, among others by Miller et al. (1991) and Ritter et al. (2000a). A motivation for such study was applications to proof-search in intuitionistic logics and to type-theoretic analysis of search spaces in classical and intuitionistic logics (Ritter et al., 2000b). This work is also to the best of our knowledge the origin of the notion of proof term in the sense we use it.

5 Type Inference and Term Synthesis

In this chapter, we demonstrate a use of proof-relevant resolution for the purpose of type inference and term synthesis in type theory. We make use of nameless LF as the language that is subject to type inference and term synthesis. The approach we present in this section consist of a preprocessing phase from nameless LF to a logic program and a proof-relevant resolution phase in the Horn-clause logic of the program. Then, solutions provided by the resolution phase are interpreted in nameless LF. In this chapter, we first explain the system by means of a detailed example, then we present formal description and discuss decidability of the preprocessing phase and soundness of the interpreted solutions.

5.1 Example by Resolution

In this section, we give a detailed example that combines preprocessing in a verified manner with the use of proof terms as a medium for communication with an external automated prover. We describe an algorithm that reduces type inference and term synthesis in type theory with dependent types to resolution in proof-relevant Horn-clause logic. In our description, we rely on an abstract syntax that closely resembles existing functional programming languages with dependent types. We will call it the *surface language*.

Example 5.1

In the surface language, we define $\texttt{maybe}_A$*, an option type over a fixed type* A*, indexed*

4 | Soundness

To prove soundness of the small-step operational semantics, we introduce a more structured relation that we call a *logical relation*. We then prove the *fundamental* theorem that the small-step operational semantics embeds into the logical relation. Further, we show that we can *escape* from logical relation to the big-step operational semantics if the judgement of the logical relation is formed for a proper proof term and a goal. Soundness of the small-step operational semantics then follows as a corollary.

4.1 Logical Relation

The logical relation exposes the structure of the big-step operational semantics while keeping track of free variables. Similarly to the big-step semantics, there are two judgements,

- $\mathcal{S}; \mathcal{P}; \Gamma \longrightarrow_{\mathcal{C}} \hat{e} : \hat{e}'$, and
- $\mathcal{S}; \mathcal{P}; \Gamma \xrightarrow{\hat{e}':D}_{\mathcal{C}} \hat{e} : A$.

Inference rules of these judgements reflect the inference rules of judgements of the big-step operational semantics, $\mathcal{S}; \mathcal{P} \longrightarrow e : G$ and $\mathcal{S}; \mathcal{P} \xrightarrow{e':D} e : A$ respectively. Unlike the big-step operational semantics, the judgements of the logical relation are equipped with context that keeps track of free variables. The logical relation is more general and proceeds on mixed terms rather than proof terms.

Besides the above two judgements we introduce one more, auxiliary judgement:

- $\mathcal{S}; \mathcal{P}; \Gamma \longrightarrow_{\mathcal{C}} \hat{e} : D$

This judgement makes explicit the invariant that the proof term and the clause that annotate the back-chaining judgement of the big-step operational semantics are well-formed.

$\boxed{\mathcal{S};\mathcal{P};\Gamma \longrightarrow_{\mathcal{C}} \hat{e} : \hat{e}'}$

$$\frac{\mathcal{S};\mathcal{P};\Gamma \xrightarrow{\hat{e}':D}_{\mathcal{C}} \hat{e} : A \qquad \mathcal{S};\mathcal{P};\Gamma \longrightarrow_{\mathcal{C}} \hat{e}' : D}{\mathcal{S};\mathcal{P};\Gamma \longrightarrow_{\mathcal{C}} \hat{e} : A}$$

$$\frac{\mathcal{S};\mathcal{P};\Gamma \longrightarrow_{\mathcal{C}} \hat{e} : G[M/x] \qquad \mathcal{S};\Gamma \vdash M : A}{\mathcal{S};\mathcal{P};\Gamma \longrightarrow_{\mathcal{C}} \langle M, \hat{e}\rangle : \exists x : A.G}$$

$$\frac{\mathcal{S};\mathcal{P},\kappa : D;\Gamma \longrightarrow_{\mathcal{C}} \hat{e} : G}{\mathcal{S};\mathcal{P};\Gamma \longrightarrow_{\mathcal{C}} \lambda\kappa.\hat{e} : D \Rightarrow G}$$

$$\frac{\mathcal{S},c : A;\mathcal{P};\Gamma \longrightarrow_{\mathcal{C}} \hat{e}[c/x] : G[c/x]}{\mathcal{S};\mathcal{P};\Gamma \longrightarrow_{\mathcal{C}} \hat{e} : \forall x : A.G}$$

$$\frac{\mathcal{S} \vdash \mathcal{P} \qquad \mathcal{S} \vdash \Gamma}{\mathcal{S};\mathcal{P};\Gamma \longrightarrow_{\mathcal{C}} A : A}$$

$$\frac{\mathcal{S};\mathcal{P};\Gamma \longrightarrow_{\mathcal{C}} \hat{e}_1 : \hat{e}_2 \qquad \mathcal{S};\Gamma' \vdash \theta : \Gamma}{\mathcal{S};\mathcal{P};\Gamma \longrightarrow_{\mathcal{C}} (\theta\hat{e})\,\hat{e}_1 : \hat{e}\,\hat{e}_2}$$

$$\frac{\mathcal{S};\mathcal{P};\Gamma \longrightarrow_{\mathcal{C}} \hat{e}_1 : \hat{e}_2 \qquad \mathcal{S};\Gamma' \vdash \theta : \Gamma}{\mathcal{S};\mathcal{P};\Gamma \longrightarrow_{\mathcal{C}} \langle \theta M, \hat{e}_1\rangle : \langle M, \hat{e}_2\rangle}$$

$$\frac{\mathcal{S};\mathcal{P};\Gamma \longrightarrow_{\mathcal{C}} \hat{e}_1 : \hat{e}_2}{\mathcal{S};\mathcal{P};\Gamma \longrightarrow_{\mathcal{C}} \lambda\kappa.\hat{e}_1 : \lambda\kappa.\hat{e}_2}$$

Figure 4.1: Logical relation, judgement $\mathcal{S};\mathcal{P};\Gamma \longrightarrow_{\mathcal{C}} \hat{e} : \hat{e}'$

Definition 4.1 (Logical relation)

The judgement $\mathcal{S};\mathcal{P};\Gamma \longrightarrow_{\mathcal{C}} \hat{e} : \hat{e}'$, the judgement $\mathcal{S};\mathcal{P};\Gamma \xrightarrow{\kappa:D}_{\mathcal{C}} \hat{e} : A$, and the judgement $\mathcal{S};\mathcal{P};\Gamma \longrightarrow_{\mathcal{C}} \hat{e} : D$ are given by inference rules in Figures 4.1, 4.2, and 4.3.

If we can form a judgement of logical relation for mixed terms $\hat{e}$ and $\hat{e}'$, we say that the mixed terms are *logically related*.

Similarly to the well-formedness judgements of the underlying term language, the logical relation possesses syntactic validity:

Proposition 4.2

- *If $\mathcal{S};\mathcal{P};\Gamma \longrightarrow_{\mathcal{C}} \hat{e} : \hat{e}'$ then $\mathcal{S} \vdash \mathcal{P}$.*
- *If $\mathcal{S};\mathcal{P};\Gamma \xrightarrow{\hat{e}_1:D}_{\mathcal{C}} \hat{e} : \hat{e}'$ then $\mathcal{S} \vdash \mathcal{P}$.*
- *If $\mathcal{S};\mathcal{P};\Gamma \longrightarrow_{\mathcal{C}} \hat{e} : D$ then $\mathcal{S} \vdash \mathcal{P}$.*

Proof. By simultaneous induction on derivations of the assumptions using implicit

Corollary 4.15 (Generalised soundness)

If $\mathcal{S};\mathcal{P} \vdash \Gamma \mid G \rightsquigarrow \Gamma' \mid e$ then $\mathcal{S};\mathcal{P} \longrightarrow e : \forall\Gamma'.G$.

Proof. Follows from the Fundamental Theorem (4.12) by generalisation (Lemma 4.14) and Escape Lemma (4.11). □

Let us conclude this chapter by recovering the notion of an *answer substitution*. Note that we can collect the substitutions that are computed in the initial sequent of the small-step resolution and compose the collected substitutions along the small resolution steps. Then, for a judgement $\mathcal{S};\mathcal{P} \vdash \Gamma \mid G \rightsquigarrow \Gamma' \mid e$, the composed substitution σ is a mapping from variables in context Γ to terms that are well-formed in context Γ' and since the partial substitutions are well formed also σ is well formed, *i.e.* $\mathcal{S};\Gamma \vdash \sigma : \Gamma'$.

4.4 Related Work

The proof of soundness in this chapter is carried out using a logical relation. The proof technique was originally introduced by Tait (1967) and used for proving strong normalisation of the simply typed lambda calculus. Initial application of logical relations include the proofs of strong normalisation for System F (Girard, 1972) and strong normalisation of Calculus of Constructions (Geuvers, 1994).

Logical relations has wide applications in programming languages research besides proofs of strong normalisation. Generally, these applications fall in two broad categories: type safety (Birkedal and Harper, 1999) and equivalence of programs (Dreyer et al., 2009, Pitts, 2000). Our use of logical relation that is relating two mixed terms is inspired by the use of logical relation for reasoning about program equivalence. A work that is relevant to our development in particular is the use of logical relations for mechanisations of metatheory of LF (Cave and Pientka, 2018, Urban et al., 2011). Logical relations has been also successfully applied to higher order type theory (Abel et al., 2018). These results provide a promising starting point for both mechanisations of results in this chapter and for extending these results beyond a first order type theory.

we obtain $\mathcal{S};\mathcal{P} \vdash \Gamma \mid A \overset{\hat{e}:D}{\rightsquigarrow} \Gamma' \mid \hat{e}''$ and $\hat{e}' = C'\{\hat{e}''\}$. By induction hypothesis we have $\mathcal{S};\mathcal{P};\Gamma' \xrightarrow{\hat{e}:D}_{\mathcal{C}} \hat{e}'' : A$. Hence, by lifting (Lemma 4.7), we obtain $\mathcal{S};\mathcal{P};\Gamma' \longrightarrow_{\mathcal{C}} \hat{e}' : C\{A\}$.

□

4.3 Soundness of Small-Step Operational Semantics

In this brief section, we bring the previous results together and prove soundness of the small-step operational semantics of proof-relevant resolution w.r.t. the big-step operational semantics. We also introduce a further lemma that allows us to prove (a strengthening of) the generalised soundness (Theorem 3.32).

First, soundness follows from the Escape Lemma and the Fundamental Theorem straightforwardly:

Corollary 4.13 (Soundness)
If $\mathcal{S};\mathcal{P} \vdash \Gamma \mid G \rightsquigarrow \cdot \mid e$ *then* $\mathcal{S};\mathcal{P} \longrightarrow e : G$.

However, recall that in Chapter 3 we stated soundness in a more general way, using generalisation of a goal with a context. Using results of the previous section, we state and prove the following lemma about generalisation of goals and the logical relation:

Lemma 4.14

1. *If* $\mathcal{S};\mathcal{P};(\Gamma, x : A) \longrightarrow_{\mathcal{C}} \hat{e} : G$ *then* $\mathcal{S};\mathcal{P};\Gamma \longrightarrow_{\mathcal{C}} \hat{e} : \forall x : A.G$.
2. *If* $\mathcal{S};\mathcal{P};\Gamma \longrightarrow_{\mathcal{C}} \hat{e} : G$ *then* $\mathcal{S};\mathcal{P};\cdot \longrightarrow_{\mathcal{C}} \hat{e} : \forall\Gamma.G$.

Proof. Part 1 Follows from substitutivity of the logical relation (Lemma 4.6) and weakening of signatures (Lemma 4.3).

Part 2 By induction on the context. The base case is by definition of generalisation, the inductive case follows from Part 1. □

Finally, we state and prove the generalised soundness of the small-step operational semantics:

$\boxed{\mathcal{S};\mathcal{P};\Gamma \xrightarrow{\hat{e}:D}_{\mathcal{C}} \hat{e} : \hat{e}'}$

$$\frac{}{\mathcal{S};\mathcal{P};\Gamma \xrightarrow{\hat{e}:A}_{\mathcal{C}} \hat{e} : A}$$

$$\frac{\mathcal{S};\mathcal{P};\Gamma \longrightarrow_{\mathcal{C}} \hat{e}_1 : A_1 \qquad \mathcal{S};\mathcal{P};\Gamma \xrightarrow{\hat{e}\,\hat{e}_1:D}_{\mathcal{C}} \hat{e}_2 : A_2}{\mathcal{S};\mathcal{P};\Gamma \xrightarrow{\hat{e}:A_1 \Rightarrow D}_{\mathcal{C}} \hat{e}_2 : A_2}$$

$$\frac{\mathcal{S};\mathcal{P};\Gamma \xrightarrow{\hat{e}:D[M/x]}_{\mathcal{C}} \hat{e}_2 : A_2 \qquad \mathcal{S};\Gamma \vdash M : A_1}{\mathcal{S};\mathcal{P};\Gamma \xrightarrow{\hat{e}:\forall x:A_1.D}_{\mathcal{C}} \hat{e}_2 : A_2}$$

Figure 4.2: Logical relation, judgement $\mathcal{S};\mathcal{P};\Gamma \xrightarrow{\hat{e}:D}_{\mathcal{C}} \hat{e} : \hat{e}'$

$\boxed{\mathcal{S};\mathcal{P};\Gamma \longrightarrow_{\mathcal{C}} \hat{e} : D}$

$$\frac{\mathcal{S} \vdash \mathcal{P} \qquad \kappa : D \in \mathcal{P} \qquad \mathcal{S} \vdash \Gamma}{\mathcal{S};\mathcal{P};\Gamma \longrightarrow_{\mathcal{C}} \kappa : D}$$

$$\frac{\mathcal{S};\mathcal{P};\Gamma \longrightarrow_{\mathcal{C}} \hat{e} : A \Rightarrow D \qquad \mathcal{S};\mathcal{P};\Gamma \longrightarrow_{\mathcal{C}} \hat{e}' : A}{\mathcal{S};\mathcal{P};\Gamma \longrightarrow_{\mathcal{C}} \hat{e}\,\hat{e}' : D}$$

$$\frac{\mathcal{S};\mathcal{P};\Gamma \longrightarrow_{\mathcal{C}} \hat{e} : \forall x : A.D \qquad \mathcal{S};\Gamma \vdash M : A}{\mathcal{S};\mathcal{P};\Gamma \longrightarrow_{\mathcal{C}} \hat{e} : D[M/x]}$$

Figure 4.3: Logical relation, judgement $\mathcal{S};\mathcal{P};\Gamma \longrightarrow_{\mathcal{C}} \hat{e} : D$

syntactic validity (Theorem 2.13). □

Further, judgements of the logical relation can be weakened with a new constant assuming that the new constant and its type or kind maintains well-formedness of the signature:

Lemma 4.3 (Weakening of signature)

1. *If* $\mathcal{S}_1, \mathcal{S}_2; \mathcal{P}; \Gamma \longrightarrow_{\mathcal{C}} e : G$ *and* $\vdash \mathcal{S}_1, c : A, \mathcal{S}_2$ *then* $\mathcal{S}_1, c : A, \mathcal{S}_2; \mathcal{P}; \Gamma \longrightarrow_{\mathcal{C}} e : G$.
2. *If* $\mathcal{S}_1, \mathcal{S}_2; \mathcal{P}; \Gamma \xrightarrow{e_1:D}_{\mathcal{C}} e : G$ *and* $\vdash \mathcal{S}_1, c : A, \mathcal{S}_2$ *then* $\mathcal{S}_1, c : A, \mathcal{S}_2; \mathcal{P}; \Gamma \xrightarrow{e_1:D}_{\mathcal{C}} e : G$.
3. *If* $\mathcal{S}_1, \mathcal{S}_2; \mathcal{P}; \Gamma \longrightarrow_{\mathcal{C}} e : D$ *and* $\vdash \mathcal{S}_1, c : A, \mathcal{S}_2$ *then* $\mathcal{S}_1, c : A, \mathcal{S}_2; \mathcal{P}; \Gamma \longrightarrow_{\mathcal{C}} e : D$.

Proof. By simultaneous structural induction on derivations of the first assumptions using Propositions 3.5 and 2.14 and syntactic validity of the logical relation (Proposition 4.2).

□

Similarly, judgements of the logical relation can be weakened with a new program clause as long as this clause maintains well-formedness of the program.

Lemma 4.4 (Weakening of program)

- *If* $\mathcal{S}; \mathcal{P}_1, \mathcal{P}_2; \Gamma \longrightarrow_{\mathcal{C}} \hat{e} : \hat{e}'$ *and* $\mathcal{S} \vdash \mathcal{P}_1, \kappa : D, \mathcal{P}_2$ *then* $\mathcal{S}; \mathcal{P}_1, \kappa : D, \mathcal{P}_2; \Gamma \longrightarrow_{\mathcal{C}} \hat{e} : \hat{e}'$.
- *If* $\mathcal{S}; \mathcal{P}_1, \mathcal{P}_2; \Gamma \xrightarrow{\hat{e}_1:D}_{\mathcal{C}} \hat{e} : \hat{e}'$ *and* $\mathcal{S} \vdash \mathcal{P}_1, \kappa : D, \mathcal{P}_2$ *then* $\mathcal{S}; \mathcal{P}_1, \kappa : D, \mathcal{P}_2; \Gamma \xrightarrow{\hat{e}_1:D}_{\mathcal{C}} \hat{e} : \hat{e}'$.
- *If* $\mathcal{S}; \mathcal{P}_1, \mathcal{P}_2; \Gamma \longrightarrow_{\mathcal{C}} \hat{e} : D$ *and* $\mathcal{S} \vdash \mathcal{P}_1, \kappa : D, \mathcal{P}_2$ *then* $\mathcal{S}; \mathcal{P}_1, \kappa : D, \mathcal{P}_2; \Gamma \longrightarrow_{\mathcal{C}} \hat{e} : D$.

Proof. By simultaneous structural induction on derivations of the first assumptions using syntactic validity of the logical relation (Proposition 4.2), syntactic validity of programs (Proposition 3.4) and implicit syntactic validity (Theorem 2.16, Part 1). □

The logical relation is stable under substitution over a program:

Theorem 4.12 (Fundamental)

Let $\mathcal{S}; \Gamma \vdash G : \mathtt{o}$.

1. *If* $\mathcal{S}; \mathcal{P} \vdash \Gamma \mid \hat{e} \rightsquigarrow \Gamma' \mid \hat{e}'$ *then* $\mathcal{S}; \mathcal{P}; \Gamma' \longrightarrow_{\mathcal{C}} \hat{e}' : \hat{e}$.
2. *If* $\mathcal{S}; \mathcal{P} \vdash \Gamma \mid \hat{e} \overset{\hat{e}_1:D}{\rightsquigarrow} \Gamma' \mid \hat{e}'$, *and* $\mathcal{S}; \mathcal{P}; \Gamma \longrightarrow_{\mathcal{C}} \hat{e}_1 : D :$ *then* $\mathcal{S}; \mathcal{P}; \Gamma' \xrightarrow{\hat{e}_1:D}_{\mathcal{C}} \hat{e}' : \hat{e}$.

Proof. By simultaneous structural induction on derivations of the judgements.

Part 1

- Let the derivation be $\dfrac{\mathcal{S}; \mathcal{P} \vdash \Gamma \mid A \overset{\kappa:D}{\rightsquigarrow} \Gamma' \mid \hat{e}}{\mathcal{S}; \mathcal{P} \vdash \Gamma \mid A \rightsquigarrow \Gamma' \mid \hat{e}}$. Using Part 2 of the lemma we have $\mathcal{S}; \mathcal{P}; \Gamma' \xrightarrow{\kappa:D}_{\mathcal{C}} \hat{e} : A$. From implicit syntactic validity (Proposition 4.2) and from lifting (Lemma 4.7) thus follows $\mathcal{S}; \mathcal{P}; \Gamma' \longrightarrow_{\mathcal{C}} \hat{e} : A$.
- Let the derivation be $\dfrac{\mathcal{S}; \mathcal{P} \vdash \Gamma, x : A \mid C\{\langle x, \hat{e}\rangle\} \rightsquigarrow \Gamma' \mid \hat{e}'}{\mathcal{S}; \mathcal{P} \vdash \Gamma \mid C\{\exists x : A.G\} \rightsquigarrow \Gamma' \mid \hat{e}'}$. By Lemma 3.29, we obtain $\mathcal{S}; \mathcal{P} \vdash \Gamma, x : A \mid \hat{e} \rightsquigarrow \Gamma' \mid \hat{e}''$ and $\hat{e}' = C'\{\langle M, \hat{e}''\rangle\}$. By induction hypothesis we have $\mathcal{S}; \mathcal{P}; \Gamma' \longrightarrow_{\mathcal{C}} \hat{e}'' : \hat{e}$. Thus using lifting (Lemma 4.7), it follows that $\mathcal{S}; \mathcal{P}; \Gamma' \longrightarrow_{\mathcal{C}} \hat{e}' : C\{\exists x : A.G\}$.
- Let the derivation be $\dfrac{\mathcal{S}; \mathcal{P}, \kappa : D \vdash \Gamma \mid C\{\lambda\kappa.\hat{e}\} \rightsquigarrow \Gamma' \mid \hat{e}'}{\mathcal{S}; \mathcal{P} \vdash \Gamma \mid C\{D \Rightarrow \hat{e}\} \rightsquigarrow \Gamma' \mid \hat{e}'}$. By Lemma 3.29, we obtain $\mathcal{S}; \mathcal{P} \vdash \Gamma \mid \lambda\kappa.\hat{e} \rightsquigarrow \Gamma' \mid \hat{e}''$ and $\hat{e}' = C'\{\lambda\kappa.\hat{e}''\}$. By induction hypothesis we have $\mathcal{S}; \mathcal{P}; \Gamma' \longrightarrow_{\mathcal{C}} \hat{e}'' : \lambda\kappa.\hat{e}$. Thus, it follows from lifting (Lemma 4.7) that $\mathcal{S}; \mathcal{P}; \Gamma' \longrightarrow_{\mathcal{C}} \hat{e}' : C\{D \Rightarrow \hat{e}\}$.
- Let the derivation be $\dfrac{\mathcal{S}; \mathcal{P}, \kappa : D \vdash \Gamma \mid C\{\hat{e}\} \rightsquigarrow \Gamma' \mid \hat{e}'}{\mathcal{S}; \mathcal{P} \vdash \Gamma \mid C\{\forall x : A.\hat{e}\} \rightsquigarrow \Gamma' \mid \hat{e}'}$. By Lemma 3.29 and the induction hypothesis, $\mathcal{S}; \mathcal{P} \vdash \Gamma \mid \hat{e} \rightsquigarrow \Gamma' \mid \hat{e}''$ and also $\hat{e}' = C'\{\hat{e}''\}$. Thus, using lifting (Lemma 4.7) we have $\mathcal{S}; \mathcal{P}; \Gamma' \longrightarrow_{\mathcal{C}} C'\{\hat{e}''\} : C\{\forall x : A.\hat{e}\}$.

Part 2

- Let the derivation be $\dfrac{\mathcal{S}; \Gamma' \vdash \sigma A = \sigma A' : \mathtt{o} \qquad \mathcal{S} \vdash \mathcal{P}}{\mathcal{S}; \mathcal{P} \vdash \Gamma \mid A' \overset{\hat{e}:A}{\rightsquigarrow} \Gamma' \mid \hat{e}}$. Then the desired judgement $\mathcal{S}; \mathcal{P}; \Gamma' \xrightarrow{\hat{e}:A}_{\mathcal{C}} \hat{e} : A'$ follows from lifting (Lemma 4.8) straightforwardly.
- Let the derivation be $\dfrac{\mathcal{S}; \mathcal{P} \vdash \Gamma \mid C\{A\} \overset{\hat{e}\, A_1:D}{\rightsquigarrow} \Gamma' \mid \hat{e}'}{\mathcal{S}; \mathcal{P} \vdash \Gamma \mid C\{A\} \overset{\hat{e}:A_1 \Rightarrow D}{\rightsquigarrow} \Gamma' \mid \hat{e}'}$. By Lemma 3.29 and the induction hypothesis we obtain $\mathcal{S}; \mathcal{P} \vdash \Gamma \mid A \overset{\hat{e}:A_1 \Rightarrow D}{\rightsquigarrow} \Gamma' \mid \hat{e}''$ and $\hat{e}' = C'\{\hat{e}''\}$. Using the induction hypothesis and the implicit syntactic validity of program for logical relation (Proposition 4.2), we obtain $\mathcal{S}; \mathcal{P}; \Gamma' \xrightarrow{\kappa:D}_{\mathcal{C}} \hat{e}'' : A$. Hence, by lifting (Lemma 4.7) we obtain $\mathcal{S}; \mathcal{P}; \Gamma' \longrightarrow_{\mathcal{C}} \hat{e}' : C\{A\}$.
- Let the derivation be $\dfrac{\mathcal{S}; \mathcal{P} \vdash \Gamma, x : A_1 \mid C\{A\} \overset{\hat{e}:D}{\rightsquigarrow} \Gamma' \mid \hat{e}'}{\mathcal{S}; \mathcal{P} \vdash \Gamma \mid C\{A\} \overset{\hat{e}:\forall x:A_1.D}{\rightsquigarrow} \Gamma' \mid \hat{e}'}$. By Lemma 3.29,

Proof. By simultaneous structural induction on derivations of $\mathcal{S};\mathcal{P};\cdot \longrightarrow_{\mathcal{C}} e : G$ and $\mathcal{S};\mathcal{P};\cdot \xrightarrow{e_1:D_1}_{\mathcal{C}} e : G$. We make implicit use of Proposition 4.9.

Part 1 The compatible cases are:

- Let the derivation be $\dfrac{\mathcal{S};\mathcal{P};\cdot \xrightarrow{e_1:D}_{\mathcal{C}} e : A}{\mathcal{S};\mathcal{P};\cdot \longrightarrow_{\mathcal{C}} e : A}$. From Part 2 of the lemma it follows that $\mathcal{S};\mathcal{P} \xrightarrow{e_1:D} e : A$. Using Lemma 4.10 it follows that $\mathcal{S};\mathcal{P} \longrightarrow e : A$.
- Let the derivation be $\dfrac{\mathcal{S};\mathcal{P};\cdot \longrightarrow_{\mathcal{C}} e : G[M/x] \qquad \mathcal{S};\cdot \vdash M : A}{\mathcal{S};\mathcal{P};\Gamma \longrightarrow_{\mathcal{C}} \langle M, e\rangle : \exists x : A.G}$. From the induction hypothesis it follows that $\mathcal{S};\mathcal{P} \longrightarrow e : G[M/X]$. Hence we form the inference $\dfrac{\mathcal{S};\mathcal{P} \longrightarrow e : G[M/X] \qquad \mathcal{S};\cdot \vdash M : A}{\mathcal{S};\mathcal{P} \longrightarrow \langle M, e\rangle : \exists X : A.G}$.
- Let the derivation be $\dfrac{\mathcal{S};\mathcal{P},\kappa : D;\Gamma \longrightarrow_{\mathcal{C}} e : G}{\mathcal{S};\mathcal{P};\Gamma \longrightarrow_{\mathcal{C}} \lambda\kappa.e : D \Rightarrow G}$. From the induction hypothesis it follows that $\mathcal{S};\mathcal{P},\kappa : D \longrightarrow e : G$. Hence we form the inference $\dfrac{\mathcal{S};\mathcal{P},\kappa : D \longrightarrow e : G}{\mathcal{S};\mathcal{P},\kappa : D \longrightarrow \lambda\kappa.e : D \Rightarrow G}$.
- Let the derivation be $\dfrac{\mathcal{S},c : A;\mathcal{P};\cdot \longrightarrow_{\mathcal{C}} e[c/x] : G[c/x]}{\mathcal{S};\mathcal{P};\cdot \longrightarrow_{\mathcal{C}} e : \forall x : A.G}$. From the induction hypothesis it follows that $\mathcal{S},c : A;\mathcal{P} \longrightarrow (\lambda\kappa.e)[c/x] : G[c/x]$. We form the inference $\dfrac{\mathcal{S},c : A;\mathcal{P} \longrightarrow (\lambda\kappa.e)[c/x] : G[c/x]}{\mathcal{S};\mathcal{P} \longrightarrow \lambda\kappa.e : \forall x : A.G}$ using the fact that substitution for proof terms and for mixed terms is defined in a uniform way.

Part 2

- Let the derivation be $\dfrac{}{\mathcal{S};\mathcal{P};\cdot \xrightarrow{e:A}_{\mathcal{C}} \kappa : A}$. Then $\dfrac{}{\mathcal{S};\mathcal{P} \xrightarrow{e:A} e : A}$.
- Let the derivation be $\dfrac{\mathcal{S};\mathcal{P};\cdot \longrightarrow_{\mathcal{C}} e_1 : A_1 \qquad \mathcal{S};\mathcal{P};\cdot \xrightarrow{e\,e_1:D}_{\mathcal{C}} \kappa : A_2}{\mathcal{S};\mathcal{P};\cdot \xrightarrow{e:A_1\Rightarrow D}_{\mathcal{C}} \kappa : A_2}$. By Part 2 of the lemma and from the assumption $\mathcal{S};\mathcal{P};\cdot \longrightarrow_{\mathcal{C}} e_1 : A_1$ it follows that $\mathcal{S};\mathcal{P} \longrightarrow e_1 : A_1$. By induction hypothesis $\mathcal{S};\mathcal{P} \xrightarrow{(\lambda\kappa.e)\,e_1:D} \kappa : A_2$. Thus we form the inference $\dfrac{\mathcal{S};\mathcal{P} \longrightarrow e_1 : A_1 \qquad \mathcal{S};\mathcal{P} \xrightarrow{(\lambda\kappa.e)\,e_1:D} \kappa : A_2}{\mathcal{S};\mathcal{P} \xrightarrow{\lambda\kappa.e:A_1\Rightarrow D} \kappa : A_2}$.
- Let the derivation be $\dfrac{\mathcal{S};\mathcal{P};\cdot \xrightarrow{e_1:D[M/x]}_{\mathcal{C}} e_2\,e_2' : A_2 \qquad \mathcal{S};\cdot \vdash M : A_1}{\mathcal{S};\mathcal{P};\cdot \xrightarrow{\kappa_1:\forall x:A_1.D}_{\mathcal{C}} e_2\,e_2' : A_2}$. Using substitutivity of logical relation (Lemma 4.6) to obtain the induction hypothesis, it follows that $\mathcal{S};\mathcal{P} \xrightarrow{e_1:D[M/x]} e_2\,e_2' : A_2$. Thus we form the required inference $\dfrac{\mathcal{S};\mathcal{P} \xrightarrow{e_1:D[M/x]} e_2\,e_2' : A_2 \qquad \mathcal{S};\cdot \vdash M : A_1}{\mathcal{S};\mathcal{P} \xrightarrow{e_1:\forall x:A_1.D} e_2\,e_2' : A_2}$.

□

Finally, we establish that two mixed terms are logically related if there is a derivation of small-step operational semantics that takes one mixed term to the other.

Proposition 4.5 (Stability)

1. *If* $\mathcal{S};\mathcal{P}_1,\kappa : D_1,\mathcal{P}_2;\Gamma \longrightarrow_{\mathcal{C}} e : G$ *then* $\mathcal{S};\mathcal{P}_1,\kappa : D_1[M/x],\mathcal{P}_2;\Gamma \longrightarrow_{\mathcal{C}} e : G$.
2. *If* $\mathcal{S};\mathcal{P}_1,\kappa : D_1,\mathcal{P}_2;\Gamma,x : A \xrightarrow{e_1:D}_{\mathcal{C}} e : G$ *then* $\mathcal{S};\mathcal{P}_1,\kappa : D_1[M/x],\mathcal{P}_2;\Gamma \xrightarrow{e_1:D}_{\mathcal{C}} e : G$.
3. *If* $\mathcal{S};\mathcal{P}_1,\kappa : D_1,\mathcal{P}_2;\Gamma,x : A \longrightarrow_{\mathcal{C}} e : D$ *then* $\mathcal{S};\mathcal{P}_1,\kappa : D_1[M/x],\mathcal{P}_2;\Gamma \longrightarrow_{\mathcal{C}} e : D$.

Proof. By simultaneous structural induction on derivations of the assumptions using Proposition 3.6. □

The logical relation possesses substitutivity property.

Lemma 4.6 (Substitutivity)

1. *If* $\mathcal{S};\mathcal{P};\Gamma,x : A \longrightarrow_{\mathcal{C}} e : G$ *and* $\mathcal{S};\Gamma \vdash M : A$ *then* $\mathcal{S};\mathcal{P};\Gamma \longrightarrow_{\mathcal{C}} e[M/x] : G[M/x]$.
2. *If* $\mathcal{S};\mathcal{P};\Gamma,x : A \xrightarrow{e_1:D}_{\mathcal{C}} e : G$ *and* $\mathcal{S};\Gamma \vdash M : A$ *then* $\mathcal{S};\mathcal{P};\Gamma \xrightarrow{e_1[M/x]:D[M/x]}_{\mathcal{C}} e : G$.
3. *If* $\mathcal{S};\mathcal{P};\Gamma,x : A \longrightarrow_{\mathcal{C}} e : D$ *and* $\mathcal{S};\Gamma \vdash M : A$ *then* $\mathcal{S};\mathcal{P};\Gamma \longrightarrow_{\mathcal{C}} e[M/x] : D[M/x]$.

Proof. By simultaneous structural induction on derivations of the first assumptions using stability of the logical relation (Proposition 4.5), Proposition 3.6, weakening of the logical relation (Lemma 4.3) and substitutivity of terms (Theorem 2.13). □

Our proof of the fundamental theorem depends on the fact that it is possible to transform judgements of the logical relation in a way that corresponds to propagation of inference rules. We call this transformation *lifting*:

Lemma 4.7 (Lifting)

Let $\mathcal{S};\Gamma' \vdash \theta : \Gamma$.

1. *If* $\mathcal{S};\mathcal{P};\Gamma \xrightarrow{\kappa:D}_{\mathcal{C}} \hat{e} : A$ *then* $\mathcal{S};\mathcal{P};\Gamma \longrightarrow_{\mathcal{C}} (\theta C)\{\hat{e}\} : C\{A\}$.
2. *If* $\mathcal{S};\mathcal{P};\Gamma \longrightarrow_{\mathcal{C}} \hat{e} : G$ *and* $\mathcal{S};\Gamma \vdash M : A$ *then* $\mathcal{S};\mathcal{P};\Gamma \longrightarrow_{\mathcal{C}} (\theta C)\{\langle M, \hat{e}\rangle\} : C\{\exists x : A.G\}$.
3. *If* $\mathcal{S};\mathcal{P},\kappa : D;\Gamma \longrightarrow_{\mathcal{C}} \hat{e} : G$ *then* $\mathcal{S};\mathcal{P};\Gamma \longrightarrow_{\mathcal{C}} (\theta C)\{\lambda\kappa.\hat{e}\} : C\{D \to G\}$.
4. *If* $\mathcal{S};\mathcal{P};\Gamma \longrightarrow_{\mathcal{C}} \hat{e} : G$ *then* $\mathcal{S};\mathcal{P};\Gamma \longrightarrow_{\mathcal{C}} (\theta C)\{\hat{e}\} : C\{\forall x : A.G\}$.

Proof. The proof of each part of the lemma proceeds by induction on the rewriting context C. Base cases for $C = \bullet$ follow as the appropriate inference rules. Remaining cases for $C = \hat{e}\,\hat{e}'$, $C = \langle M, \hat{e}\rangle$, and $C = \lambda\kappa.\hat{e}$ follow by the appropriate inference rule and the induction assumption. □

We prove lifting also for the judgement $\mathcal{S};\mathcal{P};\Gamma \xrightarrow{\hat{e}_1:D}_{\mathcal{C}} \hat{e} : \hat{e}'$:

Lemma 4.8

Let $\mathcal{S};\Gamma' \vdash \theta : \Gamma$.

1. *If* $\mathcal{S};\Gamma \vdash \sigma A \equiv \sigma A' : \mathtt{o}$, *and* $\mathcal{S};\mathcal{P};\Gamma \longrightarrow_{\mathcal{C}} \hat{e}' : D'$ *then* $\mathcal{S};\mathcal{P};\Gamma \xrightarrow{\hat{e}':A'}_{\mathcal{C}} C\{\hat{e}'\} : C\{A\}$.
2. *If* $\mathcal{S};\mathcal{P};\Gamma \longrightarrow_{\mathcal{C}} \hat{e}_1 : A_1$ *and* $\mathcal{S};\mathcal{P};\Gamma \xrightarrow{\hat{e}\,\hat{e}_1:D}_{\mathcal{C}} \hat{e} : \hat{e}'$ *then* $\mathcal{S};\mathcal{P};\Gamma \xrightarrow{\hat{e}:A_1 \Rightarrow D}_{\mathcal{C}} \hat{e} : \hat{e}'$.
3. *If* $\mathcal{S};\mathcal{P};\Gamma \xrightarrow{\hat{e}_1:D[M/x]}_{\mathcal{C}} \hat{e} : \hat{e}'$ *and* $\mathcal{S};\Gamma \vdash M : A$ *then* $\mathcal{S};\mathcal{P};\Gamma \xrightarrow{\hat{e}_1:\forall x:A.D}_{\mathcal{C}} \hat{e} : \hat{e}'$.
4. *If* $\mathcal{S};\mathcal{P};\Gamma \xrightarrow{\hat{e}_1:D[M/x]}_{\mathcal{C}} \hat{e} : \hat{e}'$, *and* $\mathcal{S};\Gamma \vdash M : A$ *then* $\mathcal{S};\mathcal{P};\Gamma \xrightarrow{\hat{e}_1:\forall x:A.D}_{\mathcal{C}} (\theta C)\{\hat{e}\} : C\{\hat{e}'\}$.

Proof. The proofs of parts 1. and 4. of the lemma proceed by induction on rewriting context C. Base cases for $C = \bullet$ follow as the appropriate inference rules. Remaining cases for $C = \hat{e}\,\hat{e}'$, $C = \langle M, \hat{e}\rangle$, and $C = \lambda\kappa.\hat{e}$ follow by the appropriate inference rule and the induction assumption. Part 4. uses part 3. in the base case.

The proofs of parts 2. and 3. of the lemma proceed by induction on the mixed term $\hat{e}$. Base cases for $\hat{e} = G$ follow as the appropriate inference rules. Remaining cases for $C = \hat{e}\,\hat{e}'$, $C = \langle M, \hat{e}\rangle$, and $C = \lambda\kappa.\hat{e}$ follow by the appropriate inference rule and the induction assumption. □

The proof of soundness depends on the fact that we can escape the logical relation if it is established for a proper proof term (*i.e.* not a mixed term). Before showing the appropriate lemma, we state an auxiliary property:

Proposition 4.9

Let $e \in \mathrm{PT}$ *be a proof term. If* $\mathcal{S};\mathcal{P};\Gamma \xrightarrow{\hat{e}_1:D_1}_{\mathcal{C}} e : A$ *then* $\hat{e}_1$ *is a proof term,* i.e. $\hat{e}_1 \in \mathrm{PT}$.

Proof. By structural induction on derivation of the judgement.

- Let the case be $\dfrac{}{\mathcal{S};\mathcal{P};\Gamma \xrightarrow{e:A}_{\mathcal{C}} e : A}$. Then $e \in \mathrm{PT}$ follows from assumptions.

- Let the case be $\dfrac{\mathcal{S};\mathcal{P};\Gamma \longrightarrow_{\mathcal{C}} \hat{e}_1 : A_1 \qquad \mathcal{S};\mathcal{P};\Gamma \xrightarrow{\hat{e}\,\hat{e}_1:D}_{\mathcal{C}} e_2 : A_2}{\mathcal{S};\mathcal{P};\Gamma \xrightarrow{\hat{e}:A_1 \Rightarrow D}_{\mathcal{C}} e_2 : A_2}$. From the induction hypothesis, $\hat{e}\,\hat{e}_1 \in \mathrm{PT}$. Hence $\hat{e} \in \mathrm{PT}$.
- Let the case be $\dfrac{\mathcal{S};\mathcal{P};\Gamma \xrightarrow{\hat{e}:D[M/x]}_{\mathcal{C}} e_2 : A_2 \qquad \mathcal{S};\mathcal{P} \vdash M : A_1}{\mathcal{S};\mathcal{P};\Gamma \xrightarrow{\hat{e}:\forall x:A_1.D}_{\mathcal{C}} e_2 : A_2}$. Then $\hat{e} \in \mathrm{PT}$ follows from the induction hypothesis.

□

Finally, we make use of the following lemma that allows us to lift an annotated judgement of the logical relation to a judgement without annotation assuming that the judgement is formed for a proper proof term, an atomic goal, and that the annotating proof term and clause are well-formed.

Lemma 4.10

Let $e \in \mathrm{PT}$ *be a proof term. If* $\mathcal{S};\mathcal{P};\Gamma \xrightarrow{e_1:D}_{\mathcal{C}} e : A$ *and* $\mathcal{S};\mathcal{P};\Gamma \longrightarrow_{\mathcal{C}} e_1 : D$ *then* $\mathcal{S};\mathcal{P};\Gamma \longrightarrow_{\mathcal{C}} e : A$.

Proof. By structural induction on the assumption using Proposition 4.9 and substitutivity of the logical relation (Lemma 4.6). □

4.2 Fundamental Escape

In this section, we state and prove two main properties that are necessary for establishing soundness of the small-step operational semantics. The *escape lemma* allows us to escape from a judgement of logical relation for a proof term and a goal to a judgement of the big-step operational semantics. The *fundamental theorem* allows us to establish that two mixed terms are logically related if there is a derivation of the small-step operational semantics for them.

We follow the order in which we introduced operational semantics and we state the escape lemma first:

Lemma 4.11 (Escape)

Let $e \in \mathrm{PT}$ *and* $e_1 \in \mathrm{PT}$ *be proof terms.*

1. *If* $\mathcal{S};\mathcal{P};\cdot \longrightarrow_{\mathcal{C}} e : G$ *then* $\mathcal{S};\mathcal{P} \longrightarrow e : G$.
2. *If* $\mathcal{S};\mathcal{P};\cdot \xrightarrow{e_1:D_1}_{\mathcal{C}} e : G$ *and* $\mathcal{S};\mathcal{P};\cdot \longrightarrow_{\mathcal{C}} e_1 : D_1$ *then* $\mathcal{S};\mathcal{P} \xrightarrow{e_1:D_1} e : G$.