Extensions of Nominal Terms

Dominic Phillip Mulligan

All work carried out under the supervision of:
Dr. Murdoch J. Gabbay and Prof. Philip W. Trinder

Submitted in total fulfilment of the requirements of the degree of:

Doctor of Philosophy

Dependable Systems Group,
Department of Computer Science,
School of Mathematics and Computer Science,
Heriot-Watt University,
Edinburgh

March 2011

Abstract

This thesis studies two major extensions of nominal terms. In particular, we study an extension with λ-abstraction over nominal unknowns and atoms, and an extension with an arguably better theory of freshness and α-equivalence.

Nominal terms possess *two* levels of variable: *atoms* a represent variable symbols, and *unknowns* X are 'real' variables. As a syntax, they are designed to facilitate metaprogramming; unknowns are used to program on syntax with variable symbols.

Originally, the role of nominal terms was interpreted narrowly. That is, they were seen solely as a syntax for representing partially-specified abstract syntax with binding.

The main motivation of this thesis is to extend nominal terms so that they can be used for metaprogramming on proofs, programs, etc. and not just for metaprogramming on abstract syntax with binding. We therefore extend nominal terms in two significant ways: adding λ-abstraction over nominal unknowns and atoms—facilitating functional programing—and improving the theory of α-equivalence that nominal terms possesses.

Neither of the two extensions considered are trivial. The *capturing substitution action* of nominal unknowns implies that our notions of scope, intuited from working with syntax possessing a *non-capturing* substitution, such as the λ-calculus, is no longer applicable. As a result, notions of λ-abstraction and α-equivalence must be carefully reconsidered.

In particular, the first research contribution of this thesis is the **two-level λ-calculus**, intuitively an intertwined pair of λ-calculi. As the name suggests, the two-level λ-calculus has two level of variable, modelled by nominal atoms and unknowns, respectively. Both levels of variable can be λ-abstracted, and requisite notions of β-reduction are provided. The result is an expressive context-calculus. The traditional problems of handling α-equivalence and the failure of commutation between instantiation and β-reduction in context-calculi are handled through the use of two distinct levels of variable, swappings, and freshness side-conditions on unknowns, i.e. 'nominal technology'.

The second research contribution of this thesis is **permissive nominal terms**, an alternative form of nominal term. They retain the 'nominal' first-order flavour of nominal terms (in fact, their grammars are almost identical) but forego the use of explicit freshness contexts. Instead, permissive nominal terms label unknowns with a *permission sort*, where permission sorts are infinite and coinfinite sets of atoms. This infinite-coinfinite nature means that permissive nominal terms recover two properties—we call them the 'always-fresh' and 'always-rename' properties—that nominal terms lack. We argue that these two properties bring the theory of α-equivalence on permissive nominal terms closer to 'informal practice'.

The reader may consider λ-abstraction and α-equivalence so familiar as to be 'solved problems'. The work embodied in this thesis stands testament to the fact

that this isn't the case. Considering λ-abstraction and α-equivalence in the context of two levels of variable poses some new and interesting problems and throws light on some deep questions related to scope and binding.

Acknowledgments

This thesis would not exist without the extensive help, support and guidance of a large number of people. Chief amongst these are my two supervisors, Murdoch J. Gabbay (Jamie) and Philip W. Trinder (Phil).

Jamie introduced me to the research area I now call my own and has gone way beyond what could reasonably be expected of a PhD supervisor in terms of support and guidance. Phil also provided extensive support and guidance, and meetings with Phil always served to focus my attention on better understanding concepts that I thought I understood (but really didn't).

Jamie and Phil also deserve recognition for teaching me how to write in a suitably academic manner. Their advice and effort proof reading this thesis helped to knock the document into its final shape. I hope they forgive me wherever I ignored their advice, and whatever errors that remain within are entirely due to me.

The fact that I now consider myself a Computer Scientist, with ideas of work that I would like to carry out on my own, is entirely due to the hard work of both Jamie and Phil.

My parents, John and Anne, along with the rest of my family, have provided extensive emotional (and financial!) support over the past three years, and never doubted that I could achieve what I wanted to achieve.

Victoria, for the past year, has given me a life outside of computer science, given me her love, kept me sane, fed, and in clothes. I promise I love you more than maths books and computer games, Vickie (even if you don't believe it).

I would also like to thank the other members of the Dependable Systems Group at Heriot-Watt, the Sunday night football gang, the players and coaches of Lismore rugby club, and my flatmate Simon, purveyor of terrible banter and jokes.

Finally, I'd like to thank Maribel and Greg for agreeing to be my examiners.[1]

[1]Naturally, my thanks is rescinded if they fail me.

For Victoria

ACADEMIC REGISTRY
Research Thesis Submission

HERIOT WATT UNIVERSITY

Name:			
School/PGI:			
Version: *(i.e. First, Resubmission, Final)*		Degree Sought (Award **and** Subject area)	

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 and have made reference to work carried out in collaboration with other persons
3) the thesis is the correct version of the thesis for submission and is the same version as any electronic versions submitted*.
4) my thesis for the award referred to, deposited in the Heriot-Watt University Library, should be made available for loan or photocopying and be available via the Institutional Repository, subject to such conditions as the Librarian may require
5) I understand that as a student of the University I am required to abide by the Regulations of the University and to conform to its discipline.

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

Signature of Candidate:		Date:	

Submission

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

For Completion in Academic Registry

Received in the Academic Registry by *(name in capitals)*:			
Method of Submission *(Handed in to Academic Registry; posted through internal/external mail)*:			
*E-thesis Submitted (**mandatory for final theses from January 2009**)*			
Signature:		Date:	

Contents

List of Figures

CHAPTER 1

Introduction

The book is divided into chapters, which are divided into sections, which are divided into paragraphs, which are divided into sentences, which are divided into words, which are divided into letters. For reference purposes, each letter in the book has a number (actually a finite sequence of positive integers) attached to it.

Carl E. Linderholm, Mathematics Made Difficult

1.1 Background and motivation

The work presented in this thesis tells a story, and like most stories, this thesis has a beginning, middle and end. We first set the scene.

1.1.1 Abstract syntax and binding

Many tasks in computer science require the automated manipulation of *abstract syntax*. By abstract syntax, we mean a usually tree-based representation of syntax, suitable for manipulation and traversal. For example: a C compiler must parse raw input strings to an abstract tree-based representation of parse-trees.

Those familiar with the implementation of proof assistants, theorem provers, source code transformation tools, interpreters, and so on, will recognise the importance of manipulating abstract syntax. So we would expect abstract syntax to be well-understood. *But is it?*

Problems arise when we wish to reason about, and manipulate, languages whose syntax involves *name binding*. Name binding is common: it is found in virtually every programming language, in the myriad calculi and formal logics that computer scientists use to reason about systems, and even in the integral calculus taught in schools. In particular when working with abstract syntax with name binding there are a few recurring operations that we would like to implement: we would like to test for α-equivalence; we would like to generate fresh names; we would like to implement capture-avoiding substitutions. Most of all we would like to do these tasks correctly and with a minimum of fuss.

Unfortunately handling name binding correctly looks deceptively simple, yet in practice is tricky to get right, and often leads to insidious and hard to find bugs as the authors of early versions of the LCF proof assistant discovered [Pau88]. Moreover techniques that were adequate for implementing and reasoning about abstract syntax *without* name binding now flounder when we naïvely implement and reason about abstract syntax *with* name binding.

The 'name binding problem'—as we'll refer to it from now on—has led many researchers to invest significant energy in trying to find elegant methods for working with abstract syntax with name binding. Many diverse techniques have been suggested: de Bruijn indices [dB72], higher-order abstract syntax [PE88], Fraenkel-Mostowski ('nominal') techniques [GP99], locally nameless representations [MM04], Sato's representation [Sat08] and other locally named approaches, and so on.

In particular, we'll focus on *nominal terms* [UPG04]. These originated with the aforementioned research into Fraenkel-Mostowski techniques by Gabbay and Pitts in the late 1990s [GP99, Gab00].

Nominal terms were proposed as a *metalanguage* for embedding syntax with binding into. Nominal terms possess two levels of variable: *atoms*, elements with a name, and no internal structure, and *unknowns*, variables with an associated substitution action. One may abstract atoms in a term, allowing one to work up to α-equivalence—the native notion of equality for syntax with binding. Syntax is embedded into nominal terms through the use of termformers, and binding at the syntax level is facilitated by atoms abstraction in nominal terms.

In particular nominal terms were proposed to try to capture queries like the following [UPG04, pg. 5]:

> Assuming a and b are distinct variables, is it possible to find a λ-term M that makes $\lambda a.\lambda b.Mb$ and $\lambda b.\lambda a.aM$ α-equivalent?

Analysing this query in detail will help to motivate nominal terms.

For instance, M may be instantiated to any term—it is a hole—even under the bound name x. This is modeled in nominal terms by making use of the two levels of variable: we may bind an atom in a term, but instantiating any unknown laying under that binder will not necessarily avoid capture. Similarly, we may have informal freshness side-condition on holes. This is modeled in nominal terms by making use of freshness assumptions on unknowns, and using a derivable freshness relation as a generalisation of the free variables of a term.

Nominal terms have a straightforward denotation in Fraenkel-Mostowski sets. They also have excellent computational properties: unification of nominal terms is efficiently decidable and produces most general unifiers. A large amount of ancillary research has been carried out imposing typing systems [FG07a], investigating rewriting [FGM04, FG05, FG07b], defining algebras [GM08a, GM09a, GM07a, GM07c], and extending Prolog's term language with nominal terms [CU08, CU04].

These applications interpret nominal terms as a language for working with abstract syntax with binding, sometimes consciously so, as in the case of αProlog (see [CU08, pg. 15]). However, a body of research exemplified by Gabbay, amongst others, has begun to look at other uses for nominal terms [Gab05, GL08, GM09a, FGM04]. Specifically, Gabbay uses nominal terms not just as a tool for embedding *syntax* with binding, but as a metalanguage for embedding *languages* with binding. Whereas the native notion of equivalence for syntax with binding is α-equivalence, languages with binding may have more complex forms of equivalence, for instance β-equivalence in the case of the λ-calculus. These forms of equivalence may be captured in a suitable nominal framework, for instance nominal rewriting [FGM04] or nominal algebra [GM09a].

Many metalanguages already exist and they collectively possess a range of features that make them suitable as a target for embedding into. This thesis asks: '*can nominal terms borrow any features from these metalanguages, and what do the resulting systems look like?*'.

1.1.2 The beginning: the two-level λ-calculus

Many metalanguages feature λ-abstraction. Can we extend nominal terms with λ-abstraction for both atoms and nominal unknowns? Further, once we have made this extension, what does the resulting system look like?

Chapter 3 introduces the two-level λ-calculus. This is as an extension of nominal terms, where both atoms and unknowns may be λ-bound. Intuitively, the two-level λ-calculus is a pair of λ-calculi intertwined around each other, one at 'level one', and the other at 'level two', with requisite notions of β-reduction at both levels.

The two-level λ-calculus is a *context-calculus*. A context is a λ-term with 'some holes', and context-calculi promote contexts to a first-class status; contexts may be returned from and passed as arguments to functions. Context-calculi were proposed to formalise the informal notion of context used regularly in computer science. For instance, observational equivalence proofs for program fragments rely on an informal notion of context (for example, see the work of Pitts [Pit94]). However, context-calculi are also interesting in their own right, as tools for investigating novel programming language designs [Has98], for investigating dynamic linking [Dam98], and many other applications (see [HO01] for an overview of possible uses).

In a context-calculus there are two levels of variable. Filling a hole—where holes are variables of the λ-calculus at level two—does not avoid capture of λ-bound variables from the level one calculus. The same holds for the two-level λ-calculus: the substitution action on unknowns, inherited from nominal terms, does not avoid capture for λ-bound atoms.

Like all context-calculi, the two-level λ-calculus must overcome two difficulties: how to handle α-equivalence in the presence of two levels of variable, and how to handle a failure of commutation between naïve β-reduction and hole filling. In the two-level λ-calculus, 'nominal technology' is brought to bear, solving these problems.

We handle α-equivalence using swappings, and freshness side-conditions. Swappings are a trademark feature of nominal techniques, and possess some excellent properties: they are self inverse, and nearly all relations defined over two-level λ-terms are equivariant.

The two-level λ-calculus doesn't just benefit from an *explicit* nominal influence—with the use of swappings and freshness constraints—but also from an *implicit* influence, in realising that there are two types of variable in context-calculi, and they should behave differently.

We prevent a term like $(\lambda a.X)t$ reducing to X, unless we specifically have hypothesised that a is fresh for X. It is the failure to prevent this reduction—the failure to realise that there are two levels of variable in a context-calculus—which leads to a non-confluent system.

The two-level λ-calculus is confluent, and the proof of this fact is non-trivial. The proof of confluence is split into two halves: we prove the confluence of the level one fragment (Theorem 3.4.24) and the confluence of the level two fragment (Theorem 3.4.36) separately, before stitching the two proofs together (Theorem 3.4.42). Level two confluence is proved using Tait's parallel reduction strategy. Level one confluence is proved using a canonical form (Definition 3.4.13) and a novel 'backwards' lemma (Lemma 3.4.23). This proof technique seems to be a powerful method for proving confluence in general, and was used to prove several other systems confluent [GM09b, GM10b].

Despite the novelty of the proof strategy, the technical 'meat' of the confluence proof is obscured by manipulations of freshness contexts. During the confluence proof, we regularly need to push substitutions under various λ-binders. This may not be possible depending on the ambient freshness context that we are working in: it simply may not have enough 'freshness' to allow us to rename our λ-abstracted variables to something fresh. For instance, if we are working in the empty freshness context, no renaming can be made.

To retain confluence, it is routinely necessary to 'freshen' a context with some fresh variables. Most lemmas and theorems within the confluence proof are parameterised by a freshness context; they take a freshness context as 'input', perform some reasoning, and 'output' a new freshness context with sufficient freshness for the lemma to hold. A lemma in this pattern is Lemma 3.4.23. However we must be careful to define the 'freshening' operation for contexts: improper management of freshness contexts would mean that *any* freshness side-condition can be met. Sometimes a freshness side-condition should never be met, regardless of the ambient freshness context we are working in.

Apart from the obscuration of confluence proofs, there are a number of other problems with the use of explicit freshness contexts. Retaining freshness contexts complicates the quotienting of terms by α-equivalence. Potential α-equivalence classes are constructed with respect to a term *and* a freshness context, and expanding or contracting this context affects the size of the α-equivalence class. A nominal term does not have an associated α-equivalence class; a nominal-term-in-context does.

Problems with explicit freshness contexts also pose problems for other nominal calculi and logics. Expanding a freshness context in nominal algebra [Mat07], for instance, results in more provable equalities. Similarly, plans for constructing a higher-order logic from the two-level λ-calculus (as discussed in Chapter 3) are complicated by the fact that expanding or contracting ambient freshness contexts affects what is provable within the logic, as more equivalences may hold, depending on the freshness context at hand!

Contexts of assumptions are common throughout theoretical computer science. For instance typing contexts are used heavily in various type theories to record the types of variables. However typing contexts and freshness contexts are fundamentally different. Expanding a typing context usually doesn't make more terms equal; expanding a freshness context does.

In order to fix these problems, we must rid ourselves of explicit freshness contexts that can contract and expand, and instead focus on some fixed, global notion of freshness. This is what we attempt to do with permissive nominal terms.

1.1.3 The middle: permissive nominal terms

Permissive nominal terms are presented in Chapter 4. In permissive nominal terms, explicit freshness contexts are elided, instead, unknowns are labeled by an infinite and coinfinite set of atoms, called a permission sort.

Permissive nominal terms retain the 'nominal' first-order flavour of nominal terms (they in fact have an almost identical grammar). They also recover some useful properties that informal presentations of syntax with binding have, but nominal terms, and by extension two-level λ-terms, lack. Namely, these are what we refer to as the 'always fresh' and 'always rename' properties.

The 'always fresh' property states that, for every term, there are always infinitely many fresh names. The 'always rename' property states that any bound name can always be renamed to something conveniently fresh. These two properties help smooth the treatment of informal syntax with binding: we can always push substitutions under binders, as we can always 'just rename'.

Informal syntax with binding is also routinely quotiented by α-equivalence. For instance in first-order logic it is routine to consider $\forall x.x = x$ and $\forall y.y = y$ as being the same formula. Permissive nominal terms use the particular form of the permission sorts to recover the 'always fresh' (Corollary 4.2.17) and 'always rename' (Corollary 4.2.18) properties, and therefore make it easy to quotient permissive nominal terms by α-equivalence. Specifically, the infinite and coinfinite nature of permission sorts makes it possible to find infinitely many fresh atoms for any term, and to always 'just rename', if necessary.

Freshness in permissive nominal terms now becomes a structural property of the term itself. There is no derivable theory of freshness, parameterised by a freshness context. Instead, a term's freshness is fixed when that term is created and permission sorts for unknowns are picked. We define a notion of the free atoms of a term by structural induction on the term itself. As a result, quotienting permissive nominal terms by α-equivalence is straightforward, and α-equivalence classes of permissive nominal terms are not liable to expand and contract depending on the ambient freshness context that we are working in.

Like nominal terms, we introduce a unification algorithm for their permissive counterparts, and the lack of explicit freshness contexts simplifies matters. A nominal unification solution consists of a substitution *and* a freshness context [UPG04]. In contrast, first- and higher-order unification algorithms return only a substitution as their solutions [BS01].

Permissive nominal unification solutions are akin to first- and higher-order unification solutions, in that they consist solely of a substitution (Definition 4.3.3). The permissive unification algorithm internalises freshness, solving constraints in a separate 'support reduction' phase. The burden of handling freshness contexts is removed from the user.

A selling point of nominal terms has been their tractable computational properties. Do permissive nominal terms also possess these properties? We prove that they do: in particular, most general unifiers exist (Theorem 4.3.56), unification of permissive nominal terms is decidable, and the algorithm of Definition 4.3.39 is correct (Theorem 4.3.58).

Despite their different treatments of freshness, permissive nominal terms and nominal terms share a close correspondence. In particular, we present a non-trivial translation between nominal terms (Definition 4.4.12), and their permissive counterparts, under the assumption that we only use atoms in the nominal world from some fixed permission sort. We prove in Theorem 4.4.24 that this translation also preserves unifiers.

Finally, the infinite and coinfinite nature of permission sorts is not a barrier to straightforward implementation. We prove this constructively, with an 'existential proof', a prototype implementation of permissive nominal terms and their unification called PNT, in Haskell. We discuss PNT in detail in Chapter 5, and demonstrate that permissive nominal terms can be implemented in a straightforward manner, without having to resort to using Haskell's laziness to represent infinite datastructures (in fact, all datastructures in the implementation are finite).

1.1.4 The end: adoption of permissive nominal terms

We introduced permissive nominal terms as a variant of nominal terms that recovered several properties that informal presentations of syntax with binding enjoy. Namely, these were the 'always fresh' and 'always rename' properties. The informal slogan of nominal techniques has always been 'ϵ-away from informal practice', and we believe that by recovering the 'always fresh' and 'always rename' properties, we move nominal techniques even closer to informal, pen-and-paper style reasoning. It remains to adopt permissive nominal techniques.

In [GM09e], Gabbay and Mulligan introduced permissive nominal algebra. A translation between the permissive nominal algebra theory ULAME and λ-theories was provided, and the translation was proved sound and complete in a suitable sense.

In [DGM10], Dowek and Gabbay prove that permissive nominal terms and higher-order patterns are 'essentially' the same thing. That is, there is a translation between permissive nominal terms and higher-order patterns which preserves unifiers, and is in some sense optimal. This result builds on a body of work by Levy and Villaret, and Cheney, but uses permissive nominal technology to simplify the proofs, cutting out 'cruft' dealing with freshness contexts.

In [DG10], Dowek and Gabbay introduced Permissive Nominal Logic. Intuitively, this can be seen as a first-order logic, with permissive nominal terms as its term language. Unknowns may be universally quantified over, and the labeling of unknowns with permission sorts makes this feasible. The logic was introduced specifically to encode, and reason about, formal systems that possess a name binding structure. Again, the use of permissive nominal terms simplifies the treatment of α-equivalence.

1.2 Contributions

The contributions of this thesis, along with references, are as follows:

The two-level λ-calculus. The two-level λ-calculus is a novel context calculus, presented in Chapter 3. It may be seen as an extension of nominal terms, where atoms and unknowns may be λ-bound. Associated notions of β-reduction are present for both levels.

Like all context-calculi, the two-level λ-calculus must overcome two hurdles: how best to handle α-equivalence in the presence of multiple levels of variable, and how best to handle a well known failure of commutation between naïve β-reduction and hole filling. The two-level λ-calculus solves both of these problems by bringing nominal techniques to bear.

We handle α-equivalence through the use of swappings, and explicit freshness constraints. Swappings have excellent properties: they are self-inverse, and all important relations defined on two-level λ-terms are equivariant (invariant under swapping).

The failure of commutation between β-reduction and hole filling is handled by imposing restrictions on the β-reduction of applications. This is similar in spirit to limitations that Sato [SSKI03] places on reductions in his calculus-of-contexts.

A strategy for proving confluence. The two-level λ-calculus is confluent (see Subsection 3.4.3 in Chapter 3). We split the proof of confluence into two halves: proving confluence of a 'level one' fragment of the calculus separately from confluence of a 'level two' fragment, then stitching the two together.

Of particular interest is the strategy for proving confluence of the level one fragment (Subsection 3.4.1). Here, we define a canonical form (Definition 3.4.13), pushing all substitutions down as far as possible, and prove that all terms eventually reduce to it (Theorem 3.4.22). We then show that if a term r reduces to s, then the canonical form of s reduces to the canonical form of r (Lemma 3.4.23). Confluence follows.

This technique appears to be useful in proving other calculi confluent, too. For instance, we reused the technique to prove one-and-a-half-level [GM09b] terms and the permissive two-level λ-calculus [GM10b] are confluent.

Permissive nominal terms. Permissive nominal terms are a variant of nominal terms [UPG04], presented in Chapter 4. Permissive nominal terms forego explicit freshness contexts. Instead, unknowns are labeled with an infinite and coinfinite set of atoms, called their permission sort.

Intuitively, a permission sort controls how an unknown is instantiated. We may only instantiate an unknown with a term whose free atoms is a subset of the unknown's permission sort. Further, the infinite and coinfinite nature of permission sorts allows us to recover two important properties that nominal terms lack: the 'always fresh' (the ability to find infinitely many fresh atoms for any term) and 'always rename' (ability to rename an abstracted atom to something fresh as needed) properties. Though these two properties are almost always assumed when working informally with the λ-calculus, for example, nominal terms lack these, and indeed we investigated permissive nominal terms because of the trouble with renaming in the confluence proof of the two-level λ-calculus.

Chapter 4 also introduces a unification algorithm for permissive nominal terms. We demonstrate that unification of permissive nominal terms is decidable, and the unification algorithm will always terminate (Theorem 4.3.42) with a 'correct' answer, in the following sense: if a unification problem has no solution, then the algorithm halts in a failing state, otherwise a most general unifier will be found. This is Theorem 4.3.58.

Finally, we provide a not entirely trivial translation between the nominal and permissive nominal worlds, and show that this translation preserves nominal unification solutions.

A prototype implementation of permissive nominal terms. The infinite and coinfinite nature of permission sorts seems to imply that permissive nominal terms has an infinitary syntax, and therefore hard to implement. Chapter 5 dispels those fears by providing an 'existential proof' that permissive nominal terms can be implemented in a straightforward manner.

Specifically, we introduce a prototype implementation of permissive nominal terms and their unification algorithm, called PNT. A frontend, allowing a user to easily define permission sorts and terms, is also provided, for ease of use.

1.3 Publications

The following is a list of publications written during the course of doctoral study. Forward links are included specifying which publications formed a basis for which chapters.

1. 'The two-level λ-calculus, part one', Murdoch J. Gabbay and Dominic P. Mulligan, **Journal of Logic and Computation**, under review, 2010.
2. 'Permissive nominal terms and their unification: an infinite, coinfinite approach to nominal techniques', Gilles Dowek, Murdoch J. Gabbay and Dominic P. Mulligan, **Logic Journal of the Interest Group in Pure and Applied Logic**, vol. 18, iss. 6, pgs. 769–822, 2010.
3. 'One-and-a-halfth-order terms: Curry-Howard for incomplete derivations', Murdoch J. Gabbay and Dominic P. Mulligan, **Journal of Information and Computation**, vol. 208, iss. 3, pgs. 230–258, 2010.
4. 'Universal algebra over lambda-terms and nominal terms: the connection in logic between nominal techniques and higher-order variables', Murdoch J. Gabbay and Dominic P. Mulligan, **Presented at the 4th International Workshop on Logical Frameworks and Meta-languages: Theory and Practice (LFMTP 2009)**.
5. 'The two-level λ-calculus', Murdoch J. Gabbay and Dominic P. Mulligan, **Electronic Notes in Theoretical Computer Science** vol. 246 pgs. 1-7–129, 2009.
6. 'Permissive nominal terms and their unification', Gilles Dowek, Murdoch J. Gabbay and Dominic P. Mulligan, **Presented at the 24th Convegno Italiano di Logica Computazionale (CILC 2009)**.
7. 'Semantic nominal terms', Murdoch J. Gabbay and Dominic P. Mulligan, **Presented at the 2nd International Workshop on the Theory and Applications of Abstraction, Substitution and Naming (TAASN 2009)**.
8. 'One-and-a-half level terms: Curry-Howard for incomplete derivations', Murdoch J. Gabbay and Dominic P. Mulligan, **Proceedings of the 15th International Workshop on Logic, Language, Information and Computation (WoLLIC 2008)**, Lecture Notes in Artificial Intelligence 5110, pgs. 179–193, 2008.
9. 'Dynamic relational rippling in HOL', Lucas Dixon and Dominic P. Mulligan, **Presented at the Automated Reasoning Workshop (ARW 2008)**.

CHAPTER 2

Background

This Chapter places nominal terms in a wider research context.

In Section 2.2 we provide a survey of the many techniques, other than the 'nominal approaches', developed by researchers for tackling the 'name binding problem'—how to work with abstract syntax and languages with binding. In Section 2.3 we provide a detailed survey of nominal techniques which have been developed to tackle the 'name binding problem' and their applications.

2.1 The λ-calculus

We provide a brief overview of the λ-calculus, enough to give the reader unfamiliar with the λ-calculus enough of a foothold to understand the rest of the thesis (particularly Chapter 3).

Functions are of fundamental importance in computer science and mathematics. Mainstream mathematicians interpret a 'function' as a set of pairs, where no two identical pairs possess identical first components (see any introductory text on set theory, for example [JW95]). For instance, in line with the standard set theoretical foundations of mathematics, the 'identity function' on real numbers is interpreted as the set:

$$\{(x, x) \mid x \in \mathbb{R}\}$$

However, there is an alternative interpretation of a 'function', not as a potentially infinite set of pairs, but rather a rule of correspondence, which maps a series of inputs to some potentially transformed output. Recast, our squaring function now can be written in the following style:

$$x \mapsto x$$

Here the input x is mapped to itself, x. This was the interpretation of function that Alonzo Church took when inventing the λ-calculus [Chu36, Bar84], a formal system intended to capture the notions of function definition via rules of correspondence, and function application.

The syntax of the λ-calculus is exceedingly simple, consisting of (in most presentations) just three syntactic classes. Let x, y, z and so on range over variables. Terms r, s, t, and so on, of the λ-calculus are then constructed using the following grammar:

$$r, s, t ::= x \mid rs \mid \lambda x.r$$

We call rs an *application* and $\lambda x.r$ an *abstraction*. Application models ordinary function application. Abstraction models the intuitive notion of abstracting a variable in order to form a function. For example, our previous example of identity function $x \mapsto x$ is modelled as the λ-term $\lambda x.x$.

Note that, for simplicity's sake, the λ-calculus is a calculus of functions that only accept single inputs. Functions taking multiple inputs can be modelled within the calculus through a process known as *Currying*. That is, all functions accepting multiple inputs are modelled as functions accepting a single input, and returning another function accepting another input, and so on. For example, the second projection function on pairs $(x, y) \mapsto y$ can be modelled within the λ-calculus as the Curried function $\lambda x.\lambda y.y$.

Further, note that in $\lambda x.r$, the λ-abstracted variable x is *bound* within the body of the abstraction, r. Reconsidering our previous example of the identity function, we note that the mappings $x \mapsto x$ and $y \mapsto y$ clearly denote the same function—the identity. So it is within the λ-calculus, where $\lambda x.x$ and $\lambda y.y$ are considered *α-equivalent*, or equivalent up to a systematic change in bound variables. α-equivalence, not syntactic identity, is the native form of equivalence on λ-terms.

We noted earlier that application rs models function application. Suppose our previous example of the identity function $x \mapsto x$. What should be the result of applying this function to 5? To apply a function to an input, we must perform a substitution, wherein x is replaced by 5 throughout the body of the function. That is, evaluating $(x \mapsto x)5 = 5$, and $((x \mapsto (y \mapsto x))5)3 = (y \mapsto 5)3 = 5$. Again, so it is in the λ-calculus, where $(\lambda x.x)y$ one-step *β-reduces* to y.

Note that function application makes use of substitution of terms for variables. We write $r[x \mapsto s]$ for the term r with free occurrences of x replaced by s throughout. Care, however, is needed when definining substitution. Through improper substitution, it's easy to destroy the functional meaning of a term by not properly respecting bound variables. For instance, the term $\lambda x.x$, the identity function, can easily have its functional meaning changed through the naïve substitution of x for y, resulting in the term $\lambda x.y$. This latter term corresponds to a constant function, always returning y, quite distinct from the identity. For this reason, substitution in the λ-calculus is carefully defined, and pushing a substitution under a λ-abstracted variable requires a careful analysis of *freshness conditions*, and possible renaming of bound variables. Concretely, with careful substitution, and suitable renaming, the result of our previous naïve substitution should properly be $(\lambda x.x)[x \mapsto y] = (\lambda x'.x')[x \mapsto y] = \lambda x'.x'$, where x' is a fresh variable.

The λ-calculus is *confluent*. If a term β-reduces in many steps to two distinct terms, then we can always close this 'gap' through more reductions, reducing both to a common term. The confluence property states that the order in which we β-reduce subterms of any particular term is irrelevant, and cements our intuition that the λ-calculus is indeed a calculus of functions.

We note that what has been presented here is the *untyped* λ-calculus. The untyped λ-calculus allows the definition of 'paradoxical' terms that never reduce to a *normal form*—a term that can no longer be reduced. For instance, Turing's ω combinator $(\lambda x.xx)(\lambda x.xx)$ reduces to itself in one step, and therefore we can construct an infinitely long reduction sequence by continually reducing this term.

Further, the untyped λ-calculus provides no facility for outlawing the application of functions to certain kinds of data, where this application would be senseless. In mathematics, it makes no sense to apply the square root function on the Reals to a Group, yet there is no way in the untyped λ-calculus to prevent a λ-encoded square root function being applied to a λ-encoded Group. These problems motivate the study of various *typed* λ-calculi, which aim to prevent various non-sensical reductions, and in many cases ensure that all terms reduce to a normal form [US06]. Further discussion of typed λ-calculi, though, is outside the scope of this introduction.

2.2 Encoding syntax and languages with binding

Abstract syntax manipulation is an important process in computer science. Many languages also incorporate name binding constructs. Finding methods for safely and elegantly working with abstract syntax with binding is therefore a major research field. This is reflected in the many different methods that have been proposed to handle the 'name binding problem'. We now survey the major approaches.

2.2.1 De Bruijn indices

Nicholaas de Bruijn invented his eponymous indices [dB72] as a 'nameless' representation of the λ-calculus for use in the Automath proof assistant [dB80]. The encoding technique does not rely on any peculiarities of the λ-calculus and as a result is commonly used as a nameless representation for other languages with binding.

De Bruijn's insight was to note that the particular choice of a bound name does not matter. However, what *does* matter is the occurence of a bound name within a term which 'points' to a particular binder. Bound variables can then be replaced by numeric pointers—indices in de Bruijn's parlance—which represent the particular binding site that is binding them.

The technique is best explained by example. Consider the following collection of λ-terms:

$$\lambda a.\lambda b.a \qquad \lambda a.\lambda b.\lambda c.bc \qquad \lambda a.a \qquad \lambda b.\lambda c.(b(\lambda d.c))$$

Through the use of underlining, we can connect a bound variable occurence with its binding site. Consider the previous collection of λ-terms, coloured, where the bound variable and respective binder are underscored with the same number of lines:

$$\underline{\lambda a}.\lambda b.\underline{a} \qquad \lambda a.\underline{\lambda b}.\underline{\underline{\lambda c}}.\underline{b}\underline{\underline{c}} \qquad \underline{\lambda a}.\underline{a} \qquad \underline{\lambda b}.\underline{\underline{\lambda c}}.(\underline{b}(\lambda d.\underline{\underline{c}}))$$

To complete the process, we now replace each underscored variable by a natural number. Intuitively, the index signifies how many binders to the right we need to traverse in order to reach the binder with the same number of underscores as the variable. We then drop variable names from terms altogether by replacing λa by λ. Performing this operation, we obtain the following de Bruijn encodings of the previous collection of λ-terms:

$$\lambda\lambda\ 2 \qquad \lambda\lambda\lambda\ (2\ 1) \qquad \lambda\ 1 \qquad \lambda\lambda\ (2\ (\lambda\ 2))$$

Free variables are assumed 'bound at top level'; a free variable is represented by a numerical index into a global context, associating indices with concrete names.

De Bruijn also derives a number of common operations, for instance capture-avoiding substitution, for working on languages encoded with his representation. The details of these operations involve some quite involved arithmetic in order to ensure that variables aren't inadvertently captured. This arithmetical manipulation is often cited as a key disadvantage of the de Bruijn representation along with the fact that the terms aren't human readable [BU06]. This makes them unsuitable for use in 'customer facing roles'—the interface between human users and the implemented system—for instance.

A related disadvantage of the de Bruijn representation comes when reasoning needs to be performed on an encoded language. Berghofer and Urban [BU06] used a de Bruijn encoding to formalise in Isabelle the substitution lemma from the λ-calculus. They point out that additional lemmas are particularly hard to spot, due to the arithmetic involved in substitutions, and when the lemmas finally are found they are extremely brittle. They quote Nipkow, in support of their claim [BU06, pg. 6]:

> Initially, I tried to find and prove these lemmas from scratch but soon decided to steal them from Rasmussen's ZF proofs, instead, which has obvious advantages:
> — I did not have to find this collection of non-obvious lemmas myself ...

Despite this, Berghofer and Urban point out that the lemmas do not change much between different encoded languages, and are particularly amenable to this form of borrowing.

Despite these drawbacks, de Bruijn indices *do* have a number of nice properties. Perhaps most significant is the fact that α-equivalence test now coincides precisely with syntactic identity on de Bruijn terms. For instance, it should be intuitively clear that the two λ-terms $\lambda a.\lambda b.a$ and $\lambda c.\lambda d.c$ are both mapped to the same de Bruijn representation, $\lambda\lambda\ 2$, and in general the de Bruijn encoding is invariant under α-conversion. This is a particularly important feature of de Bruijn encodings as it allows us to easily compute functions that operate on α-equivalence classes.

Another key advantage is the first-order nature of the encoding. We do not need to resort to anything as complex as, for instance, function spaces to represent binding. Instead, a de Bruijn representation allows a programmer to use standard features of many programming languages, like algebraic datatypes, to encode a language.

Berkling's encoding [Ber76] of the λ-calculus is closely related to the de Bruijn encoding. In Berkling's encoding aliased bound names are annotated with an index, which denotes how many binders need to be skipped for that particular variable name, not counting binders for variables with different names. The Calculus of Indexed Names and Named Indices (CINNI) [Ste00], a calculus of explicit substitutions designed for embedding object languages into, uses Berkling's notation.

Many other techniques are based on, or variations of, de Bruijn's encoding [MM04, Pol93, MP97]. Subsection 2.2.2 will introduce a series of closely related techniques.

Pros: Syntactic identity and α-equivalence coincide. As a first-order approach, any implementation can use existing infrastructure provided by most functional programming languages: recursive datatypes, pattern matching etc.
Cons: The definition of standard operations requires tricky arithmetic. Reasoning about encoded languages in a proof assistant requires 'non-obvious' lemmas. Encodings are hard to read for humans.

2.2.2 Locally nameless and locally named approaches

Locally nameless Locally nameless approaches [MM04, Pol93, Gor93] use a mixed representation of de Bruijn indices and variable names. Bound variables are represented using de Bruijn indices whilst free variables are given explicit names. This goes some way to ameliorating the readability problem of de Bruijn indices. Like de Bruijn encodings, locally nameless encodings have the advantage of syntactic identity and α-equivalence coinciding.

McBride and McKinna use a locally nameless approach within the implementation of Epigram, a dependently typed functional language [McB04]. However, locally nameless approaches have a history dating back to Pollack's work on the LEGO proof assistant [Pol93] and publications by Gordon [Gor93] implementing the technique in the HOL proof assistant.

Locally nameless approaches still require some arithmetical manipulation when defining operations on terms, but compared to de Bruijn indices, this is reduced.

Several tools have been implemented for automatically generating locally nameless encodings of languages with binding from a grammar. Ott [SNO$^+$10] compiles a grammar, similar to an informal pen-and-paper based presentation of syntax with binding, into a locally nameless encoding provided as a series of definitions in an Isabelle, HOL or Coq theory file, or alternatively into a LaTeX file for publication. LNGen [AW10] further extends Ott by automating many of the routine 'nameplate' lemmas that need to be proved about locally nameless encodings. However, LNGen presently only targets Coq.

Pros: Syntactic identity and α-equivalence coincide. As a first-order approach, any implementation can use existing infrastructure provided by most functional programming languages: recursive datatypes, pattern matching etc. Encodings use explicit names for free variables, so are more readable than de Bruijn encodings.
Cons: The definition of standard operations requires some arithmetic, though this is reduced compared to de Bruijn encodings.

Locally named Locally named approaches [MP97] use names for both bound and free variables. To handle issues with α-equivalence, bound and free variables are assumed to come from two disjoint syntactic classes: *parameters* are used for free variables, whilst *variables* are used for binding.

The use of parameters facilitates a straightforward definition of capture-avoiding substitution, but unlike de Bruijn indices, α-equivalence and syntactic identity no longer necessarily coincide. Care must also be taken to ensure that terms are well-formed, and have no dangling free variables (in Pollack and McKinna's terminology, ensure that the terms are 'vclosed').

The use of parameters for bound variables has a long history in logic (for instance, Smullyan's book on predicate logic [Smu95]). Pollack and McKinna used the locally nameless approach to formalise a large amount of type theory in the LEGO proof assistant [MP97]. Locally nameless approaches were also applied to formalising the metatheory of programming languages [ACP$^+$08], and to the POPLMark challenge [Ler07].

A variation of the locally named approach is the Sato representation [Sat08, SP10]. This is 'canonical' in the sense that terms are placed into an α-normal form, where α-equivalence and syntactic identity coincide, restoring this important property that de Bruijn encodings enjoy but other locally named approaches do not.

Pros: As a first-order approach, any implementation can use existing infrastructure provided by most functional programming languages: recursive datatypes, pattern matching etc. Unlike de Bruijn and locally nameless approaches, encodings use no numerical indices.
Cons: Syntactic identity and α-equivalence need not coincide. Care must be taken to ensure encoded terms are well formed.

2.2.3 Higher-order abstract syntax

Pfenning and Elliott introduced the slogan 'higher-order abstract syntax' [PE88] referring to a mechanism that they were using for representing object languages with name binding constructs in Ergo, a language generic development environment.

Higher-order abstract syntax uses metalevel function spaces as the means of representing name binding in object level languages. Typically, this is achieved by using a typed λ-calculus as the meta-language in which object languages are embedded. For instance, Pfenning and Elliot originally used the simply typed λ-calculus extended with products and polymorphism. However, the term is also widely used to refer to similar embeddings in other meta-languages, such as Haskell (for instance, see [Aug06]).

Pfenning and Elliott merely popularised the *term* 'higher-order abstract syntax' not the technique with which it later became synonymous. A similar technique, where object level binders were being implemented in terms of metalevel binders, was already being used in the Edinburgh Logical Framework at the time of publication of [PE88]. Further, what exactly constitutes 'higher-order abstract syntax' is open to dispute. For instance, Felty and Pientka [FP10, pg 1] characterise higher-order abstract syntax as:

> Our focus in this paper is on encoding metatheory of programming languages using higher-order abstract syntax (HOAS), where we encode object-level binders with metalevel binders.

Washburn and Weirich also give a similar definition [WW03, pg 1]:

> The main idea is elegant: instead of representing object variables explicitly, we use metalanguage variables.

As does Atkey [Atk09, pg 2]:

> Another common approach is to use higher-order abstract syntax [13]. In this approach, we use the binding structure of the meta-language to represent binding in the object-language.

Such definitions, whilst useful as a very high level overview of the technique, are clearly deficient as *encodings of object languages in nominal terms also use metalevel binding for encoding object level binders*! However nominal terms are routinely characterised as an extended first-order technique not higher-order.

To properly distinguish between nominal encodings and higher-order encodings we insist that, when discussing higher-order abstract syntax, the meta-language in question is the λ-calculus, or some variant of. This isn't an idiosyncratic choice as many authors wishing to discuss the advantages and disadvantages of higher-order abstract syntax also make a similar assumption. For instance, Cheney's critique of higher-order abstract syntax [Che05c] makes little sense without the implicit assumption that the metalevel calculus in question is some form of the simply-typed λ-calculus.

The use of metalevel function spaces for representing object level name binding can be traced back to Church's work on the simple theory of types [Chu40]. Church worked in the simply typed λ-calculus, and hypothesised two base types: ι the type of individuals and o the type of truth values. A constant symbol $\forall$ with type $(\iota \to o) \to o$ was introduced and Church embedded the object level universal quantifier into the simply typed λ-calculus using this constant. For example, $\forall(\lambda x.\bot)$, where $\bot$ is a constant of type o representing falsehood, can be correctly typed with o.

In the previous example, we see that object level binding (the universal quantifier) is handled by the metalevel binder (the λ of the simply typed λ-calculus). Using this pattern, we obtain a generic method for embedding languages with binding. In addition, tricky operations such as capture-avoiding substitutions, fresh name generation and α-equivalence tests are inherited by the object languages from the meta-language. These operations need therefore only be implemented once at the metalevel and forgotten about. This is a significant time saver and also serves to reduces bugs.

Higher-order abstract syntax, widely construed, is implemented in a large range of systems. For instance, Isabelle, amongst many other proof assistants, use higher-order abstract syntax for representing its object logics [Pau10]. A recent trend has seen programming languages being implemented with the explicit goal of working with higher-order data, representing deductive systems implemented in Edinburgh LF. A language in this mould is Beluga [PD08, PD10], though Delphin [PS08b] (and Elphin, it's simply typed cousin [SPS05]) predate Beluga. This trend throws up additional challenges, for instance, it is now useful to be able to perform case analysis on higher-order data [DP08].

Higher-order abstract syntax has a number of advantages: it is widely implemented, relatively conceptually simple and does save significant time and effort, as many operations on encodings can be relegated to the metalevel. However, the technique also has a number of disadvantages. The next few subsections will characterise the major flaws of higher-order abstract syntax, and attempt to survey the large body of research that has been carried out seeking solutions to these problems.

2.2.3.1 Higher-order unification

Higher-order unification is the unification up to $\beta\eta$-equivalence of simply-typed λ-terms (though the phrase is also used to describe the unification of terms with more expressive types). Higher-order unification was investigated in the hope of finding an automated theorem proving method for higher-order logic [Dow01].

Higher-order unification is widely used. Notably, when working with higher-order abstract syntax first-order unification is not sufficient and higher-order unification must be used due to the presence of function variables. In addition, higher-order unification finds uses in computational linguistics [GK96a, GK96b] and in type inference for some programming languages [Pfe88], amongst other varied uses.

Higher-order unification is undecidable and this can be shown via a reduction to Hilbert's tenth problem [Dow01]. (In fact, Goldfarb proved a more general result, that second order unification is undecidable, and the undecidability of higher-order unification follows from that [Gol81].) By undecidable, we mean that no algorithm exists that takes a higher-order unification problem and outputs an answer if it has a solution or not. However, higher-order unification is semi-decidable in that a unification algorithm exists that terminates and returns an answer if a unification problem has a solution, possibly diverging otherwise. The most famous semi-decision procedure along these lines is Huet's algorithm [Hue75], though a semi-decision procedure using explicit substitutions is also widely known [DHK95].

Intuitively, Huet's algorithm follows a 'generate and test' pattern. Unification problems are split into three groups: rigid-rigid, flexible-rigid and flexible-flexible. A λ-term is said to be rigid if its head symbol is a constant or bound variable; a term is flexible if its head symbol is a free variable.

Rigid-rigid terms may be simplified directly, whilst flexible-rigid terms require all possible substitutions making the two terms $\beta\eta$-equivalent to be enumerated (fortunately, this process may be restricted) and applied to the terms, one by one. However, flexible-flexible terms also require the enumeration of all possible substitutions, but there is no way to restrict this blind enumeration process. To counter this, Huet demonstrated that every flexible-flexible pair of terms has a unifier—useful if all we care about is the existence of a unifier.

When compared with first-order unification, higher-order unification has poor computational properties. Higher-order unification may not terminate, and, if it does, the resulting unifier may not be most general (in fact, unique most general unifiers need not exist). These poor computational properties have led many researchers to attempt to identify subsets of λ-terms with better computational properties. One subset of particular importance is Miller's higher-order patterns.

Higher-order patterns Miller [Mil91a] identified a subset of λ-terms for use in the logic programming language L_λ that admitted a decidable unification algorithm with most general unifiers. Nipkow later named this subset 'higher-order patterns' [Nip93a] and distilled the rigorous but high-level description of the algorithm by Miller into an algorithm for unifying higher-order patterns suitable for implementation in a functional programming language. Higher-order patterns are used widely, for instance in the logic programming language λProlog [NM88], and the proof assistants Abella [Gac08] and Isabelle [Pau98].

Intuitively higher-order patterns are a linearised form of λ-term. Miller first noted that every λ-term may be placed into β-normal-η-long form where no β-redexes exist and no η-expansion can be performed without introducing another β-reduction. Normal forms of this sort correspond to the following grammar (due to Cheney [Che05d]):

$$g ::= \lambda \bar{x}.y\ \bar{g} \mid \lambda \bar{x}.Y\ \bar{g}$$

Here, Y is a metavariable, and $\bar{g}$ denotes a vector of λ-terms.

Miller's insight was in noting that the nondeterminism in higher-order unification stems from the uncertainty in how a metavariable Y may act on its argument list of terms $\bar{g}$. This list of arguments may itself contain other metavariables or repeated terms. This insight was further motivated by noting that higher-order unification can be decomposed into higher-order pattern unification and a suitable search strategy, separating issues dealing with name binding from the search aspect [Mil91b].

Miller removed the uncertainty surrounding the behaviour of metavariables on argument lists by placing restrictions on what form an argument list may take. Each list $\bar{g}$ must consist of distinct bound variables, and, as a result of this restriction, the flexible-flexible pairs of Huet's algorithm may always be solved.

The unification theory of higher-order patterns is well-understood. A linear time and space algorithm for higher-order patterns was investigated by Qian [Qia93], although never implemented[1]. Dowek and Pfenning designed a pattern unification algorithm using explicit substitutions [HDKP98]. Pientka and Pfenning proposed several optimisations, notably a form of linearisation which further removes occurs checks, and justified the use of these optimisations with a modal type theory [PP03]. Pfenning extended higher-order pattern unification to dependent types (specifically the Calculus of Constructions) [Pfe91]. Pientka investigated optimisations of higher-order pattern unification in a dependently typed setting [Pie06].

In addition to (syntactic) higher-order pattern unification, pattern antiunification [Pfe91], pattern E-unification (i.e. semantic unification with respect to an equational theory) [Bou00, BC97, BC01], pattern complement [MP03] and relative complement [MP99], and disunification, with Lugiez identifying some decidable subcases [Lug94], have all been investigated. A slight relaxation of higher-order patterns have been profitably used in a program transformation framework [YHT04]. Higher-order patterns have also been incorporated into rewriting frameworks; Nipkow defined higher-order rewrite systems using higher-order patterns and pattern unification [Nip91, Nip93b], for instance.

Though undecidable higher-order unification has been reported to work well in practice [Pau88]. Empirical evidence suggests that higher-order pattern unification is sufficient to solve the overwhelming majority of higher-order unification problems [Pau98]. Nipkow discovered that 97% of all higher-order unification problems that arise through everyday use of Isabelle can be solved by higher-order pattern unification alone. This suggests a possible explanation for higher-order

[1] Dale Miller, personal communication.

unification's good computational behaviour in practice (in Paulson's words: higher-order unification 'performs sufficiently well in practice' [Pau88]), and Isabelle now defaults to solving higher-order unification problems with higher-order pattern unification, only using full higher-order unification when necessary.

Higher-order pattern unification and nominal unification share a close relationship with each other. A recent body of research suggests that higher-order patterns and nominal terms have 'morally' the same expressive power [Che05d, LV08, DGM10]. This will be discussed later in the thesis.

2.2.3.2 Comparing names: α-inequality

Unlike nominal approaches bound names in higher-order abstract syntax do not enjoy an independent denotational existence, as names are taken to α-vary. This makes defining natural relations between terms, such as α-inequivalence, tricky. The Twelf implementation of the Edinburgh Logical Framework, a major user of higher-order abstract syntax, employs hypothetical judgments for handling this problem [CH06].

2.2.3.3 Use in Coq and adequacy

Imagine a higher-order abstract syntax embedding of the λ-calculus. A type trm is introduced representing the type of λ-terms, and two constants, $app : trm \to (trm \to trm)$ and $lam : (trm \to trm) \to trm$ representing application and λ-abstraction, respectively. These constants are not the constructors of an inductive type, as there is a negative appearance of trm in the type of lam. (To maintain the consistency of their logics, type theory based proof assistants like Coq or Matita incorporate a *positivity checker*, which checks that types do not incorporate a *negative occurence*, leading to possible non-termination.) As a result, an induction principle for this type cannot be formulated, and implementing this embedding in type-theories like the Calculus of Constructions, as used by Coq [BC04], is impossible.

Further, suppose we extend our simply typed metalanguage with a feature for case distinction. Then, not every well-typed term of trm represents a valid λ-term. For instance:

$$lam(\lambda x.\text{case } x \text{ of } app\ l\ r \Rightarrow app\ r\ l \mid lam\ b \Rightarrow lam\ b)$$

This term can be successfully typed as trm but does not represent any valid λ-term. By allowing case distinction on bound variables we have lost *adequacy* in our encoding.

How can these problems be handled?

As Coq is a widely used proof assistant, the problems with higher-order abstract syntax in Coq have led many researchers to investigate alternative formulations of the technique, suitable for use in type-theory based proof assistants. Of particular note is 'weak higher-order abstract syntax' [DFH95]. Weak higher-order abstract syntax introduces a dedicated type for variables var, injected with a dedicated constructor $variable : var \to trm$, and replaces the negative occurence of trm in the type of lam with var. That is, the type of lam now becomes $(var \to lam) \to lam$.

Parametric higher-order abstract [Chl08] syntax expands upon this idea, by limiting how inhabitants of this var type may be analysed with parametricity.

A significant body of research has been undertaken addressing the adequacy problems of higher-order abstract syntax. Washburn and Weirich used System-F style polymorphism to limit how bound variables may be analysed [WW03]. That is, they claim the following type

$$\forall a.((a \to a) \to a) \to (a \to a \to a) \to a$$

represents exactly the untyped λ-terms. Here, the carrier type a has been universally quantified over, but the two operations of higher-order abstract syntax, *lam* and *app* are captured in the type. Washburn and Weirich provide some fold-like operators and associated reasoning principles for working with inhabitants of this type.

Intuitively, it seems obvious that this type *does* capture untyped λ-terms, as the quantification over the carrier type means it is no longer possible to perform the case analysis that destroys adequacy (the parametricity means that we cannot assume anything about the carrier type). However, Washburn and Weirich do not offer a formal proof that this is the case. Similarly, Coquand and Huet state without proof that this type represents untyped λ-terms [CH85], and Chlipala's previously mentioned parametric higher-order abstract syntax uses a similar idea [Chl08], again, without proofs of correctness.

Recently, Atkey provided a proof that the previous type does in fact represent untyped λ-terms [Atk09]. Using Kripke logical relations, Atkey demonstrated that the denotation of the type is isomorphic to closed de Bruijn terms (de Bruijn terms without dangling pointers). The proof was formalised inside a version of Coq.

An alternative approach to regaining adequacy was investigated by Fegaras and Sheard [FS96]. Here, case analysis and pattern matching of bound variables is restricted, without the use of Atkey's System-F style of polymorphism, ensuring adequacy.

Pros: Pushes frequently needed operations on encoded languages to the metalevel. The technique is widely implemented. Common restricted cases use efficient higher-order pattern unification.
Cons: There are problems with adequacy and use in type theory based proof assistants. The general case requires undecidable higher-order unification. There are problems defining 'natural' relations on encoded languages, such as α-inequality.

2.2.4 The Pronominal Approach

The pronominal approach [HLZ09, LZH08] may be seen as a variation of higher-order abstract syntax. Subsubsection 2.2.3.3 has already summarised the problems of negative occurences in higher-order abstract syntax, especially with regards to implementation in type-theory based proof assistants. The pronominal approach counters this problem by introducing two function spaces: one with positive polarity for ordinary computation in the style of the λ-calculus, and another function space with negative polarity for constructing datatypes with binding. The negative polarity of the binding function space makes pattern matching and recursion under binders easy to implement.

The pronominal approach retains many of the advantages of higher-order abstract syntax. For instance, binding is still handled at the metalevel by a function space, just one of a different polarity to standard higher-order abstract syntax, and hence capture-avoiding substitution, α-equivalence tests and fresh name generation can all be relegated to the metalevel, once and for all.

The most recently investigated pronominal type theory is simply typed. However, plans exist for a dependently typed version along the lines of Edinburgh LF [LZH08].

Pros: Pushes frequently needed operations on encoded languages to the metalevel. One can easily construct data types with binding that can be analysed using pattern matching in a straightforward manner.
Cons: The technique is not very widely implemented. Two function spaces, one for computation, and another for name binding, are needed.

2.2.5 Calculus of Nominal Inductive Constructions (CNIC)

The Calculus of Constructions, a dependent type theory, is the most expressive of all eight systems in the λ-cube [US06]. As a consequence, the calculus and its immediate extensions, notably the Calculus of Inductive Constructions (CIC), has been used as foundations for many widely used proof assistants, including Coq [BC04], Lego [Bur91] and Matita [ACTZ07].

Many users of proof assistants based on CIC wish to formalise the metatheory of programming languages, logics, calculi, and other formal systems with name binding structures. However, techniques like higher-order abstract syntax are awkward to use in CIC proof assistants like Coq, as discussed in Subsubsection 2.2.3.3.

Westbrook et al propose the Calculus of Nominal Inductive Constructions (CNIC) [WSA09]. This extends CIC with an intensional name binding construct ν which introduces a fresh name in a local scope. Despite the name CNIC is not closely related to the nominal techniques described in Section 2.3.

CNIC's ν binder satisfies four properties: name freshness (the generated name is distinct from all others), terms including ν are equal up to α-equivalence, a name cannot escape from its scope, and different types of variables can be bound. To complement ν CNIC also introduces features for deconstructing terms with bound names.

Westbrook et al prove that CNIC is a suitable for use in a logical framework by proving consistency and strong normalisation via a suitable reduction to CIC. A prototype implementation of CNIC called Cinic is also described. Work on CNIC and Cinic continues.[2]

Pros: The technique supports straightforward encodings of languages with name binding, and manipulating those encodings, in a dependent type theory.
Cons: The technique is not very widely implemented (prototype implementation still under active development).

[2]Edwin Westbrook, personal e-mail.

2.3 Nominal techniques

Nominal techniques were introduced just over a decade ago, by Gabbay and Pitts [GP99, Gab00]. Since then, a large amount of work has been carried out, extending the field in different directions. What follows is a brief summary of this work, sectioned into related areas for easy navigation.

2.3.1 Nominal sets and foundations

Fraenkel-Mostowski (FM) is a set theory with atoms (*urelemente* in set theory parlance). FM set theory was applied by Fraenkel to the study of the independence of the axiom of choice from other axioms in first Zermelo-Fraenkel with Atoms (ZFA) [Fra22], and later—by Cohen—ZF [Coh63]. Eventually, FM was rediscovered and applied to the problem of abstract syntax with name binding by Gabbay and Pitts [GP99, Gab00].

Several set theories, and classes of sets within these theories, related to nominal techniques exist: ZFA, FM, equivariant FM sets and nominal sets. The FM sets hierarchy resides inside the ZFA hierarchy. Within the FM sets hierarchy resides the class of equivariant FM sets. Equivariant FM sets provide a model for 'nominal sets'. Nominal sets are an axiomatisation of a set with a finitely supported permutation action upon it, and related operations, that facilitate straightforward concrete definitions of abstract syntax with binding.

As standard set-theoretic constructions such as functions and products can also be defined in the FM set hierarchy, atom abstraction can be used to model bound names in structures other than abstract syntax. In addition, the notion of 'equivariance', or invariance under permutation, captures the idea that particular choices of name do not matter.

Various case studies support the claim that FM sets correctly capture abstract syntax with name bining up to α-equivalence. For instance, Gabbay encoded the π-calculus in FM sets [Gab03], and encodings of the untyped λ-calculus up to α-equivalence also exist [GP99]. Gabbay also demonstrated that FM sets provide a model for capture-avoiding substitution, as well as abstract syntax up to α-equivalence [GG08]. Clouston and Pitts used FM sets to develop Nominal Equational Logic [CP07], a simple logic of equality, with a natural interpretation in nominal sets. Turner and Winskel applied domains constructed within FM sets to the study of concurrent calculi, obtaining adequacy and soundness results [TW09].

Gabbay made an attempt at providing automated support for FM-style reasoning in the Isabelle proof assistant [Gab00, Gab02b]. Nominal Isabelle continued in this vein (Section 2.3.7).

FM sets facilitate a generic atoms abstraction definition, but only single atoms may be abstracted in a set. Gabbay later generalised nominal sets to allow the binding of infinitely many atoms [Gab07b].

2.3.2 Nominal terms and unification

Terms Nominal terms [UPG04] were developed as a metalanguage for encoding object languages with binding with a concrete semantics in FM sets. Specifically, nominal terms were proposed in order to capture statements like the following, taken from the metatheory of the λ-calculus, inside a formal language:

$$(\lambda x.r)[y{\mapsto}t] = \lambda x.(r[y{\mapsto}t]) \quad (\text{if } x \notin fv(t)) \tag{2.1}$$

Looking closely at this statement, we see the following unique features:

- The 'terms' r and t are really metavariables which can be instantiated to any term
- The name x is bound in the metavariable r
- There exists a freshness side-condition on t, stating that the name x must not occur in whatever t is instantiated to

Nominal terms attempt to internalise these features inside a formal syntax:

$$r, s, t ::= a \mid \pi{\cdot}X \mid [a]r \mid \mathsf{f}(r_1, \ldots, r_n)$$

Here, a ranges over *atoms*, π over *permutations*, X over *unknowns*, and f over *term-formers*. A term of the form $[a]r$ is an *abstraction*, and in $\pi{\cdot}X$ we state that π is *suspended* on X.

Atoms a model 'names' and are intended to capture the role of x in Equation 2.1. They may be compared with other atoms for equality and inequality, but have no other internal structure. Many 'natural' relations between terms, such as α-inequivalence, rely on the ability to compare atoms for inequality. An atom may be abstracted in a term $[a]r$. This abstraction is intensional: it aims to capture what λ, π, ν, and all other name binders, have in common.

Unknowns X, on the other hand, model metavariables, such as r and t. Unknowns have a capturing substitution action, intended to model metavariable instantiation. We write $[X{:=}t]$ for the substitution mapping X to t, and we can easily extend substitutions to a *substitution action* on terms $r[X{:=}t]$. For example, $([a]X)[X{:=}a] \equiv [a]a$, whilst $([b]a)[X{:=}c] \equiv [b]a$, where $\equiv$ denotes syntactic equality between nominal terms.

Permutations π are finitely supported bijections on atoms[3]. Nominal terms use permutations, specifically 'swappings' of atoms, to handle α-equivalence, as, in Cheney's words, they are 'inherently capture avoiding' [Che05c]; swapping fresh atoms in a term always leads to an α-equivalent term. Swappings also have some other properties that make them preferable to using substitutions: they are self-inverse, easy to compose, and most relations are equivariant, or invariant under permutation.

A *permutation action* $\pi{\cdot}r$ on terms is also easy to define: permutations 'just' traverse through the structure of a term until they reach an atom, in which case they act, or hit an unknown, in which case they suspend. For instance, writing $(b\ a)$ for the permutation that swaps b with a with b, and leaves all other c fixed, we have:

$$(b\ a){\cdot}[a]\mathsf{f}(b, X, c) \equiv [b]\mathsf{f}(a, (b\ a){\cdot}X, c)$$

[3] Alternatively, permutations are often taken as finite lists of pairs of atoms. This is the approach taken in the original nominal unification paper [UPG04], for instance, and also within earlier versions of Nominal Isabelle [UNB07]. The latest versions of Nominal Isabelle now assume permutations are finitely supported bijections of atoms [HU10].

Intuitively, a suspended permutation, such as $(b\ a){\cdot}X$, in the example above, waits to act on whatever X is eventually instantiated to. This creates a sort of 'memory', where nominal unknowns, modelling metavariables, remember name changes modelling α-equivalence for abstracted atoms that may lie above them in the term.

These are several alternative presentations of nominal terms, tailored to specific purposes. For instance [GM09e] elides termformer arguments, using constants and term application instead:

$$r, s, t ::= a \mid \pi{\cdot}X \mid \mathsf{f} \mid rs \mid [a]r$$

This style of nominal term was proposed to make proofs relating nominal algebras and λ-algebras more convenient. Calvès also introduces several variants of nominal terms better suited to implementation [Cal10] in his thesis. For instance, he introduces *compact nominal terms* with explicit suspended substitutions, a technical device used in the implementation of efficient unification algorithms:

$$r, s, t ::= a \mid X \mid \mathsf{f} \mid rs \mid [a]r \mid \pi{\cdot}t$$

Derivable freshness and α-equivalence Nominal terms may contain unknowns X. Unknowns may be substituted for any term, and therefore behave as if they have an infinite set of free atoms. A generalisation of the free atoms of a term therefore needs to be found.

Nominal terms use a *derivable freshness* relation; syntax directed rules dictate when an atom a may be considered fresh for a term r, with respect to some assumptions about the freshness of atoms relative to unknowns. We write $\Delta \vdash a\#r$ for 'a is derivably fresh from r using the assumptions in Δ'. Here, Δ is a finite set of *freshness assumptions* of the form $a\#X$; we read $a\#X$ as 'a is assumed fresh for X'.

With freshness described, we can express Equation 2.1:

$$a\#Y \vdash \mathsf{Sub}([b](\mathsf{Lam}([a]X)), Y) \longrightarrow \mathsf{Lam}([a](\mathsf{Sub}([b]X, Y)))$$

Here, we render Equation 2.1 inside the framework of nominal rewriting (discussed in Subsection 2.3.5), where the informal equality of Equation 2.1 is replaced by a directed equality, interpreted as a rewrite arrow $\longrightarrow$. Two termformers, Lam and Sub, model λ-abstraction and substitution. The holes, r and t, in Equation 2.1 are modeled by the nominal unknowns X and Y. The bound variable x in r is modeled by abstracting a in X. The informal freshness side-condition is captured by the freshness assumption $a\#X$.

Derivable freshness is a central component in α-equivalence checking. Nominal terms introduce an intensional name binding construct $[a]r$, and this implies that the native notion of equality on nominal terms should be α-equivalence. Like freshness, α-equivalence is defined through a derivable relation, induced by a series of syntax directed rules, with respect to a freshness context. Derivable α-equivalence is parameterised by a freshness context because we need to check α-equality for 'open terms'—terms with occurences of unknowns. We write $\Delta \vdash r \approx s$ for 'r and s are derivably equal, under the freshness assumptions in Δ'.

Unification Nominal terms have a computationally tractable unification algorithm: nominal unification is decidable, despite nominal terms possessing a notion of name binding. The unification algorithm of [UPG04] works by recursive descent, simplifying nominal unification problem (finite sets of pairs of terms of the form $r \overset{?}{=} s$). Nominal unification returns, as output, a substitution θ and a freshness context Δ, where given $r \overset{?}{=} s$ as input $\Delta \vdash r\theta \approx s\theta$ is derivable.

The naïve algorithm of [UPG04] works by recursive descent, and has exponential worst-case running time (the same examples that make naïve first-order unification algorithms exponential also work for the recursive descent nominal unification algorithm). However, a large amount of research, by Calvès and Fernández, has been expended trying to define a more efficient form of the algorithm than the naïve recursive descent algorithm presented in [UPG04]. A polynomial time algorithm for nominal unification, based on optimisations taken from implementations of first-order unification, was discovered and implemented [CF07, CF08a, CF08b].

These polynomial asymptotic time bounds for nominal unification have recently been sharpened by two independent results. Fernández and Calvès introduced a quadratic time nominal unification algorithm, based on the Martelli and Montari first-order unification algorithm [Cal10]. Similarly, Levy and Villaret have also produced a quadratic time nominal version of the Patterson and Wegman first-order unification algorithm [LV10].

Further, Kumar and Norrish have also investigated more efficient forms of nominal unification [KN10]. Their unification algorithm uses 'triangular substitutions', which unlike the substitutions returned as a solution from the standard nominal unification algorithm, need not be necessarily idempotent, but is better suited to backtracking search in logic programming. They use their unification algorithm in the αLeanTAP theorem prover [NBF09].

Calvès later implemented nominal terms in the Haskell Nominal Toolkit [Cal09]. He provided associated zipper data structures for traversing them, and with several algorithms defined over them, including nominal unification and matching.

Nominal unification is unification of nominal terms up to α-equivalence, i.e. given two terms r and s, find a θ such that $\Delta \vdash r\theta \approx s\theta$. However, for some applications (for instance rewriting and αProlog) nominal unification is not enough, and equivariant unification is required. Equivariant unification unifies terms up to permutation, i.e. given two terms r and s, find a θ and π such that $\Delta \vdash r\theta \approx \pi \cdot s\theta$. Unfortunately, equivariant unification is known to be NP-complete [Che04a, Che05a].

Fernández and Gabbay first anticipated this phenomenom whilst investigating nominal rewriting (see Subsection 2.3.5 for a description of the problem in the context of rewriting systems), and introduced the notion of 'closed terms' which do not require equivariant unification [FGM04]. Cheney later formally demonstrated that the equivariant unification problem is NP-complete [Che04a, Che05a]. However, Cheney and Urban were able to mitigate this by showing that in a large percentage of cases equivariant unification can be avoided in αProlog [UC05] by using the notion of closed terms, introduced by Fernández and Gabbay.

2.3.3 Nominal algebras

Nominal algebra [GM06, GM07b, Mat07, GM09e, GM09a] is a simple logic of equality over nominal terms. Nominal algebra may be seen as a variant of universal algebra, with the addition of a name binding construct and freshness side-conditions. The λ-calculus [GM07a, GM10a], capture avoiding substitution [GM08a] and first-order logic [GM09a], amongst other theories, have all been axiomatised within nominal algebra.

Nominal algebra is the application of nominal techniques to algebraic reasoning in the presence of atoms, unknowns, atoms abstraction and freshness side-conditions. Informal equivalences may be internalised and formalised within nominal algebra with a close approximation to informal practice. For instance, η-equality in the λ-calculus may be formally captured by the following axiom:

$$a\#X \vdash \lambda[a](Xa) = X \tag{2.2}$$

Here, λ is a term-former, and $a\#X$ is a freshness side-condition, stating that a must be fresh for X for the equality to hold. In Equation 2.2, we sugar $\mathsf{App}(X, a)$, where App is another term-former, to Xa, for reasons of legibility.

Equation 2.2 also reveals something interesting about nominal algebra. Namely, nominal algebra axioms appear to have an implicational flavour, in that they are conditional on a freshness side-condition, in this case $a\#X$, being met. In what sense, then, is nominal algebra an algebra—a simple logic of equality?

Birkhoff's theorem, otherwise known as the HSP theorem for 'homomorphism, subalgebra, product', is a fundamental theorem in universal algebra [BS81]. The theorem characterises models of an algebraic theory as a class closed under the operations of homomorphism, subalgebra and product. Intuitively, it allows one to 'factor out' complexity in models of algebraic theories. The HSP theorem states that classes of algebras satisfying some equations must be closed under the HSP operations, and also the converse, namely, classes of algebras satisfying the HSP operations must be equational. Proving that nominal algebra satisfies the HSP theorem, or a variant thereof, will therefore adequately demonstrate that nominal algebra is in fact an algebra, or equational logic.

Gabbay showed that a suitably strengthened form of Birkhoff's theorem, called HSPA, holds for nominal algebra [Gab09], and therefore we are justified in calling nominal algebra a logic of equality. HSPA extends 'homomorphism, subalgebra, product' with 'atoms abstraction', taking into account the ability to abstract atoms in nominal algebra.

Kurtz and Petrişan have also studied the HSP theorem for nominal algebra [KP09, KPV10, KP10], using a different proof technique. Intuitively, Kurtz and Petrişan have discovered a way of reducing HSPA in nominal algebra and nominal equational logic to HSP in first-order logic, through a 'compilation' process. Work on this technique is ongoing.

Permissive nominal terms (Chapter 4) are a form of nominal term with an alternative notion of freshness, relative to nominal terms. Permissive nominal algebra shares the same relationship with permissive nominal terms as nominal algebra shares with nominal terms: it is a simple logic of equality over permissive nominal terms (Chapter 4) [GM09e].

Permissive nominal algebra and λ-algebras, logics of equality over λ-terms, are closely related [GM09e]. Gabbay and Mulligan provided a non-trivial translation between a permissive nominal algebra theory ULAME and λ-theories, and proved it sound and complete in a suitable sense.

Nominal equational logic [CP07] is another 'nominal' equational logic. The equational fragment of both nominal equational logic and nominal algebra are sound and complete with respect to their models in nominal sets. With respect to the freshness fragment of both algebras, Clouston and Pitts claim that [CP07, pg. 32]

> ... nominal algebra does not provide a complete axiomatisation of the semantic notion of freshness within nominal sets.

This is in some sense true, but also misleading, as freshness within nominal sets can be characterised in purely equational terms, and one may write down equational axioms in nominal algebra that capture this semantic notion of freshness. Since both logics are sound and complete for equality in nominal sets, they are equivalent in expressive power.

The difference in the treatment of freshness affects the computational properties of the two algebras. Freshness in nominal algebra is usually handled with the derivable freshness relation defined on nominal terms, whereas freshness in nominal equational logic is taken as the underlying 'semantic freshness', inherited from nominal sets. Freshness in nominal algebra is therefore decidable, whereas freshness in nominal equational logic is not.

2.3.4 Nominal type systems

When implementing a type system for a nominal calculus what should the type of atoms be? Should atoms inhabit only their own sort, or collection of sorts, or should atoms inhabit every type?

The first rudimentary typing (sorting) system for nominal terms was introduced by Urban et al [UPG04]. This was used to provide a 'nominal signature', where every term-former in scope is ascribed a fixed sort. Here, there are a collection of atom sorts, and every atom is assumed to belong to one of these sorts. The correctness of the derivable α-equivalence relation for nominal terms was proved with respect to a particular signature, i.e. if $\Delta \vdash r \approx s$ then r and s have the same sort.

A more advanced type system with atom sorts was Schöpp and Stark's dependent type theory with names and binders [SS04, Sch06]. Introduced with the aim of providing 'a dependent type theory for programming and reasoning with such names', the type theory is able to capture many idioms familiar from FreshML. For instance, FreshML introduced a 'new' operator which introduced an arbitrarily chosen fresh name into a local scope [PG00]. Schöpp and Stark's type theory also can express this, but the fact that the new name is chosen completely fresh is also captured by the typing system.

Fernández and Gabbay introduced a Curry-style rank 1 polymorphic types to nominal terms, where atoms could inhabit any type [FG07a]. Care was taken defining permutations, which if left unrestrained may change the type of a term when acting upon it. Further, the presence of unknowns, and their capturing substitution action, also complicated the design of the typing system, as terms could be moved from the global typing context to a local context under a binder. Solving this problem required a notion of consistency for typing contexts. In the same paper, Fernández and Gabbay introduced a notion of typed rewriting over nominal terms, and proved subject reduction.

Gabbay and Mulligan defined a typing system for a subset of two-level λ-terms where types corresponded to first-order predicate logic [GM09b]. Their aim was to define a notion of proof term, under the Curry-Howard correspondence, for incomplete derivations in first-order logic. Typed nominal unknowns represent incomplete derivations—parts of the derivation tree that are yet to be specified. This mode of incomplete derivation is often found in the proof states of proof assistants like Isabelle or Matita. Like the typing system of Fernández and Gabbay, atoms could populate any type. Soundness and completeness for closed terms, with respect to Natural Deduction, are proved.

Pitts introduced a nominal version of Gödel's System T [Pit10], which introduces a new recursion principle for inductive data with bound names modulo α-equivalence. Pitts demonstrated the adequacy of his system using a novel normalization-by-evaluation argument, making use of a notion of local names in Gabbay-Pitts nominal sets, introduced by Fernández and Gabbay [FG05]. Atoms are assumed to belong to a sort of atoms.

Cheney introduced Simple Nominal Type Theory [Che08]. This is an extension of the simply-typed λ-calculus with facilities for abstracting an atom in a term, with the aim of providing a foundation for developing more complex type theories with name binding, along the lines of Schöpp and Stark. Confluence and strong normalisation hold for the system, and atoms are assumed to belong to a sort of atoms.

2.3.5 Nominal rewriting

Term rewriting systems are a generic formalism for capturing the dynamic behaviour of various formal systems. However, first-order rewrite systems (i.e. rewriting on first-order terms) struggle to capture rewriting on abstract syntax trees with binding. For instance, capturing the β-reduction rule from λ-calculus involves freshness side-conditions, working up to α-equivalence, and the definition of capture-avoiding substitution which also requires more freshness side-conditions. Various prior attempts had been made to define rewriting systems that capture these notions effectively, therefore Gabbay and Fernández introduced a 'nominal system'.

These problems motivated investigation into nominal rewriting [FG07b, FGM04], that is, term rewrite systems defined over nominal terms. Nominal rewriting incorporates freshness side-conditions and α-equivalence, without sacrificing the 'nameful' syntax (i.e. resorting to de Bruijn indices, or some other nameless approach). First-order matching is replaced by nominal matching.

Nominal terms were later imbued with a Curry-style polymorphic typing scheme [FG07a]. Nominal rewriting was extended to typed nominal terms, and subject reduction was proved. As a result, typed calculi, like the simply-typed λ-calculus, can just as easily be captured in nominal rewriting as their untyped counterparts.

Recent work by Fernández and Rubio has generalised the recursive path ordering [Nip98], a widely used measure for proving termination of a rewrite system, to nominal terms [FR10]. A general method for proving termination of nominal rewrite systems, and a completion procedure, follows.

Nominal rewrite systems have one small problem: selecting a suitable nominal rewrite rule to apply to a term is in general NP-complete. This inefficiency is for the same reasons that equivariant unification is inefficient; Cheney's theorem [Che04a, Theorem 1 and Corollary 3] explained the reason for the inefficiency. However, closed nominal rewriting rules are a restricted form of rewrite rule where matching rules with terms becomes much more efficient (polynomial) [FGM04, Theorem 4.9]. Intuitively, closed nominal rewrite rules are rules where no atom appears free (i.e. all atoms are bound by some abstraction).

Closed nominal rules are very expressive. All systems that arise naturally in functional programming (including the λ-calculus) are closed. In addition, nominal rewrite systems using closed rules are at least as expressive as higher-order rewrite systems. However, some 'natural' systems, like the π-calculus, fall outside of this restriction. Attempting to capture these systems in a nominal rewriting framework led to research on nominal rewrite systems with name generation [FG05].

There exists a close correspondence between closed nominal rewriting systems and derivable equality in nominal algebra [FG10]. Nominal rewriting is sound and complete for nominal algebra when all axioms are closed, but only complete for open axioms. If an equational theory can be represented by a confluent and normalising rewrite system, then the soundness and completeness result immediately implies a method for efficiently deciding the equality of two nominal terms with respect to this equational theory.

2.3.6 Languages and libraries

The need to write programs, such as compilers, interpreters and proof assistants, that manipulate abstract syntax with name binders was the original *raison d'être* of nominal techniques. Accordingly, a large amount of work was put into developing extensions of existing programming languages, and developing libraries for existing languages, that could handle this task.

αProlog αProlog [CU03, CU04, CU08, Che04b] is a logic programming system, similar to Prolog, whose term language has been enriched with facilities for name binding and expressing freshness side-conditions. αProlog has driven a lot of investigation into the semantics of nominal logic, on which it takes a large amount of inspiration, and nominal unification, which it uses heavily.

Cheney investigated the semantics of nominal logic programs [Che06b]. He provided a denotational semantics for αProlog programs, as well as proof theoretic and CLP semantics. Cheney also defined a notion of uniform proof, suitable for αProlog.

Cheney later went on to develop a system similar to Haskell's QuickCheck for αProlog [CM07].

αKanren αKanren [BF07] is an implementation of nominal logic programming in R5 Scheme. It is built atop the Kanren Scheme logic programming system.

αLeanTAP [NBF09], a theorem prover for first-order classical logic, is built atop αKanren.

Haskell Nominal Toolkit The Haskell Nominal Toolkit [Cal09] (HNT) is an implementation of nominal terms and their unification algorithm

FreshLib FreshLib is a Haskell library that provides nominal abstract syntax to Haskell programmers [Che05b]. Binding constructs are introduced, as well as monads for fresh names generation, and generic functions for swapping names. The library's aim is to reduce the amount of *nameplate* code that a programmer, wanting to encode object languages with binding, has to write.

FreshML and Fresh O'Caml FreshML [PG00, SPG03] and FreshO'Caml [Shi03, Shi05] are extensions of Standard ML and O'Caml, respectively, with support for binding, fresh name generation and writing functions that operate on α-equivalence classes. FreshO'Caml is the direct descendent of FreshML.

In early versions of FreshML [Gab00] the typing system maintained purity. However, the compile time typing checks for maintaining purity were expensive and conservative in that valid and useful programs could not be typed. Shinwell, in later versions of FreshML and FreshO'Caml, relaxed the typing constraints, at the expense of purity. However, Pottier was able to recover purity for a restricted subset of FreshML by definining a proof system that sits atop the FreshML typing system [Pot07]. In Pottier's approach, freshness constraints are reduced to SAT problems, and then solved. In another extension to the FreshML typing system, Pitts and Shinwell demonstrated that many other relations, such as total orderings, in addition to equality, can be placed on atoms, without disturbing the fundamental properties of the type system [PS08a]. Further, as a result of investigations into FreshML and FreshO'Caml, Pitts and Shinwell investigated nominal domain theory [SP05].

The designs of FreshML and FreshO'Caml were influential in the design of a calculus for name management by D'Ancona and Moggi [DM04], which uses FreshML style names.

MLSOS and αML MLSOS [LP08] and αML [Lak09] are metalanguages designed for animating structural operational semantics. αML is functional logic programming language, and is the direct descendent of the now defunct MLSOS.

Nominal calculi typically employ the *permutative convention*, popularised by Gabbay, where distinct names are assumed to range over distinct atoms [GM08a]. αML makes use of an alternative form of nominal abstract syntax—non-permutative nominal abstract syntax (NPNAS)—which shuns the permutative convention [LP10]. Instead, αML uses ordinary metavariables to range over atoms, where distinctly named metavariables may denote the same atom. Freshness side-conditions are encoded as constraints.

NPNAS was developed in response to the problems with incomplete proof search in 'nominal' programming languages such as αProlog, where equivariant unification, or syntactic restrictions on terms, are needed, as described in Subsection 2.3.5 and Subsection 2.3.2. In particular, the constraint language of αML and equivariant unification are both NP-complete, and therefore both polynomial-time reducible to each other. However, αML's proof search is shown to be complete for a simply yet powerful class of inductive definitions [LP09].

2.3.7 Nominal logics

LGN$^\omega$ LGN$^\omega$ is a logic which incorporates generic judgments and equivariant reasoning, as inspired by nominal techniques [Tiu08]. Names are represented by a predicate, and although there are infinitely many of them, only finitely many names can be mentioned by the judgments of the logic. As a result, judgments are finitely supported, and hence the validity of judgments is invariant under swapping of names.

'Nominal' logics arising from FM Sets What we would now call a 'nominal' logic was discovered in the form of FM Sets and used in the study of the independence of the axiom of choice from the other axioms of set theory [Fra22] (see Subsection 2.3.1). Thus, the first 'nominal' logic was FM set theory. It was applied to study *abstract syntax with binding* in [GP99, Gab00].

Nominal Logic [Pit03] is another first-order theory—a set of axioms. The difference from FM Sets is that Nominal Logic does not commit to a cumulative hierarchy of sets. Nominal Logic has influenced in particular αProlog and Nominal Isabelle.

One striking feature of nominal style reasoning is the NEW quantifier И. This quantifier expresses 'true for all but finitely many names'. This is not a new idea; for instance Krivine used it (interestingly, also for reasoning on syntax) in [KC93]. What makes NEW different, in the context of nominal techniques, is its interaction with equivariance [Gab00]. This gives NEW properties of both a universal and existential quantifier; its characteristic some/any property as discussed for example in [GP99, pg. 5].

This raises the question: 'What is a good proof-theory for NEW?'. Several proof theories have been suggested. Gabbay introduced the first sequent calculus style presentation of NEW [Gab07a]. Gabbay and Cheney later developed another sequent calculus for NEW [GC04]. This was later simplified by Cheney [Che05e]. A very different treatment of NEW is in [DG10, fig. 2, pg. 4].

Cheney showed completeness for Nominal Logic and developed Herbrand models for it [Che06a]. Staton used ideas from nominal sets and Nominal Logic to investigate name passing calculi [Sta06], and worked with Fiore in the same area [FS09]. Miculan showed how to translated specifications in Nominal Logic into the Calculus of Inductive Constructions, using the Theory of Contexts [MSH05]. Yasmeen implemented Mobile Ambients in Nominal Logic [YG08].

Schöpp investigated encoding generic judgments and provided a semantics for Miller and Tiu's ∇ quantifier. A translation between Nominal Logic and Miller and Tiu's logic, was recovered [Sch07].

One-and-a-halfth order logic One-and-a-halfth order logic is a logic similar to first-order logic, but with predicate unknowns in a nominal style [GM07d]. The purpose of the logic is to capture common metalevel statements, as often used when writing down axioms for logics and calculi, mentioning freshness conditions acting on meta-variables.

Cut elimination is proved for the logic, as well as the correctness of an interpretation of first-order logic within it.

Spatial logic Caires and Cardelli introduced Spatial Logic [CC02], a modal logic for describing the spatial behaviour of distributed systems. The logic includes primitives for composition, as well as the nominal NEW quantifier, for name hiding.

2.3.8 Use in proof assistants

Specialised support for object languages with binders is needed in proof assistants in order to simplify working with and using these languages, and also to ensure that non-theorems are not proved [Urb08]. Nominal techniques have been extremely influential in the design and implementation of such packages.

Agda Pouillard and Pottier [PP10] introduced a library consisting of a number of types and definitions for working with abstract syntax with binding in the dependently typed proof assistant Agda [Nor09]. The library uses the notion of a 'world' in order to control the use of atoms. If worlds are taken to be integers, then a de Bruijn index style of programming is obtained. If worlds are taken to be atoms, then a nominal style of programming is obtained. That is, the true nature of name is kept completely abstract.

The dependent typing of Agda is used to maintain several invariants. These invariants are used to ensure the following three conditions are never violated: name abstraction cannot be violated, names do not escape their scope, and names with different scopes cannot be mixed.

Coq and HOL Aydemir et al developed a nominal module in the Coq proof assistant and proved it correct with respect to an encoding of the untyped λ-calculus [ABW07]. Aydemir's development demonstrated that nominal techniques are implementable in a dependently typed system, such as Coq, but require axiomatisation, which makes some operations inconvenient (for example, the axiomatisation of key concepts means unfolding is impossible [ABW07, pg. 6]).

Norrish mechanized the metatheory of the λ-calculus using a first-order representation in the HOL proof assistant, using techniques borrowed from Nominal Isabelle (such as permutations) [Nor06].

Isabelle The first attempt at providing proof assistant support for languages with binding, based on nominal techniques, was by Gabbay [Gab02a, Gab02b]. Gabbay attempted to 'retarget' Isabelle/ZF as Isabelle/ZFA. On a much larger scale, Urban subsequently developed Nominal Isabelle [UN05, UT05, UNB07], an extension of Isabelle/HOL. Since then, a large amount of work on Nominal Isabelle has been carried out, attempting to automate as much as possible: the automatic derivation of inversion principles [BU08], Barendregt style induction rules for nominal datatypes [UBN07]) and attempting to make working with the Nominal Isabelle package as straightforward as possible (such as developing recursion operators for easy encodings of recursive functions over α-equivalence classes [UB06]).

A number of large theorems and algorithms have been verified using Nominal Isabelle. These include a verification of the W principal typing algorithm for Hindley-Milner style type systems [NU09], the verification of some typical structural operational semantics proofs [UN09], the formalisation of a key lemma from Urban's PhD thesis on cut elimination, demonstrating that the proof is correct [UZ08], a mechanization of a proof of Craig's interpolation theorem [CMU08], a formalisation of Crary's proof of completeness for equivalence checking [NU08], an implementation of the Spi calculus [KM08], various studies of the π-calculus [BP07, BP09a], an implementation of the ψ-calculus (an extension of the π-calculus) [BP09b], a formalisation of Intuitionistic Linear Logic, and a study of proof planning of dialogue [DST09], the study of a small functional programming language with references (a la Standard ML) [BKBH07], and a mechanization of the metatheory of Edinburgh LF [UCB08].

The formalisation of Crary's completeness proof resulted in several changes from the pen-and-paper proof. Similarly, the proof of correctness for Urban's key lemma resulted in several errors in supporting lemmas from his PhD thesis being discovered [UZ08, pg. 3].

Norrish demonstrated that de Bruijn terms really are isomorphic to λ-terms in Isabelle using techniques (permutations, and equivariance) inspired by nominal work [NV07].

2.3.9 Language semantics

Abramsky et al provided the first fully abstract model for the ν-calculus, a calculus with fresh name generation, using nominal games—games constructed within nominal sets [AGM$^+$04]. Nominal games were also employed by Tzvelekos to provide a fully abstract model for the ν-calculus with integer references [Tze07].

Turner and Winskel used nominal domain theory to provide semantics for higher-order concurrent calculi, proving adequacy and soundness results [TW09].

Finally, Mousavi et al made some brief investigations into nominal operation semantics [MGR06].

CHAPTER 3

The two-level λ-calculus

Abstract

The two-level λ-calculus is an extension of nominal terms where both atoms and nominal unknowns may be λ-bound, with requisite notions of β-reduction defined for both 'levels' of the calculus.

In particular, the two-level λ-calculus is a novel context calculus. A context may be though of as a λ-term with 'some holes', and context-calculi promote contexts to first class entities. Context-calculi have two levels of variable: variables of level two modeling holes, and variables of level one which can be bound in holes. Notably, 'filling' a hole in a context calculus is not capture avoiding for λ-bound variables of level one.

Context-calculi have been used as a research tool for many purposes, including investigating dynamic binding, module systems, and novel programming languages, amongst other uses. Context-calculi may also be used to formalise the informal notion of context as a 'term with some holes' familiar from work on contextual equivalence of program fragments.

Intuitively, the two-level λ-calculus behaves like a pair of λ-calculi wrapped around each other; for instance, substitution within a level is capture avoiding in the usual sense. However, the interaction between levels is interesting, as substitution of unknowns *does not* avoid capture of λ-bound atoms, but substitution of atoms *does* avoid capture of λ-bound unknowns.

Like all context-calculi, the two-level λ-calculus must overcome two problems: how to α-convert bound variables in the presence of 'holes', and how to avoid a well-known failure of commutation between β-reduction and hole filling. In handling these problems, the 'nominal influence' on the calculus shines through, both explicitly (in our handling of α-equivalence) and implicitly (in our handling of β-reduction).

We handle α-conversion through swappings and explicit freshness side-conditions. Swappings are particularly associated with nominal techniques, and possess useful properties: they are self-inverse and all relations defined on two-level λ-terms are equivariant.

We handle the failure of commutation between hole filling and β-reduction by prohibiting the reduction of a term like $(\lambda a.X)t$ to X, unless a freshness side-condition states that it is safe to do so. This is the implicit nominal influence: atoms and unknowns are different types of variable, and should absolutely not be conflated with each other.

The two-level λ-calculus is confluent. Proving this is non-trivial. We split the confluence proof in two: we prove the confluence of the level one and level two fragments separately, before finally stitching the two proofs together to obtain confluence for the whole calculus. The proof of confluence for the level one fragment uses a novel proof technique which appears to be quite general.

The Chapter concludes with a survey of prior art.

3.1 Introduction

This Chapter introduces a novel *context-calculus*—the result of extending nominal terms with λ-binders for atoms and nominal unknowns—in the spirit of [SSKI03, GL08, HO01], called the two-level λ-calculus. Intuitively, a context is a λ-term with some 'holes'.

The two-level λ-calculus has, as the name suggests, two levels of variable. We use atoms and unknowns, familiar from nominal terms [UPG04], to model the variables of the respective calculi. Atoms, a and b in the example reductions above, model *level one* variables. Unknowns, X and Y in the example reductions above, model *level two* variables. Both levels of variable may be freely λ-abstracted and the requisite notions of reduction exist at both levels. In effect, the two-level λ-calculus is an intertwining of two copies of the ordinary, untyped λ-calculus, and the essence of the two-level λ-calculus is best revealed through a series of example reductions:

$$\Delta \vdash (\lambda X.(\lambda a.X))a \to \lambda a.a \quad \Delta \vdash \lambda X.(\lambda Y.X)Y \to \lambda X'.Y \quad \Delta \vdash \lambda a.(\lambda b.a)b \to \lambda b'.b$$

In the leftmost reduction, substitution of unknowns is *not* capture avoiding for λ-bound atoms. The remaining two reductions, centre and rightmost, reinforce the intuition that the two-level λ-calculus is 'just' two copies of the untyped λ-calculus intertwined: substitution within a level *is* capture-avoiding. Note, however, that the following is *not* a valid reduction in the two-level λ-calculus:

$$\Delta \vdash \lambda a.(\lambda X.a)X \not\to \lambda X.X$$

That is, the levels of variable induce a kind of 'power structure', where, compared to level one variables, level two variables are strictly more 'powerful'. Level two variables may ignore the bindings of level one variables, but the reverse does not hold.

The reader will no doubt have noticed the numerous occurrences of Δ in the example reductions above. We use Δ to range over *freshness contexts*. For the purposes of this introduction, these can be safely ignored, but are important cogs in the machinery of the two-level λ-calculus (see Section 3.3 for a full explanation).

In principle, it is possible to extend the two-level λ-calculus, to three, four, an indeed, an infinity of levels (presumably obtaining the ω-level λ-calculus in the process). A few other context-calculi have employed an infinity of levels [GL08, SSKI03]. However, we make the conservative choice of sticking to just two levels, as this suffices for studying the interesting interactions between levels of the calculus, without additional notational overload.

So far, we have focussed our attention on *what* we have done, as opposed to *why* we have done it. We produced another context-calculus because context-calculi are incredibly useful, not just as a means of formalising the informal concept of a 'term with holes', familiar from work on e.g. contextual equivalence of program fragments as exemplified by Pitts [Pit97], but also because we believe that context-calculi have an independent mathematical interest. Further, we thought nominal techniques could bring some new ideas to the research area, in the handling of α-equivalence in the presence of two levels of variable.

The general idea of a context-calculus has existed since at least the 1970s. De Bruijn investigated a limited type of context, called a segment, as part of the Automath project [dB78]. Here, de Bruijn was interested in providing definitional mechanisms, and the ability to create shorthands, for mathematical expressions. Section 3.5 studies de Bruijn's segments in more detail.

Later investigation revealed that contexts, or 'open terms', appear to be quite common in everyday mathematical parlance. For instance, note:

$$\forall x.(\phi \to \psi) = \phi \to \forall x.\psi \quad (\text{if } x \notin fv(\phi))$$

Here, ϕ and ψ are both 'holes', which can be plugged with any formula, even under a binder (in this case, $\forall$). Examples of this sort are readily apparent when studying the metatheory of formal calculi, from programming languages to logics. Indeed, possibly the best way to spot a context 'in the wild' is to open a textbook on logic.

Another excellent source of open terms are proof assistants, such as Isabelle [Pau88]. The predominant mode of reasoning in Isabelle is backward reasoning: to prove a conjecture, we work backwards, splitting our conjecture into subgoals (and therein recursively splitting subgoals into hopefully simpler subgoals), until we reach a state where all open subgoals can be solved directly. Now, consider the (**ExI**) rule, familiar from Natural Deduction:

$$\frac{\Gamma \vdash P(t)}{\Gamma \vdash \exists x.P(x)}\,(\mathbf{ExI})$$

Read backwards, this rule states that to prove a goal of the form $\Gamma \vdash \exists x.P(x)$ we must exhibit some t such that $\Gamma \vdash P(t)$ holds. From the viewpoint of practical proof construction, this is rather awkward. Ideally, we'd really like to build t incrementally, as opposed to 'pulling a rabbit from a hat', so to speak, and producing t on the spot. Isabelle handled this by introducing Skolemised metavariables, i.e. by working with open formulae, allowing the user to introduce a metavariable in lieu of a concrete choice of t [Pau98]. This metavariable may later be instantiated, but this instantiation is nothing more than the hole filling operation in a context-calculus.

A similar phenomenon occurs in type theory based proof assistants such as Coq, Lego or Matita which depend heavily on the Curry-Howard correspondence. Here, the types of some typed-λ calculus (typically the Calculus of Constructions, or some extension thereof) correspond to formulae in some logic (in the case of the Calculus of Constructions, higher-order logic). Theorems are proved in the system by attempting to inhabit a type with a proof term. Again, for purposes of practical proof construction, this proof term is ideally built incrementally. In order to facilitate this, proof assistants enrich their typed λ-calculi with metavariables, standing for unknown, or yet to be completed sections of a proof.

For example, suppose we work in an imaginary type theory based proof assistant based on the simply typed λ-calculus enriched with metavariables (corresponding, under Curry-Howard, to propositional logic). To prove $\phi \to \phi$ we introduce a typed metavariable $\vdash X : \phi \to \phi$. Incrementally constructing a proof term to inhabit this type, we introduce a fresh variable of type ϕ into our context $a : \phi \vdash X : \phi \to \phi$ before λ-abstracting over a fresh metavariable of type ϕ, as follows $a : \phi \vdash \lambda a : \phi.X' : \phi \to \phi$. Finally, completing the construction of the proof term, we instantiate X' to a, discharging a, to obtain $\vdash \lambda a : \phi.a : \phi \to \phi$. This term type-checks with the correct type, confirming that our conjecture is indeed a theorem of propositional logic.

Above, we constructed a proof incrementally by slowly building a witnessing λ-term to inhabit a type. In proof assistants, these construction steps are handled by 'tactics', or small programs that exist at the metalevel, written in a programming language independent from the logical layer of the assistant itself. However, it's worth noting here that tactics are merely functions from open proof terms to open proof terms. If we accept that $\lambda a{:}\phi.X$ is an open proof term, then what is $\lambda X.\lambda a : \phi.X$? This also appears to be a function from open proof terms to open proof terms—or, a rudimentary tactic.

Here, then, we also obtain a further motivation for investigating context calculi. The metavariables of type theory based proof assistants are clearly the 'holes', or unknowns X of a context calculus. Abstracting over these holes allows us to write functions from open proof terms to open proof terms, or tactics. Therefore, a typed λ-calculus is itself a primitive type theory based proof assistant, where tactics can be written by the user at level two, and level one corresponds to the logical level of the assistant.

Further motivation for context calculi also exist, outwith proof assistants. Contexts also arise naturally when trying to formalise what it means for two expressions, in some given programming language, to be equivalent. Intuitively, two expressions are *contextually equivalent* when, after plugging each expression into a complete program, the results of executing that program are identical. Again, this process of 'plugging in' an expression is nothing more than the hole filling operation in a context-calculus.

The previous series of examples are evidence that contexts and open terms are in fact surprisingly common. However, the role of contexts, and how contexts can be manipulated, is usually glossed over. Sato [SSKI03] and Gabbay [GP99], amongst others, both investigated calculi of contexts to provide a firm foundational footing for working with open terms.

Context-calculi also appear to possess some independent mathematical interest, and we see an allegory with the history of the λ-calculus along these lines. Despite being initially proposed as part of a foundational framework for mathematics, the λ-calculus has positioned itself as the *de facto* syntax for expressing functions in computer science, and as a fundamental tool for investigating programming language denotations. It therefore seems natural to ask what additional uses we may have for context-calculi. In fact, a large amount of work has applied context-calculi, and other calculi with open terms, to various problems, including investigating dynamic binding [Dam98], module systems [LF96], novel programming languages [Has98] and proof terms for incomplete derivations [Muñ97, GJ02, Joj03, GM09b].

Hopefully, we have now given the reader an idea of why context-calculi are of interest. However, many context-calculi already exist: why investigate another, and what do we see as the novel features of the two-level λ-calculus?

Any context-calculus design must overcome two well-known problems: how to perform α-conversion on contexts, and how to avoid the problems commuting naïve β-reduction and hole filling (for instance, these problems are discussed in [SSKI03]). Consider the two contexts $\Delta \vdash \lambda a.X$ and $\Delta \vdash \lambda b.X$. We simply cannot state that $\Delta \vdash \lambda a.X$ and $\Delta \vdash \lambda b.X$ are α-equivalent. To see why, consider filling X with a, which leads to two α-inequivalent terms $\Delta \vdash \lambda a.a$ and $\Delta \vdash \lambda b.a$.

One possible solution to this problem is to work with 'nameless' terms using de Bruijn's encodings, for instance. We view de Bruijn encodings as being potentially hard to read, and for this reason, we prefer a 'nameful' syntax. Another possible solution is to use a calculus of explicit substitutions, in the spirit of [Ste00] or [Muñ97]. Here, $\Delta \vdash \lambda a.X$ can be α-converted to $\Delta \vdash \lambda b.(X[a{\mapsto}b])$, with the substitution $[a{\mapsto}b]$ *suspending* on the hole X, waiting to act on whatever X is filled with. However, the design of calculi of explicit substitutions is a subtle affair, and hard to perfect.

Instead of explicit substitutions, we use the much weaker notion of *swappings* on atoms. Now, $\Delta \vdash \lambda a.X$ is considered α-equivalent to $\Delta \vdash \lambda b.((b\ a){\cdot}X)$, where $(b\ a)$ is a *swapping*. Swappings map from b to a to b and fix all other c. Swappings, and permutations in general, are the 'poster child' of nominal techniques[1], and possess some useful properties: they are self-inverse, commute nicely with each other, and all relations that we define on two-level λ-terms are pleasantly *equivariant* (invariant under swappings). Leveraging nominal techniques also opens up the possibility of using nominal sets for a concrete denotation (see Section 3.5 for ideas along these lines).

The other main problem that any context-calculus must overcome is that naïve β-reduction and hole filling do not commute. To see this consider naïvely β-reducing $\Delta \vdash (\lambda a.X)t \to X$ and filling X with a to obtain $\Delta \vdash (\lambda a.a)t$. Reducing we obtain $\Delta \vdash (\lambda a.a)t \to t$. However, $t \not\equiv a$, and therefore there is a failure of commutativity and ultimately, a failure of confluence.

We address this problem within the two-level λ-calculus is by noting that reducing $(\lambda a.X)t$ to X does not make any sense, because a and X are different kinds of variable. At any point, X may be filled, and the behaviour of the final, closed term, under reduction will depend on what X is filled with. We therefore prevent $\Delta \vdash (\lambda a.X)t$ from reducing further (it is 'stuck') until it is filled with another term. However, we make one exception: if we explicitly hypothesise within our freshness context that a is 'fresh for' X, we permit the 'safe' reduction $\Delta \vdash (\lambda a.X)t \to X$. Following our reduction scheme, we note that open terms may still be reduced, if there is opportunity for doing so. For example, using the rules in Definition 3.4.3, it is still the case that $\Delta \vdash ((\lambda a.X)t)((\lambda a.a)b) \to ((\lambda a.X)t)b$, regardless if a is fresh for X or not.

We now provide a high-level overview of the Chapter. Section 3.2 introduces the syntax of the calculus. Subsection 3.2.1 introduces a permutation and instantiation action on terms. Permutations are used to handle bound variable renamings in the presence of holes, and instantiations are the mechanism through which said holes are filled. Section 3.3 introduces a theory of derivable freshness, a generalisation of the notion of the free atoms of a term—a generalisation necessary when working with open terms (open terms essentially have an infinite set of free atoms).

Section 3.4 introduces a reduction relation for the calculus, and proves it confluent. The proof of confluence is split into two; one proof of confluence for each level. Subsection 3.4.1 handles the proof of confluence for level one reductions, Subsection 3.4.2 handles the proof of confluence for level two reductions. The two halves of the full confluence proof are finally 'stitched together' in Subsection 3.4.3, which includes the main theorem of the Chapter, Theorem 3.4.42, overall confluence of the system.

Finally, Section 3.5 concludes the Chapter. We include a survey of related work in Subsection 3.5.1.

[1]By this, we mean that permutations have come to be associated closely with the nominal approach. For instance, see [Tiu08], wherein Tiu describes his approach as in a 'nominal-style', due to his use of permutations on names and equivariance.

The work presented within this Chapter is an expanded version of a workshop paper [GM09d], authored with Murdoch J. Gabbay. A journal version of this Chapter is currently under review [GM10c]. The initial idea for the work was Gabbay's. The design of the term language was made jointly with Gabbay, based on a modified initial suggestion by Gabbay. The mathematics was carried out by myself, with supervision and advice from Gabbay.

3.2 Terms

Fix a countably infinite set of **atoms** $\mathbb{A}$. We use atom to model **variables of level one**. We use a, b, c, and so on, to range over atoms (variables of level one), and employ a permutative convention (that is, a and b denote *distinct* atoms).

Fix a countably infinite set of **unknowns**. We use unknowns to model **variables of level two**. We use X, Y, Z to range over variables of level two. We also employ a permutative convention with unknowns (variables of level two), so X and Y always denote distinct unknowns.

Fix a countably infinite set of **constant symbols**. We used $\mathsf{c}, \mathsf{d}, \mathsf{e}$, and so on, to range over constant symbols permutatively.

Definition 3.2.1: Suppose f is a function from atoms to atoms. Define:

$$\mathit{nontriv}(f) = \{a \mid f(a) \neq a\}$$

Call a bijection π on variables of level one a **permutation** whenever $\mathit{nontriv}(\pi)$ is finite. Requiring finite $\mathit{nontriv}(\pi)$ is the familiar notion of 'finite support' from nominal techniques recast. We use π, π', π'', and so on, to range over permutations.

We write π^{-1} for the **inverse** of a permutation, π. We write $\pi \circ \pi'$ for the functional **composition** of two permutations, so $(\pi \circ \pi')(a) = \pi(\pi'(a))$. Further, we write id for the **identity** permutation, so that $id(a) = a$ for all variables of level one. Note, in particular, that $(\pi^{-1} \circ \pi) = id = (\pi \circ \pi^{-1})$.

Permutations of particular importance are **swappings**. We write $(b\ a)$ for the permutation mapping b to a to b, and fixing all other c. All swappings are self-inverse.

Definition 3.2.2: Define **terms** of the two-level λ-calculus by:

$$r, s, t ::= a \mid \mathsf{c} \mid \pi \cdot X \mid \lambda a.r \mid \lambda X.r \mid rs$$

We equate terms up to α-equivalence of λX bound variables, but *do not* do this for λa bound variables. We write $\equiv$ for syntactic equivalence up to λX α-conversion. As a corollary, we write $\not\equiv$ for syntactic inequivalence. For example:

$$\lambda X.X \equiv \lambda Y.Y \qquad \lambda X.(\lambda a.X) \equiv \lambda Y.(\lambda a.Y) \qquad \lambda a.a \not\equiv \lambda b.b \qquad \lambda a.(\lambda X.a) \not\equiv \lambda b.(\lambda Y.b)$$

We write $r[a \mapsto t]$ as a shorthand for $(\lambda a.r)t$. Note that X *isn't* a term, but an unknown, but $id \cdot X$ is. For typographical convenience, however, we will often abbreviate $id \cdot X$ as simply X.

In '$\pi \cdot X$', we say that π is **suspended** on X. Intuitively, π waits to act (in the sense of Definition 3.2.4) on whatever X is instantiated with. Suspended permutations may be thought of as a special case of suspended substitutions, familiar from calculi of explicit substitutions. For instance, the following term, with an explicit substitution $\lambda x.(r[y \mapsto x])$, roughly corresponds to $\lambda a.((b\ a)X)$ in our syntax.

$$\pi{\cdot}a \equiv \pi(a) \quad \pi{\cdot}(\pi'{\cdot}X) \equiv (\pi{\circ}\pi'){\cdot}X \quad \pi{\cdot}\mathsf{c} \equiv \mathsf{c} \quad \pi{\cdot}rs \equiv (\pi{\cdot}r)(\pi{\cdot}s)$$
$$\pi{\cdot}\lambda a.r \equiv \lambda\pi(a).(\pi{\cdot}r) \quad \pi{\cdot}\lambda X.r \equiv \lambda X.(\pi{\cdot}r[X{:=}\pi^{-1}{\cdot}X])$$

$$a[X{:=}r] \equiv a \quad \mathsf{c}[X{:=}r] \equiv \mathsf{c} \quad (\pi{\cdot}X)[X{:=}r] \equiv \pi{\cdot}r \quad (\pi{\cdot}Y)[X{:=}r] \equiv \pi{\cdot}Y$$
$$(st)[X{:=}r] \equiv (s[X{:=}r])(t[X{:=}r]) \quad (\lambda b.s)[X{:=}r] \equiv \lambda b.(s[X{:=}r])$$
$$(\lambda X.r)[X{:=}r] \equiv \lambda X.r$$
$$(\lambda Y.s)[X{:=}r] \equiv \lambda Y.(s[X{:=}r]) \quad (\text{if } Y \notin fV(r))$$

Figure 3.1 Level one permutation and level two instantiation actions

3.2.1 Permutation and instantiation actions

Permutations and capturing substitutions for unknowns (instantiations) are hallmarks of nominal terms. Permutations are used to handle α-equivalence in nominal calculi in an elegant manner, as in e.g. nominal algebra [Mat07, Definition 2.3.1]. We can model capture-avoiding substitutions with instantiations by carefully managing freshness assumptions.

This Subsection introduces the permutation and instantiation actions, as well as some useful results for manipulating permutations and instantiations. Definition 3.2.4 introduces the permutation and instantiation actions. Definition 3.2.7 introduces a notion of 'depth' for a term, useful in later inductive proofs.

Lemmas 3.2.13 and 3.2.16 are important results for manipulating permutations and instantiations, will be used extensively throughout the rest of the Chapter, and may be thought of as the most important results in this Subsection. Lemma 3.2.13 states that two permutations acting on a term in sequence are equivalent to the composition of those two permutations acting on the term. Lemma 3.2.16 states that the permutation and instantiation actions commute.

Definition 3.2.3: Define the **free variables of level two** of a term by:

$$fV(a) = \emptyset \qquad fV(\mathsf{c}) = \emptyset \qquad fV(\pi{\cdot}X) = X \qquad fV(rs) = fV(r) \cup fV(s)$$
$$fV(\lambda a.r) = fV(r) \quad fV(\lambda X.r) = fV(r) \setminus \{X\}$$

Definition 3.2.4: Define level one **permutation** and level two **instantiation** actions by the rules in Figure 3.1. The side-condition $Y \notin fV(r)$ for the $\lambda Y.s$ case in the definition of the instantiation action can always be guaranteed, simply by renaming.

The instantiation action $r[X{:=}t]$ pushes an instantiation as far inside a term as possible, acting on unknowns, or evaporating on atoms and constants. Note that instantiation is capture-avoiding for λ-bound unknowns, but is *not* capture-avoiding for λ-bound atoms. Similarly, the permutation action attempts to push a permutation as far as possible inside a term, acting on atoms or suspending on unknowns. The permutation action has one oddity—the case of $\lambda X.r$—which requires a partial intertwining of the instantiation and permutation actions. This coupling of the two action is explained below.

We note that there is a distinction between $\pi{\cdot}a$ and $\pi(a)$. The latter is an atom, the image of a under π, and itself a term of the calculus. In contrast, the permutation action on terms $\pi{\cdot}r$

is a 'recipe' for how to make a term, by pushing a permutation as far as possible inside it, from r.

Further, we note that the permutation and instantiation actions are intertwined, defined mutually, unlike previous definitions of permutation and instantiation actions in other nominal calculi. This is due to the presence of λX. To see why this is so, consider the following series of equivalences, easily verified using the rules in Figure 3.1:

$$\pi{\cdot}\lambda X.X \equiv \lambda X.(\pi{\cdot}X[X{:=}\pi^{-1}{\cdot}X]) \equiv \lambda X.(\pi{\cdot}(\pi^{-1}{\cdot}X)) \equiv \lambda X.((\pi\circ\pi^{-1}){\cdot}X) \equiv \lambda X.X$$

Permutations acting on $\lambda X.r$ distribute into the body of the term. Doing this incorrectly may 'break' the bindings of bound occurrences of X. To counteract this effect, the action of the permutation on bound occurences of X must be fixed with a corrective instantiation. Note that:

$$\pi{\cdot}\lambda X.(XY) \equiv \lambda X.((\pi{\cdot}X[X{:=}\pi^{-1}{\cdot}X])(\pi{\cdot}Y)) \equiv \lambda X.(X(\pi{\cdot}Y))$$

That is, only bound unknowns are affected by the actions of counteracting instantiations.

Note, however, that we could have made an alternative design decision, and allowed permutations π to distribute throughout $\lambda X.r$ in the following manner:

$$\pi{\cdot}\lambda X.r \equiv \lambda\pi{\cdot}X.(\pi{\cdot}r)$$

This could have been made to work, with slight alterations to the mathematics. However, we chose the alternative design for the simple reason that we think that suspending a permutation on a λ-abstracted unknown, as in $\lambda\pi{\cdot}X.r$, looks ugly[2].

Lemma 3.2.5: $fV(r) = fV(r[X{:=}\pi{\cdot}X])$

Proof. By induction on r. See Appendix A. ⊠

Lemma 3.2.6 is a standard equivariance result for the set of free variables of level one of a term.

Lemma 3.2.6: $fV(r) = fV(\pi{\cdot}r)$

Proof. By induction on r, using Lemma 3.2.5. See Appendix A. ⊠

Definition 3.2.7 will be used in later inductive proofs. See, for instance, the proof of Lemma 3.2.13. Lemma 3.2.8 and Lemma 3.2.9 are invariance and equivariance results for the depth of a term.

Definition 3.2.7: Define a notion of **depth** on a term by:

$$\mathit{depth}(a) = 1 \quad \mathit{depth}(\mathsf{c}) = 1 \quad \mathit{depth}(\pi{\cdot}X) = 1 \quad \mathit{depth}(rs) = \mathit{depth}(r) + \mathit{depth}(s)$$
$$\mathit{depth}(\lambda a.r) = 1 + \mathit{depth}(r) \quad \mathit{depth}(\lambda X.r) = 1 + \mathit{depth}(r)$$

Lemma 3.2.8: $\mathit{depth}(r) = \mathit{depth}(r[X{:=}\pi{\cdot}Y])$

[2]More seriously, the instantiation $[X{:=}\pi^{-1}{\cdot}X]$ is reminiscent of the conjugation action on Nominal Sets [Gab00].

Proof. By induction on r. See Appendix A. ⊠

Lemma 3.2.9: $\mathit{depth}(r) = \mathit{depth}(\pi{\cdot}r)$

Proof. By induction on r, using Lemma 3.2.8. See Appendix A. ⊠

The following technical lemma is used in the proof of Lemma 3.2.13:

Lemma 3.2.10: $r[X{:=}\pi{\cdot}X][X{:=}\pi'{\cdot}X] \equiv r[X{:=}(\pi\circ\pi'){\cdot}X]$

Proof. By induction on r. See Appendix A. ⊠

Lemma 3.2.11 states that identity instantiations 'evaporate' on terms[3]. Similarly, Lemma 3.2.12 states that the identity permutation evaporates on terms.

Lemma 3.2.11: $r[X{:=}\mathit{id}{\cdot}X] \equiv r$.

Proof. By induction on r. See Appendix A. ⊠

Lemma 3.2.12: $\mathit{id}{\cdot}r \equiv r$

Proof. By induction on r, using Lemma 3.2.11. See Appendix A. ⊠

Lemma 3.2.13 will be used extensively throughout the rest of the Chapter. Permutations are used extensively for handling α-equivalence. Lemma 3.2.13 formalises the intuition that α-renaming a term twice is precisely equivalent to performing a single composite renaming.

Lemma 3.2.13: $\pi{\cdot}(\pi'{\cdot}r) \equiv (\pi\circ\pi'){\cdot}r$

Proof. By induction on $\mathit{depth}(r)$.

- The case a. By Definition 3.2.4 we have $\pi{\cdot}(\pi'{\cdot}a) \equiv \pi{\cdot}\pi'(a)$. By Definition 3.2.4 we have $\pi{\cdot}\pi'(a) \equiv \pi(\pi'(a))$. It is a fact that $\pi(\pi'(a)) \equiv (\pi\circ\pi')(a)$. By Definition 3.2.4 we have $(\pi\circ\pi')(a) \equiv (\pi\circ\pi'){\cdot}a$. The result follows.
- The case c. By Definition 3.2.4 we have $\pi{\cdot}(\pi'{\cdot}\mathsf{c}) \equiv \mathsf{c}$. By Definition 3.2.4 we have $(\pi\circ\pi'){\cdot}\mathsf{c} \equiv \mathsf{c}$. The result follows.
- The case $\pi''{\cdot}X$. By Definition 3.2.4 we have $\pi{\cdot}(\pi'{\cdot}(\pi''{\cdot}X)) \equiv \pi{\cdot}((\pi'\circ\pi''){\cdot}X)$. Definition 3.2.4 we have $\pi{\cdot}((\pi'\circ\pi''){\cdot}X) \equiv (\pi\circ(\pi'\circ\pi'')){\cdot}X$. It is a fact that $(\pi\circ(\pi'\circ\pi'')){\cdot}X \equiv ((\pi\circ\pi')\circ\pi''){\cdot}X$. By Definition 3.2.4 we have $((\pi\circ\pi')\circ\pi''){\cdot}X \equiv (\pi\circ\pi'){\cdot}(\pi''{\cdot}X)$. The result follows.
- The case rs. By Definition 3.2.4 we have $\pi{\cdot}(\pi'{\cdot}rs) \equiv \pi{\cdot}((\pi'{\cdot}r)(\pi'{\cdot}s))$. By Definition 3.2.4 we have $\pi{\cdot}((\pi'{\cdot}r)(\pi'{\cdot}s)) \equiv (\pi{\cdot}(\pi'{\cdot}r))(\pi{\cdot}(\pi'{\cdot}s))$. By inductive hypothesis $(\pi{\cdot}(\pi'{\cdot}r))(\pi{\cdot}(\pi'{\cdot}s)) \equiv ((\pi\circ\pi'){\cdot}r)((\pi\circ\pi'){\cdot}s)$. By Definition 3.2.4 we have $((\pi\circ\pi'){\cdot}r)((\pi\circ\pi'){\cdot}s) \equiv (\pi\circ\pi'){\cdot}rs$. The result follows.

[3]Of course, 'identity instantiations' are not unique; $[X{:=}\mathit{id}{\cdot}X]$ and $[Y{:=}\mathit{id}{\cdot}Y]$ both act like the identity.

- The case $\lambda a.r$. By Definition 3.2.4 we have $\pi{\cdot}(\pi'{\cdot}\lambda a.r) \equiv \pi{\cdot}(\lambda\pi'(a).(\pi'{\cdot}r))$. Definition 3.2.4 we have $\pi{\cdot}(\lambda\pi'(a).(\pi'{\cdot}r)) \equiv \lambda\pi(\pi'(a)).(\pi{\cdot}(\pi'{\cdot}r))$. By inductive hypothesis $\lambda\pi(\pi'(a)).(\pi{\cdot}(\pi'{\cdot}r)) \equiv \lambda\pi(\pi'(a)).((\pi{\circ}\pi'){\cdot}r)$. It is a fact that $\lambda\pi(\pi'(a)).((\pi{\circ}\pi'){\cdot}r) \equiv \lambda(\pi{\circ}\pi')(a).((\pi{\circ}\pi'){\cdot}r)$. By Definition 3.2.4 we have $\lambda(\pi{\circ}\pi')(a).((\pi{\circ}\pi'){\cdot}r) \equiv (\pi{\circ}\pi'){\cdot}\lambda a.r$. The result follows.
- The case $\lambda X.r$. First, we note $(\pi{\circ}\pi')^{-1} = \pi'^{-1}{\circ}\pi^{-1}$. By Definition 3.2.4 we have $\pi{\cdot}(\pi'{\cdot}\lambda X.r) \equiv \pi{\cdot}\lambda X.(\pi'{\cdot}r[X{:=}\pi'^{-1}{\cdot}X])$. By Definition 3.2.4 we have $\pi{\cdot}\lambda X.(\pi'{\cdot}r[X{:=}\pi'^{-1}{\cdot}X]) \equiv \lambda X.(\pi{\cdot}(\pi'{\cdot}r)[X{:=}\pi'^{-1}{\cdot}X][X{:=}\pi^{-1}{\cdot}X])$. By Lemma 3.2.10 we have $\lambda X.(\pi{\cdot}(\pi'{\cdot}r)[X{:=}\pi'^{-1}{\cdot}X][X{:=}\pi^{-1}{\cdot}X]) \equiv \lambda X.(\pi{\cdot}(\pi'{\cdot}r)[X{:=}(\pi'^{-1}{\circ}\pi^{-1}){\cdot}X])$. By inductive hypothesis and Lemma 3.2.8 we have $\lambda X.(\pi{\cdot}(\pi'{\cdot}r)[X{:=}(\pi'^{-1}{\circ}\pi^{-1}){\cdot}X]) \equiv \lambda X.((\pi{\circ}\pi'){\cdot}r[X{:=}(\pi'^{-1}{\circ}\pi^{-1}){\cdot}X])$. It is a fact that $\lambda X.((\pi{\circ}\pi'){\cdot}r[X{:=}(\pi'^{-1}{\circ}\pi^{-1}){\cdot}X]) \equiv \lambda X.((\pi{\circ}\pi'){\cdot}r[X{:=}(\pi{\circ}\pi')^{-1}{\cdot}X])$. By Definition 3.2.4 we have $\lambda X.((\pi{\circ}\pi'){\cdot}r[X{:=}(\pi{\circ}\pi')^{-1}{\cdot}X]) \equiv (\pi{\circ}\pi'){\cdot}\lambda X.r$. The result follows.

⊠

Lemma 3.2.14: If $Y \notin fV(r)$ then $r[Y{:=}s] \equiv r$.

Proof. By induction on r.

- The case a. By Definition 3.2.4 we have $a[X{:=}t] \equiv a$. The result follows.
- The case c. By Definition 3.2.4 we have $\mathsf{c}[X{:=}t] \equiv \mathsf{c}$. The result follows.
- The case $\pi{\cdot}Y$. By Definition 3.2.4 we have $(\pi{\cdot}Y)[X{:=}t] \equiv \pi{\cdot}Y$. The result follows.
- The case rs. Suppose $Z \notin fV(rs)$. By Definition 3.2.3 we have $Z \notin fV(r) \cup fV(s)$, therefore $Z \notin fV(r)$ and $Z \notin fV(s)$. By Definition 3.2.4 we have $(rs)[Z{:=}t] \equiv (r[Z{:=}t])(s[Z{:=}t])$. By inductive hypothesis $(r[Z{:=}t])(s[Z{:=}t]) \equiv rs$. The result follows.
- The case $\lambda a.r$. Suppose $Y \notin fV(\lambda a.r)$. By Definition 3.2.3 we have $X \notin fV(r)$. By Definition 3.2.4 we have $(\lambda a.r)[Y{:=}s] \equiv \lambda a.(r[Y{:=}s])$. By inductive hypothesis $\lambda a.(r[Y{:=}s]) \equiv \lambda a.r$. The result follows.
- The case $\lambda X.r$. By Definition 3.2.4 we have $(\lambda X.r)[X{:=}r] \equiv \lambda X.r$. The result follows.
- The case $\lambda Y.s$. Suppose $X \notin fV(\lambda Y.s)$, and $Y \notin fV(r)$, which can be guaranteed by renaming. By Definition 3.2.3 we have $X \notin fV(s) \setminus \{Y\}$, therefore $X \notin fV(s)$. By Definition 3.2.4 we have $(\lambda Y.s)[X{:=}r] \equiv \lambda Y.(s[X{:=}r])$. By inductive hypothesis $\lambda Y.(s[X{:=}r]) \equiv \lambda Y.s$. The result follows.

⊠

Lemma 3.2.15: If $Y \notin fV(t)$ then $r[Y{:=}s][Z{:=}t] \equiv r[Z{:=}t][Y{:=}s[Z{:=}t]]$.

Proof. By induction on r.

- The case a. By Definition 3.2.4 we have $a[Y{:=}s][X{:=}t] \equiv a$. By Definition 3.2.4 we have $a[Z{:=}t][Y{:=}s[Z{:=}t]] \equiv a$. The result follows.
- The case c. By Definition 3.2.4 we have $\mathsf{c}[Y{:=}s][X{:=}t] \equiv \mathsf{c}$. By Definition 3.2.4 we have $\mathsf{c}[Z{:=}t][Y{:=}s[Z{:=}t]] \equiv \mathsf{c}$. The result follows.
- The case $\pi{\cdot}X$. By Definition 3.2.4 we have $(\pi{\cdot}X)[Y{:=}s][Z{:=}t] \equiv (\pi{\cdot}X)[Z{:=}t]$. By Definition 3.2.4 we have $(\pi{\cdot}X)[Z{:=}t] \equiv \pi{\cdot}X$. By Definition 3.2.4 we have $\pi{\cdot}X \equiv (\pi{\cdot}X)[Y{:=}s[Z{:=}t]]$. By Definition 3.2.4 we have $(\pi{\cdot}X)[Y{:=}s[Z{:=}t]] \equiv (\pi{\cdot}X)[Z{:=}t][Y{:=}s[Z{:=}t]]$. The result follows.

- The case $\pi{\cdot}Y$. By Definition 3.2.4 we have $(\pi{\cdot}Y)[Y{:=}s][Z{:=}t] \equiv (\pi{\cdot}s)[Z{:=}t]$. By Definition 3.2.4 we have $(\pi{\cdot}s)[Z{:=}t] \equiv (\pi{\cdot}Y)[Y{:=}s[Z{:=}t]])$. By Definition 3.2.4 and the fact that $Y \notin fV(t)$ we have $(\pi{\cdot}Y)[Y{:=}s[Z{:=}t]]) \equiv (\pi{\cdot}Y)[Z{:=}t][Y{:=}s[Z{:=}t]]$. The result follows.
- The case $\pi{\cdot}Z$. By Definition 3.2.4 we have $(\pi{\cdot}Z)[Y{:=}s][Z{:=}t] \equiv (\pi{\cdot}Z)[Z{:=}t]$. By Definition 3.2.4 we have $(\pi{\cdot}Z)[Z{:=}t] \equiv \pi{\cdot}t$. By Lemma 3.2.14 and the fact that $Y \notin fV(t)$ we have $\pi{\cdot}t \equiv (\pi{\cdot}t)[Y{:=}s[Z{:=}t]]$. By Definition 3.2.4 and the fact that $Y \notin fV(t)$ we have $(\pi{\cdot}t)[Y{:=}s[Z{:=}t]] \equiv (\pi{\cdot}Z)[Z{:=}t][Y{:=}s[Z{:=}t]]$. The result follows.
- The case rs. By Definition 3.2.4 we have $(rs)[Y{:=}s][Z{:=}t] \equiv ((r[Y{:=}s])(s[Y{:=}s]))[Z{:=}t]$. By Definition 3.2.4 we have $((r[Y{:=}s])(s[Y{:=}s]))[Z{:=}t] \equiv (r[Y{:=}s][Z{:=}t])(s[Y{:=}s][Z{:=}t])$. By inductive hypothesis $(r[Y{:=}s][Z{:=}t])(s[Y{:=}s][Z{:=}t]) \equiv (r[Z{:=}t][Y{:=}s[Z{:=}t]])(s[Z{:=}t][Y{:=}s[Z{:=}t]])$. By Definition 3.2.4 we have $(r[Z{:=}t][Y{:=}s[Z{:=}t]])(s[Z{:=}t][Y{:=}s[Z{:=}t]]) \equiv ((r[Z{:=}t])(s[Z{:=}t]))[Y{:=}s[Z{:=}t]]$. By Definition 3.2.4 we have $((r[Z{:=}t])(s[Z{:=}t]))[Y{:=}s[Z{:=}t]] \equiv (rs)[Z{:=}t][Y{:=}s[Z{:=}t]]$. The result follows.
- The case $\lambda a.r$. By Definition 3.2.4 we have $(\lambda a.r)[Y{:=}s][Z{:=}t] \equiv (\lambda a.(r[Y{:=}s]))[Z{:=}t]$. By Definition 3.2.4 we have $(\lambda a.(r[Y{:=}s]))[Z{:=}t] \equiv \lambda a.(r[Y{:=}s][Z{:=}t])$. By inductive hypothesis $\lambda a.(r[Y{:=}s][Z{:=}t]) \equiv \lambda a.(r[Z{:=}t][Y{:=}s[Z{:=}t]])$. By Definition 3.2.4 we have $\lambda a.(r[Z{:=}t][Y{:=}s[Z{:=}t]]) \equiv (\lambda a.(r[Z{:=}t]))[Y{:=}s[Z{:=}t]]$. By Definition 3.2.4 we have $(\lambda a.(r[Z{:=}t]))[Y{:=}s[Z{:=}t]] \equiv (\lambda a.r)[Z{:=}t][Y{:=}s[Z{:=}t]]$. The result follows.
- The case $\lambda X.r$. By Definition 3.2.4 we have $(\lambda X.r)[Y{:=}s][Z{:=}t] \equiv (\lambda X.r[Y{:=}s])[Z{:=}t]$. By Definition 3.2.4 we have $(\lambda X.r[Y{:=}s])[Z{:=}t] \equiv \lambda X.(r[Y{:=}s][Z{:=}t])$. By inductive hypothesis $\lambda X.(r[Y{:=}s][Z{:=}t]) \equiv \lambda X.(r[Z{:=}t][Y{:=}s[Z{:=}t]])$. By Definition 3.2.4 we have $\lambda X.(r[Z{:=}t][Y{:=}s[Z{:=}t]]) \equiv \lambda X.(r[Z{:=}t])[Y{:=}s[Z{:=}t]]$. By Definition 3.2.4 we have $\lambda X.(r[Z{:=}t])[Y{:=}s[Z{:=}t]] \equiv (\lambda X.r)[Z{:=}t][Y{:=}s[Z{:=}t]]$. The result follows.
- The case $\lambda Y.s$. By Definition 3.2.4 we have $(\lambda Y.s)[Y{:=}s][Z{:=}t] \equiv (\lambda Y.(s)[Z{:=}t]$. By Definition 3.2.4 we have $(\lambda Y.(s)[Z{:=}t] \equiv \lambda Y.(s[Z{:=}t])$. By Definition 3.2.4 we have $\lambda Y.(s[Z{:=}t]) \equiv \lambda Y.(s[Z{:=}t])[Y{:=}s[Z{:=}t]]$. By Definition 3.2.4 we have $\lambda Y.(s[Z{:=}t])[Y{:=}s[Z{:=}t]] \equiv \lambda Y.(s)[Z{:=}t][Y{:=}s[Z{:=}t]]$. The result follows.
- The case $\lambda Z.t$. By Definition 3.2.4 we have $(\lambda Z.t)[Y{:=}s][Z{:=}t] \equiv (\lambda Z.(t[Y{:=}s])[Z{:=}t]$. By Definition 3.2.4 we have $(\lambda Z.(t[Y{:=}s])[Z{:=}t] \equiv \lambda Z.(t[Y{:=}s])$. By Definition 3.2.4 we have $\lambda Z.(t[Y{:=}s]) \equiv (\lambda Z.t)[Y{:=}s]$. By Definition 3.2.4 we have $(\lambda Z.t)[Y{:=}s] \equiv (\lambda Z.t)[Z{:=}t[Y{:=}s]][Y{:=}s]$. The result follows.

⊠

Lemma 3.2.16, that the permutation and instantiation actions commute, is another result that will be used extensively throughout the rest of the Chapter. Lemma 3.2.16 makes formal the intuition that the order that one chooses to α-rename a bound level one variable, or instantiate an unknown, is irrelevant.

Lemma 3.2.16: $(\pi{\cdot}r)[Y{:=}s] \equiv \pi{\cdot}(r[Y{:=}s])$

Proof. By induction on $depth(r)$.

- The case a. By Definition 3.2.4 we have $\pi{\cdot}(a[Y{:=}s]) \equiv \pi{\cdot}a$. By Definition 3.2.4 we have $\pi{\cdot}a \equiv \pi(a)$. By Definition 3.2.4 we have $\pi(a) \equiv \pi(a)[Y{:=}s]$. By Definition 3.2.4 we have $\pi(a)[Y{:=}s] \equiv (\pi{\cdot}a)[Y{:=}s]$. The result follows.

- The case c. By Definition 3.2.4 we have $\pi{\cdot}(\mathsf{c}[Y{:=}s]) \equiv \pi{\cdot}\mathsf{c}$. By Definition 3.2.4 we have $\pi{\cdot}\mathsf{c} \equiv \mathsf{c}$. By Definition 3.2.4 we have $\mathsf{c} \equiv \mathsf{c}[Y{:=}s]$. By Definition 3.2.4 we have $\mathsf{c}[Y{:=}s] \equiv (\pi{\cdot}\mathsf{c})[Y{:=}s]$. The result follows.
- The case $\pi'{\cdot}X$. By Definition 3.2.4 we have $\pi{\cdot}((\pi'{\cdot}X)[Y{:=}s]) \equiv \pi{\cdot}(\pi'{\cdot}X)$. By Definition 3.2.4 we have $\pi{\cdot}(\pi'{\cdot}X) \equiv (\pi{\circ}\pi'){\cdot}X$. By Definition 3.2.4 we have $(\pi{\circ}\pi'){\cdot}X \equiv ((\pi{\circ}\pi'){\cdot}X)[Y{:=}s]$. By Definition 3.2.4 we have $((\pi{\circ}\pi'){\cdot}X)[Y{:=}s] \equiv (\pi{\cdot}(\pi'{\cdot}X))[Y{:=}s]$. The result follows.
- The case $\pi'{\cdot}Y$. By Lemma 3.2.13 we have $(\pi{\cdot}(\pi'{\cdot}Y))[Y{:=}s] \equiv ((\pi{\circ}\pi'){\cdot}Y)[Y{:=}s]$. By Definition 3.2.4 we have $((\pi{\circ}\pi'){\cdot}Y)[Y{:=}s] \equiv (\pi{\circ}\pi'){\cdot}s$. By Lemma 3.2.13 we have $(\pi{\circ}\pi'){\cdot}s \equiv \pi{\cdot}(\pi'{\cdot}s)$. By Definition 3.2.4 we have $\pi{\cdot}(\pi'{\cdot}s) \equiv \pi{\cdot}((\pi'{\cdot}Y)[Y{:=}s])$. The result follows.
- The case rs. By Definition 3.2.4 we have $(\pi{\cdot}rs)[Z{:=}t] \equiv ((\pi{\cdot}r)(\pi{\cdot}s))[Z{:=}t]$. By Definition 3.2.4 we have $((\pi{\cdot}r)(\pi{\cdot}s))[Z{:=}t] \equiv ((\pi{\cdot}r)[Z{:=}t])((\pi{\cdot}s)[Z{:=}t])$. By inductive hypothesis $((\pi{\cdot}r)[Z{:=}t])((\pi{\cdot}s)[Z{:=}t]) \equiv (\pi{\cdot}(r[Z{:=}t]))(\pi{\cdot}(s[Z{:=}t]))$. By Definition 3.2.4 we have $(\pi{\cdot}(r[Z{:=}t]))(\pi{\cdot}(s[Z{:=}t])) \equiv \pi{\cdot}((r[Z{:=}t])(s[Z{:=}t]))$. By Definition 3.2.4 we have $\pi{\cdot}((r[Z{:=}t])(s[Z{:=}t])) \equiv \pi{\cdot}((rs)[Z{:=}t])$. The result follows.
- The case $\lambda a.r$. By Definition 3.2.4 we have $(\pi{\cdot}\lambda a.r)[Y{:=}s] \equiv (\lambda\pi(a).(\pi{\cdot}r))[Y{:=}s]$. By Definition 3.2.4 we have $(\lambda\pi(a).(\pi{\cdot}r))[Y{:=}s] \equiv \lambda\pi(a).((\pi{\cdot}r)[Y{:=}s])$. By inductive hypothesis $\lambda\pi(a).((\pi{\cdot}r)[Y{:=}s]) \equiv \lambda\pi(a).(\pi{\cdot}(r[Y{:=}s]))$. By Definition 3.2.4 we have $\lambda\pi(a).(\pi{\cdot}(r[Y{:=}s])) \equiv \pi{\cdot}(\lambda a.(r[Y{:=}s]))$. By Definition 3.2.4 we have $\pi{\cdot}(\lambda a.(r[Y{:=}s])) \equiv \pi{\cdot}((\lambda a.r)[Y{:=}s])$. The result follows.
- The case $\lambda X.r$. Suppose $X \notin fV(s)$, which can be guaranteed by renaming. By Definition 3.2.4 we have $(\pi{\cdot}\lambda X.r)[Y{:=}s] \equiv (\lambda X.(\pi{\cdot}r[X{:=}\pi^{-1}{\cdot}X]))[Y{:=}s]$. By Definition 3.2.4 we have $(\lambda X.(\pi{\cdot}r[X{:=}\pi^{-1}{\cdot}X]))[Y{:=}s] \equiv (\lambda X.(\pi{\cdot}r[X{:=}\pi^{-1}{\cdot}X][Y{:=}s]))$. By Lemma 3.2.15 we have $(\lambda X.(\pi{\cdot}r[X{:=}\pi^{-1}{\cdot}X][Y{:=}s])) \equiv \lambda X.(\pi{\cdot}r[Y{:=}s][X{:=}\pi^{-1}{\cdot}X])$. By Definition 3.2.4 we have $\lambda X.(\pi{\cdot}r[Y{:=}s][X{:=}\pi^{-1}{\cdot}X]) \equiv \pi{\cdot}(\lambda X.r[Y{:=}s])$. By Definition 3.2.4 we have $\pi{\cdot}(\lambda X.r[Y{:=}s]) \equiv \pi{\cdot}((\lambda X.r)[Y{:=}s])$. The result follows.
- The case $\lambda Y.s$. By Definition 3.2.4 we have $\pi{\cdot}((\lambda Y.s)[Y{:=}s]) \equiv \pi{\cdot}\lambda Y.s$. By Definition 3.2.4 we have $\pi{\cdot}\lambda Y.s \equiv \lambda Y.((\pi{\cdot}s)[Y{:=}\pi^{-1}{\cdot}Y])$. By Definition 3.2.4 we have $\lambda Y.((\pi{\cdot}s)[Y{:=}\pi^{-1}{\cdot}Y]) \equiv (\lambda Y.((\pi{\cdot}s)[Y{:=}\pi^{-1}{\cdot}Y]))[Y{:=}s]$. By Definition 3.2.4 we have $(\lambda Y.((\pi{\cdot}s)[Y{:=}\pi^{-1}{\cdot}Y]))[Y{:=}s] \equiv (\pi{\cdot}\lambda Y.s)[Y{:=}s]$. The result follows.

⊠

3.3 The theory of derivable freshness

Derivable freshness is a generalisation of the 'free variables' of a term in the presence of nominal unknowns, as unknowns behave like they have an infinite set of free atoms, that is, they may be instantiated to any term. This Section introduces the theory of derivable freshness for terms of the two-level λ-calculus. Definition 3.3.3 introduces the derivable freshness judgment. Lemmas 3.3.8 and 3.3.9 are basic results regarding derivable freshness, used throughout the rest of the Chapter, and may be considered the main results in this Section.

$$\frac{}{\Delta \vdash a\#b}\,(\#\mathbf{b}) \qquad \frac{}{\Delta \vdash a\#\mathsf{c}}\,(\#\mathsf{c}) \qquad \frac{(\pi^{\text{-}1}(a)\#X \in \Delta)}{\Delta \vdash a\#\pi{\cdot}X}\,(\#\mathbf{X})$$

$$\frac{\Delta \vdash a\#r \quad \Delta \vdash a\#s}{\Delta \vdash a\#rs}\,(\#\mathbf{rs}) \qquad \frac{}{\Delta \vdash a\#\lambda a.r}\,(\#\lambda\mathbf{a}) \qquad \frac{\Delta \vdash a\#s}{\Delta \vdash a\#\lambda b.s}\,(\#\lambda\mathbf{b})$$

$$\frac{\Delta, a\#X \vdash \pi(a)\#\pi{\cdot}r}{\Delta \vdash \pi(a)\#\pi{\cdot}(\lambda X.r)}\,(\#\lambda\mathbf{X})$$

Figure 3.2 Derivable freshness rules

Definition 3.3.1: A **freshness** is a pair of the form $a\#r$. We call a pair $a\#X$ (where $r \equiv X$) a **primitive freshness**.

Definition 3.3.2: Call a finite set of primitive freshnesses a **freshness context**. $\Delta, \Delta', \Delta''$, and so on, will range over freshness contexts.

For typographical convenience, we drop set brackets, and write $a\#X, b\#Y$ instead of $\{a\#X, b\#Y\}$ and write $\Delta, a\#X$ instead of $\Delta \cup \{a\#X\}$.

Definition 3.3.3: Define a notion of **derivable freshness** using the rules in Figure 3.2.

We may write $\Delta \vdash a\#r$ as a shorthand for '$\Delta \vdash a\#r$ is derivable'. Write $\Delta \nvdash a\#r$ as a shorthand for '$\Delta \vdash a\#r$ is *not* derivable'.

Freshness is a generalisation of 'the free variables of' relation familiar from the λ-calculus, applied to nominal terms. This generalisation is necessary as nominal unknowns X behave as if they have an infinite set of free atoms, due to the possibility of their instantiation with any term.

Note, also, that the freshness relation of Definition 3.3.3 is a 'positive' statement, compared to the 'negativity' of the usual 'free variables of' relation. That is, freshness asserts something positive: $\Delta \vdash a\#r$ states that a is fresh for r, whereas $a \notin fv(r)$ states that a is not in the free variables of a term r. This difference, once again, can be explained by looking at nominal unknowns: we need to hypothesise the freshness of individual atoms with respect to unknowns in a freshness context Δ. This is why (#**X**) in Definition 3.3.3 is necessary.

The only other freshness derivation rule of note is (#λ**X**). This is essentially in the form that it is to make Lemma 3.3.9 work.

Definition 3.3.4: An **instantiation** θ is a finitely supported map from level two variables to terms. (In this context, finitely supported means that the set $\{X \mid \theta(X) \not\equiv id{\cdot}X\}$ is finite.)

Instantiations act on terms $r\theta$ in the natural way, extending the instantiation action of Definition 3.2.4.

Definition 3.3.5: Define the **range** of an instantiation by:

$$rng(\theta) = \bigcup\{fV(\theta(X)) \mid \text{ for every } X \text{ such that } \theta(X) \not\equiv id{\cdot}X\ \}$$

Definition 3.3.6: Define a **pointwise instantiation action** on freshness contexts by:

$$\Delta\theta = \{a\#\theta(X) \mid a\#X \in \Delta\}$$

Definition 3.3.7: Suppose $\mathcal{F}$ is a finite set of freshnesses. Write $\Delta \vdash \mathcal{F}$ whenever $\Delta \vdash a\#r$ for all $a\#r \in \mathcal{F}$.

Lemma 3.3.8 intuitively states that derivable freshness is invariant under instantiation.

Lemma 3.3.8: If $\Delta' \vdash \Delta\theta$ then if $\Delta \vdash a\#r$ then $\Delta' \vdash a\#(r\theta)$.

Proof. By induction on the derivation of $\Delta \vdash a\#r$.

- The case ($\#\mathbf{b}$). By Definition 3.2.4 we have $b\theta \equiv b$. The result follows.
- The case ($\#\mathsf{c}$). By Definition 3.2.4 we have $\mathsf{c}\theta \equiv \mathsf{c}$. The result follows.
- The case ($\#\mathbf{X}$). Suppose $\pi^{-1}(a)\#X \in \Delta$, therefore $\Delta \vdash a\#\pi{\cdot}X$. It is a fact that $\pi^{-1}(a)\#X\theta \in \Delta\theta$ therefore $\Delta\theta \vdash a\#\pi{\cdot}(X\theta)$. By Lemma 3.2.16 we have $\Delta\theta \vdash a\#(\pi{\cdot} X)\theta$. The result follows.
- The case ($\#\mathbf{rs}$). Suppose $\Delta \vdash a\#r$ and $\Delta \vdash a\#s$. By inductive hypotheses $\Delta\theta \vdash a\#r\theta$ and $\Delta\theta \vdash a\#s\theta$. Using ($\#\mathbf{rs}$) we obtain $\Delta\theta \vdash a\#(r\theta)(s\theta)$. By Definition 3.2.4 we have $(r\theta)(s\theta) \equiv (rs)\theta$. The result follows.
- The case ($\#\lambda\mathbf{a}$). By Definition 3.2.4 we have $(\lambda a.r)\theta \equiv \lambda a.(r\theta)$. Using ($\#\lambda\mathbf{a}$) we obtain $\Delta\theta \vdash a\#\lambda a.(r\theta)$. The result follows.
- The case ($\#\lambda\mathbf{b}$). Suppose $\Delta \vdash a\#s$. By inductive hypothesis $\Delta\theta \vdash a\#s\theta$. Using ($\#\lambda\mathbf{a}$) we obtain $\Delta\theta \vdash a\#\lambda b.(s\theta)$. By Definition 3.2.4 we have $\lambda b.(s\theta) \equiv (\lambda b.s)\theta$. The result follows.
- The case ($\#\lambda\mathbf{X}$). Suppose $\Delta \vdash a\#\lambda X.r$ and suppose $X \notin rng(\theta)$, which can be guaranteed by renaming. It is a fact that $\Delta, a\#X \vdash a\#r$. By inductive hypothesis $\Delta', a\#X\theta \vdash a\#r\theta$. By assumption $X \notin rng(\theta)$ therefore $\Delta', a\#X \vdash r\theta$. Using ($\#\lambda\mathbf{X}$) we obtain $\Delta' \vdash a\#\lambda X.(r\theta)$. By Definition 3.2.4 we have $\lambda X.(r\theta) \equiv (\lambda X.r)\theta$. The result follows.

⊠

Lemma 3.3.9 is a basic equivariance result for derivable freshness.

Lemma 3.3.9: If $\Delta \vdash a\#r$ then $\Delta \vdash \pi(a)\#\pi{\cdot}r$.

Proof. By induction on the derivation of $\Delta \vdash a\#r$.

- The case ($\#\mathbf{b}$). By Definition 3.2.4 we have $\pi{\cdot}b \equiv \pi(b)$. Using ($\#\mathbf{b}$) we obtain $\Delta \vdash \pi(a)\#\pi(b)$. The result follows.
- The case ($\#\mathsf{c}$). By Definition 3.2.4 we have $\pi{\cdot}\mathsf{c} \equiv \mathsf{c}$. Using ($\#\mathsf{c}$) we obtain $\Delta \vdash \pi(a)\#\mathsf{c}$. The result follows.
- The case ($\#\mathbf{X}$). Suppose $a\#X \in \Delta$. It is a fact that $(\pi\circ\pi')^{-1}(\pi(a))\#X \in \Delta$. Using ($\#\mathbf{X}$) we obtain $\Delta \vdash \pi(a)\#(\pi\circ\pi'){\cdot}X$. By Definition 3.2.4 we have $(\pi\circ\pi'){\cdot}X \equiv \pi{\cdot}(\pi'{\cdot}X)$. The result follows.

- The case (#**rs**). Suppose $\Delta \vdash a\#r$ and $\Delta \vdash a\#s$. By inductive hypotheses $\Delta \vdash \pi(a)\#\pi{\cdot}r$ and $\Delta \vdash \pi(a)\#\pi{\cdot}s$. Using (#**rs**) we obtain $\Delta \vdash \pi(a)\#((\pi{\cdot}r)(\pi{\cdot}s))$. By Definition 3.2.4 we have $((\pi{\cdot}r)(\pi{\cdot}s)) \equiv \pi{\cdot}rs$. The result follows.
- The case ($\#\lambda$**a**). Using ($\#\lambda$**a**) we obtain $\Delta \vdash \pi(a)\#\lambda\pi(a).(\pi{\cdot}r)$. By Definition 3.2.4 we have $\lambda\pi(a).(\pi{\cdot}r) \equiv \pi{\cdot}\lambda a.r$. The result follows.
- The case ($\#\lambda$**b**). Suppose $\Delta \vdash a\#s$. By inductive hypothesis $\Delta \vdash \pi(a)\#\pi{\cdot}s$. Using ($\#\lambda$**b**) we obtain $\Delta \vdash \pi(a)\#\lambda\pi(b).(\pi{\cdot}s)$. By Definition 3.2.4 we have $\lambda\pi(b).(\pi{\cdot}s) \equiv \pi{\cdot}\lambda b.s$. The result follows.
- The case ($\#\lambda$**X**). Suppose $\Delta, a\#X \vdash \pi'(a)\#\pi'{\cdot}r$. By inductive hypothesis $\Delta, a\#X \vdash \pi(\pi'(a))\#\pi{\cdot}(\pi'{\cdot}r)$. By Lemma 3.2.13 we have $\Delta, a\#X \vdash (\pi\circ\pi'){\cdot}a\#(\pi\circ\pi'){\cdot}r$. Using ($\#\lambda$**X**) we obtain $\Delta \vdash (\pi\circ\pi'){\cdot}a\#(\pi\circ\pi'){\cdot}\lambda X.r$. By Lemma 3.2.13 we have $\Delta \vdash \pi(\pi'(a))\#\pi{\cdot}(\pi'{\cdot}\lambda X.r)$. The result follows.

⊠

Derivable freshness is invariant under freshness context weakening. Lemma 3.3.10 makes this formal.

Lemma 3.3.10: If $\Delta \vdash a\#r$ and $Y \notin \Delta$ then $\Delta, b\#Y \vdash a\#r$.

Proof. By induction on the derivation of $\Delta \vdash a\#r$. See Appendix A. ⊠

3.4 Reductions and confluence

In this Section, we prove various soundness and correctness results of the calculus, centering on a reduction relation defined in Definition 3.4.3. The main result of this Section is a proof that the reduction relation of Definition 3.4.3 is confluent (Theorem 3.4.42 in Subsection 3.4.1).

Lemma 3.4.4 is 'subject reduction for freshness', stating that reduction preserves the freshness of a term. Lemma 3.4.5 states that reduction is invariant under instantiation. Lemma 3.4.8 states that the reduction relation is equivariant. The confluence proof is split across three Subsections, corresponding to a division of the reduction rules into 'level one' and 'level two' reductions. Subsection 3.4.1 handles the confluence of level one reductions. Subsection 3.4.2 handles the confluence of level two reductions. Finally, Subsection 3.4.3 handles the confluence of level one and level two reductions together by stitching the two preceeding confluence proofs into a single confluence result.

Definition 3.4.1: Define the **level** of a term by the following rules:

$$level(a) = 1 \qquad level(\mathsf{c}) = 1 \qquad level(\pi{\cdot}X) = 2$$
$$level(rs) = \max(level(r), level(s)) \quad level(\lambda a.r) = level(r) \quad level(\lambda X.r) = 2$$

Definition 3.4.1 is used in the definition of the reduction relation (Definition 3.4.3). Intuitively, if a term r mentions a variable of level two, then $level(r) = 2$, otherwise $level(r) = 1$. For instance, $level(\lambda X.X) = 2$, $level(\lambda X.a) = 2$ and $level(\lambda a.a) = 1$.

$$\frac{\Delta \vdash r \rhd s}{\Delta \vdash \lambda a.r \rhd \lambda a.s}\,(\rhd\lambda\mathbf{a}) \qquad \frac{\Delta \vdash r \rhd t}{\Delta \vdash rs \rhd ts}\,(\rhd\mathbf{rs1}) \qquad \frac{\Delta \vdash s \rhd t}{\Delta \vdash rs \rhd rt}\,(\rhd\mathbf{rs2})$$

$$\frac{\Delta \vdash r \rhd s \quad (X \notin \Delta)}{\Delta \vdash \lambda X.r \rhd \lambda X.s}\,(\rhd\lambda\mathbf{X}) \qquad \frac{\Delta \vdash r \rhd s \quad \Delta \vdash a\#r \quad \Delta \vdash b\#r}{\Delta \vdash (b\ a)\cdot r \rhd s}\,(\rhd\alpha)$$

$$\frac{}{\Delta \vdash a[a{\mapsto}t] \to t}\,(\to\mathbf{a}) \qquad \frac{\Delta \vdash a\#r}{\Delta \vdash r[a{\mapsto}t] \to r}\,(\to\#)$$

$$\frac{\Delta \vdash a\#s}{\Delta \vdash (rs)[a{\mapsto}t] \to (r[a{\mapsto}t])s}\,(\to\mathbf{rs1})$$

$$\frac{(level(r) = 1)}{\Delta \vdash (rs)[a{\mapsto}t] \to (r[a{\mapsto}t])(s[a{\mapsto}t])}\,(\to\mathbf{rs2})$$

$$\frac{\Delta \vdash b\#t}{\Delta \vdash (\lambda b.s)[a{\mapsto}t] \to \lambda b.(s[a{\mapsto}t])}\,(\to\lambda\mathbf{b})$$

$$\frac{(X \notin fV(t))}{\Delta \vdash (\lambda X.r)[a{\mapsto}t] \to \lambda X.(r[a{\mapsto}t])}\,(\to\lambda\mathbf{X}) \qquad \frac{}{\Delta \vdash (\lambda X.r)t \to r[X{:=}t]}\,(\to\beta)$$

$$\frac{}{\Delta \vdash a \Rightarrow a}\,(\Rightarrow\mathbf{a}) \qquad \frac{}{\Delta \vdash \pi\cdot X \Rightarrow \pi\cdot X}\,(\Rightarrow\mathbf{X}) \qquad \frac{}{\Delta \vdash \mathsf{c} \Rightarrow \mathsf{c}}\,(\Rightarrow\mathsf{c})$$

$$\frac{\Delta \vdash r \Rightarrow t \quad \Delta \vdash s \Rightarrow u}{\Delta \vdash rs \Rightarrow tu}\,(\Rightarrow\mathbf{rs}) \qquad \frac{\Delta \vdash r \Rightarrow s}{\Delta \vdash \lambda a.r \Rightarrow \lambda a.s}\,(\Rightarrow\lambda\mathbf{a})$$

$$\frac{\Delta \vdash r \Rightarrow s}{\Delta \vdash \lambda X.r \Rightarrow \lambda X.s}\,(\Rightarrow\lambda\mathbf{X}) \qquad \frac{\Delta \vdash r \Rightarrow t \quad \Delta \vdash s \Rightarrow u \quad \Delta \vdash tu \overset{\mathbf{(level2)}}{\to} v}{\Delta \vdash rs \Rightarrow v}\,(\Rightarrow\epsilon)$$

$$\frac{\Delta \vdash r \Rightarrow s \quad \Delta \vdash a\#r \quad \Delta \vdash b\#r}{\Delta \vdash (b\ a)\cdot r \Rightarrow s}\,(\Rightarrow\alpha)$$

Figure 3.3 Congruence, reduction and parallel reduction rules

Definition 3.4.2: Write $\Delta \vdash -\triangleright-$ for a binary relation on terms, parameterised by a freshness context, Δ. Call a relation $\triangleright$ a **congruence** when it is closed under the congruence rules in Figure 3.3.

Definition 3.4.2 intuitively will allow us to reduce, when applicable, at any point inside a term. Our notion of congruence also incorporates a notion of α-renaming with $(\triangleright\alpha)$. This is a design decision; we could easily have defined an independent notion of α-equivalence.

Definition 3.4.3: Let $\Delta \vdash r \to s$ be the least congruence closed under the reduction rules in Figure 3.3.

Recall that $r[a\mapsto t]$ is sugar for $(\lambda a.r)t$. In rule $(\to\beta)$ level 2 β-reduction occurs in one step using $[X{:=}t]$. The rest of the rules are implementing a multi-step reduction for level 1 β-reduction, in the style of a calculus of explicit substitutions (though we do not put the explicit substitution in as a distinct syntactic form).

This is a design decision. A one-step substitution action $[a{:=}t]$ can indeed be defined (see Definition 3.4.12); it could also be used in Figure 3.3. We do not do this because level 1 reduction is more complex due to the presence of level 2 variables X, so it is convenient to treat it one small step at a time.

The rules (**rs1**) and (**rs2**) in Definition 3.4.3 have particularly interesting side-conditions. To understand why they are there, consider adding the following faulty reduction rule:

$$\Delta \vdash (rs)[a\mapsto t] \overset{(\mathbf{BAD})}{\to} (r[a\mapsto t])(s[a\mapsto t])$$

Then we have the following divergent reduction paths:

$$\begin{array}{rcl}
\Delta \vdash ((\lambda X.((b\ a)\cdot X))b)[b\mapsto c] & \overset{(\to\beta)}{\to} & (((b\ a)\cdot X)[X{:=}b])[b\mapsto c] \\
& \equiv & a[b\mapsto c] \\
& \overset{(\to\#)}{\to} & a
\end{array}$$

$$\begin{array}{rcl}
\Delta \vdash ((\lambda X.((b\ a)\cdot X))b)[b\mapsto c] & \overset{(\mathbf{BAD})}{\to} & ((\lambda X.((b\ a)\cdot X)[b\mapsto c])(b[b\mapsto c]) \\
& \overset{(\to\lambda\mathbf{X})}{\to} & (\lambda X.(((b\ a)\cdot X)[b\mapsto c]))(b[b\mapsto c]) \\
& \overset{(\to\mathbf{a})}{\to} & (\lambda X.(((b\ a)\cdot X)[b\mapsto c]))c \\
& \overset{(\to\beta)}{\to} & (((b\ a)\cdot X)[b\mapsto c])[X{:=}c] \\
& \equiv & (c[b\mapsto c]) \\
& \overset{(\to\#)}{\to} & c
\end{array}$$

Where a and c are unjoinable. Therefore it is clear that some side-condition on (**BAD**) must be imposed.

To close a divergence in $((\lambda X.r)t)[b\mapsto u]$ between $(\to\beta)$ and (**BAD**) we must join $r[X{:=}t][b\mapsto u]$ and $r[b\mapsto u][X{:=}t[b\mapsto u]]$ where $X \notin fV(u)$ and $X \notin fV(t)$, in a freshness context such that $\Delta \vdash b\#u$. It is not possible to close this divergence in general: take $r \equiv (b\ a)\cdot X$, $t \equiv a$ and $u \equiv c$. However, it *is* possible to close this divergence if $level(r) = 1$ or if $\Delta \vdash b\#t$—motivating the side-conditions we see on (**rs1**) and (**rs2**) of Definition 3.4.3.

Lemma 3.4.4 is our first correctness result, stating that freshness is preserved by the reduction relation. Intuitively, the lemma states that reduction does not create any new atoms, and corresponds to the fact that reduction in the λ-calculus reduces the free variables of a term.

Lemma 3.4.4: If $\Delta \vdash a\#r$ and $\Delta \vdash r \to s$ then $\Delta \vdash a\#s$.

Proof. By induction on the derivation of $\Delta \vdash r \to s$.

- The case $(\to\mathbf{a})$. Suppose $\Delta \vdash b[b\mapsto u] \to u$ and suppose $\Delta \vdash a\#b[b\mapsto u]$. By definition $b[b\mapsto u] \equiv (\lambda b.b)u$ therefore $\Delta \vdash a\#\lambda b.b$ and $\Delta \vdash a\#u$. The result follows.
- The case $(\to\#)$. There are two cases:
  - The case $r[a\mapsto t]$. Suppose $\Delta \vdash r[a\mapsto t] \to r$ and $\Delta \vdash a\#r$ and $\Delta \vdash a\#r[a\mapsto t]$. The result follows by assumption.
  - The case $r[b\mapsto u]$. Suppose $\Delta \vdash r[b\mapsto u] \to r$ and $\Delta \vdash a\#r$ and $\Delta \vdash a\#r[b\mapsto u]$. By definition $r[b\mapsto u] \equiv (\lambda b.r)u$ therefore $\Delta \vdash a\#\lambda b.r$, therefore $\Delta \vdash a\#r$. The result follows.
- The case $(\to\mathbf{rs1})$. There are two cases:
  - The case $(rs)[a\mapsto t]$. Suppose $\Delta \vdash a\#s$ so that $\Delta \vdash (rs)[a\mapsto t] \to (r[a\mapsto t])s$ and $\Delta \vdash a\#(rs)[a\mapsto t]$. By definition $(rs)[a\mapsto t] \equiv (\lambda a.(rs))t$ therefore $\Delta \vdash a\#t$. The result follows.
  - The case $(rs)[b\mapsto u]$. Suppose $\Delta \vdash b\#s$ so that $\Delta \vdash (rs)[b\mapsto u] \to (r[b\mapsto u])s$ and $\Delta \vdash a\#(rs)[b\mapsto u]$. By definition $(rs)[b\mapsto u] \equiv (\lambda b.(rs))u$ therefore $\Delta \vdash a\#r$, $\Delta \vdash a\#s$ and $\Delta \vdash a\#u$. The result follows.
- The case $(\to\mathbf{rs2})$. Suppose $level(r) = 1$. There are two cases:
  - The case $(rs)[a\mapsto t]$. Suppose $\Delta \vdash (rs)[a\mapsto t] \to (r[a\mapsto t])(s[a\mapsto t])$ and $\Delta \vdash a\#(rs)[a\mapsto t]$. By definition $(rs)[a\mapsto t] \equiv (\lambda a.(rs))t$ therefore $\Delta \vdash a\#t$. The result follows.
  - The case $(rs)[b\mapsto u]$. Suppose $\Delta \vdash (rs)[b\mapsto u] \to (r[b\mapsto u])(s[b\mapsto u])$ and $\Delta \vdash a\#(rs)[b\mapsto u]$. By definition $(rs)[b\mapsto u] \equiv (\lambda b.(rs))u$ therefore $\Delta \vdash a\#r$, $\Delta \vdash a\#s$ and $\Delta \vdash a\#u$. The result follows.
- The case $(\to\lambda\mathbf{b})$. There are two cases:
  - The case $(\lambda b.s)[a\mapsto t]$. Suppose $\Delta \vdash b\#t$ so that $\Delta \vdash (\lambda b.s)[a\mapsto t] \to \lambda b.(s[a\mapsto t])$. By definition $(\lambda b.s)[a\mapsto t] \equiv (\lambda a.(\lambda b.s))t$, therefore $\Delta \vdash a\#t$. The result follows.
  - The case $(\lambda b.s)[c\mapsto u]$. Suppose $\Delta \vdash b\#u$ so that $\Delta \vdash (\lambda b.s)[c\mapsto u] \to \lambda b.(s[c\mapsto u])$. By definition $(\lambda b.s)[c\mapsto u] \equiv (\lambda c.(\lambda b.s))u$, therefore $\Delta \vdash a\#s$ and $\Delta \vdash a\#u$. The result follows.
- The case $(\to\beta)$. Suppose $\Delta \vdash a\#(\lambda X.r)t$ so $\Delta \vdash a\#t$ and $\Delta, a\#X \vdash a\#r$. By Lemma 3.3.8 we have $\Delta \vdash a\#r[X{:=}t]$. The result follows.
- The case $(\to\lambda\mathbf{X})$. There are two cases:
  - The case $(\lambda X.r)[a\mapsto t]$. Suppose $\Delta \vdash a\#(\lambda X.r)[a\mapsto t]$ therefore $\Delta \vdash a\#t$. Suppose also that $X \notin fV(t)$, which can be guaranteed, so that $\Delta \vdash (\lambda X.r)[a\mapsto t] \to \lambda X.(r[a\mapsto t])$. By definition $\lambda X.(r[a\mapsto t]) \equiv \lambda X.((\lambda a.r)t)$. Using $(\#\lambda\mathbf{X})$ we have $\Delta \vdash a\#\lambda X.((\lambda a.r)t)$ whenever $\Delta, a\#X \vdash a\#(\lambda a.r)t$. By assumption and Lemma 3.3.10 we have $\Delta, a\#X \vdash a\#t$. The result follows.
  - The case $(\lambda X.r)[b\mapsto u]$. Suppose $\Delta \vdash a\#(\lambda X.r)[b\mapsto u]$ so that $\Delta, a\#X \vdash a\#r$ and $\Delta \vdash a\#u$. Suppose also that $X \notin fV(u)$, which can be guaranteed, so that $\Delta \vdash (\lambda X.r)[b\mapsto u] \to \lambda X.(r[b\mapsto u])$. By definition $\lambda X.(r[b\mapsto u]) \equiv \lambda X.((\lambda b.r)u)$. Using $(\#\lambda\mathbf{X})$

we have $\Delta \vdash a\#\lambda X.((\lambda b.r)u)$ whenever $\Delta, a\#X \vdash a\#r$ and $\Delta \vdash a\#u$. By assumption and Lemma 3.3.10 we have $\Delta, a\#X \vdash a\#r$ and $\Delta \vdash a\#u$. The result follows.

- The case ($\triangleright\lambda\mathbf{a}$). There are two cases:
  - The case $\lambda a.r$. Since $\Delta \vdash a\#\lambda a.r$ always.
  - The case $\lambda b.s$. Suppose $\Delta \vdash a\#\lambda b.s$ therefore $\Delta \vdash a\#s$. Suppose also that $\Delta \vdash s \to t$ therefore $\Delta \vdash a\#s$ implies $\Delta \vdash a\#t$ by inductive hypothesis. Then $\Delta \vdash a\#\lambda b.t$. The result follows.
- The case ($\triangleright\mathbf{rs1}$). Suppose $\Delta \vdash a\#rs$ therefore $\Delta \vdash a\#r$ and $\Delta \vdash a\#s$. Suppose also that $\Delta \vdash r \to t$ therefore $\Delta \vdash a\#r$ implies $\Delta \vdash a\#t$ by inductive hypothesis. Then $\Delta \vdash a\#ts$. The result follows.
- The case ($\triangleright\mathbf{rs2}$). Suppose $\Delta \vdash a\#rs$ therefore $\Delta \vdash a\#r$ and $\Delta \vdash a\#s$. Suppose also that $\Delta \vdash s \to t$ therefore $\Delta \vdash a\#s$ implies $\Delta \vdash a\#t$ by inductive hypothesis. Then $\Delta \vdash a\#rt$. The result follows.
- The case ($\triangleright\lambda\mathbf{X}$). Suppose $\Delta \vdash a\#\lambda X.r$ therefore $\Delta, a\#X \vdash a\#r$. Suppose also that $\Delta \vdash a\#r$ implies $\Delta \vdash a\#s$ by inductive hypothesis. By Lemma 3.3.10 we have $\Delta, a\#X \vdash a\#r$ implies $\Delta, a\#X \vdash a\#s$, therefore $\Delta \vdash a\#\lambda X.s$. The result follows.
- The case ($\triangleright\alpha$). Suppose $\Delta \vdash r \to s$ so that $\Delta \vdash c\#r$ implies $\Delta \vdash c\#s$ by inductive hypothesis. Suppose also that $\Delta \vdash a\#s$ and $\Delta \vdash b\#s$. Suppose $\Delta \vdash c\#r$, therefore $\Delta \vdash c\#s$. By Lemma 3.3.9 we have $\Delta \vdash c\#(b\ a)\cdot s$. The result follows.

$\boxtimes$

We have already noted that for any term r not mentioning a variable of level two we have $\mathit{level}(r) = 1$ by Definition 3.4.1. The following result therefore encodes common sense. As there are no variables of level two present within the term to instantiate, there is therefore no opportunity to increase the level of the term.

Lemma 3.4.5: If $\mathit{level}(r) = 1$ then $\mathit{level}(r\theta) = 1$.

Proof. By induction on r. See Appendix A. $\boxtimes$

Lemma 3.4.6 is our second correctness result, stating that reduction is invariant under instantiation.

Lemma 3.4.6: If $\Delta' \vdash \Delta\theta$ and $\Delta \vdash r \to s$ then $\Delta' \vdash r\theta \to s\theta$.

Proof. By induction on the derivation of $\Delta \vdash r \to s$.

- The case ($\to\mathbf{a}$). Suppose $\Delta \vdash a[a\mapsto t] \to t$. By definition, $a[a\mapsto t] \equiv (\lambda a.a)t$ and therefore $(a[a\mapsto t])\theta \equiv a[a\mapsto t\theta]$. Using ($\to\mathbf{a}$) we obtain $\Delta' \vdash a[a\mapsto t\theta] \to t\theta$. The result follows.
- The case ($\to\#$). Suppose $\Delta \vdash a\#r$ therefore $\Delta \vdash r[a\mapsto t] \to r$. By definition $r[a\mapsto t] \equiv (\lambda a.r)t$ therefore $(r[a\mapsto t])\theta \equiv r\theta[a\mapsto t\theta]$. By Lemma 3.3.8 we have $\Delta' \vdash a\#r\theta$. Using ($\to\#$) we obtain $\Delta' \vdash (r\theta)[a\mapsto t\theta] \to r\theta$. The result follows.

- The case ($\rightarrow$**rs1**). Suppose $\Delta \vdash a\#s$ therefore $\Delta \vdash (rs)[a\mapsto t] \rightarrow (r[a\mapsto t])s$. By Lemma 3.3.8 we have $\Delta' \vdash a\#s\theta$. By definition $(rs)[a\mapsto t] \equiv (\lambda a.rs)t$ therefore $((rs)[a\mapsto t])\theta \equiv ((r\theta)(s\theta))[a\mapsto t\theta]$. Using ($\rightarrow$**rs1**) we obtain $\Delta' \vdash ((r\theta)(s\theta))[a\mapsto t\theta] \rightarrow ((r\theta)[a\mapsto t\theta])(s\theta)$. As $((r\theta)[a\mapsto t\theta])(s\theta) \equiv ((r[a\mapsto t])s)\theta$, the result follows.
- The case ($\rightarrow$**rs2**). Suppose $level(r) = 1$ therefore $\Delta \vdash (rs)[a\mapsto t] \rightarrow (r[a\mapsto t])(s[a\mapsto t])$. By Lemma 3.4.5 we have $level(r\theta) = 1$. By definition $(rs)[a\mapsto t] \equiv (\lambda a.rs)t$ therefore $((rs)[a\mapsto t])\theta \equiv ((r\theta)(s\theta))[a\mapsto t\theta]$. Using ($\rightarrow$**rs2**) we obtain $\Delta' \vdash ((r\theta)(s\theta))[a\mapsto t\theta] \rightarrow ((r\theta)[a\mapsto t\theta])((s\theta)[a\mapsto t\theta])$. As $((r\theta)[a\mapsto t\theta])((s\theta)[a\mapsto t\theta]) \equiv ((r[a\mapsto t])(s[a\mapsto t]))\theta$, the result follows.
- The case ($\rightarrow\lambda$**b**). Suppose $\Delta \vdash b\#t$ therefore $\Delta \vdash (\lambda b.s)[a\mapsto t] \rightarrow \lambda b.(s[a\mapsto t])$. By Lemma 3.3.8 we have $\Delta' \vdash b\#t\theta$. By definition $(\lambda b.s)[a\mapsto t] \equiv (\lambda a.(\lambda b.s))t$ therefore $((\lambda b.s)[a\mapsto t])\theta \equiv (\lambda b.(s\theta))[a\mapsto t\theta]$. Using ($\rightarrow\lambda$**b**) we obtain $\Delta' \vdash (\lambda b.(s\theta))[a\mapsto t\theta] \rightarrow \lambda b.((s\theta)[a\mapsto t\theta])$. As $\lambda b.((s\theta)[a\mapsto t\theta]) \equiv \lambda b.(s[a\mapsto t])\theta$, the result follows.
- The case ($\rightarrow\lambda$**X**). Suppose $X \notin rng(\theta)$, which can be guaranteed by renaming. We have $((\lambda X.r)[a\mapsto t])\theta \equiv \lambda X.(r\theta)[a\mapsto t\theta]$. Using ($\rightarrow\lambda$**X**) we obtain $\Delta' \vdash \lambda X.(r\theta)[a\mapsto t\theta] \rightarrow \lambda X.(r\theta[a\mapsto t\theta])$. As $\lambda X.(r\theta[a\mapsto t\theta]) \equiv \lambda X.(r[a\mapsto t])\theta$, the result follows.
- The case ($\rightarrow\beta$). Suppose $X \notin rng(\theta)$, which can be guaranteed by renaming. We have $((\lambda X.r)t)\theta \equiv (\lambda X.(r\theta))t\theta$. Using ($\rightarrow\beta$) we obtain $\Delta' \vdash (\lambda X.(r\theta))t\theta \rightarrow (r\theta)[X{:=}t\theta]$. By Lemma 3.2.15 we have $(r\theta)[X{:=}t\theta] \equiv (r[X{:=}t])\theta$. The result follows.
- The case ($\triangleright\lambda$**a**). Suppose $\Delta \vdash r \rightarrow s$. By inductive hypothesis $\Delta' \vdash r\theta \rightarrow s\theta$. Using ($\triangleright\lambda$**a**) we obtain $\Delta' \vdash \lambda a.(r\theta) \rightarrow \lambda a.(s\theta)$. As $\lambda a.(r\theta) \equiv (\lambda a.r)\theta$, the result follows.
- The case ($\triangleright$**rs1**). Suppose $\Delta \vdash r \rightarrow t$. By inductive hypothesis $\Delta' \vdash r\theta \rightarrow t\theta$. Using ($\triangleright$**rs1**) we obtain $\Delta' \vdash (r\theta)(s\theta) \rightarrow (t\theta)(s\theta)$. As $(r\theta)(s\theta) \equiv (rs)\theta$, the result follows.
- The case ($\triangleright$**rs2**). Suppose $\Delta \vdash s \rightarrow t$. By inductive hypothesis $\Delta' \vdash s\theta \rightarrow t\theta$. Using ($\triangleright$**rs2**) we obtain $\Delta' \vdash (r\theta)(s\theta) \rightarrow (r\theta)(t\theta)$. As $(r\theta)(s\theta) \equiv (rs)\theta$, the result follows.
- The case ($\triangleright\lambda$**X**). Suppose $\Delta \vdash r \rightarrow s$, $X \notin \Delta$, and also $X \notin rng(\theta)$, picking fresh X if necessary. By inductive hypothesis $\Delta' \vdash r\theta \rightarrow s\theta$. Using ($\triangleright\lambda$**X**) we obtain $\Delta' \vdash \lambda X.(r\theta) \rightarrow \lambda X.(s\theta)$. As $\lambda X.(r\theta) \equiv (\lambda X.r)\theta$, the result follows.
- The case ($\triangleright\alpha$). Suppose $\Delta \vdash r \rightarrow s$, $\Delta \vdash a\#r$ and $\Delta \vdash b\#r$. By inductive hypothesis $\Delta' \vdash r\theta \rightarrow s\theta$. By Lemma 3.3.8 we have $\Delta' \vdash a\#r\theta$ and $\Delta' \vdash b\#r\theta$. Using ($\triangleright\alpha$) we obtain $\Delta' \vdash (b\ a)\cdot r\theta \rightarrow s\theta$. By Lemma 3.2.16 we have $\Delta' \vdash ((b\ a)\cdot r)\theta \rightarrow s\theta$. The result follows.

⊠

Lemma 3.4.7 is a standard equivariance result for the level of a term, and used in the proof of Lemma 3.4.8.

Lemma 3.4.7: $level(r) = level(\pi\cdot r)$

Proof. By induction on r, using Lemma 3.2.13. See Appendix A. ⊠

Lemma 3.4.8 is our third correctness result, stating that the reduction relation is equivariant.

Lemma 3.4.8: If $\Delta \vdash r \rightarrow s$ then $\Delta \vdash \pi\cdot r \rightarrow \pi\cdot s$.

Proof. By induction on the derivation of $\Delta \vdash r \to s$.

- The case $(\to\mathbf{a})$. Suppose $\Delta \vdash a[a{\mapsto}t] \to t$. By definition $a[a{\mapsto}t] \equiv (\lambda a.a)t$. By Definition 3.2.4 we have $\pi{\cdot}(a[a{\mapsto}t]) \equiv \pi(a)[\pi(a){\mapsto}\pi{\cdot}t]$. The result follows.
- The case $(\to\#)$. Suppose $\Delta \vdash a\#r$ therefore $\Delta \vdash r[a{\mapsto}t] \to r$. By definition $r[a{\mapsto}t] \equiv (\lambda a.r)t$. By Definition 3.2.4 we have $\pi{\cdot}(r[a{\mapsto}t]) \equiv (\pi{\cdot}r)[\pi(a){\mapsto}\pi{\cdot}t]$. By Lemma 3.3.9 we have $\Delta \vdash \pi(a)\#\pi{\cdot}r$. The result follows.
- The case $(\to\mathbf{rs1})$. Suppose $\Delta \vdash a\#s$ therefore $\Delta \vdash (rs)[a{\mapsto}t] \to (r[a{\mapsto}t])s$. By definition $(rs)[a{\mapsto}t] \equiv (\lambda a.(rs))t$. By Definition 3.2.4 we have $\pi{\cdot}((rs)[a{\mapsto}t]) \equiv ((\pi{\cdot}r)(\pi{\cdot}s))[\pi(a){\mapsto}\pi{\cdot}t]$. By Lemma 3.3.9 we have $\Delta \vdash \pi(a)\#\pi{\cdot}s$. The result follows.
- The case $(\to\mathbf{rs2})$. Suppose $\mathit{level}(r) = 1$ therefore $\Delta \vdash (rs)[a{\mapsto}t] \to (r[a{\mapsto}t])(s[a{\mapsto}t])$. By definition $(rs)[a{\mapsto}t] \equiv (\lambda a.(rs))t$. By Definition 3.2.4 we have $\pi{\cdot}((rs)[a{\mapsto}t]) \equiv ((\pi{\cdot}r)(\pi{\cdot}s))[\pi(a){\mapsto}\pi{\cdot}t]$. By Lemma 3.4.7 we have $\mathit{level}(\pi{\cdot}r) = 1$. The result follows.
- The case $(\to\lambda\mathbf{b})$. Suppose $\Delta \vdash b\#t$ therefore $\Delta \vdash (\lambda b.s)[a{\mapsto}t] \to \lambda b.(s[a{\mapsto}t])$. By definition $(\lambda b.s)[a{\mapsto}t] \equiv (\lambda a.(\lambda b.s))t$. By Definition 3.2.4 we have $\pi{\cdot}((\lambda b.s)[a{\mapsto}t]) \equiv (\lambda\pi(b).(\pi{\cdot}s))[\pi(a){\mapsto}(\pi{\cdot}t)]$. By Lemma 3.3.9 we have $\Delta \vdash \pi(b)\#\pi{\cdot}t$. The result follows.
- The case $(\to\lambda\mathbf{X})$. Suppose $X \notin fV(t)$, which can be guaranteed, therefore $\Delta \vdash (\lambda X.r)[a{\mapsto}t] \to \lambda X.(r[a{\mapsto}t])$. By definition $(\lambda X.r)[a{\mapsto}t] \equiv (\lambda a.(\lambda X.r))t$. By Definition 3.2.4 we have $\pi{\cdot}(\lambda a.(\lambda X.r))t \equiv \lambda\pi(a).(\lambda X.(\pi{\cdot}r[X{:=}\pi^{-1}{\cdot}X]))(\pi{\cdot}t)$. By definition $\lambda\pi(a).(\lambda X.(\pi{\cdot}r[X{:=}\pi^{-1}{\cdot}X]))(\pi{\cdot}t) \equiv \lambda X.(\pi{\cdot}r[X{:=}\pi^{-1}{\cdot}X])[\pi(a){\mapsto}(\pi{\cdot}t)]$. Using $(\to\lambda\mathbf{X})$ we obtain $\lambda X.(\pi{\cdot}r[X{:=}\pi^{-1}{\cdot}X][\pi(a){\mapsto}(\pi{\cdot}t)])$. By Lemma 3.2.6 we have $X \notin fV(\pi{\cdot}t)$ therefore $\lambda X.(\pi{\cdot}r[X{:=}\pi^{-1}{\cdot}X][\pi(a){\mapsto}(\pi{\cdot}t)]) \equiv \lambda X.(\pi{\cdot}r[X{:=}\pi^{-1}{\cdot}X][\pi(a){\mapsto}(\pi{\cdot}t)[X{:=}\pi^{-1}{\cdot}X]])$. It is a fact that $\lambda X.(\pi{\cdot}r[X{:=}\pi^{-1}{\cdot}X][\pi(a){\mapsto}(\pi{\cdot}t)[X{:=}\pi^{-1}{\cdot}X]]) \equiv \lambda X.(\pi{\cdot}r[\pi(a){\mapsto}(\pi{\cdot}t)][X{:=}\pi^{-1}{\cdot}X])$. By Lemma 3.2.16 we have $\lambda X.(\pi{\cdot}r[\pi(a){\mapsto}(\pi{\cdot}t)][X{:=}\pi^{-1}{\cdot}X]) \equiv \lambda X.((\pi{\cdot}r)[\pi(a){\mapsto}(\pi{\cdot}t)][X{:=}\pi^{-1}{\cdot}X])$. By Definition 3.2.4 we have $\lambda X.((\pi{\cdot}r)[\pi(a){\mapsto}(\pi{\cdot}t)][X{:=}\pi^{-1}{\cdot}X]) \equiv \pi{\cdot}\lambda X.(r[a{\mapsto}t])$. The result follows.
- The case $(\to\beta)$. Suppose $\Delta \vdash (\lambda X.r)t \to r[X{:=}t]$. By Definition 3.2.4 we have $\pi{\cdot}((\lambda X.r)t) \equiv (\lambda X.(\pi{\cdot}r[X{:=}\pi^{-1}{\cdot}X]))(\pi{\cdot}t)$. By Lemma 3.2.16 we have $(\lambda X.((\pi{\cdot}r)[X{:=}\pi^{-1}{\cdot}X]))(\pi{\cdot}t)$. Using $(\to\beta)$ we obtain $(\pi{\cdot}r)[X{:=}\pi^{-1}{\cdot}X][X{:=}\pi{\cdot}t]$. By Lemma 3.2.10 we have $(\pi{\cdot}r)[X{:=}(\pi^{-1}{\circ}\pi){\cdot}t]$ therefore $(\pi{\cdot}r)[X{:=}t]$. By Lemma 3.2.16 we have $\pi{\cdot}(r[X{:=}t])$, as required.
- The case $(\rhd\lambda\mathbf{a})$. Suppose $\Delta \vdash r \to s$. By inductive hypothesis $\Delta \vdash \pi{\cdot}r \to \pi{\cdot}s$. Using $(\rhd\lambda\mathbf{a})$ we obtain $\Delta \vdash \lambda\pi(a).(\pi{\cdot}r) \to \lambda\pi(a).(\pi{\cdot}s)$. As $\lambda\pi(a).(\pi{\cdot}r) \equiv \pi{\cdot}\lambda a.r$, the result follows.
- The case $(\rhd\mathbf{rs1})$. Suppose $\Delta \vdash r \to t$. By inductive hypothesis $\Delta \vdash \pi{\cdot}r \to \pi{\cdot}t$. Using $(\rhd\mathbf{rs1})$ we obtain $\Delta \vdash (\pi{\cdot}r)(\pi{\cdot}s) \to (\pi{\cdot}t)(\pi{\cdot}s)$. As $(\pi{\cdot}r)(\pi{\cdot}s) \equiv \pi{\cdot}rs$, the result follows.
- The case $(\rhd\mathbf{rs2})$. Suppose $\Delta \vdash s \to u$. By inductive hypothesis $\Delta \vdash \pi{\cdot}s \to \pi{\cdot}u$. Using $(\rhd\mathbf{rs2})$ we obtain $\Delta \vdash (\pi{\cdot}r)(\pi{\cdot}s) \to (\pi{\cdot}r)(\pi{\cdot}u)$. As $(\pi{\cdot}r)(\pi{\cdot}s) \equiv \pi{\cdot}rs$, the result follows.
- The case $(\rhd\lambda\mathbf{X})$. Suppose $\Delta \vdash r \to s$ with $X \notin \Delta$. By inductive hypothesis $\Delta \vdash \pi{\cdot}r \to \pi{\cdot}s$. By Lemma 3.4.6 we have $\Delta[X{:=}\pi^{-1}{\cdot}X] \vdash (\pi{\cdot}r)[X{:=}\pi^{-1}{\cdot}X] \to (\pi{\cdot}s)[X{:=}\pi^{-1}{\cdot}X]$. As $X \notin \Delta$ it is the case that $\Delta[X{:=}\pi^{-1}{\cdot}X] = \Delta$. By Lemma 3.2.16 we have $\Delta \vdash \pi{\cdot}r[X{:=}\pi^{-1}{\cdot}X] \to \pi{\cdot}s[X{:=}\pi^{-1}{\cdot}X]$. Using $(\rhd\lambda\mathbf{X})$ we obtain $\Delta \vdash \lambda X.(\pi{\cdot}r[X{:=}\pi^{-1}{\cdot}X]) \to \lambda X.(\pi{\cdot}s[X{:=}\pi^{-1}{\cdot}X])$. As $\lambda X.(\pi{\cdot}r[X{:=}\pi^{-1}{\cdot}X]) \equiv \pi{\cdot}(\lambda X.r)$, the result follows.
- The case $(\rhd\alpha)$. Suppose $\Delta \vdash r \to s$, $\Delta \vdash a\#r$ and $\Delta \vdash b\#r$. By inductive hypothesis $\Delta \vdash \pi{\cdot}r \to \pi{\cdot}s$. By Lemma 3.3.9 we have $\Delta \vdash \pi(a)\#\pi{\cdot}r$ and $\Delta \vdash \pi(b)\#\pi{\cdot}r$. Using $(\rhd\alpha)$

we obtain $\Delta \vdash ((\pi(b)\ \pi(a))\cdot(\pi\cdot r)) \to \pi\cdot s$. By Lemma 3.2.13 we have $\Delta \vdash ((\pi(b)\ \pi(a))\circ\pi)\cdot r \to \pi\cdot s$. By elementary properties of permutations we have $(\pi(b)\ \pi(a))\circ\pi = \pi\circ(b\ a)$ therefore $\Delta \vdash (\pi\circ(b\ a))\cdot r \to \pi\cdot s$. By Lemma 3.2.13 we have $\Delta \vdash \pi\cdot((b\ a)\cdot r) \to \pi\cdot s$. The result follows.

⊠

3.4.1 Confluence of level one reductions

This Subsection presents a proof of confluence of the 'level one' fragment of the two-level λ-calculus.

Definition 3.4.13 presents a canonical form for terms. Theorem 3.4.22 states that all terms eventually reduce to their canonical form. Lemma 3.4.23 states that if r reduces to s then the canonical form of s reduces eventually to the canonical form of r. Theorem 3.4.24, the confluence result for the level one fragment of the calculus, and the main result of this Subsection, essentially follows as a corollary of the previous two results.

Definition 3.4.9: Let (**level1**) be the set $\{(\to\mathbf{a}), (\to\#), (\to\mathbf{rs1}), (\to\mathbf{rs2}), (\to\lambda\mathbf{a}), (\to\lambda\mathbf{X})\}$ from Figure 3.3. Let $\Delta \vdash r \overset{(\mathbf{level1})}{\to} s$ be the least congruence closed under the rules in (**level1**).

Definition 3.4.10: Say that Δ^+ is a **fresh extension** of Δ whenever $\Delta^+ = \Delta \cup \Delta'$, Δ and Δ' are disjoint, and for all $a\#X \in \Delta'$ we have $a \notin \Delta$ but $X \in \Delta$.

To reduce a term fully, we may need to introduce some fresh atoms into our freshness context. For instance, consider an attempt to reduce $\Delta \vdash (\lambda a.r)[b\mapsto u]$ where $\Delta \nvdash a\#u$. This *should* reduce as one would expect, $\Delta \vdash (\lambda a.r)[b\mapsto u] \to \lambda a.(r[b\mapsto u])$, but is being prevented from doing so by 'insufficient freshness' in Δ. Without reductions of this sort, confluence fails.

To restore confluence, we introduce the notion of fresh extensions in Definition 3.4.10. The purpose of these extensions is to enrich a given freshness context with additional fresh atoms. Using these fresh atoms, we can rename abstracted atoms with something suitably fresh, guaranteeing that no reduction paths become needlessly stuck due to insufficient freshness.

Definition 3.4.11: Choose some arbitrary yet fixed ordering on atoms. If S is a finite set of atoms say 'for the first atom not in S' to mean 'for the least atom in our arbitrary yet fixed ordering, that is not in S'.

Definition 3.4.12: For a given Δ define a **level one substitution action** $r[a{:=}t]$ as follows:

- $r[a{:=}t] \equiv r$ provided $\Delta \vdash a\#r$
- $a[a{:=}t] \equiv t$
- $(\pi\cdot X)[a{:=}t] \equiv (\pi\cdot X)[a\mapsto t]$
- $(rs)[a{:=}t] \equiv (r[a{:=}t])(s[a{:=}t])$ provided $level(r) = 1$
- $(rs)[a{:=}t] \equiv (r[a{:=}t])s$ provided $\Delta \vdash a\#s$
- $(rs)[a{:=}t] \equiv (rs)[a\mapsto t]$ provided $\Delta \nvdash a\#s$ and $level(r) = 2$

- $(\lambda b.s)[a{:=}t] \equiv \lambda b.(s[a{:=}t])$ provided $\Delta \vdash b\#t$
- $(\lambda b.s)[a{:=}t] \equiv \lambda c.(((c\ b)\cdot s)[a{:=}t])$ provided $\Delta \nvdash b\#t$. Here c is the first atom, distinct from a and b, not mentioned in s or t, and such that $\Delta \vdash c\#s$ and $\Delta \vdash c\#t$, if such an atom exists.
- $(\lambda b.s)[a{:=}t] \equiv (\lambda b.s)[a{\mapsto}t]$ provided $\Delta \nvdash b\#t$ and no fresh atom exists.
- $(\lambda X.r)[a{:=}t] \equiv \lambda X.(r[a{:=}t])$ renaming λX if needed so as to ensure $X \notin fV(t)$.

Definition 3.4.13: For a given Δ define a **canonical form** $r^\star$ as follows:

$$a^\star \equiv a \qquad \mathsf{c}^\star \equiv \mathsf{c} \qquad (\pi\cdot X)^\star \equiv \pi\cdot X \qquad (\lambda a.r)^\star \equiv \lambda a.(r^\star) \qquad (\lambda X.r)^\star \equiv \lambda X.(r^\star) \quad (\text{if } X \notin \Delta)$$
$$(r[a{\mapsto}t])^\star \equiv r^\star[a{:=}t^\star] \qquad ((\lambda X.r)t)^\star \equiv r^\star[X{:=}t^\star] \qquad (rs)^\star \equiv r^\star s^\star$$

Here, earlier rules have priority, read left-to-right and top-to-bottom. We can always rename λX to ensure $X \notin \Delta$.

Intuitively $r^\star$ is a canonical form of r, pushing all unreduced β-reducts $r[a{\mapsto}t]$ as far as possible inside r, and collecting all garbage.[4]

The strategy we use for proving confluence is vaguely reminiscent of a strategy developed by Takahashi [Tak95]. We define a canonical form (Definition 3.4.13). In Takahashi's terminology, the canonical form is referred to as a *complete development*. We then prove that all terms eventually reduce to their canonical form (Theorem 3.4.22). Finally, we prove that a term r reducing to another term s implies the canonical form of s reduces to the canonical form of r (Lemma 3.4.23). As an almost immediate corollary, we obtain the confluence of the calculus (Theorem 3.4.24).

It is important to note that the canonical form of Definition 3.4.13 is *not* necessarily a normal form. The calculus is untyped and normal forms need not exist. For example, consider the following pair of reduction paths, both of whom fail to terminate like their counterparts in the untyped λ-calculus:

$$\Delta \vdash (\lambda a.aa)(\lambda a.aa) \to^* (\lambda a.aa)(\lambda a.aa) \to^* (\lambda a.aa)(\lambda a.aa) \to^* \ldots$$
$$\Delta \vdash (\lambda X.XX)(\lambda X.XX) \to^* (\lambda X.XX)(\lambda X.XX) \to^* (\lambda X.XX)(\lambda X.XX) \to^* \ldots$$

Definition 3.4.14: Call $\Delta \vdash -\triangleright-$ **reflexive** when $\Delta \vdash r \triangleright r$ always. Call $\Delta \vdash -\triangleright-$ **transitive** when $\Delta \vdash r \triangleright s$ and $\Delta \vdash s \triangleright t$ imply $\Delta \vdash r \triangleright t$ always. Write $\Delta \vdash - \triangleright^* -$ for the least transitive reflexive relation containing $\Delta \vdash - \triangleright -$.

The following six lemmas—Lemmas 3.4.15 through to 3.4.21—are technical results used in the proof of Theorem 3.4.22.

Lemma 3.4.15: For every Δ, r, a, and t there exists a Δ^+ such that $\Delta^+ \vdash r[a{\mapsto}t] \to^* r[a{:=}t]$.

Proof. By induction on the depth of r.

[4]By 'garbage' we mean terms of the form $r[a{\mapsto}t]$ where $\Delta \vdash a\#r$.

- The case a. By Definition 3.4.12 we have $a[a{:=}t] \equiv t$ for every Δ. Using $(\to\mathbf{a})$ we obtain $\Delta \vdash a[a{\mapsto}t] \to t$ for every Δ. We take $\Delta^+ = \Delta$. The result follows.
- The case b. By Definition 3.4.12 we have $b[a{:=}t] \equiv b$ for every Δ. Using $(\to\#)$ we obtain $\Delta \vdash b[a{\mapsto}t] \to b$ for every Δ. We take $\Delta^+ = \Delta$. The result follows.
- The case c. By Definition 3.4.12 we have $\mathsf{c}[a{:=}t] \equiv \mathsf{c}$ for every Δ. Using $(\to\#)$ we obtain $\Delta \vdash \mathsf{c}[a{\mapsto}t] \to \mathsf{c}$ for every Δ. We take $\Delta^+ = \Delta$. The result follows.
- The case $\pi{\cdot}X$. There are two cases:
  - The case $\Delta \vdash a\#\pi{\cdot}X$. By Definition 3.4.12 we have $(\pi{\cdot}X)[a{:=}t] \equiv \pi{\cdot}X$. Using $(\to\#)$ we obtain $\Delta \vdash (\pi{\cdot}X)[a{\mapsto}t] \to \pi{\cdot}X$. We take $\Delta^+ = \Delta$. The result follows.
  - The case $\Delta \nvdash a\#\pi{\cdot}X$. By Definition 3.4.12 we have $(\pi{\cdot}X)[a{:=}t] \equiv (\pi{\cdot}X)[a{\mapsto}t]$. By Definition 3.4.14 we have $\Delta \vdash (\pi{\cdot}X)[a{\mapsto}t] \to^* (\pi{\cdot}X)[a{\mapsto}t]$ for every Δ. We take $\Delta^+ = \Delta$. The result follows.
- The case rs. There are three cases:
  - The case $\Delta \vdash a\#s$. By Definition 3.4.12 we have $(rs)[a{:=}t] \equiv (r[a{:=}t])s$. By inductive hypothesis Δ'^+ exists such that $\Delta'^+ \vdash r[a{\mapsto}t] \to^* r[a{:=}t]$. By Definition 3.4.14 we have $\Delta \vdash s \to^* s$ for every Δ. Using $(\to\mathbf{rs2})$ we obtain $\Delta \vdash (rs)[a{\mapsto}t] \to (r[a{\mapsto}t])s$. We take $\Delta^+ = \Delta'^+$. The result follows.
  - The case $\mathit{level}(r) = 1$. By Definition 3.4.12 we have $(rs)[a{:=}t] \equiv (r[a{:=}t])(s[a{:=}t])$. By inductive hypothesis there exists some Δ'^+ such that $\Delta'^+ \vdash r[a{\mapsto}t] \to^* r[a{:=}t]$ and some Δ''^+ such that $\Delta''^+ \vdash s[a{\mapsto}t] \to^* s[a{:=}t]$. Using $(\to\mathbf{rs1})$ we obtain $\Delta \vdash (rs)[a{\mapsto} t] \to r[a{\mapsto}t](s[a{\mapsto}t])$. We take $\Delta^+ = \Delta'^+ \cup \Delta''^+$. The result follows.
  - The case $\Delta \nvdash a\#s$ and $\mathit{level}(r) = 2$. By Definition 3.4.12 we have $(rs)[a{:=}t] \equiv (rs)[a{\mapsto}t]$. By Definition 3.4.14 we have $\Delta \vdash (rs)[a{\mapsto}t] \to^* (rs)[a{\mapsto}t]$ for every Δ. We take $\Delta^+ = \Delta$. The result follows.
- The case $\lambda a.r$. For every Δ, we have $(\lambda a.r)[a{:=}t] \equiv \lambda a.r$ by Definition 3.4.12. Similarly, for every Δ, we have $\Delta \vdash (\lambda a.r)[a{\mapsto}t] \to \lambda a.r$ by $(\to\#)$. We take $\Delta^+ = \Delta$. The result follows.
- The case $\lambda b.s$. There are three cases:
  - The case $\Delta \vdash a\#\lambda b.s$. By Definition 3.4.12 we have $\Delta \vdash (\lambda b.s)[a{:=}t] \equiv \lambda b.s$. Using $(\to\#)$ we obtain $\Delta \vdash (\lambda b.s)[a{\mapsto}t] \to \lambda b.s$. We take $\Delta^+ = \Delta$. The result follows.
  - The case $\Delta \vdash b\#t$. By Definition 3.4.12 we have $\Delta \vdash (\lambda b.s)[a{:=}t] \equiv \lambda b.(s[a{:=}t])$. Using $(\to\lambda\mathbf{b})$ we obtain $\Delta \vdash (\lambda b.s)[a{\mapsto}t] \to \lambda b.(s[a{\mapsto}t])$. By inductive hypothesis there exists an Δ^+ such that $\Delta^+ \vdash s[a{\mapsto}t] \to^* s[a{:=}t]$. Using $(\rhd\lambda\mathbf{a})$ we obtain $\Delta^+ \vdash \lambda b.(s[a{\mapsto}t]) \to^* \lambda b.(s[a{:=}t])$. The result follows.
  - The case $\Delta \nvdash b\#s$. Take Δ'^+ to be Δ extended with the first fresh c, according to our arbitrary yet fixed ordering, such that $\Delta'^+ \vdash c\#s$ and $\Delta'^+ \vdash c\#t$. By Definition 3.4.12 we have $(\lambda b.s)[a{:=}t] \equiv \lambda c.(((c\ b){\cdot}s)[a{:=}t])$, calculated for Δ'^+. By Lemma 3.2.9 we have $\mathit{depth}((b\ a){\cdot}s) = \mathit{depth}(s)$. By inductive hypothesis Δ''^+ exists such that $\Delta''^+ \vdash ((c\ b){\cdot} r)[a{\mapsto}t] \to^* ((c\ b){\cdot}s)[a{:=}t]$. Using $(\rhd\alpha)$ we obtain $\Delta'^+ \vdash (\lambda b.s)[a{\mapsto}t] \to \lambda c.((c\ b){\cdot} s)[a{\mapsto}t]$. Using $(\to\lambda\mathbf{a})$ we obtain $\Delta'^+ \vdash (\lambda b.s)[a{\mapsto}t] \to \lambda c.(((c\ b){\cdot}s)[a{\mapsto}t])$. We take $\Delta^+ = \Delta'^+ \cup \Delta''^+$. The result follows.

- The case $\lambda X.r$. Suppose $X \notin \Delta$ and $X \notin fV(t)$, which can always be guaranteed by renaming. By Definition 3.4.12 we have $(\lambda X.r)[a{:=}t] \equiv \lambda X.(r[a{:=}t])$. Using $(\rightarrow\lambda\mathbf{X})$ we obtain $\Delta \vdash (\lambda X.r)[a{\mapsto}t] \rightarrow \lambda X.(r[a{\mapsto}t])$. By inductive hypothesis there exists Δ^+ such that $\Delta^+ \vdash r[a{\mapsto}t] \rightarrow^* r[a{:=}t]$. Using $(\triangleright\lambda\mathbf{X})$ we obtain $\Delta^+ \vdash \lambda X.(r[a{\mapsto}t]) \rightarrow^* \lambda X.(r[a{:=}t])$. The result follows.

⊠

Lemma 3.4.16: Fix Δ. Then for canonical forms calculated for Δ, we have:

1. $(\pi{\cdot}r)^\star \equiv \pi{\cdot}r^\star$.
2. If $(\pi{\cdot}t)^\star \equiv \pi{\cdot}t^\star$ for all π then $(r[X{:=}t])^\star \equiv r^\star[X{:=}t^\star]$.

Proof. By mutual induction on $depth(r)$. We handle the claims separately. The first claim:

- The case a. By Definition 3.2.4 we have $\pi{\cdot}a \equiv \pi(a)$. By Definition 3.4.13 we have $(\pi(a))^\star \equiv \pi(a)$ and $a^\star \equiv a$. The result follows.
- The case c. By Definition 3.2.4 we have $\pi{\cdot}\mathsf{c} \equiv \mathsf{c}$. By Definition 3.4.13 we have $\mathsf{c}^\star \equiv \mathsf{c}$. The result follows.
- The case $\pi'{\cdot}X$. By Definition 3.2.4 we have $\pi{\cdot}(\pi'{\cdot}X) \equiv (\pi\circ\pi'){\cdot}X$. By Definition 3.4.13 we have $((\pi\circ\pi'){\cdot}X)^\star \equiv (\pi\circ\pi'){\cdot}X$ and $(\pi'{\cdot}X)^\star \equiv \pi'{\cdot}X$. The result follows.
- The case rs. There are two cases:
  - The case $(\lambda X.r)t$. By Definition 3.2.4 we have $\pi{\cdot}((\lambda X.r)t) \equiv (\lambda X.((\pi{\cdot}r)[X{:=}\pi^{-1}{\cdot}X]))(\pi{\cdot}t)$. By Definition 3.4.13 we have $((\lambda X.((\pi{\cdot}r)[X{:=}\pi^{-1}{\cdot}X]))(\pi{\cdot}t))^\star \equiv ((\pi{\cdot}r)[X{:=}\pi^{-1}{\cdot}X])^\star[X{:=}(\pi{\cdot}t)^\star]$. By the second claim we have $((\pi{\cdot}r)[X{:=}\pi^{-1}{\cdot}X])^\star[X{:=}t^\star] \equiv (\pi{\cdot}r)^\star[X{:=}(\pi^{-1}{\cdot}X)^\star][X{:=}(\pi{\cdot}t)^\star]$. By Definition 3.4.13 we have $(\pi{\cdot}r)^\star[X{:=}\pi^{-1}{\cdot}X^\star][X{:=}t^\star] \equiv (\pi{\cdot}r)^\star[X{:=}\pi^{-1}{\cdot}X][X{:=}(\pi{\cdot}t)^\star]$. By inductive hypothesis $(\pi{\cdot}r)^\star[X{:=}\pi^{-1}{\cdot}X][X{:=}(\pi{\cdot}t)^\star] \equiv (\pi{\cdot}r^\star)[X{:=}\pi^{-1}{\cdot}X][X{:=}(\pi{\cdot}t^\star)]$. By Lemma 3.2.10 we have $(\pi{\cdot}r^\star)[X{:=}\pi^{-1}{\cdot}X][X{:=}(\pi{\cdot}t^\star)] \equiv (\pi{\cdot}r^\star)[X{:=}t^\star]$. By Lemma 3.2.16 we have $(\pi{\cdot}r^\star)[X{:=}t^\star] \equiv \pi{\cdot}(r^\star[X{:=}t^\star]$. By Definition 3.4.13 we have $\pi{\cdot}(r^\star[X{:=}t^\star] \equiv \pi{\cdot}((\lambda X.r)t)^\star$. The result follows.
  - The case rs. By Definition 3.2.4 we have $\pi{\cdot}(rs) \equiv (\pi{\cdot}r)(\pi{\cdot}s)$. By Definition 3.4.13 we have $(\pi{\cdot}r)(\pi{\cdot}s)^\star \equiv (\pi{\cdot}r)^\star(\pi{\cdot}s)^\star$. By inductive hypotheses we have $(\pi{\cdot}r)^\star(\pi{\cdot}s)^\star \equiv (\pi{\cdot}r^\star)(\pi{\cdot}s^\star)$. By Definition 3.2.4 we have $(\pi{\cdot}r^\star)(\pi{\cdot}s^\star) \equiv \pi{\cdot}(r^\star s^\star)$. By Definition 3.4.13 we have $\pi{\cdot}(r^\star s^\star) \equiv \pi{\cdot}(rs)^\star$. The result follows.
- The case $\lambda a.r$. By Definition 3.2.4 we have $\pi{\cdot}\lambda a.r \equiv \lambda\pi(a).(\pi{\cdot}r)$. By Definition 3.4.13 we have $(\lambda\pi(a).(\pi{\cdot}r))^\star \equiv \lambda\pi(a).(\pi{\cdot}r)^\star$. By inductive hypothesis $\lambda\pi(a).(\pi{\cdot}r)^\star \equiv \lambda\pi(a).(\pi{\cdot}r^\star)$. By Definition 3.2.4 we have $\lambda\pi(a).(\pi{\cdot}r^\star) \equiv \pi{\cdot}(\lambda a.r^\star)$. By Definition 3.4.13 we have $\pi{\cdot}(\lambda a.r^\star) \equiv \pi{\cdot}(\lambda a.r)^\star$. The result follows.
- The case $\lambda X.r$. By Definition 3.2.4 we have $\pi{\cdot}\lambda X.r \equiv \lambda X.((\pi{\cdot}r)[X{:=}\pi^{-1}{\cdot}X])$. By Definition 3.4.13 we have $(\lambda X.((\pi{\cdot}r)[X{:=}\pi^{-1}{\cdot}X]))^\star \equiv \lambda X.((\pi{\cdot}r)[X{:=}\pi^{-1}{\cdot}X])^\star$. By the second claim we have $\lambda X.((\pi{\cdot}r)[X{:=}\pi^{-1}{\cdot}X])^\star \equiv \lambda X.((\pi{\cdot}r)^\star[X{:=}(\pi^{-1}{\cdot}X)^\star])$. By Definition 3.4.13 we have $\lambda X.((\pi{\cdot}r)^\star[X{:=}(\pi^{-1}{\cdot}X)^\star]) \equiv \lambda X.((\pi{\cdot}r)^\star[X{:=}\pi^{-1}{\cdot}X])$. By inductive hypothesis $\lambda X.((\pi{\cdot}r)^\star[X{:=}\pi^{-1}{\cdot}X]) \equiv \lambda X.((\pi{\cdot}r^\star)[X{:=}\pi^{-1}{\cdot}X])$. By Definition 3.2.4 we have $\lambda X.((\pi{\cdot}r^\star)[X{:=}\pi^{-1}{\cdot}X]) \equiv \pi{\cdot}(\lambda X.r^\star)$. By Definition 3.4.13 we have $\pi{\cdot}(\lambda X.r^\star) \equiv \pi{\cdot}(\lambda X.r)^\star$. The result follows.

The second claim:

- The case a. By Definition 3.2.4 we have $a[X{:=}t] \equiv a$. By Definition 3.4.13 we have $a^\star \equiv a$. The result follows.
- The case c. By Definition 3.2.4 we have $\mathsf{c}[X{:=}t] \equiv \mathsf{c}$. By Definition 3.4.13 we have $\mathsf{c}^\star \equiv \mathsf{c}$. The result follows.
- The case $\pi{\cdot}X$. By Definition 3.2.4 we have $(\pi{\cdot}X)[X{:=}t] \equiv \pi{\cdot}t$. By assumption $(\pi{\cdot}t)^\star \equiv \pi{\cdot}t^\star$. By Definition 3.4.13 we have $(\pi{\cdot}X)^\star[X{:=}t^\star] \equiv \pi{\cdot}t^\star$. The result follows.
- The case $\pi{\cdot}Y$. By Definition 3.2.4 we have $(\pi{\cdot}Y)[X{:=}t] \equiv \pi{\cdot}Y$. By Definition 3.4.13 we have $(\pi{\cdot}Y)^\star \equiv \pi{\cdot}Y$. The result follows.
- The case rs. By Definition 3.2.4 we have $(rs)[X{:=}t] \equiv (r[X{:=}t])(s[X{:=}t])$. By Definition 3.4.13 we have $((r[X{:=}t])(s[X{:=}t]))^\star \equiv (r[X{:=}t])^\star(s[X{:=}t])^\star$. By inductive hypotheses $(r[X{:=}t])^\star(s[X{:=}t])^\star \equiv (r^\star[X{:=}t^\star])(s^\star[X{:=}t^\star])$. By Definition 3.2.4 we have $(r^\star[X{:=}t^\star])(s^\star[X{:=}t^\star]) \equiv (r^\star s^\star)[X{:=}t^\star]$. By Definition 3.4.13 we have $(r^\star s^\star)[X{:=}t^\star] \equiv (rs)^\star[X{:=}t^\star]$. The result follows.
- The case $\lambda a.r$. By Definition 3.2.4 we have $(\lambda a.r)[X{:=}t] \equiv \lambda a.(r[X{:=}t])$. By Definition 3.4.13 we have $(\lambda a.(r[X{:=}t]))^\star \equiv \lambda a.(r[X{:=}t])^\star$. By inductive hypothesis $\lambda a.(r[X{:=}t])^\star \equiv \lambda a.(r^\star[X{:=}t^\star])$. By Definition 3.2.4 we have $\lambda a.(r^\star[X{:=}t^\star]) \equiv (\lambda a.r^\star)[X{:=}t^\star]$. By Definition 3.4.13 we have $(\lambda a.r^\star)[X{:=}t^\star] \equiv (\lambda a.r)^\star[X{:=}t^\star]$. The result follows.
- The case $\lambda X.r$. By Definition 3.2.4 we have $(\lambda X.r)[X{:=}t] \equiv \lambda X.r$. The result follows.
- The case $\lambda Y.s$. Suppose $Y \not\in fV(t)$ which can be guaranteed. By Definition 3.2.4 we have $(\lambda Y.s)[X{:=}t] \equiv \lambda Y.(s[X{:=}t])$. By Definition 3.4.13 we have $(\lambda Y.(s[X{:=}t]))^\star \equiv \lambda Y.(s[X{:=}t])^\star$. By inductive hypothesis $\lambda Y.(s[X{:=}t])^\star \equiv \lambda Y.(s^\star[X{:=}t^\star])$. By Definition 3.2.4 we have $\lambda Y.(s^\star[X{:=}t^\star]) \equiv (\lambda Y.s^\star)[X{:=}t^\star]$. By Definition 3.4.13 we have $(\lambda Y.s^\star)[X{:=}t^\star] \equiv (\lambda Y.s)^\star[X{:=}t^\star]$. The result follows.

⊠

Lemma 3.4.17: If $\Delta \vdash a\#r$ and $\Delta \vdash a\#t$ then $\Delta \vdash a\#r[a{:=}t]$.

Proof. By induction on *depth*(r), using Lemma 3.2.9. See Appendix A. ⊠

Lemma 3.4.18: If $\Delta \vdash a\#r$ and $\Delta \vdash a\#u$ then $\Delta \vdash a\#r[b{:=}u]$.

Proof. By induction on *depth*(r), using Lemma 3.2.9. See Appendix A. ⊠

Intuitively Lemma 3.4.19 states that the canonical form does not introduce any extraneous variable of level one.

Lemma 3.4.19: If $\Delta \vdash a\#r$ then $\Delta \vdash a\#r^\star$ where $r^\star$ is calculated for Δ.

Proof. By induction on the derivation of $\Delta \vdash a\#r$.

- The case $(\#\mathbf{b})$. Suppose $\Delta \vdash a\#b$ by $(\#\mathbf{b})$. By Definition 3.4.13 we have $b^\star \equiv b$. The result follows.

- The case ($\#\mathsf{c}$). Suppose $\Delta \vdash a\#\mathsf{c}$ by ($\#\mathsf{c}$). By Definition 3.4.13 we have $\mathsf{c}^\star \equiv \mathsf{c}$. The result follows.
- The case ($\#\mathbf{X}$). Suppose $\pi^{-1}(a)\#X \in \Delta$ so that $\Delta \vdash a\#\pi{\cdot}X$ by ($\#\mathbf{X}$). By Definition 3.4.13 we have $(\pi{\cdot}X)^\star \equiv \pi{\cdot}X$. The result follows.
- The case ($\#\mathbf{rs}$). There are several cases:
  - The case $(\lambda a.r)t$. Suppose $\Delta \vdash a\#t$ so that $\Delta \vdash a\#(\lambda a.r)t$ by ($\#\mathbf{rs}$). By Definition 3.4.13 we have $((\lambda a.r)t)^\star \equiv r^\star[a{:=}t^\star]$. By inductive hypothesis $\Delta \vdash a\#t^\star$. By Lemma 3.4.17 we have $\Delta \vdash a\#r^\star[a{:=}t^\star]$. The result follows.
  - The case $(\lambda b.s)t$. Suppose $\Delta \vdash a\#\lambda b.s$ and $\Delta \vdash a\#t$ so that $\Delta \vdash a\#(\lambda b.s)t$ by ($\#\mathbf{rs}$). By Definition 3.4.13 we have $((\lambda b.s)t)^\star \equiv s^\star[b{:=}t^\star]$. By inductive hypothesis $\Delta \vdash a\#(\lambda b.s)^\star$ and $\Delta \vdash a\#t^\star$. By Definition 3.4.13 we have $\Delta \vdash a\#\lambda b.(s^\star)$. By ($\#\lambda\mathbf{b}$) we have $\Delta \vdash a\#s^\star$. By Lemma 3.4.18 we have $\Delta \vdash a\#s^\star[a{:=}t^\star]$. The result follows.
  - The case $(\lambda X.r)t$. Suppose $\Delta \vdash a\#\lambda X.r$ and $\Delta \vdash a\#t$ so that $\Delta \vdash a\#(\lambda X.r)t$. By inductive hypotheses $\Delta \vdash a\#\lambda X.r^\star$ and $\Delta \vdash a\#t^\star$. By Definition 3.4.13 we have $\lambda X.r^\star \equiv \lambda X.(r^\star)$. By Definition 3.4.13 we have $((\lambda X.r)t)^\star \equiv r^\star[X{:=}t^\star]$. By Lemma 3.3.8 and calculation we have $\Delta \vdash a\#r^\star[X{:=}t^\star]$. The result follows.
  - All other cases. Suppose $\Delta \vdash a\#r$ and $\Delta \vdash a\#s$. By inductive hypotheses $\Delta \vdash a\#r^\star$ and $\Delta \vdash a\#s^\star$. Using ($\#\mathbf{rs}$) we obtain $\Delta \vdash a\#r^\star s^\star$. By Definition 3.4.13 we have $r^\star s^\star \equiv rs^\star$. The result follows.
- The case ($\#\lambda\mathbf{a}$). By Definition 3.4.13 $\lambda a.r^\star \equiv \lambda a.(r^\star)$. Using ($\#\lambda\mathbf{a}$) we obtain $\Delta \vdash a\#\lambda a.(r^\star)$. The result follows.
- The case ($\#\lambda\mathbf{b}$). Suppose $\Delta \vdash a\#s$. By inductive hypothesis $\Delta \vdash a\#s^\star$. Using ($\#\lambda\mathbf{a}$) we obtain $\Delta \vdash a\#\lambda b.(s^\star)$. By Definition 3.4.13 we have $\lambda b.(s^\star) \equiv (\lambda b.s)^\star$. The result follows.
- The case ($\#\lambda\mathbf{X}$). Suppose $\Delta, a\#X \vdash \pi(a)\#\pi{\cdot}r$. By inductive hypothesis $\Delta, a\#X \vdash \pi(a)\#\pi{\cdot}r^\star$. By Lemma 3.4.16 we have $\pi{\cdot}r^\star \equiv \pi{\cdot}r^\star$. Using ($\#\lambda\mathbf{X}$) we obtain $\Delta \vdash \pi(a)\#\pi{\cdot}\lambda X.(r^\star)$. By Definition 3.4.13 we have $\pi{\cdot}\lambda X.(r^\star) \equiv \pi{\cdot}(\lambda X.r)^\star$. By Lemma 3.4.16 we have $\pi{\cdot}(\lambda X.r)^\star \equiv (\pi{\cdot}\lambda X.r)^\star$. The result follows.

⊠

Intuitively, Lemma 3.4.20 states that the canonical form does not introduce any extraneous variables of level two.

Lemma 3.4.20: Fix Δ. Then for $r^\star$ calculated for Δ we have $fV(r^\star) \subseteq fV(r)$.

Proof. By induction on r. See Appendix A. ⊠

Lemma 3.4.21: Fix Δ. Then for canonical forms calculated for Δ we have $(r[a{:=}t])^\star \equiv r^\star[a{:=}t^\star]$.

Proof. By induction on $depth(r)$.

- The case a. By Definition 3.4.12 we have $(a[a{:=}t])^\star \equiv t^\star$. By Definition 3.4.12 we have $t^\star \equiv a[a{:=}t^\star]$. By Definition 3.4.13 we have $a[a{:=}t^\star] \equiv a^\star[a{:=}t^\star]$. The result follows.

- The case b. By Definition 3.4.12 we have $b[a{:=}t]^\star \equiv b^\star$. By Definition 3.4.13 we have $b^\star \equiv b$. By Definition 3.4.12 we have $b \equiv b[a{:=}t^\star]$. By Definition 3.4.13 we have $b[a{:=}t^\star] \equiv b^\star[a{:=}t^\star]$. The result follows.
- The case c. For every Δ we have $\Delta \vdash a\#\mathsf{c}$. By Definition 3.4.12 we have $\mathsf{c}[a{:=}t]^\star \equiv \mathsf{c}^\star$. By Definition 3.4.13 we have $\mathsf{c}^\star \equiv \mathsf{c}$. By Definition 3.4.12 we have $\mathsf{c} \equiv \mathsf{c}[a{:=}t^\star]$. The result follows.
- The case $\pi{\cdot}X$. There are two cases:
  - The case $\Delta \vdash a\#\pi{\cdot}X$. By Definition 3.4.13 we have $(\pi{\cdot}X)^\star[a{:=}t^\star] \equiv (\pi{\cdot}X)[a{:=}t^\star]$. By Definition 3.4.12 we have $(\pi{\cdot}X)[a{:=}t^\star] \equiv \pi{\cdot}X$. By Definition 3.4.12 we have $\pi{\cdot}X \equiv (\pi{\cdot}X)[a{:=}t]$. By Definition 3.4.13 we have $(\pi{\cdot}X)[a{:=}t] \equiv ((\pi{\cdot}X)[a{:=}t])^\star$. The result follows.
  - The case $\Delta \nvdash a\#\pi{\cdot}X$. By Definition 3.4.13 we have $(\pi{\cdot}X)^\star[a{:=}t^\star] \equiv (\pi{\cdot}X)[a{:=}t^\star]$. By Definition 3.4.12 we have $(\pi{\cdot}X)[a{:=}t^\star] \equiv (\pi{\cdot}X)[a{\mapsto}t^\star]$. By definition $(\pi{\cdot}X)[a{\mapsto}t^\star] \equiv (\lambda a.(\pi{\cdot}X))t^\star$. By Definition 3.4.13 we have $(\lambda a.(\pi{\cdot}X))t^\star \equiv ((\lambda a.(\pi{\cdot}X))t)^\star$. By definition $((\lambda a.(\pi{\cdot}X))t)^\star \equiv ((\pi{\cdot}X)[a{\mapsto}t])^\star$. By Definition 3.4.12 we have $((\pi{\cdot}X)[a{\mapsto}t])^\star \equiv ((\pi{\cdot}X)[a{:=}t])^\star$. The result follows.
- The case rs. By Definition 3.4.12 we have $((rs)[a{:=}t])^\star \equiv (r[a{:=}t])(s[a{:=}t])^\star$. By inductive hypothesis $(r[a{:=}t])(s[a{:=}t])^\star \equiv (r^\star[a{:=}t^\star])(s^\star[a{:=}t^\star])$. By Definition 3.4.12 we have $(r^\star[a{:=}t^\star])(s^\star[a{:=}t^\star]) \equiv (r^\star s^\star)[a{:=}t^\star]$. By Definition 3.4.13 we have $(r^\star s^\star)[a{:=}t^\star] \equiv (rs)^\star[a{:=}t^\star]$. The result follows.
- The case $\lambda a.r$. By Definition 3.4.12 we have $((\lambda a.r)[a{:=}t])^\star \equiv (\lambda a.r)^\star$. By Definition 3.4.13 we have $(\lambda a.r)^\star \equiv \lambda a.(r^\star)$. By Definition 3.4.12 we have $\lambda a.(r^\star) \equiv (\lambda a.(r^\star))[a{:=}t^\star]$. By Definition 3.4.13 we have $(\lambda a.(r^\star))[a{:=}t^\star] \equiv (\lambda a.r)^\star[a{:=}t^\star]$. The result follows.
- The case $\lambda b.s$. There are three cases:
  - The case $\Delta \vdash b\#t$. By Definition 3.4.12 we have $((\lambda b.s)[a{:=}t])^\star \equiv (\lambda b.(s[a{:=}t]))^\star$. By Definition 3.4.13 we have $(\lambda b.(s[a{:=}t]))^\star \equiv \lambda b.((s[a{:=}t])^\star)$. By inductive hypothesis $\lambda b.((s[a{:=}t])^\star) \equiv \lambda b.(s^\star[a{:=}t^\star])$. By Definition 3.4.12 we have $\lambda b.(s^\star[a{:=}t^\star]) \equiv (\lambda b.(s^\star))[a{:=}t^\star]$. By Definition 3.4.13 we have $(\lambda b.(s^\star))[a{:=}t^\star] \equiv (\lambda b.s)^\star[a{:=}t^\star]$. The result follows.
  - The case $\Delta \nvdash b\#t$ with Δ having sufficient freshness. Suppose $\Delta \nvdash b\#t$. By Definition 3.4.12 we have $((\lambda b.s)[a{:=}t])^\star \equiv (\lambda c.(((c\ b){\cdot}s)[a{:=}t]))^\star$ where c is the first fresh atom picked with respect to our arbitrary but fixed ordering such that $\Delta \vdash c\#s$ and $\Delta \vdash c\#t$. By Definition 3.4.13 we have $(\lambda c.(((c\ b){\cdot}s)[a{:=}t]))^\star \equiv \lambda c.(((c\ b){\cdot}s)[a{:=}t])^\star$. By Lemma 3.2.9 we have $\mathit{depth}((c\ b){\cdot}s) = \mathit{depth}(s)$. By inductive hypothesis $\lambda c.(((c\ b){\cdot}s)[a{:=}t])^\star \equiv \lambda c.(((c\ b){\cdot}s)^\star[a{:=}t^\star])$. By Lemma 3.4.16 we have $\lambda c.(((c\ b){\cdot}s)^\star[a{:=}t^\star]) \equiv \lambda c.(((c\ b){\cdot}s^\star)[a{:=}t^\star])$. By Definition 3.4.12 we have $\lambda c.(((c\ b){\cdot}s^\star)[a{:=}t^\star]) \equiv (\lambda b.s^\star)[a{:=}t^\star]$. By Definition 3.4.13 we have $(\lambda b.s^\star)[a{:=}t^\star] \equiv (\lambda b.s)^\star[a{:=}t^\star]$. The result follows.
  - The case $\Delta \nvdash b\#t$ with Δ not having sufficient freshness. By Definition 3.4.12 we have $((\lambda b.s)[a{:=}t])^\star \equiv ((\lambda b.s)[a{\mapsto}t])^\star$. By definition $(\lambda b.s)[a{\mapsto}t] \equiv (\lambda a.(\lambda b.s))t$. By Definition 3.4.13 we have $((\lambda a.(\lambda b.s))t)^\star \equiv \lambda a.(\lambda b.(s^\star))t^\star$. By definition $\lambda a.(\lambda b.$

$(s^\star))t^\star \equiv (\lambda b.s^\star)[a{\mapsto}t^\star]$. By Definition 3.4.12 we have $(\lambda b.s^\star)[a{\mapsto}t^\star] \equiv (\lambda b.s^\star)[a{:=}t^\star]$. By Definition 3.4.13 we have $(\lambda b.s^\star)[a{:=}t^\star] \equiv (\lambda b.s)^\star[a{:=}t^\star]$. The result follows.

- The case $\lambda X.r$. Suppose $X \notin fV(t)$, which can be guaranteed by renaming. By Definition 3.4.12 we have $((\lambda X.r)[a{:=}t])^\star \equiv \lambda X.(r[a{:=}t])^\star$. By Definition 3.4.13 we have $\lambda X.(r[a{:=}t])^\star \equiv \lambda X.(r[a{:=}t]^\star)$. By inductive hypothesis $\lambda X.(r[a{:=}t]^\star) \equiv \lambda X.(r^\star[a{:=}t^\star])$. By Definition 3.4.12 and Lemma 3.4.20 we have $\lambda X.(r^\star[a{:=}t^\star]) \equiv (\lambda X.(r^\star))[a{:=}t^\star]$. By Definition 3.4.13 we have $(\lambda X.(r^\star))[a{:=}t^\star] \equiv (\lambda X.r)^\star[a{:=}t^\star]$. The result follows.

⊠

Theorem 3.4.22 is an important piece in the machinery of the confluence proof for the level one fragment. In short, it states that every term eventually reduces to its canonical form.

Theorem 3.4.22: For every Δ and r there exists Δ^+ freshly extending Δ such that $\Delta^+ \vdash r \overset{(\mathbf{level1})}{\to^*} r^\star$.

Proof. By induction on $depth(r)$.

- The case a. By Definition 3.4.13 we have $a^\star \equiv a$. We take $\Delta^+ = \Delta$. The result follows.
- The case c. By Definition 3.4.13 we have $\mathsf{c}^\star \equiv \mathsf{c}$. We take $\Delta^+ = \Delta$. The result follows.
- The case $\pi{\cdot}X$. By Definition 3.4.13 we have $\pi{\cdot}X^\star \equiv \pi{\cdot}X$. We take $\Delta^+ = \Delta$. The result follows.
- The case rs. By definition $r[a{\mapsto}t] \equiv (\lambda a.r)t$. There are multiple cases:
  - The case $(\lambda a.r)[a{\mapsto}t]$. By Definition 3.4.13 we have $((\lambda a.r)[a{\mapsto}t])^\star \equiv \lambda a.(r^\star)$. Taking $\Delta^+ = \Delta$, by $(\to\#)$ we have $\Delta \vdash (\lambda a.r)[a{\mapsto}t] \to \lambda a.r$. By inductive hypothesis and $(\triangleright\lambda\mathbf{a})$ we have $\Delta \vdash \lambda a.r \overset{(\mathbf{level1})}{\to^*} \lambda a.r^\star$. By Definition 3.4.13 we have $\lambda a.(r^\star) \equiv (\lambda a.r)^\star$. The result follows.
  - The case $(\lambda b.s)[a{\mapsto}t]$ with $\Delta \vdash b\#t$. By Definition 3.4.13 we have $(\lambda b.s)[a{\mapsto}t]^\star \equiv \lambda b.(r^\star[a{:=}t^\star])$. By $(\to\lambda\mathbf{a})$ we have $\Delta \vdash (\lambda b.s)[a{\mapsto}t] \to \lambda b.(s[a{\mapsto}t])$. By Lemma 3.4.15 there exists a Δ_1^+ freshly extending Δ such that ${\Delta_1}^+ \vdash s[a{\mapsto}t] \to^* s[a{:=}t]$. By inductive hypothesis and $(\triangleright\lambda\mathbf{a})$ there exists Δ_2^+ freshly extending Δ_1^+ such that $\Delta_2^+ \vdash s[a{:=}t] \overset{(\mathbf{level1})}{\to^*} (s[a{:=}t])^\star$. By Lemma 3.4.21 we have $(s[a{:=}t])^\star \equiv s^\star[a{:=}t^\star]$. By Definition 3.4.13 we have $\lambda b.(s^\star[a{:=}t^\star]) \equiv (\lambda b.(s[a{\mapsto}t]))^\star$. We take $\Delta^+ = \Delta_2^+$, and the result follows.
  - The case $(\lambda b.s)[a{\mapsto}t]$ with $\Delta \nvdash b\#t$. There are two cases:
    - The case when Δ contains sufficient freshness. By Definition 3.4.13 we have $((\lambda b.s)[a{\mapsto}t])^\star \equiv \lambda c.(((c\ b){\cdot}s^\star)[a{:=}t^\star])$ where c is a fresh atom, distinct from a and b, chosen so that $\Delta \vdash c\#s$ and $\Delta \vdash c\#t$. By $(\triangleright\alpha)$ we have $\Delta \vdash (\lambda b.s)[a{\mapsto}t] \to^* (\lambda c.((c\ b){\cdot}s))[a{\mapsto}t]$. By $(\to\lambda\mathbf{a})$ we have $\Delta \vdash (\lambda c.((c\ b){\cdot}s))[a{\mapsto}t] \to \lambda c.(((c\ b){\cdot}s)[a{\mapsto}t])$. By Lemma 3.4.15 there exists Δ_1^+ freshly extending Δ such that $\Delta^+ \vdash ((c\ b){\cdot}s)[a{\mapsto}t] \overset{(\mathbf{level1})}{\to^*} ((c\ b){\cdot}s)[a{:=}t]$. By inductive hypothesis there exists Δ_2^+ freshly extending Δ_1^+ such that $\Delta_2^+ \vdash ((c\ b){\cdot}s)[a{:=}t] \overset{(\mathbf{level1})}{\to^*} (((c\ b){\cdot}s)[a{:=}t])^\star$. By Lemma 3.4.21 we have $(((c\ b){\cdot}s)[a{:=}t])^\star \equiv ((c\ b){\cdot}s)^\star[a{:=}t^\star]$. By Lemma 3.4.16 we have $((c\ b){\cdot}s)^\star[a{:=}$

$t^\star] \equiv ((b\ c)\cdot s^\star)[a{:=}t^\star]$. By Definition 3.4.13 we have $\lambda b.(((b\ c)\cdot s^\star)[a{:=}t^\star]) \equiv \lambda b.(((c\ b)\cdot s)[a{:=}t])$. We take $\Delta^+ = \Delta_2^+$, and the result follows.

- The case when Δ does not contain sufficient freshness. Let Δ_1^+ be a fresh extension of Δ such that $\Delta^+ \vdash c\#s$ and $\Delta^+ \vdash c\#t$, where c is the first atom, not mentioned in s or t, in the ordering of Definition 3.4.12. By Definition 3.4.13 we have $((\lambda b.s)[a{\mapsto}t])^\star \equiv \lambda c.(((c\ b)\cdot s^\star)[a{:=}t^\star])$ where c is our fresh atom. By $(\rhd\alpha)$ we have $\Delta_1^+ \vdash (\lambda b.s)[a{\mapsto}t] \to^* (\lambda c.((c\ b)\cdot s))[a{\mapsto}t]$. By $(\to\lambda\mathbf{a})$ we have $\Delta_1^+ \vdash (\lambda c.((c\ b)\cdot s))[a{\mapsto}t] \to \lambda c.(((c\ b)\cdot s)[a{\mapsto}t])$. By Lemma 3.4.15 there exists Δ_2^+ freshly extending Δ_1^+ such that $\Delta_2^+ \vdash ((c\ b)\cdot s)[a{\mapsto}t] \overset{(\mathbf{level1})}{\to^*} ((c\ b)\cdot s)[a{:=}t]$. By inductive hypothesis there exists Δ_3^+ freshly extending Δ_2^+ such that $\Delta_2^+ \vdash ((c\ b)\cdot s)[a{:=}t] \overset{(\mathbf{level1})}{\to^*} (((c\ b)\cdot s)[a{:=}t])^\star$. By Lemma 3.4.21 we have $(((c\ b)\cdot s)[a{:=}t])^\star \equiv ((c\ b)\cdot s)^\star[a{:=}t^\star]$. By Lemma 3.4.16 we have $((c\ b)\cdot s)^\star[a{:=}t^\star] \equiv ((b\ c)\cdot s^\star)[a{:=}t^\star]$. By Definition 3.4.13 we have $\lambda b.(((b\ c)\cdot s^\star)[a{:=}t^\star]) \equiv \lambda b.(((c\ b)\cdot s)[a{:=}t])$. We take $\Delta^+ = \Delta_3^+$, and the result follows.

- The case $(\lambda X.r)[a{\mapsto}t]$. Suppose $X \notin fV(t)$, which can be guaranteed. By $(\to\lambda\mathbf{X})$ we have $\Delta \vdash (\lambda X.r)[a{\mapsto}t] \to \lambda X.(r[a{\mapsto}t])$. By inductive hypothesis Δ^+ exists such that $\Delta^+ \vdash r[a{\mapsto}t] \to^* (r[a{\mapsto}t])^\star$. By Definition 3.4.13 we have $(r[a{\mapsto}t])^\star \equiv r^\star[a{:=}t^\star]$. By Definition 3.4.13 and Lemma 3.4.20 we have $((\lambda X.r)[a{\mapsto}t])^\star \equiv \lambda X.(r^\star[a{:=}t^\star])$. The result follows.

- The case $(\lambda X.r)t$. By Definition 3.4.13 we have $((\lambda X.r)t)^\star \equiv r^\star[X{:=}t^\star]$. By inductive hypotheses there exists Δ_1^+ freshly extending Δ such that $\Delta_1^+ \vdash t \overset{(\mathbf{level1})}{\to^*} t^\star$, and Δ_2^+ freshly extending Δ such that $\Delta_2^+ \vdash r \overset{(\mathbf{level1})}{\to^*} r^\star$. Take $\Delta^+ = \Delta_1^+ \cup \Delta_2^+$. By $(\rhd\mathbf{app2})$ we have $\Delta^+ \vdash (\lambda X.r)t \overset{(\mathbf{level1})}{\to^*} (\lambda X.r)t^\star$. By $(\rhd\lambda\mathbf{X})$ we have $\Delta^+ \vdash (\lambda X.r)t^\star \overset{(\mathbf{level1})}{\to^*} (\lambda X.r^\star)t^\star$. By $(\to\beta)$ we have $\Delta^+ \vdash (\lambda X.r^\star)t^\star \to r^\star[X{:=}t^\star]$. By Lemma 3.4.16 we have $r^\star[X{:=}t^\star] \equiv (r[X{:=}t])^\star$. The result follows.

- All other cases. By inductive hypotheses there exists Δ_1^+ freshly extending Δ such that $\Delta_1^+ \vdash r \overset{(\mathbf{level1})}{\to^*} r^\star$ and Δ_2^+ freshly extending Δ such that $\Delta_2^+ \vdash s \overset{(\mathbf{level1})}{\to^*} s^\star$. Taking $\Delta^+ = \Delta_1^+ \cup \Delta_2^+$, we have $\Delta^+ \vdash rs \overset{(\mathbf{level1})}{\to^*} r^\star s^\star$. By Definition 3.4.13 we have $r^\star s^\star \equiv (rs)^\star$. The result follows.

- The case $\lambda a.r$. By inductive hypothesis there exists Δ^+ freshly extending Δ such that $\Delta^+ \vdash r \overset{(\mathbf{level1})}{\to^*} r^\star$. By $(\rhd\lambda\mathbf{a})$ we have $\Delta^+ \vdash \lambda a.r \overset{(\mathbf{level1})}{\to^*} \lambda a.r^\star$. The result follows.
- The case $\lambda X.r$. By inductive hypothesis there exists Δ^+ freshly extending Δ such that $\Delta^+ \vdash r \overset{(\mathbf{level1})}{\to^*} r^\star$. By $(\rhd\lambda\mathbf{X})$ we have $\Delta^+ \vdash \lambda X.r \overset{(\mathbf{level1})}{\to^*} \lambda X.r^\star$. The result follows.

⊠

Lemma 3.4.23 is the final missing component in the proof of confluence for the level one fragment.

Lemma 3.4.23: For every Δ, r and s there exists Δ^+ freshly extending Δ such that $\Delta \vdash r \overset{(\mathbf{level1})}{\to} s$ implies $\Delta^+ \vdash s^\star \overset{(\mathbf{level1})}{\to^*} r^\star$ where $r^\star$ and $s^\star$ are calculated for Δ^+.

Proof. By induction on the derivation of $\Delta \vdash r \overset{(\mathbf{level1})}{\to} s$.

- The case ($\rightarrow$**a**). Suppose $\Delta \vdash a[a{\mapsto}t] \overset{(\textbf{level1})}{\rightarrow} t$. By Definition 3.4.13 we have $(a[a{\mapsto}t])^\star \equiv t^\star$. We obtain $\Delta \vdash t^\star \overset{(\textbf{level1})}{\rightarrow^*} t^\star$. We take $\Delta^+ = \Delta$. The result follows.
- The case ($\rightarrow$#). Suppose $\Delta \vdash a\#r$ so that $\Delta \vdash r[a{\mapsto}t] \rightarrow r$. By Definition 3.4.13 we have $(r[a{\mapsto}t])^\star \equiv r^\star$. We obtain $\Delta \vdash r^\star \overset{(\textbf{level1})}{\rightarrow^*} r^\star$. We take $\Delta^+ = \Delta$. The result follows.
- The case ($\rightarrow$**rs1**). Suppose $\Delta \vdash a\#s$ so that $\Delta \vdash (rs)[a{\mapsto}t] \overset{(\textbf{level1})}{\rightarrow} (r[a{\mapsto}t])s$. By Definition 3.4.13 we have $((rs)[a{\mapsto}t])^\star \equiv ((r^\star)[a{:=}t^\star])s^\star$. By Definition 3.4.13 we have $((r[a{\mapsto}t])s)^\star \equiv ((r^\star)[a{:=}t^\star])s^\star$. We obtain $\Delta \vdash ((r^\star)[a{:=}t^\star])s^\star \overset{(\textbf{level1})}{\rightarrow^*} ((r^\star)[a{:=}t^\star])s^\star$. We take $\Delta^+ = \Delta$. The result follows.
- The case ($\rightarrow$**rs2**). Suppose $\mathit{level}(r) = 1$ so that $\Delta \vdash (rs)[a{\mapsto}t] \overset{(\textbf{level1})}{\rightarrow} (r[a{\mapsto}t])(s[a{\mapsto}t])$. By Definition 3.4.13 we have $((rs)[a{\mapsto}t])^\star \equiv (r^\star[a{:=}t^\star])(s^\star[a{:=}t^\star])$. By Definition 3.4.13 we have $((r[a{\mapsto}t])(s[a{\mapsto}t]))^\star \equiv (r^\star[a{:=}t^\star])(s^\star[a{:=}t^\star])$. We obtain $\Delta \vdash (r^\star[a{:=}t^\star])(s^\star[a{:=}t^\star]) \overset{(\textbf{level1})}{\rightarrow^*} (r^\star[a{:=}t^\star])(s^\star[a{:=}t^\star])$. We take $\Delta^+ = \Delta$. The result follows.
- The case ($\rightarrow\lambda$**a**). Suppose $\Delta \vdash a\#u$ so that $\Delta \vdash (\lambda a.r)[b{\mapsto}u] \overset{(\textbf{level1})}{\rightarrow} \lambda a.(r[b{\mapsto}u])$. By Definition 3.4.13 we have $((\lambda a.r)[b{\mapsto}u])^\star \equiv \lambda a.(r^\star[b{:=}u^\star])$. By Definition 3.4.13 we have $(\lambda a.(r[b{\mapsto}u]))^\star \equiv \lambda a.(r^\star[b{:=}u^\star])$. We obtain $\Delta \vdash \lambda a.(r^\star[b{:=}u^\star]) \overset{(\textbf{level1})}{\rightarrow^*} \lambda a.(r^\star[b{:=}u^\star])$. We take $\Delta^+ = \Delta$. The result follows.
- The case ($\rightarrow\lambda$**X**). Suppose $X \notin fV(t)$ so that $\Delta \vdash (\lambda X.r)[a{\mapsto}t] \overset{(\textbf{level1})}{\rightarrow} \lambda X.(r[a{\mapsto}t])$. By Definition 3.4.13 we have $((\lambda X.r)[a{\mapsto}t])^\star \equiv \lambda X.(r^\star[a{:=}t^\star])$. By Definition 3.4.13 we have $(\lambda X.(r[a{\mapsto}t]))^\star \equiv \lambda X.(r^\star[a{:=}t^\star])$. We obtain $\Delta \vdash \lambda X.(r^\star[a{:=}t^\star]) \overset{(\textbf{level1})}{\rightarrow^*} \lambda X.(r^\star[a{:=}t^\star])$. We take $\Delta^+ = \Delta$. The result follows.
- The case ($\rightarrow\beta$). Suppose $\Delta \vdash (\lambda X.r)t \overset{(\textbf{level1})}{\rightarrow} r[X{:=}t]$. By Definition 3.4.13 we have $((\lambda X.r)t)^\star \equiv r^\star[X{:=}t^\star]$. By Lemma 3.4.16 we have $(r[X{:=}t])^\star \equiv r^\star[X{:=}t^\star]$. We obtain $\Delta \vdash r^\star[X{:=}t^\star] \overset{(\textbf{level1})}{\rightarrow^*} r^\star[X{:=}t^\star]$. We take $\Delta^+ = \Delta$. The result follows.
- The case ($\triangleright\lambda$**a**). Suppose $\Delta \vdash r \overset{(\textbf{level1})}{\rightarrow} s$. By inductive hypothesis Δ^+ exists such that $\Delta^+ \vdash s^\star \overset{(\textbf{level1})}{\rightarrow^*} r^\star$. Using ($\triangleright\lambda$**a**) we obtain $\Delta^+ \vdash \lambda a.(s^\star) \overset{(\textbf{level1})}{\rightarrow^*} \lambda a.(r^\star)$. By Definition 3.4.13 we have $\lambda a.(r^\star) \equiv (\lambda a.r)^\star$. The result follows.
- The case ($\triangleright$**rs1**) and ($\triangleright$**rs2**). We consider only the first case, as the second is similar.
- The case ($\triangleright\lambda$**X**). Suppose $\Delta \vdash r \overset{(\textbf{level1})}{\rightarrow} s$. By inductive hypothesis Δ^+ exists such that $\Delta^+ \vdash s^\star \overset{(\textbf{level1})}{\rightarrow^*} r^\star$. Using ($\triangleright\lambda$**X**) we obtain $\Delta^+ \vdash \lambda X.(s^\star) \overset{(\textbf{level1})}{\rightarrow^*} \lambda X.(r^\star)$. By Definition 3.4.13 we have $\lambda X.(r^\star) \equiv (\lambda X.r)^\star$. The result follows.
- The case ($\triangleright\alpha$). Suppose $\Delta \vdash r \overset{(\textbf{level1})}{\rightarrow} s$, $\Delta \vdash a\#r$ and $\Delta \vdash b\#r$. By inductive hypothesis there exists Δ'^+ such that $\Delta'^+ \vdash s^\star \overset{(\textbf{level1})}{\rightarrow^*} r^\star$. By Lemma 3.4.19 we have Δ''^+ exists such that $\Delta''^+ \vdash a\#r^\star$ and $\Delta''^+ \vdash b\#r^\star$. By Lemma 3.4.4 we have $\Delta''^+ \vdash a\#s^\star$ and $\Delta''^+ \vdash b\#r^\star$. Using ($\triangleright\alpha$) we obtain $\Delta''^+ \vdash (b\ a){\cdot}s^\star \overset{(\textbf{level1})}{\rightarrow^*} r^\star$. By Lemma 3.4.8 we have $\Delta''^+ \vdash (b\ a){\cdot}((b\ a){\cdot}s^\star) \overset{(\textbf{level1})}{\rightarrow^*} (b\ a){\cdot}r^\star$. By Lemma 3.2.13 we have $\Delta''^+ \vdash ((b\ a){\circ}(b\ a)){\cdot}s^\star \overset{(\textbf{level1})}{\rightarrow^*} (b\ a){\cdot}r^\star$. It is a fact that $((b\ a){\circ}(b\ a)){\cdot}s^\star \equiv s^\star$. By Lemma 3.4.16 we have $\Delta''^+ \vdash s^\star \overset{(\textbf{level1})}{\rightarrow^*} ((b\ a){\cdot}r)^\star$. We take $\Delta^+ = \Delta''^+$. The result follows.

⊠

Theorem 3.4.24: Suppose $\Delta \vdash r \overset{(\textbf{level1})}{\rightarrow^*} s$ and $\Delta \vdash r \overset{(\textbf{level1})}{\rightarrow^*} t$. Then there exists Δ^+ freshly extending Δ and some term u such that $\Delta^+ \vdash s \overset{\textbf{(level1)}}{\rightarrow^*} u$ and $\Delta^+ \vdash t \overset{\textbf{(level1)}}{\rightarrow^*} u$.

That is, $\Delta \vdash - \overset{\textbf{(level1)}}{\rightarrow} -$ is confluent.

Proof. Suppose $\Delta \vdash r \overset{\textbf{(level1)}}{\rightarrow^*} s$ and $\Delta \vdash r \overset{\textbf{(level1)}}{\rightarrow^*} t$. By Theorem 3.4.22 there exists ${\Delta_1}^+$ such that ${\Delta_1}^+ \vdash r \overset{\textbf{(level1)}}{\rightarrow^*} r^\star$, ${\Delta_2}^+$ such that ${\Delta_2}^+ \vdash s \overset{\textbf{(level1)}}{\rightarrow^*} s^\star$ and ${\Delta_3}^+$ such that ${\Delta_3}^+ \vdash t \overset{\textbf{(level1)}}{\rightarrow^*} t^\star$.

By Lemma 3.4.23 there exists ${\Delta_4}^+$ and ${\Delta_5}^+$ such that ${\Delta_4}^+ \vdash s^\star \overset{\textbf{(level1)}}{\rightarrow^*} r^\star$ and ${\Delta_5}^+ \vdash t^\star \overset{\textbf{(level1)}}{\rightarrow^*} r^\star$. Taking $\Delta^+ = \bigcup_{1 \leq i \leq 5} {\Delta_i}^+$, we have the result. ⊠

Theorem 3.4.24, the confluence of the level one fragment, is the main result in this Subsection. The proof of Theorem 3.4.24 can be readily visualised with the following diagram:

$$\begin{tikzcd}
& r \arrow[dl, "*"'] \arrow[dd, "*"] \arrow[dr, "*"] & \\
s \arrow[d, "*"] & & t \arrow[d, "*"] \\
s^\star \arrow[r, "*"] & r^\star & t^\star \arrow[l, "*"']
\end{tikzcd}$$

3.4.2 Confluence of level two reductions

This Subsection handles the proof of confluence for the level two fragment of the reduction relation, introduced in Definition 3.4.25. We prove confluence for level two using Tait and Martin-Löf's parallel reduction method (explained in Barendregt's tome [Bar84]).

Theorem 3.4.31 demonstrates that the notions of parallel reduction and level two reduction coincide. Lemma 3.4.35 states that parallel reduction satisfies the 'diamond property'. Theorem 3.4.36, the confluence of the level two fragment, is the main result in this Subsection.

Definition 3.4.25: Let (**level2**) be the set $\{(\rightarrow\beta), (\rightarrow\lambda\mathbf{X})\}$. Let $\Delta \vdash r \overset{\textbf{(level2)}}{\rightarrow} s$ be the least congruence closed under the rules of (**level2**).

Lemma 3.4.26 through to Corollary 3.4.30 are technical lemmas used in the proof of Theorem 3.4.31.

Lemma 3.4.26: If $\Delta \vdash r \Rightarrow s$ then $\Delta \vdash r \overset{\textbf{(level2)}}{\rightarrow^*} s$.

Proof. By induction on the derivation of $\Delta \vdash r \Rightarrow s$.

- The case ($\Rightarrow$**a**). Suppose $\Delta \vdash a \Rightarrow a$. By Definition 3.4.14 we have $\Delta \vdash a \overset{\textbf{(level2)}}{\rightarrow^*} a$. The result follows.
- The case ($\Rightarrow$**X**). Suppose $\Delta \vdash \pi{\cdot}X \Rightarrow \pi{\cdot}X$. By Definition 3.4.14 we have $\Delta \vdash \pi{\cdot}X \overset{\textbf{(level2)}}{\rightarrow^*} \pi{\cdot}X$. The result follows.
- The case ($\Rightarrow\mathsf{c}$). Suppose $\Delta \vdash \mathsf{c} \Rightarrow \mathsf{c}$. By Definition 3.4.14 we have $\Delta \vdash \mathsf{c} \overset{\textbf{(level2)}}{\rightarrow^*} \mathsf{c}$. The result follows.
- The case ($\Rightarrow$**rs**). Suppose $\Delta \vdash r \Rightarrow s$ and $\Delta \vdash t \Rightarrow u$. By inductive hypotheses $\Delta \vdash r \overset{\textbf{(level2)}}{\rightarrow^*} s$ and $\Delta \vdash t \overset{\textbf{(level2)}}{\rightarrow^*} u$. Using ($\triangleright$**rs1**) and ($\triangleright$**rs2**) we obtain $\Delta \vdash rs \overset{\textbf{(level2)}}{\rightarrow^*} tu$. The result follows.

- The case ($\Rightarrow$$\lambda$**a**). Suppose $\Delta \vdash r \Rightarrow s$. By inductive hypothesis $\Delta \vdash r \overset{(\mathbf{level2})}{\to^*} s$. Using ($\triangleright\lambda$**a**) we obtain $\Delta \vdash \lambda a.r \overset{(\mathbf{level2})}{\to^*} \lambda a.s$. The result follows.
- The case ($\Rightarrow$$\lambda$**X**). Suppose $\Delta \vdash r \Rightarrow s$. By inductive hypothesis $\Delta \vdash r \overset{(\mathbf{level2})}{\to^*} s$. Using ($\triangleright\lambda$**X**) we obtain $\Delta \vdash \lambda X.r \overset{(\mathbf{level2})}{\to^*} \lambda X.s$. The result follows.
- The case ($\Rightarrow\epsilon$). Suppose $\Delta \vdash r \Rightarrow s$, $\Delta \vdash t \Rightarrow u$ and $\Delta \vdash su \overset{(\mathbf{level2})}{\to} v$. By inductive hypotheses $\Delta \vdash r \overset{(\mathbf{level2})}{\to^*} s$ and $\Delta \vdash t \overset{(\mathbf{level2})}{\to^*} u$. By Definition 3.4.14 we have $\Delta \vdash su \overset{(\mathbf{level2})}{\to^*} v$. Using ($\triangleright$**rs1**) and ($\triangleright$**rs2**) we obtain $\Delta \vdash rt \overset{(\mathbf{level2})}{\to^*} v$. The result follows.
- The case ($\Rightarrow\alpha$). Suppose $\Delta \vdash r \Rightarrow s$, $\Delta \vdash a\#r$ and $\Delta \vdash b\#r$. By inductive hypothesis $\Delta \vdash r \overset{(\mathbf{level2})}{\to^*} s$. Using ($\triangleright\alpha$) we obtain $\Delta \vdash (b\ a)\cdot r \overset{(\mathbf{level2})}{\to^*} s$. The result follows.

⊠

Corollary 3.4.27: If $\Delta \vdash r \Rightarrow^* s$ then $\Delta \vdash r \to^* s$.

Proof. Immediate from Lemma 3.4.26 and the definition of reflexive, transitive closure. ⊠

Lemma 3.4.28: $\Rightarrow$ is reflexive, that is $\Delta \vdash r \Rightarrow r$, for all r.

Proof. By induction on r. See Appendix A. ⊠

Lemma 3.4.29: If $\Delta \vdash r \overset{(\mathbf{level2})}{\to} s$ then $\Delta \vdash r \Rightarrow s$.

Proof. An immediate corollary of Lemma 3.4.28 and ($\Rightarrow\epsilon$). ⊠

Corollary 3.4.30: If $\Delta \vdash r \overset{(\mathbf{level2})}{\to^*} s$ then $\Delta \vdash r \Rightarrow^* s$.

Proof. Immediate from Lemma 3.4.29 and the definition of reflexive, transitive closure. ⊠

Theorem 3.4.31 states that the notions of parallel reduction and level two reduction coincide.

Theorem 3.4.31: $\Delta \vdash r \Rightarrow^* s$ if and only if $\Delta \vdash r \overset{(\mathbf{level2})}{\to^*} s$.

Proof. Immediate, from Corollaries 3.4.27 and 3.4.30. ⊠

Lemma 3.4.32 states that parallel reduction is invariant under identity instantiations.

Lemma 3.4.32: If $\Delta \vdash r \Rightarrow s$ and $\Delta' \vdash \Delta[X{:=}\pi\cdot X]$ then $\Delta' \vdash r[X{:=}\pi\cdot X] \Rightarrow s[X{:=}\pi\cdot X]$.

Proof. By induction on the derivation of $\Delta \vdash r \Rightarrow s$, using Lemmas 3.4.6 and 3.2.16. See Appendix A. ⊠

Lemma 3.4.33 is a standard equivariance result for parallel reductions:

Lemma 3.4.33: If $\Delta \vdash r \Rightarrow s$ then $\Delta \vdash \pi\cdot r \Rightarrow \pi\cdot s$.

Proof. By induction on the derivation of $\Delta \vdash r \Rightarrow s$.

- The case ($\Rightarrow$**a**). By Definition 3.2.4 we have $\pi{\cdot}a \equiv \pi(a)$. Using ($\Rightarrow$**a**) we obtain $\Delta \vdash \pi(a) \Rightarrow \pi(a)$. The result follows.
- The case ($\Rightarrow$**X**). By Definition 3.2.4 we have $\pi{\cdot}(\pi'{\cdot}X) \equiv (\pi{\circ}\pi'){\cdot}X$. Using ($\Rightarrow$**X**) we obtain $\Delta \vdash (\pi{\circ}\pi'){\cdot}X \Rightarrow (\pi{\circ}\pi'){\cdot}X$. The result follows.
- The case ($\Rightarrow$$\mathsf{c}$). By Definition 3.2.4 we have $\pi{\cdot}\mathsf{c} \equiv \mathsf{c}$. Using ($\Rightarrow$$\mathsf{c}$) we obtain $\Delta \vdash \mathsf{c} \Rightarrow \mathsf{c}$. The result follows.
- The case ($\Rightarrow$**rs**). Suppose $\Delta \vdash r \Rightarrow s$ and $\Delta \vdash t \Rightarrow u$. By inductive hypotheses $\Delta \vdash \pi{\cdot}r \Rightarrow \pi{\cdot}s$ and $\Delta \vdash \pi{\cdot}t \Rightarrow \pi{\cdot}u$. Using ($\Rightarrow$**rs**) we obtain $\Delta \vdash (\pi{\cdot}r)(\pi{\cdot}s) \Rightarrow (\pi{\cdot}t)(\pi{\cdot}u)$. By Definition 3.2.4 we have $(\pi{\cdot}t)(\pi{\cdot}u) \equiv \pi{\cdot}tu$. The result follows.
- The case ($\Rightarrow\lambda$**a**). Suppose $\Delta \vdash r \Rightarrow s$. By inductive hypothesis $\Delta \vdash \pi{\cdot}r \Rightarrow \pi{\cdot}s$. Using ($\Rightarrow\lambda$**a**) we obtain $\Delta \vdash \lambda\pi(a).(\pi{\cdot}r) \Rightarrow \lambda\pi(a).(\pi{\cdot}s)$. By Definition 3.2.4 we have $\lambda\pi(a).(\pi{\cdot}s) \equiv \pi{\cdot}\lambda a.s$. The result follows.
- The case ($\Rightarrow\lambda$**X**). Suppose $\Delta \vdash r \Rightarrow s$. By inductive hypothesis $\Delta \vdash \pi{\cdot}r \Rightarrow \pi{\cdot}s$. By Lemma 3.4.32 we have $\Delta \vdash (\pi{\cdot}r)[X{:=}\pi^{-1}{\cdot}X] \Rightarrow (\pi{\cdot}s)[X{:=}\pi^{-1}{\cdot}X]$. Using ($\Rightarrow\lambda$**X**) we obtain $\Delta \vdash \pi{\cdot}\lambda X.r \Rightarrow \pi{\cdot}\lambda X.s$. The result follows.
- The case ($\Rightarrow\epsilon$). Suppose $\Delta \vdash r \Rightarrow t$, $\Delta \vdash s \Rightarrow u$ and $\Delta \vdash tu \overset{\textbf{(level2)}}{\to} v$. By inductive hypotheses $\Delta \vdash \pi{\cdot}r \Rightarrow \pi{\cdot}t$ and $\Delta \vdash \pi{\cdot}s \Rightarrow \pi{\cdot}u$. By Lemma 3.4.8 we have $\Delta \vdash (\pi{\cdot}s)(\pi{\cdot}u) \overset{\textbf{(level2)}}{\to} \pi{\cdot}v$. Using ($\Rightarrow\epsilon$) we have $\Delta \vdash (\pi{\cdot}r)(\pi{\cdot}s) \Rightarrow \pi{\cdot}v$. By Definition 3.2.4 we have $(\pi{\cdot}r)(\pi{\cdot}s) \equiv \pi{\cdot}rs$. The result follows.
- The case ($\Rightarrow\alpha$). Suppose $\Delta \vdash r \Rightarrow s$, $\Delta \vdash a\#r$ and $\Delta \vdash b\#r$. By inductive hypothesis $\Delta \vdash \pi{\cdot}r \Rightarrow \pi{\cdot}s$. By Lemma 3.3.9 we have $\Delta \vdash \pi(a)\#\pi{\cdot}r$ and $\Delta \vdash \pi(b)\#\pi{\cdot}r$. Using ($\Rightarrow\alpha$) we obtain $\Delta \vdash (\pi(b)\ \pi(a)){\cdot}(\pi{\cdot}r) \Rightarrow \pi{\cdot}s$. It is a fact that $(\pi(b)\ \pi(a)){\cdot}(\pi{\cdot}r) \equiv \pi{\cdot}((b\ a){\cdot}r)$. The result follows.

⊠

Lemma 3.4.32 is a technical result used in the proof of Lemma 3.4.35.

Lemma 3.4.34: If $\Delta \vdash r \Rightarrow s$ and $\Delta \vdash t \Rightarrow u$ then $\Delta \vdash r[X{:=}t] \Rightarrow s[X{:=}u]$.

Proof. By induction on the derivation of $\Delta \vdash r \Rightarrow s$, using Lemmas 3.4.33, 3.4.6, and 3.2.16. See Appendix A. ⊠

Lemma 3.4.35 states that parallel reduction satisfies the 'diamond property':

Lemma 3.4.35: If $\Delta \vdash r \Rightarrow s$ and $\Delta \vdash r \Rightarrow t$ then there exists some u such that $\Delta \vdash s \Rightarrow u$ and $\Delta \vdash t \Rightarrow u$.

Proof. We examine all possible non-trivial divergences (where $s \not\equiv t$).

- The case $(\lambda X.r)[b{\mapsto}u]$. Suppose that $X \notin fV(u)$. Suppose also that $\Delta \vdash (\lambda X.r)[b{\mapsto}u] \Rightarrow (\lambda X.r')[b{\mapsto}u']$ and $\Delta \vdash (\lambda X.r)[b{\mapsto}u] \Rightarrow \lambda X.(r''[b{\mapsto}u''])$. By inductive hypothesis u''' exists such that $\Delta \vdash u' \Rightarrow u'''$ and $\Delta \vdash u'' \Rightarrow u'''$. Similarly, by inductive hypothesis r''' exists such that $\Delta \vdash r' \Rightarrow r'''$ and $\Delta \vdash r'' \Rightarrow r'''$. Using ($\Rightarrow\epsilon$) with ($\to\lambda$**X**) we obtain $\Delta \vdash (\lambda X.r')[b{\mapsto}u'] \Rightarrow \lambda X.(r'''[b{\mapsto}u'''])$. Using ($\Rightarrow\lambda$**X**) we obtain $\Delta \vdash \lambda X.(r''[b{\mapsto}u'']) \Rightarrow \lambda X.(r'''[b{\mapsto}u'''])$. The result follows.

- The case $(\lambda a.r)[b{\mapsto}u]$. Suppose that $\Delta \vdash a\#u$. Suppose also that $\Delta \vdash (\lambda a.r)[b{\mapsto}u] \Rightarrow (\lambda a.r')[b{\mapsto}u']$ and $\Delta \vdash (\lambda a.r)[b{\mapsto}u] \Rightarrow \lambda a.(r''[b{\mapsto}u''])$. By inductive hypothesis u''' exists such that $\Delta \vdash u' \Rightarrow u'''$ and $\Delta \vdash u'' \Rightarrow u'''$. Similarly, by inductive hypothesis r''' exists such that $\Delta \vdash r' \Rightarrow r'''$ and $\Delta \vdash r'' \Rightarrow r'''$. By Lemma 3.4.4 we have $\Delta \vdash a\#u'''$. Using $(\Rightarrow\epsilon)$ with $(\rightarrow\lambda\mathbf{a})$ we obtain $\Delta \vdash (\lambda a.r')[b{\mapsto}u'] \Rightarrow \lambda a.(r'''[b{\mapsto}u'''])$. Using $(\Rightarrow\lambda\mathbf{a})$ we obtain $\Delta \vdash \lambda a.(r''[b{\mapsto}u'']) \Rightarrow \lambda a.(r'''[b{\mapsto}u'''])$. The result follows.
- The case $(\lambda X.r)t$. Suppose $\Delta \vdash (\lambda X.r)t \Rightarrow (\lambda X.r')t'$ and $\Delta \vdash (\lambda X.r)t \Rightarrow r''[X{:=}t'']$. By inductive hypothesis t''' exists such that $\Delta \vdash t' \Rightarrow t'''$ and $\Delta \vdash t'' \Rightarrow t'''$. Similarly, by inductive hypothesis r''' exists that that $\Delta \vdash r' \Rightarrow r'''$ and $\Delta \vdash r'' \Rightarrow r'''$. By Lemma 3.4.34 we have $\Delta \vdash r''[X{:=}t''] \Rightarrow r'''[X{:=}t''']$. Using $(\Rightarrow\epsilon)$ with $(\rightarrow\beta)$ we obtain $\Delta \vdash (\lambda X.r')t \Rightarrow r'''[X{:=}t''']$. The result follows.

$\boxtimes$

Theorem 3.4.36, confluence of the level two fragment of the reduction relation, is the most important result in this Subsection, and an immediate corollary of the diamond property of parallel reductions, and the fact that parallel reduction and level two reduction coincide.

Theorem 3.4.36: $\Delta \vdash - \overset{(\mathbf{level2})}{\rightarrow} -$ is confluent.

Proof. An immediate corollary of Lemma 3.4.35 and Theorem 3.4.31. $\boxtimes$

3.4.3 Confluence of level one and level two reductions

This Subsection stitches together the proofs of confluence for the level one and level two fragments of the reduction relation.

The main result in this Subsection, and in the Chapter itself, is Theorem 3.4.42, stating the reduction relation of Definition 3.4.3 is confluent.

Lemma 3.4.37 is a technical lemma used in the proof of Lemma 3.4.38.

Lemma 3.4.37: If $\Delta \vdash r \rightarrow s$ then $level(s) \leq level(r)$.

Proof. By induction on the derivation of $\Delta \vdash r \rightarrow s$, using Lemma 3.4.7. See Appendix A. $\boxtimes$

Intuitively, Lemma 3.4.38 states that if a parallel reduction and a level one reduction diverge, then we can close this divergence using only parallel reductions and level one reductions.

Lemma 3.4.38: If $\Delta \vdash r \Rightarrow s$ and $\Delta \vdash r \overset{(\mathbf{level1})}{\rightarrow} t$ then there exists a u such that $\Delta \vdash s \overset{(\mathbf{level1})}{\rightarrow^*} u$ and $\Delta \vdash t \Rightarrow u$.

Proof. By considering all possible non-trivial divergences (where $s \not\equiv t$), using Lemmas 3.4.4 and 3.4.37. See Appendix A. $\boxtimes$

The last case of Lemma 3.4.38 demonstrates why $(\rightarrow\lambda\mathbf{X})$ must be in both $(\mathbf{level1})$ *and* $(\mathbf{level2})$.

Corollary 3.4.39: If $\Delta \vdash r \Rightarrow s$ and $\Delta \vdash r \overset{(\mathbf{level1})}{\to^*} t$ then there exists a u such that $\Delta \vdash s \overset{(\mathbf{level1})}{\to^*} u$ and $\Delta \vdash t \Rightarrow u$.

Proof. Immediate, from Lemma 3.4.38 and Definition 3.4.14. ⊠

Lemmas 3.4.40 and 3.4.41 are technical lemmas used in the proof of Theorem 3.4.42.

Lemma 3.4.40: If $\Delta \vdash r \overset{(\mathbf{level1})}{\to} s$ and $\Delta \vdash r \to^* t$ then there exists some u and some Δ^+ such that $\Delta^+ \vdash r \to^* u$ and $\Delta^+ \vdash t \overset{(\mathbf{level1})}{\to} u$.

Proof. By induction on the path length of $\Delta \vdash r \to^* t$.

- The base case. Suppose $r \equiv t$. Take $\Delta^+ = \Delta$ and $u \equiv s$. The result follows.
- The inductive step. Suppose $\Delta \vdash r \overset{(\mathbf{level1})}{\to} s$ and $\Delta \vdash r \to^* t' \to t$. By inductive hypothesis Δ'^+ and u' exists such that $\Delta'^+ \vdash s \to^* u'$ and $\Delta'^+ \vdash t \overset{(\mathbf{level1})}{\to} u'$. There are two cases:
  - The case $t' \overset{(\mathbf{level1})}{\to} u'$. By Theorem 3.4.24 we have Δ''^+ and u exist such that $\Delta''^+ \vdash t \overset{(\mathbf{level1})}{\to^*} u$ and $\Delta''^+ \vdash u' \overset{(\mathbf{level1})}{\to^*} u$. We take $\Delta^+ = \Delta''^+$. The result follows.
  - The case $t' \overset{(\mathbf{level2})}{\to} u'$. By Lemma 3.4.38 and Theorem 3.4.31 we have Δ''^+ and u exist such that $\Delta''^+ \vdash t \overset{(\mathbf{level1})}{\to^*} u$ and $\Delta''^+ \vdash u' \overset{(\mathbf{level2})}{\to^*} u$. We take $\Delta^+ = \Delta''^+$. The result follows.

⊠

Lemma 3.4.41: If $\Delta \vdash r \overset{(\mathbf{level2})}{\to} s$ and $\Delta \vdash r \to^* t$ then there exists some u and some Δ^+ such that $\Delta^+ \vdash r \to^* u$ and $\Delta^+ \vdash t \overset{(\mathbf{level2})}{\to} u$.

Proof. By induction on the path length of $\Delta \vdash r \to^* t$.

- The base case. Suppose $r \equiv t$. Take $\Delta^+ = \Delta$ and $u \equiv s$. The result follows.
- The inductive step. Suppose $\Delta \vdash r \overset{(\mathbf{level2})}{\to} s$ and $\Delta \vdash r \to^* t' \to t$. By inductive hypothesis Δ'^+ and u' exists such that $\Delta'^+ \vdash s \to^* u'$ and $\Delta'^+ \vdash t \overset{(\mathbf{level2})}{\to} u'$. There are two cases:
  - The case $t' \overset{(\mathbf{level2})}{\to} u'$. By Theorem 3.4.36 we have Δ''^+ and u exist such that $\Delta''^+ \vdash t \overset{(\mathbf{level2})}{\to^*} u$ and $\Delta''^+ \vdash u' \overset{(\mathbf{level2})}{\to^*} u$. We take $\Delta^+ = \Delta''^+$. The result follows.
  - The case $t' \overset{(\mathbf{level1})}{\to} u'$. By Lemma 3.4.38 and Theorem 3.4.31 we have Δ''^+ and u exist such that $\Delta''^+ \vdash t \overset{(\mathbf{level1})}{\to^*} u$ and $\Delta''^+ \vdash u' \overset{(\mathbf{level2})}{\to^*} u$. We take $\Delta^+ = \Delta''^+$. The result follows.

⊠

Theorem 3.4.42 is the main result in this Subsection, and the main result in this Chapter. It states that the reduction relation of Definition 3.4.3 is confluent.

Theorem 3.4.42: $\Delta \vdash - \to -$ (reduction with rules in $(\mathbf{level1}) \cup (\mathbf{level2})$) is confluent. That is, if $\Delta \vdash r \to^* s$ and $\Delta \vdash r \to^* t$ then there exists some u and some Δ^+ such that $\Delta^+ \vdash s \to^* u$ and $\Delta^+ \vdash t \to^* u$.

Proof. By induction on the path length of $\Delta \vdash r \to^* s$.

- The base case. Suppose $r \equiv s$. We take $\Delta^+ = \Delta$ and $u \equiv t$. The result follows.
- The inductive step. There are two cases:
  - The case (**level1**). Suppose $\Delta \vdash r \to^* s' \overset{\textbf{(level1)}}{\to} s$ and $\Delta \vdash r \to^* t$. By inductive hypothesis Δ'^+ and u' exist such that $\Delta'^+ \vdash s \to^* u'$ and $\Delta^+ \vdash t \to^* u'$. By Lemma 3.4.40 Δ''^+ and u exist such that $\Delta''^+ \vdash u' \overset{\textbf{(level1)}}{\to} u$ and $\Delta''^+ \vdash s \to^* u$. We take $\Delta^+ = \Delta''^+$. The result follows.
  - The case (**level2**). Suppose $\Delta \vdash r \to^* s' \overset{\textbf{(level2)}}{\to} s$ and $\Delta \vdash r \to^* t$. By inductive hypothesis Δ'^+ and u' exist such that $\Delta'^+ \vdash s \to^* u'$ and $\Delta^+ \vdash t \to^* u'$. By Lemma 3.4.41 Δ''^+ and u exist such that $\Delta''^+ \vdash u' \overset{\textbf{(level2)}}{\to} u$ and $\Delta''^+ \vdash s \to^* u$. We take $\Delta^+ = \Delta''^+$. The result follows.

⊠

The λ-calculus may be considered as an equational theory. Intuitively, two λ-terms are considered equal when they are β-equivalent. In a similar fashion, we may also consider the two-level λ-calculus as an equational theory. Definition 3.4.43 formalises the notion of equality on two-level λ-terms.

Definition 3.4.43: Define $\Delta \vdash - = -$ as the least reflexive, transitive and symmetric congruence closed under $\Delta \vdash - \to -$.

Corollary 3.4.44, the consistency of the two-level equational theory is an immediate consequence of Theorem 3.4.42.

Corollary 3.4.44: $\Delta \vdash - = -$ is consistent. That is, there are two terms not related by reflexive, transitive and symmetric closure of $\Delta \vdash - \to -$.

Proof. Take $\lambda a.\lambda b.a$ and $\lambda a.\lambda b.b$. These terms are distinct, and do not reduce to a common term. By Theorem 3.4.42 we have that $\Delta \vdash - = -$ is consistent. ⊠

3.5 Conclusions

This Chapter introduced the two-level λ-calculus, a novel context-calculus which uses nominal techniques. α-equivalence between terms is handled elegantly through the use of swappings, and problems with commuting β-reduction and hole filling are sidestepped by restricting reduction. The calculus is confluent (Theorem 3.4.42) and hence the associated equational theory over two-level λ-terms is consistent (Corollary 3.4.44).

Context-calculi aim to formalise the informal notion of context, as a 'term with holes', present in many areas of computer science research, most noticably equivalence proofs between program fragments (see, for example, the work of Pitts [Pit94]). In addition, context-calculi have an independent research interest, as they can be applied to the study of dynamic binding, module systems, and novel programming language designs. We survey these applications in Subsection 3.5.1.

3.5.1 Related work

Contexts have been the subject of a large amount of prior work, and many researchers have attempted to adapt context-calculi to many varied applications.

In particular, the two-level λ-calculus fits into a broader research programme, wherein various nominal context-calculi are investigated [Gab05, GL08, GM10b]. The two-level λ-calculus can be seen as a 'successor' to the NEW calculus of contexts [Gab05] and the λ context-calculus [GL08] with a more sophisticated notion of α-equivalence, and therefore more permissible reductions.

This Subsection goes on to survey a large body of existing work on other context-calculi.

Gabbay and Lengrand's λ-context calculus As mentioned, the λ-context calculus of Gabbay and Lengrand [GL08] may be viewed as a predecessor calculus to the two-level λ-calculus. The treatment of α-equivalence in the λ-context calculus is much less sophisticated than in the two-level λ-calculus. Practically, this means that reductions that are available in the two-level λ-calculus become stuck in the λ-context calculus.

Another notable difference between the two calculi are the number of levels of variables that each possess. As its name suggests, the two-level λ-calculus possesses two levels of variable. In contrast, the λ-context calculus possesses an infinity of levels of variable. We do not see this as a major disadvantage of the two-level λ-calculus, as we believe an infinity of levels of variable can be added quite easily.

Sato's $\lambda\mathcal{M}$. Sato's $\lambda\mathcal{M}$ calculus of metavariables [SSKI03] is the closest calculus in spirt to the two-level λ-calculus.

Sato places restrictions on reductions in $\lambda\mathcal{M}$ similar to the restrictions found in ($\rightarrow$**rs1**) and ($\rightarrow$**rs2**), in order to preserve confluence of the system. Notably, Sato prevents a redex from being reduced if it contains a variable of higher level than the level of the redex itself, where the 'level of a redex' is the level of the bound variable, suitably defined. Sato's calculus has an infinity of levels, compared to the two levels of the two-level λ-calculus, though in principle the two-level λ-calculus could be extended.

However, Sato's calculus differs from the two-level λ-calculus in a number of ways. Certain reductions in the two-level λ-calculus have no analogue in Sato's $\lambda\mathcal{M}$. For instance, in $\lambda\mathcal{M}$ the term $(\lambda a.(\lambda X.X))b$ is 'stuck', but reduces in the two-level λ-calculus: $\Delta \vdash (\lambda a.(\lambda X.X))b \rightarrow \lambda X.(X[a{\mapsto}b])$, providing we are working in a suitable freshness context. Sato characterises terms of this form as 'meaningless', and imposes a simple sorting system in order to rule them out as valid terms (his calculus is strongly normalising). In contrast, we prefer to allow as many reductions as possible.

Sato's $\lambda\kappa\epsilon$ calculus. Sato's $\lambda\kappa\epsilon$ calculus [SSK02] has first-class contexts and environments, and simple types. The calculus is 'pure' in the sense that it is known to be a conservative extension of the $\lambda\beta$ calculus, is strongly normalising and is confluent.

$\lambda\kappa\epsilon$ includes a notion of first-class environment that the two-level λ-calculus lacks. First-class environments are introduced to solve the well-known failure of commutation between hole-filling and β-reduction, whilst recovering a straightforward notion of reduction for open terms.

The two-level λ-calculus, $\lambda\mathcal{M}$ and Hashimoto's calculus, amongst others, solve this failure by placing restrictions on the reduction of applications (see ($\rightarrow$**rs1**) and ($\rightarrow$**rs2**)), or prohibiting such problem redexes from being reduced outright.

Hashimoto's calculus with first-class contexts. Hashimoto and Ohori consider a calculus with first-class contexts [HO01]. This calculus was later developed into an extension of the functional language Standard ML with first-class contexts [Has98], and later interpreted by Hideki in the $\lambda\epsilon$ calculus of environments [Hid00].

Hashimoto's calculus is typed to ensure confluence though strong-normalisation is not established. Only closed redexes may be reduced, and Hashimoto handles the α-equivalence of open terms by using contexts decorated with *variable renamers*.

Pitts' and Sands' higher-order approach to contexts. Sands [San98] introduced a context-calculus inspired by a notion of context due to Pitts[5]. Pitts represents contexts with higher-order function variables. This representation has the pleasing property that open terms may be identified up to α-equivalence [Pit94].

Pitts' technique is best demonstrated with an example. Suppose ϑ is a function variable. In Pitt's approach, all function variables have an explicit arity, dictating how many arguments they expect. A context is represented by a function variable applied to a vector of arguments, whose length is the same as the arity of the function variable. For instance, the two-level λ-calculus term $\Delta \vdash (\lambda X.X)a$ is represented in Pitts' and Sands' approach by the term $(\lambda x.\vartheta(x))y$. α-equivalence is handled in the normal manner, even in the presence of open terms. For example, $\lambda x.(\vartheta(x)) =_\alpha \lambda y.(\vartheta(y))$.

Hole filling is established by substituting for the function variable, using a *meta-abstraction*. The hole in $(\lambda x.(\vartheta(x)))y$ is filled with x by the substitution $[\vartheta{:=}\Lambda x.x]$, where $\Lambda x.x$ belongs to a copy of the λ-calculus existing at the metalevel. Performing the substitution, we obtain the term $(\lambda x.((\Lambda x.x)x))y \equiv (\lambda x.x)y$, as desired, where $\equiv$ is syntactic equality.

Pitts' meta-abstractions and Sato's first-class environments are structurally very similar. The salient difference between the two is where the features exist; first-class environments are object level entities, meta-abstractions exist at the metalevel.

De Bruijn's calculus of segments. De Bruijn introduced the calculus of segments [dB78] as part of the Automath project. Its intended purpose was the facilitation of definitions and abbreviations of mathematical expressions. Intuitively, a segment may be viewed as a λ-term with a single hole in a special place (in the words of Balsters, who studied a simply typed calculus of segments [Bal87], segments are 'terms with a kind of open end on the extreme right' [Bal94, pg. 346]).

[5]Mason traces the technique back further, to Aczel and Klopp [Mas99].

Jojgov and Geuvers' open proof terms. Jojgov and Geuvers [GJ02, Joj03] investigate proof terms for incomplete derivations in higher-order logic. Incomplete proofs often appear within theorem proving environments such as Isabelle, due to the preferred backward style of proof used, where complex goals are simplified into subgoals for solving. Proof terms can be obtained from complete derivations using the Curry-Howard correspondence in a straightforward manner. However, proof terms for incomplete derivations are harder to obtain, as refining a proof state may mean instantiating a hole with a bound variable.

Jojgov and Geuvers therefore turn to contexts to provide proof terms for incomplete derivations. However, unlike the two-level λ-calculus, there is no λ-abstraction of holes, and therefore no notion β-reduction. Rather, terms contain metavariables, labeled with sorts and equipped with a list of parameters, that can be instantiated later. The notion of substitution for holes used by Jojgov and Geuvers is more complex than the notion used in the two-level λ-calculus, where holes are simply filled with a term; metavariable substitution in Jojgov and Geuvers' calculus sets off a series of other substitutions.

Jojgov and Geuvers have a different focus than we do for investigating their calculus. However, a subset of the two-level λ-calculus, one-and-a-half level terms, are used by Gabbay and Mulligan to also provide proof terms for incomplete derivations (see [GM09b]).

Lee and Friedman's context enriched λ-calculus. Lee and Friedman [LF96] enrich the λ-calculus with contexts and four additional mechanisms: abstractions, simulating compiled evolved contexts (here an evolved context is a context without holes), an execution operator, to execute compiled code, and two additional operators, for building and destructing compiled code. Their context-calculus is geared toward applications in programming language design, specifically the design of module systems. As a consequence, they demonstrate how to express the linking of separately developed programs within the calculus, by careful α-renaming, and also simulate Garigue's label-selective λ-calculus [AKG95] and Lamping's transparent data parameters [Lam88].

Bognar and de Vrijer's λc calculus. Bognar and de Vrijer introduced a context-calculus λc [Bog02, BdV01]. The calculus comes equipped with a notion of *pretyping*. By varying the pretyping of terms, λc can emulate various other context-calculi, including de Bruijn's segments, and Hashimoto and Ohori's simply typed calculus. For this reason, λc is probably best understood as a family of context-calculi, rather than a single calculus.

The treatment of α-equivalence in λc is markedly different to the treatment in the two-level λ-calculus. Intuitively, problems with α-equivalence of open terms are handled using substitutions suspending on a hole. However, Bognar and de Vrijer observe that substitutions emerge as the result of a β-reduction step, and reductions needn't be applied eagerly. Unreduced redexes are therefore used to handle α-renamings. Such a scheme works better with multiple substitutions. λc therefore introduces special operators for binding multiple variables and applying multiple arguments.

Bognar's thesis [Bog02] also introduced the context cube $\lambda[]$. This is a contextual analogue of all eight corners of Barendregt's λ-cube [US06], replacing the λ-calculus with a context-calculus. Holes may appear not just in terms, but also in types.

Mason's calculus. Mason's context-calculus [Mas99] introduces two notions of variable replacements, *weak* and *strong* substitutions, which differ in how they behave when they encounter holes. Holes in terms may be labeled with weak substitutions, and this mechanism is used to handle α-equivalence. Strong substitutions are used to fill holes in terms.

CHAPTER 4

Permissive nominal terms and their unification

Abstract

The informal slogan of nominal techniques is 'ϵ-away from informal practice'. Informal presentation of syntax with binding possess two useful properties: the ability to always find a fresh name for any term ('always fresh'), and the ability to always rename a bound variable to something fresh ('always rename'). Nominal terms exist in a context of freshness assumptions. Depending on the context, it may not be possible to find some fresh atom to 'just rename' with, that is, nominal terms-in-context do not possess the 'always fresh' and 'always rename' properties that informal syntax enjoys. This became apparent in the confluence proof of the two-level λ-calculus, where 'freshening' operations for contexts were extensively employed.

Can we move nominal techniques closer to informal practice?

Permissive nominal terms are a variant of nominal terms that elide explicit freshness contexts. Instead, unknowns are labeled with an infinite and coinfinite set of atoms, called their permission sort, which controls how an unknown may be instantiated. The infinite and coinfinite nature of these permission sorts means that permissive nominal terms recover the 'always fresh' and 'always rename' properties that informal syntax possess, but nominal terms-in-context do not.

Freshness is also simplified through the use of permission sorts. A term's freshness now becomes a structural property of the term itself, and we may directly define the notion of 'free atoms' by induction on the structure of the term, as opposed to using a derivable freshness relation.

Like nominal terms, we introduce permissive nominal terms in the context of a unification algorithm. One appeal of nominal terms has been their excellent computational properties. We prove that these are not sacrificed by using permissive nominal techniques. In particular, permissive nominal unification is decidable and most general unifiers are computed. Further, permissive nominal techniques admit simpler solutions to unification problems: like first- and higher-order unification, permissive nominal unification solutions consist solely of a substitution, as opposed to the substitution-freshness constraint pair that constitutes a nominal unification solution.

Despite the differences in how freshness is handled, there remains a close correspondence between permissive nominal terms and nominal terms. We make this formal, by providing a non-trivial translation between the two, and demonstrating that unifiers are preserved under this translation.

4.1 Introduction

This Chapter introduces a new form of nominal term, the *permissive nominal term*. Permissive nominal terms possess all the flavour of their older counterparts (referred to as UPG nominal terms from hereonin), yet also possess reasoning properties that make working with permissive nominal terms closer to informal reasoning (that is, closer to the *style* of reasoning one uses when working with the λ-calculus, to take one example, with pencil and paper). These new properties, and why UPG nominal terms lack them, are now explained.

Nominal terms were introduced by Urban, Pitts and Gabbay as a 'toolkit' for encoding languages with binding [UPG04]. The decidability of the nominal unification algorithm has made nominal terms (and by extension, nominal techniques) an attractive proposition. αProlog, for instance, uses nominal terms as its term language, and replaces the standard first-order unification used by Prolog with nominal unification. The decidability of the nominal unification algorithm is in stark contrast with higher-order unification, which is known to be undecidable. As a bonus, following a body of work by Calvès and Fernández, nominal unification is known to be not only decidable, but efficiently so [CF08b, CF08a, Cal10].

Another widely perceived advantage of nominal techniques has been the ease with which languages with binding can be encoded, and the encodings manipulated formally. Various stock phrases have come to be associated with nominal encodings: 'close to informal practice', 'close to intuition', 'close to pen-and-paper proofs' etc. (see below for some concrete examples). Further, existing approaches (usually higher-order abstract syntax [PE88] and de Bruijn encodings [dB72]) for handling name binding are often considered markedly different from informal practice. For instance [Che05c, pgs 12 and 4]:

> The above proof should seem trivial, and this is the point: nominal abstract syntax facilitates a rigorous style of reasoning with names and binding that is close to intuition and informal practice.

> Higher-order language encodings are often so different from their informal "paper" presentations that proving "adequacy" (that is, equivalence of the encoding and the real language) is nontrivial, and elegant-looking encodings can be incorrect for subtle reasons.

Similar claims of the 'naturality' of nominal techniques, and their associated encodings, were made by Pitts [Pit03, pg 2]:

> Indeed, the work reported in [16,17,35] does do better, by providing a mathematical notion of 'sufficiently fresh name' that remains very close to the informal practice described above while enabling α-equivalence classes of parse trees to gain useful inductive/recursive properties.

Further, Berghofer and Urban have made similar claims about the 'naturality' of encodings in Nominal Isabelle compared to similar de Bruijn encodings. For instance [BU06, pg 13]:

One big advantage of the nominal data type package, we feel, is the relatively small "gap" between an informal proof on "paper" and an actual proof in a theorem prover.

The purpose of the above litany of quotations is to *build a case*, and justify the following claim:

Claim 4.1.1: The nominal research community believes one of the major advances of nominal techniques over competing approaches is the ease through which we can make, and reason about, encodings of languages with binding.

This claim represents the views of a number of scientists. Although subjective, the above quotes support the idea that the nominal research community thinks nominal encodings are more natural than encodings made with competing techniques.

However, recent research extending nominal techniques (presented in Chapter 3 and in recent publications [GM08b, GM09b]) has demonstrated that in fact nominal techniques aren't as close to informal practice as they could be. Consider the following two points, which are supported by references to Chapter 3 and elsewhere:

1. Suppose we are working on paper with the λ-calculus (to take a specific example—what follows works just as well with any other calculus or logic). For every λ-term g, there exists infinitely many $a \notin fv(g)$. Furthermore, at any time, we can always 'simply rename' a λ-bound variable to some fresh variable, as convenient.

   Now, let's examine the confluence proof in the two-level λ-calculus (Section 3.4, Chapter 3). Is it still true that, for a fixed Δ and term r, we have infinitely many a such that $\Delta \vdash a\#r$ (the 'always fresh' property)? Is it still true that we can 'simply rename' a λ-bound atom with a fresh atom, as necessary (the 'always rename' property)? On both counts, the answer is *no*.[1]

   In particular, these significant deficiencies force us to include Definition 3.4.10 in Chapter 3 (similar problems can also be seen in the confluence proof of [GM08b, GM09b]). As a direct result, we also have to 'thread' a freshness context through the statement of every lemma in the confluence proof. This is because we may need to push a substitution under a binder $\Delta \vdash (\lambda b.s)[a{\mapsto}t] \to \lambda b.(s[a{\mapsto}t])$. In order to do so, we need to meet a freshness side-condition $\Delta \vdash b\#t$, yet depending on the 'ambient' freshness context, we may not be able to meet this freshness side-condition.

   Here we see a significant mismatch between informal practice and what (current) nominal techniques readily permit. Such a mismatch is *dangerous* in that it gives the impression that nominal encodings are harder to work with than they need be.

2. Consider trying to unify two higher-order terms with Huet's higher-order unification algorithm. The solution, if one exists (and if the algorithm halts), will consist of a single substitution σ unifying the two terms.

[1] Consider trying to rename a bound atom in the empty freshness context.

> Suppose we try to unify two nominal terms with the Urban-Pitts-Gabbay algorithm [UPG04]. The solution, if one exists, will consists of a substitution paired with a series of freshness constraints. That is, nominal unification solutions are more complex than their higher- and first-order counterparts. Not only is this bad for 'selling' nominal techniques as a solution to the 'name binding problem', but programmers who would like to use the nominal unification algorithm are also forced into extra book keeping, keeping track of freshness constraints returned by the algorithm.

It seems obvious that, if these two deficiencies with nominal terms were fixed, we'd get something that allowed us to reason even closer to informal practice. However, is it the case that these problems are somehow *inherent* to nominal terms?

This chapter introduces a body of work that we believe demonstrates the two cited problems above are not an inherent deficiency of nominal terms, but an artefact of the design choices made by Urban, Pitts and Gabbay. In particular, we introduce a slightly modified form of nominal term, which we call *permissive nominal terms* (Definition 4.2.7). Here, unknowns are labeled with an infinite and coinfinite set of atoms, their permission sort (Definition 4.2.2). As the name suggests, a permission sort is used to control how an unknown is instantiated, and this mechanism allows us to recover the 'always fresh' (Corollary 4.2.17) and 'always rename' (Corollary 4.2.18) properties missing from nominal terms.

In addition, we go on to study the unification of permissive nominal terms (Section 4.3). We show that, like nominal unification, permissive nominal unification is correct (Theorem 4.3.58), decidable and synthesises principal unifiers (Theorem 4.3.56). However, unlike nominal unification, solutions to a permissive unification problem are simplified, consisting solely of a substitution. This brings the permissive nominal unification problem in line with its higher- and first-order counterparts. As a side-effect, from a programmer's perspective, the interface of the algorithm is significantly simplified. For this reason, the permissive unification algorithm is factored into two sub-algorithms: one dealing with *support reduction* (Subsubsection 4.3.2.1), and one that synthesises a principal unifier by decomposing an equality problem into simpler subproblems (Subsubsection 4.3.2.2).

Following the work introduced in this Chapter, we have two alternative forms of nominal term. Section 4.4 analyses the precise relationship between permissive nominal terms and their older counterparts. In particular, a translation between the two is presented, and we show that solutions in the permissive world are mapped to solutions in the nominal world, and vice-versa (Theorem 4.4.24).

This chapter is based on joint work with Gilles Dowek and Murdoch J. Gabbay, and can be found in two published papers [DGM09a, DGM10]. The initial idea for the work was due to Dowek and Gabbay. The majority of the mathematics was carried out by myself, though some was joint work with Gabbay.

4.2 Permissive terms

In this Section, we introduce basic definitions. Of particular interest will be Definition 4.2.2, the definition of *permission sorts*. These control how unknowns are instantiated, and their infinite and coinfinite nature is what gives permissive nominal terms their unique flavour. Of further note is Definition 4.2.7, the definition of *permissive terms*, which differs only slightly from the UPG notion of nominal term (compare with Definition 4.4.5).

Fix a countably infinite set of **atoms** $\mathbb{A}$. Use a, b, c, and so on, to range over atoms. We employ a permutative convention (that is, a and b denote *distinct* atoms). We note that, like all nominal work, we assume that (in)equality on atoms is a decidable relation.

Fix a countably infinite set of **term formers**. Use $\mathsf{f}, \mathsf{g}, \mathsf{h}$, and so on, to range over term formers. We employ a permutative convention (that is, f and g denote *distinct* term formers).

Definition 4.2.1: Call a set **coinfinite** when its setwise complement is infinite.

A set may be both infinite and coinfinite. For example, consider the even natural numbers, an infinite set whose complement (the odd natural numbers) is also infinite.

Definition 4.2.2: Fix an infinite and coinfinite set of atoms *comb*. Define **permission sorts** by the following rules:

- The set *comb* is a permission sort.
- If S is a permission sort, and A is a finite set, then $S \setminus A$ is a permission sort.
- If S is a permission sort, and A is a finite set, then $S \cup A$ is a permission sort.

Use S, S', T, T', and so on, to range over permission sorts.

Remark 4.2.3: We note that if S and T are permission sorts, then $S \cup T$ and $S \cap T$ are also permission sorts. We use this fact without comment in various proofs later in the Chapter.

We implicitly assume that subset, union, set difference and equality on permission sorts are decidable relations. We use these assumptions without comment throughout the rest of the Chapter.

Definition 4.2.4: To every permission sort S fix a countably infinite set of **(permissive) unknowns** of sort S. Use X, Y, Z, and so on, to range over unknowns. We do not employ a permutative convention with unknowns (that is X and Y may refer to the *same* unknown). Write X^S for 'X has sort S'.

If $S \neq S'$ then there is no particular connection between X^S and $X^{S'}$. However, to avoid confusion, we will avoid name clashes of this sort, wherever possible.

Definition 4.2.5: Suppose f is a function from atoms to atoms. Then:

$$nontriv(f) = \{a \mid f(a) \neq a\}$$

Definition 4.2.5 is potentially problematic, in that, depending on f, computing $nontriv(f)$ may be impossible. However, we are careful to only use *nontriv* with functions f where $nontriv(f)$ can easily be computed, or with functions that have an 'obvious' concrete realisation where this is true (for example, permutations, which may be implemented as finite lists of swappings).

Definition 4.2.6: Call a bijection π on atoms a **permutation** when $\mathit{nontriv}(\pi)$ is finite. Use π, π', π'', and so on, to range over permutations.

Write $\pi^{\text{-}1}$ for the **inverse** of a permutation, π. Write $\pi{\circ}\pi'$ for the functional **composition** of two permutations, so $(\pi{\circ}\pi')(a) = \pi(\pi'(a))$. Further, write id for the **identity** permutation, so that $id(a) = a$ for all atoms. Note, in particular, that $(\pi^{\text{-}1}{\circ}\pi) = id = (\pi{\circ}\pi^{\text{-}1})$.

Permutations of particular importance are **swappings**. Write $(b\ a)$ for the permutation mapping b to a to b, leaving all other c unchanged. We note that all swappings are self-inverse.

Definition 4.2.7: Define **permissive terms** by:

$$r, s, t ::= a \mid \pi{\cdot}X^S \mid [a]r \mid \mathsf{f}(r, \ldots, r)$$

We call $[a]r$ an abstraction, and say that 'a is abstracted in r'. Intuitively, $[a]r$ is an intensional binding construct; you may envisage $[-]-$ as what $\lambda -.-$, $\pi -.-$, etc. have in common, i.e. that a in $[a]r$ is bound within r.

Write $\equiv$ for syntactic identity. That is, $r \equiv s$ whenever r and s denote identical terms. For typographic convenience, we may abbreviate $id{\cdot}X^S$ as X^S.

The reader will notice the definition of terms given in Definition 4.2.7 is *extremely* similar to the traditional definition of nominal terms (for instance [GM09a, Definition 2.4]). The only difference is the presence of a permission sort tagging permissive unknowns, yet the following Subsection will demonstrate that this small change makes a significant difference to the reasoning properties that the two notions of term enjoy.

Remark 4.2.8: We remark that, despite permission sorts being infinite and coinfinite in nature, permissive nominal syntax is *not* infinitary syntax, any more than the Church-typed λ-term $\lambda a : \mathbb{N}.a : \mathbb{N} \to \mathbb{N}$ is infinitary syntax. In both cases, terms are taken to be finite trees (finitary syntax), albeit labeled with an infinite set. These infinite sets need not be expanded.

4.2.1 Permutations and α-equivalence

This Subsection introduces a permutation action, and then a notion of α-equivalence for permissive nominal terms. The majority of results are routine checks of correctness for the α-equivalence relation (equivarience is checked in Lemma 4.2.23, and free atom preservation is checked in Lemma 4.2.24), as well as routine lemmas concerning the permutation action. Corollaries 4.2.17 and 4.2.18 are important, and mark a significant departure from the properties of nominal terms. However, the main result in this Subsection is Theorem 4.2.27, demonstrating that α-equivalence is an equivalence relation.

Definition 4.2.9: Define a **permutation action** on terms by:

$$\pi{\cdot}a \equiv \pi(a) \quad \pi{\cdot}(\pi'{\cdot}X^S) \equiv (\pi{\circ}\pi'){\cdot}X^S \quad \pi{\cdot}[a]r \equiv [\pi(a)](\pi{\cdot}r) \quad \pi{\cdot}\mathsf{f}(r_1, \ldots, r_n) \equiv \mathsf{f}(\pi{\cdot}r_1, \ldots, \pi{\cdot}r_n)$$

One of the defining features of nominal techniques is the handling of α-equivalence through injective permutations.

$$\frac{}{a =_\alpha a}\,(=_\alpha\mathbf{a}) \qquad \frac{r_i =_\alpha s_i \quad (1 \leq i \leq n)}{\mathsf{f}(r_1, \dots, r_n) =_\alpha \mathsf{f}(s_1, \dots, s_n)}\,(=_\alpha\mathsf{f}) \qquad \frac{r =_\alpha s}{[a]r =_\alpha [a]s}\,(=_\alpha[\mathbf{a}])$$

$$\frac{(b\ a)\cdot r =_\alpha s \quad (b \notin fa(r))}{[a]r =_\alpha [b]s}\,(=_\alpha[\mathbf{b}]) \qquad \frac{\pi|_S = \pi'|_S}{\pi\cdot X^S =_\alpha \pi'\cdot X^{S'}}\,(=_\alpha\mathbf{X})$$

Figure 4.1 Rules for α-equivalence

Definition 4.2.10: If $S \subseteq \mathbb{A}$ define the **pointwise permutation action** by:

$$\pi\cdot S = \{\pi(a) \mid a \in S\}$$

Definition 4.2.11: Define the **free atoms** of a term by:

$$fa(a) = \{a\} \qquad fa(\pi\cdot X^S) = \pi\cdot S \qquad fa([a]r) = fa(r) \setminus \{a\} \qquad fa(\mathsf{f}(r_1, \dots, r_n)) = \bigcup_{1 \leq i \leq n} fa(r_i)$$

Definition 4.2.12: If $S \subseteq \mathbb{A}$ then $\pi|_S$ is the **restriction** of π to S. That is, $\pi|_S$ is the partial function such that:

$$\pi|_S(a) = \pi(a) \text{ if } a \in S \qquad \pi|_S(a) \text{ is undefined, otherwise}$$

Definition 4.2.13: Define a notion of α**-equivalence** on terms, by the rules in Figure 4.1.

Definition 4.2.14: Define the **unknowns** of a term by:

$$fV(a) = \emptyset \qquad fV([a]r) = fV(r) \qquad fV(\pi\cdot X^S) = \{X^S\} \qquad fV(\mathsf{f}(r_1, \dots, r_n)) = \bigcup_{1 \leq i \leq n} fV(r_i)$$

Definition 4.2.15: Define the **atoms** of a term by:

$$\mathit{one}(a) = \{a\} \quad \mathit{one}(\mathsf{f}(r_1, \dots, r_n)) = \bigcup_{1 \leq i \leq n} \mathit{one}(r_i) \quad \mathit{one}([a]r) = \mathit{one}(r) \cup \{a\}$$
$$\mathit{one}(\pi\cdot X^S) = \mathit{nontriv}(\pi)$$

The set of atoms of a term is a purely technical notion used in the proof of Corollary 4.2.17. Similarly, the following is a technical result used solely in the same proof.

Lemma 4.2.16: $fa(r) \subseteq \mathit{one}(r) \cup \bigcup\{S \mid X^S \in fV(r)\}$

Proof. By induction on r.

- The case a. By Definition 4.2.15 we have $\mathit{one}(a) = \{a\}$. By Definition 4.2.11 we have $fa(a) = \{a\}$. The result follows.
- The case $\pi\cdot X^S$. By Definition 4.2.15 we have $\mathit{one}(\pi\cdot X^S) = \mathit{nontriv}(\pi) \cup S$. By Definition 4.2.11 we have $fa(\pi\cdot X^S) = \pi\cdot S$. The result follows.
- The case $\mathsf{f}(r_1, \dots, r_n)$. By Definition 4.2.11 we have $fa(\mathsf{f}(r_1, \dots, r_n)) = \bigcup_{1 \leq i \leq n} fa(r_i)$. By inductive hypothesis $\bigcup_{1 \leq i \leq n} fa(r_i) \subseteq \bigcup_{1 \leq i \leq n} \mathit{one}(r_i) \cup \bigcup\{S \mid X^S \in fV(r_i)\}$. By Definition 4.2.15 we have $\bigcup_{1 \leq i \leq n} \mathit{one}(r_i) \cup \bigcup\{S \mid X^S \in fV(r_i)\} = \mathit{one}(\mathsf{f}(r_1, \dots, r_n)) \cup \bigcup\{S \mid X^S \in fV(\mathsf{f}(r_1, \dots, r_n))\}$. The result follows.

- The case $[a]r$. By Definition 4.2.15 we have $\mathit{one}([a]r) = \{a\} \cup \mathit{one}(r)$. By Definition 4.2.11 we have $fa([a]r) = fa(r) \setminus \{a\}$. By inductive hypothesis $fa(r) \subseteq \mathit{one}(r) \cup \bigcup\{S \mid X^S \in fV(r)\}$. The result follows.

⊠

The following two Corollaries mark a significant difference between permissive nominal terms and nominal terms. Corollary 4.2.17 states that, for any finite collection of terms, we can always find an atom that is fresh for that collection (the 'always fresh' property). Corollary 4.2.18 states that, we can always rename a bound atom to something fresh when needed (the 'always rename' property).

Corollary 4.2.17: For every $r_1, \dots, r_n$ there are infinitely many b such that $b \notin \{fa(r_i) \mid 1 \leq i \leq n\}$.

Proof. By Lemma 4.2.16, $fa(r_i) \subseteq \mathit{one}(r_i) \cup \bigcup\{S \mid X^S \in fV(r_i)\}$. Since the syntax of r_i is finite, $\mathit{one}(r_i)$ and $fV(r_i)$ are finite also. It follows that $\bigcup\{S \mid X^S \in fV(r_i)\}$ for some i is coinfinite. The result follows. ⊠

Corollary 4.2.18: For every r and a there exists infinitely many b such that $[a]r =_\alpha [b]s$ for some s.

Proof. Immediate, from Corollary 4.2.17 and $(=_\alpha[\mathbf{b}])$. ⊠

Lemma 4.2.19: $id{\cdot}r \equiv r$

Proof. By induction on r. See Appendix B. ⊠

Lemma 4.2.20: $\pi{\cdot}(\pi'{\cdot}r) \equiv (\pi \circ \pi'){\cdot}r$

Proof. By induction on r. See Appendix B. ⊠

The following two lemmas are checks that the set of free atoms and the set of unknowns of a term are equivariant:

Lemma 4.2.21: $\pi{\cdot}fa(r) = fa(\pi{\cdot}r)$

Proof. By induction on r. See Appendix B. ⊠

Lemma 4.2.22: $fV(\pi{\cdot}r) = fV(r)$

Proof. By induction on r. See Appendix B. ⊠

The following two lemmas are basic correctness checks for the α-equivalence relation. Lemma 4.2.23 is a check that α-equivalence is equivariant (invariant under permutation of atoms). Lemma 4.2.24 is a check that α-equivalence only adjusts *bound* atoms, and neither increases nor reduces the set of free atoms of a term.

Lemma 4.2.23: If $r =_\alpha s$ then $\pi\cdot r =_\alpha \pi\cdot s$.

Proof. By induction on the derivation of $r =_\alpha s$.

- The case $(=_\alpha\mathbf{a})$. Since $\pi(a) =_\alpha \pi(a)$ always, using $(=_\alpha\mathbf{a})$.
- The case $(=_\alpha\mathsf{f})$. Suppose $r_i =_\alpha s_i$ for $1 \leq i \leq n$. By inductive hypotheses $\pi\cdot r_i =_\alpha \pi\cdot s_i$ for $1 \leq i \leq n$. Using $(=_\alpha\mathsf{f})$ we have $\mathsf{f}(\pi\cdot r_1, \dots, \pi\cdot r_n) =_\alpha \mathsf{f}(\pi\cdot s_1, \dots, \pi\cdot s_n)$. By Definition 4.2.9, $\mathsf{f}(\pi\cdot r_1, \dots, \pi\cdot r_n) \equiv \pi\cdot\mathsf{f}(r_1, \dots, r_n)$, and the result follows.
- The case $(=_\alpha[\mathbf{a}])$. Suppose $r =_\alpha s$. By inductive hypothesis $\pi\cdot r =_\alpha \pi\cdot s$. Using $(=_\alpha[\mathbf{a}])$ we have $[\pi(a)](\pi\cdot r) =_\alpha [\pi(a)](\pi\cdot s)$. By Definition 4.2.9, $[\pi(a)](\pi\cdot r) \equiv \pi\cdot[a]r$, and the result follows.
- The case $(=_\alpha[\mathbf{b}])$. Suppose $(b\ a)\cdot r =_\alpha s$ and $b \notin fa(r)$. By inductive hypothesis $\pi\cdot((b\ a)\cdot r) =_\alpha \pi\cdot s$. By Lemma 4.2.21, $\pi(b) \notin fa(\pi\cdot r)$. By elementary properties of permutations, $\pi\cdot((b\ a)\cdot r) \equiv (\pi(b)\ \pi(a))\cdot(\pi\cdot r)$. Using $(=_\alpha[\mathbf{b}])$ we have $[\pi(a)](\pi\cdot r) =_\alpha [\pi(b)](\pi\cdot s)$. By Definition 4.2.9, $[\pi(a)](\pi\cdot r) \equiv \pi\cdot[a]r$, and the result follows.
- The case $(=_\alpha\mathbf{X})$. Suppose $\pi'|_S = \pi''|_S$ so that $\pi\cdot X^S =_\alpha \pi'\cdot X^S$. Then $\pi\circ\pi'|_S = \pi\circ\pi''|_S$, therefore $(\pi\circ\pi')\cdot X^S =_\alpha (\pi\circ\pi'')\cdot X^S$. By Definition 4.2.9, $(\pi\circ\pi')\cdot X^S \equiv \pi\cdot(\pi'\cdot X^S)$, and the result follows.

⊠

Lemma 4.2.24: If $r =_\alpha s$ then $fa(r) = fa(s)$.

Proof. By induction on the derivation of $r =_\alpha s$.

- The case $(=_\alpha\mathbf{a})$. Since $fa(a) = fa(a)$ always.
- The case $(=_\alpha\mathsf{f})$. Suppose $r_i =_\alpha s_i$ for $1 \leq i \leq n$. By inductive hypotheses $fa(r_i) = fa(s_i)$ for $1 \leq i \leq n$. Therefore $\bigcup_{1\leq i\leq n} fa(r_i) = \bigcup_{1\leq i\leq n} fa(s_i)$. Then $fa(\mathsf{f}(r_1, \dots, r_n)) = fa(\mathsf{f}(s_1, \dots, s_n))$, as required.
- The case $(=_\alpha[\mathbf{a}])$. Suppose $r =_\alpha s$. By inductive hypothesis $fa(r) = fa(s)$. Therefore $fa(r) \setminus \{a\} = fa(s) \setminus \{a\}$. Then $fa([a]r) = fa([a]s)$, as required.
- The case $(=_\alpha[\mathbf{b}])$. Suppose $[a]r =_\alpha [b]s$ by $(=_\alpha[\mathbf{b}])$ so that $b \notin fa(r)$. We aim to show $fa([a]r) = fa([b]s)$, or $fa(r) \setminus \{a\} = fa(s) \setminus \{b\}$. As $b \notin fa(r)$ we have $(b\ a)\cdot(fa(r) \setminus \{a\}) = fa((b\ a)\cdot r) \setminus \{b\}$. By inductive hypothesis $fa((b\ a)\cdot r) = fa(s)$. The result follows.
- The case $(=_\alpha\mathbf{X})$. Suppose $\pi|_S = \pi'|_S$ therefore $\pi\cdot S = \pi'\cdot S$. By Definition 4.2.11 we have $fa(\pi\cdot X^S) = \pi\cdot S$. It is a fact that $\pi\cdot S = \pi'\cdot S$. By Definition 4.2.11 we have $\pi'\cdot S = fa(\pi'\cdot X^S)$. The result follows.

⊠

We introduce the following notion as an aid for inductive proofs (for instance, the proof of Theorem 4.2.27). Lemma 4.2.26 checks that the size of a term is invariant under atom permutation.

Definition 4.2.25: Define the **size** of a term by:

$$size(a) = 0 \qquad size(\pi\cdot X^S) = 0 \qquad size([a]r) = 1 + size(r) \qquad size(\mathsf{f}(r_1, \dots, r_n)) = 1 + \sum_{1\leq i\leq n} size(r_i)$$

Lemma 4.2.26: $size(\pi{\cdot}r) = size(r)$

Proof. By induction on r. See Appendix B. ⊠

Theorem 4.2.27, demonstrating that α-equivalence is an equivalence relation, is our main result in this Subsection.

Theorem 4.2.27: $=_\alpha$ is reflexive, symmetric and transitive.

Proof. We handle the three cases separately.

- The reflexivity case. We show $r =_\alpha r$ by induction on r.
  - The case a. Using $(=_\alpha\mathbf{a})$ we obtain $a =_\alpha a$. The result follows.
  - The case $\pi{\cdot}X^S$. It is a fact that $\pi|_S = \pi|_S$. Using $(=_\alpha\mathbf{X})$ we obtain $\pi{\cdot}X^S =_\alpha \pi{\cdot}X^S$. The result follows.
  - The case $\mathsf{f}(r_1, \ldots, r_n)$. Suppose $r_i =_\alpha r_i$ for $1 \leq i \leq n$. Using $(=_\alpha\mathsf{f})$ we obtain $\mathsf{f}(r_1, \ldots, r_n) =_\alpha \mathsf{f}(r_1, \ldots, r_n)$. The result follows.
  - The case $[a]r$. Suppose $r =_\alpha r$. Using $(=_\alpha[\mathbf{a}])$ we obtain $[a]r =_\alpha [a]r$. The result follows.
- The symmetry case. We show $s =_\alpha r$ if $r =_\alpha s$ by induction on the derivation of $r =_\alpha s$.
  - The case $(=_\alpha\mathbf{a})$. Using $(=_\alpha\mathbf{a})$ we obtain $a =_\alpha a$. The result follows.
  - The case $(=_\alpha\mathsf{f})$. Suppose $r_i =_\alpha s_i$ for $1 \leq i \leq n$. By inductive hypotheses $s_i =_\alpha r_i$ for $1 \leq i \leq n$. Using $(=_\alpha\mathsf{f})$ we obtain $\mathsf{f}(s_1, \ldots, s_n) =_\alpha \mathsf{f}(r_1, \ldots, r_n)$. The result follows.
  - The case $(=_\alpha[\mathbf{a}])$. Suppose $r =_\alpha s$. By inductive hypothesis $s =_\alpha r$. Using $(=_\alpha[\mathbf{a}])$ we obtain $[a]s =_\alpha [a]r$. The result follows.
  - The case $(=_\alpha[\mathbf{b}])$. Suppose $(b\ a){\cdot}r =_\alpha s$ with $b \notin fa(r)$. By Lemma 4.2.23 we have $(b\ a){\cdot}((b\ a){\cdot}r) =_\alpha (b\ a){\cdot}s$. By Lemma 4.2.20 we have $((b\ a)\circ(b\ a)){\cdot}r =_\alpha (b\ a){\cdot}s$. It is a fact that swappings are self inverse, so $id{\cdot}r =_\alpha (b\ a){\cdot}s$. By Lemma 4.2.19 we have $r =_\alpha (b\ a){\cdot}s$. By inductive hypothesis $(b\ a){\cdot}s =_\alpha r$. By Lemma 4.2.21 we have $a \notin fa((b\ a){\cdot}r)$. By Lemma 4.2.24 we have $a \notin fa(s)$. Using $(=_\alpha[\mathbf{b}])$ we obtain $[b]s =_\alpha [a]r$. The result follows.
  - The case $(=_\alpha\mathbf{X})$. It is a fact that equality on partial functions is symmetric. The result follows.
- The transitivity case. We show $r =_\alpha t$ if $r =_\alpha s$ and $s =_\alpha t$ by induction on the size of r.
  - The case a. Suppose $a =_\alpha s$. By the structure of the derivation rules we have $s \equiv a$. Similarly, suppose $a =_\alpha t$. By the structure of the derivation rules we have $t \equiv a$. Using $(=_\alpha\mathbf{a})$ we obtain $a =_\alpha a$. The result follows.
  - The case $\pi{\cdot}X^S$. It is a fact that equality on partial functions is transitive. The result follows.

- The case $\mathsf{f}(r_1,\ldots,r_n)$. Suppose $r_i =_\alpha s_i$ and $s_i =_\alpha t_i$ for $1 \leq i \leq n$. By inductive hypotheses $r_i =_\alpha t_i$ for $1 \leq i \leq n$. Using $(=_\alpha\mathsf{f})$ we obtain $\mathsf{f}(r_1,\ldots,r_n) =_\alpha \mathsf{f}(t_1,\ldots,t_n)$. The result follows.
- The case $[a]r$. We consider the most general case, as all other cases are similar. Suppose $(b\ a)\cdot r =_\alpha s$ and $(c\ b)\cdot s =_\alpha t$ with $b \notin fa(r)$ and $c \notin fa(s)$. By Lemma 4.2.23 we have $(c\ b)\cdot((b\ a)\cdot r) =_\alpha (c\ b)\cdot s$. By Lemma 4.2.26 we have $(c\ b)\cdot((b\ a)\cdot r) =_\alpha t$. By Lemma 4.2.20 we have $((c\ b)\circ(b\ a))\cdot r =_\alpha t$ therefore $(c\ a)\cdot r =_\alpha t$. By Lemma 4.2.24 we have $c \notin fa((b\ a)\cdot r)$. By Lemma 4.2.21 we have $c \notin (b\ a)\cdot fa(r)$ therefore $c \notin fa(r)$. Using $(=_\alpha[\mathbf{b}])$ we obtain $[a]r =_\alpha [c]t$. The result follows.

⊠

4.2.2 Substitutions

This Subsection introduces a notion of substitution for unknowns, and a substitution action on terms. The main results are Theorem 4.2.30, a demonstration that the substitution action reduces or preserves the free atoms of a term, Theorem 4.2.33, and Theorem 4.2.36 demonstrating that performing a substitution twice on a term is equivalent to performing a single, composed substitution.

Definition 4.2.28: A **substitution** θ is a function from unknowns to terms such that $fa(\theta(X^S)) \subseteq S$. Use $\theta, \theta', \theta''$, and so on, to range over substitutions.

Write id for the **identity substitution** mapping X^S to $id\cdot X^S$. It will always be clear from context whether id refers to the identity permutation or identity substitution.

Suppose $fa(t) \subseteq S$. Write $[X^S{:=}t]$ for the substitution such that:

$$[X^S{:=}t](X^S) \equiv t \quad \text{and} \quad [X^S{:=}t](Y^T) \equiv id\cdot Y^T \text{ for all other } Y^T$$

In Definition 4.2.28, we note again that, if $S \neq S'$, then there is no particular connection between X^S and $X^{S'}$.

Definition 4.2.29: Define a **substitution action** on terms by:

$$a\theta \equiv a \quad (\pi\cdot X^S)\theta \equiv \pi\cdot\theta(X^S) \quad ([a]r)\theta \equiv [a](r\theta) \quad \mathsf{f}(r_1,\ldots,r_n)\theta \equiv \mathsf{f}(r_1\theta,\ldots,r_n\theta)$$

As is standard for nominal terms, permissive nominal substitutions make no attempt to avoid capture by bound atoms. For instance, if $S = \mathit{comb}$ and $a \in \mathit{comb}$, we have $([a]X^S)[X^S{:=}a] \equiv [a]a$.

Theorem 4.2.30: $fa(r\theta) \subseteq fa(r)$

Proof. By induction on r.

- The case a. By Definition 4.2.29 we have $a\theta \equiv a$. The result follows.
- The case $\pi\cdot X^S$. By Definition 4.2.29 we have $(\pi\cdot X^S)\theta \equiv \pi\cdot\theta(X^S)$. There are two cases:

- The case $\theta(X^S) = id{\cdot}X^S$. The result follows.
- The case $\theta(X^S) \neq id{\cdot}X^S$. Suppose that $\theta(X^S) = id{\cdot}Y^T$. By Definition 4.2.28 we have $fa(id{\cdot}Y^T) \subseteq S$. By Definition 4.2.11 we have $fa(id{\cdot}Y^T) = id{\cdot}T = T$, therefore $T \subseteq S$. By Definition 4.2.9 we have $\pi{\cdot}(id{\cdot}Y^T) \equiv (\pi \circ id){\cdot}Y^T$. It is a fact that $\pi \circ id = \pi$ and as $T \subseteq S$ then $\pi{\cdot}T \subseteq \pi{\cdot}S$. By Definition 4.2.11 we have $fa(\pi{\cdot}Y^T) \subseteq fa(\pi{\cdot}X^S)$. The result follows.

- The case $[a]r$. By Definition 4.2.29 we have $fa(([a]r)\theta) = fa([a](r\theta))$. By Definition 4.2.11 we have $fa([a](r\theta)) = fa(r\theta) \setminus \{a\}$. By inductive hypothesis $fa(r\theta) \setminus \{a\} \subseteq fa(r) \setminus \{a\}$. By Definition 4.2.11 we have $fa(r) \setminus \{a\} = fa([a]r)$. The result follows.
- The case $\mathsf{f}(r_1, \ldots, r_n)$. By Definition 4.2.29 we have $fa(\mathsf{f}(r_1, \ldots, r_n)\theta) = fa(\mathsf{f}(r_1\theta, \ldots, r_n\theta))$. By Definition 4.2.11 we have $fa(\mathsf{f}(r_1\theta, \ldots, r_n\theta)) = \bigcup_{1 \leq i \leq n} fa(r_i\theta)$. By inductive hypotheses $\bigcup_{1 \leq i \leq n} fa(r_i\theta) \subseteq \bigcup_{1 \leq i \leq n} fa(r_i)$. By Definition 4.2.11 we have $\bigcup_{1 \leq i \leq n} fa(r_i) = fa(\mathsf{f}(r_1, \ldots, r_n))$. The result follows.

⊠

The following is a basic commutation property between permutations and substitutions. Intuitively, Lemma 4.2.31 states that renaming first, then substituting, or substituting and then renaming both amount to the same thing.

Lemma 4.2.31: $\pi{\cdot}(r\theta) \equiv (\pi{\cdot}r)\theta$

Proof. By induction on r.

- The case a. By Definition 4.2.29 we have $a\theta \equiv a$. The result follows.
- The case $\pi'{\cdot}X^S$. By Definition 4.2.29 we have $\pi{\cdot}((\pi'{\cdot}X^S)\theta) \equiv \pi{\cdot}(\pi'{\cdot}\theta(X^S))$. By Lemma 4.2.20 we have $\pi{\cdot}(\pi'{\cdot}\theta(X^S)) \equiv (\pi \circ \pi'){\cdot}\theta(X^S)$. By Definition 4.2.29 we have $(\pi \circ \pi'){\cdot}\theta(X^S) \equiv ((\pi \circ \pi'){\cdot}X^S)\theta$. By Definition 4.2.9 we have $((\pi \circ \pi'){\cdot}X^S)\theta \equiv (\pi{\cdot}(\pi'{\cdot}X^S))\theta$. The result follows.
- The case $[a]r$. By Definition 4.2.29 we have $\pi{\cdot}(([a]r)\theta) \equiv \pi{\cdot}[a](r\theta)$. By Definition 4.2.9 we have $\pi{\cdot}[a](r\theta) \equiv [\pi(a)](\pi{\cdot}(r\theta))$. By inductive hypothesis $[\pi(a)](\pi{\cdot}(r\theta)) \equiv [\pi(a)]((\pi{\cdot}r)\theta)$. By Definition 4.2.29 we have $[\pi(a)]((\pi{\cdot}r)\theta) \equiv ([\pi(a)](\pi{\cdot}r))\theta$. By Definition 4.2.9 we have $([\pi(a)](\pi{\cdot}r))\theta \equiv (\pi{\cdot}[a]r)\theta$. The result follows.
- The case $\mathsf{f}(r_1, \ldots, r_n)$. By Definition 4.2.29 we have $\pi{\cdot}(\mathsf{f}(r_1, \ldots, r_n)\theta) \equiv \pi{\cdot}\mathsf{f}(r_1\theta, \ldots, r_n\theta)$. By Definition 4.2.9 we have $\pi{\cdot}\mathsf{f}(r_1\theta, \ldots, r_n\theta) \equiv \mathsf{f}(\pi{\cdot}(r_1\theta), \ldots, \pi{\cdot}(r_n\theta))$. By inductive hypotheses $\mathsf{f}(\pi{\cdot}(r_1\theta), \ldots, \pi{\cdot}(r_n\theta)) \equiv \mathsf{f}((\pi{\cdot}r_1)\theta, \ldots, (\pi{\cdot}r_n)\theta)$. By Definition 4.2.29 we have $\mathsf{f}((\pi{\cdot}r_1)\theta, \ldots, (\pi{\cdot}r_n)\theta) \equiv \mathsf{f}(\pi{\cdot}r_1, \ldots, \pi{\cdot}r_n)\theta$. By Definition 4.2.9 we have $\mathsf{f}(\pi{\cdot}r_1, \ldots, \pi{\cdot}r_n)\theta \equiv (\pi{\cdot}\mathsf{f}(r_1, \ldots, r_n))\theta$. The result follows.

⊠

The following is a technical result used in the proof of Lemma 4.3.49. As it concerns substitutions, we leave it here.

Lemma 4.2.32: $fV(r[X^S{:=}t]) \subseteq fV(r) \cup fV(t)$

Proof. By induction on r, using Lemma 4.2.22. See Appendix B. ⊠

Intuitively, the following result states that, if a pair of substitutions 'agree' on all unknowns occurring in a term, then performing the substitutions on that term leaves us with an α-equivalent pair of terms.

Theorem 4.2.33: If $\theta(X^S) =_\alpha \theta'(X^S)$ for all $X^S \in fV(r)$ then $r\theta =_\alpha r\theta'$.

Proof. By induction on r.

- The case a. By Definition 4.2.14 we have $fV(a) = \emptyset$. There is nothing to prove.
- The case $\pi{\cdot}X^S$. Suppose $\theta(X^S) =_\alpha \theta'(X^S)$. By Lemma 4.2.23 we have $\pi{\cdot}\theta(X^S) =_\alpha \pi{\cdot}\theta'(X^S)$. By Definition 4.2.29 we have $(\pi{\cdot}X^S)\theta =_\alpha (\pi{\cdot}X^S)\theta'$. The result follows.
- The case $[a]r$. Suppose $\theta(X^S) =_\alpha \theta'(X^S)$ for all $X^S \in fV([a]r)$. By Definition 4.2.14 we have $\theta(X^S) =_\alpha \theta'(X^S)$ for all $X^S \in fV(r)$, as $fV([a]r) = fV(r)$. By inductive hypothesis $r\theta =_\alpha r\theta'$. Using $(=_\alpha[\mathbf{a}])$ we obtain $[a](r\theta) =_\alpha [a](r\theta')$. By Definition 4.2.29 we have $([a]r)\theta =_\alpha ([a]r)\theta'$. The result follows.
- The case $\mathsf{f}(r_1, \dots, r_n)$. Suppose $\theta(X^S) =_\alpha \theta'(X^S)$ for all $X^S \in fV(\mathsf{f}(r_1, \dots, r_n))$. By Definition 4.2.14 $\theta(X^S) =_\alpha \theta'(X^S)$ for all $X^S \in fV(r_i)$ for $1 \leq i \leq n$, as $fV(\mathsf{f}(r_1, \dots, r_n)) = \bigcup_{1 \leq i \leq n} fV(r_i)$. By inductive hypotheses $r_i\theta =_\alpha r_i\theta'$ for $1 \leq i \leq n$. Using $(=_\alpha\mathsf{f})$ we have $\mathsf{f}(r_1\theta, \dots, r_n\theta) =_\alpha \mathsf{f}(r_1\theta', \dots, r_n\theta')$. By Definition 4.2.29 $\mathsf{f}(r_1, \dots, r_n)\theta =_\alpha \mathsf{f}(r_1, \dots, r_n)\theta$ as required.

⊠

Lemma 4.2.34 checks that the substitution action preserves α-equivalence.

Lemma 4.2.34: If $r =_\alpha s$ then $r\theta =_\alpha s\theta$.

Proof. By induction on the derivation of $r =_\alpha s$.

- The case $(=_\alpha\mathbf{a})$. By Definition 4.2.29 we have $a\theta \equiv a$. The result follows.
- The case $(=_\alpha\mathbf{X})$. Suppose $\pi|_S = \pi'|_S$ so that $\pi{\cdot}X^S =_\alpha \pi'{\cdot}X^S$. By Definition 4.2.28 we have $fa(\theta(X^S)) \subseteq S$. The result follows.
- The case $(=_\alpha[\mathbf{a}])$. Suppose $r =_\alpha s$. By inductive hypothesis $r\theta =_\alpha s\theta$. Using $(=_\alpha[\mathbf{a}])$ we obtain $[a](r\theta) =_\alpha [a](s\theta)$. By Definition 4.2.29 we have $[a](r\theta) \equiv ([a]r)\theta$. The result follows.
- The case $(=_\alpha[\mathbf{b}])$. Suppose $(b\ a){\cdot}r =_\alpha s$ with $b \notin fa(r)$. By inductive hypothesis $((b\ a){\cdot}r)\theta =_\alpha s\theta$. By Theorem 4.2.30 we have $b \notin fa(r\theta)$. By Lemma 4.2.31 we have $(b\ a){\cdot}(r\theta) =_\alpha s\theta$. Using $(=_\alpha[\mathbf{b}])$ we obtain $[a](r\theta) =_\alpha [b](s\theta)$. By Definition 4.2.29 we have $[a](r\theta) \equiv ([a]r)\theta$. The result follows.
- The case $(=_\alpha\mathsf{f})$. Suppose $r_i =_\alpha s_i$ for $1 \leq i \leq n$. By inductive hypothesis $r_i\theta =_\alpha s_i\theta$. Using $(=_\alpha\mathsf{f})$ we obtain $\mathsf{f}(r_1\theta, \dots, r_n\theta) =_\alpha \mathsf{f}(s_1\theta, \dots, s_n\theta)$. By Definition 4.2.29 we have $\mathsf{f}(r_1\theta, \dots, r_n\theta) \equiv \mathsf{f}(r_1, \dots, r_n)\theta$. The result follows.

⊠

Definition 4.2.35: Define **composition** of substitutions by $(\theta\circ\theta')(X^S) \equiv (\theta(X^S))\theta'$.

We note that, compared to composition of permutations (Definition 4.2.6), composition of substitutions is *reversed*. This reflects the fact that we usually write substitutions in a postfix style (for instance, $r[X^S{:=}t]$).

Theorem 4.2.36: $r(\theta\circ\theta') \equiv (r\theta)\theta'$

Proof. By induction on r.

- The case a. By Definition 4.2.29 we have $a\theta \equiv a$. The result follows.
- The case $\pi\cdot X^S$. By Definition 4.2.29 we have $(\pi\cdot X^S)(\theta\circ\theta') \equiv \pi\cdot((\theta\circ\theta')(X^S))$. By Definition 4.2.35 we have $\pi\cdot((\theta\circ\theta')(X^S)) \equiv \pi\cdot(\theta(X^S)\theta')$. By Lemma 4.2.31 we have $\pi\cdot(\theta(X^S)\theta') \equiv (\pi\cdot(\theta(X^S)))\theta'$. By Definition 4.2.29 we have $(\pi\cdot(\theta(X^S)))\theta' \equiv ((\pi\cdot X^S)\theta)\theta'$. By Lemma 4.2.31 we have $((\pi\cdot X^S)\theta)\theta' \equiv ((\pi\cdot X^S)\theta)\theta'$. The result follows.
- The case $[a]r$. By Definition 4.2.29 we have $([a]r)(\theta\circ\theta') \equiv [a](r(\theta\circ\theta'))$. By inductive hypothesis $[a](r(\theta\circ\theta')) \equiv [a]((r\theta)\theta')$. By Definition 4.2.29 we have $[a]((r\theta)\theta') \equiv ([a](r\theta))\theta'$. By Definition 4.2.29 we have $([a](r\theta))\theta' \equiv (([a]r)\theta)\theta'$. The result follows.
- The case $\mathsf{f}(r_1,\dots,r_n)$. By Definition 4.2.29 we have $\mathsf{f}(r_1,\dots,r_n)(\theta\circ\theta') \equiv \mathsf{f}(r_1(\theta\circ\theta'),\dots,r_n(\theta\circ\theta'))$. By inductive hypotheses $\mathsf{f}(r_1(\theta\circ\theta'),\dots,r_n(\theta\circ\theta')) \equiv \mathsf{f}((r_1\theta)\theta',\dots,(r_n\theta)\theta')$. By Definition 4.2.29 we have $\mathsf{f}((r_1\theta)\theta',\dots,(r_n\theta)\theta') \equiv \mathsf{f}(r_1\theta,\dots,r_n\theta)\theta'$. By Definition 4.2.29 we have $\mathsf{f}(r_1\theta,\dots,r_n\theta)\theta' \equiv (\mathsf{f}(r_1,\dots,r_n)\theta)\theta'$. The result follows.

⊠

4.3 Unification of terms

4.3.1 Unification problems and their solutions

This Subsection introduces the important notion of a unification problem (Definition 4.3.2), as well as what it means for a substitution to be a solution to said unification problems (Definition 4.3.3).

Definition 4.3.1: Call a pair $r \overset{?}{=} s$ an **equality**.

Definition 4.3.2: Call a finite set of equalities a **unification problem**. Use $\mathcal{P}, \mathcal{P}', \mathcal{P}''$, and so on, to range over unification problems.

Definition 4.3.3: Call a substitution θ a **solution** to a unification problem $\mathcal{P}$ when:

$$r\theta =_\alpha s\theta \quad \text{for every} \quad r \overset{?}{=} s \in \mathcal{P}$$

Definition 4.3.4: Write $\mathit{Sol}(\mathcal{P})$ for the set of solutions of a unification problem $\mathcal{P}$. Call $\mathcal{P}$ **solvable** when $\mathit{Sol}(\mathcal{P})$ is non-empty.

4.3.2 A unification algorithm

This Subsection introduces the permissive nominal unification algorithm proper. As stated, unlike the nominal unification algorithm, the permissive nominal unification algorithms is factored into two separate sub-algorithms, and to reflect this, the Subsection is divided into two. Subsubsection 4.3.2.1 describes the support reduction procedure, a subprocedure that's used by the rest of the algorithm to remove the burden of handling freshness constraints from the user. Subsubsection 4.3.2.2 describes the remainder of the algorithm, and proves important correctness results.

4.3.2.1 Support reduction

This Subsubsection introduces the notion of support reduction. At first glance, support reduction, and its motivation, may seem mysterious. However, the following example may help to remove the mystery from the process.

Suppose $a, b \notin \mathit{comb}$ such that $S = \mathit{comb} \cup \{a\}$ and $T = \mathit{comb} \cup \{b\}$. Suppose also that we wish to unify $\mathit{id}{\cdot}X^S$ and $\mathit{id}{\cdot}Y^T$. We saw in an earlier section that a permissive substitutions must reduce the free atoms of a term that it acts on (Theorem 4.2.30). With this in mind, we see that there is no *direct* substitution that makes $\mathit{id}{\cdot}X^S$ and $\mathit{id}{\cdot}Y^T$ α-equivalent.

However, what we can do is first substitute $\mathit{id}{\cdot}X^S$ and $\mathit{id}{\cdot}Y^T$ for the fresh unknown Z with sort $U = \mathit{comb} \setminus \{a, b\}$. It's easy to check that for both X^S and Y^T a substitution of the form $[X^S{:=}\mathit{id}{\cdot}Z^U]{\circ}[Y^T{:=}\mathit{id}{\cdot}Z^U]$ is support reducing, and solves the problem of unifying $\mathit{id}{\cdot}X^S$ and $\mathit{id}{\cdot}Y^T$. Support reduction is the process of finding support reducing substitutions of this form.

Definition 4.3.6 introduces the notion of a support reduction problem and Definition 4.3.7 introduces the important concept of a solution to support reduction problems. Definition 4.3.8 introduces a support reduction algorithm given by a series of syntax-directed rewrite rules. The rest of the Subsubsection is dedicated to proving that this rewrite relation generates solutions of the correct form (Lemma 4.3.23) and always terminates (Theorem 4.3.13).

Definition 4.3.5: A **support reduction** is a pair $r \sqsubseteq S$ of a term and a permission sort.

Definition 4.3.6: A **support reduction problem** is a finite set of support reductions. Inc, Inc', Inc'', and so on, will range over support reduction problems.

Definition 4.3.7: Call a substitution θ a **solution** to a support reduction problem Inc when:

$$fa(r\theta) \subseteq S \quad \text{for every} \quad r \sqsubseteq S \in Inc$$

Write $\mathit{Sol}(Inc)$ for the set of solutions to a support reduction problem. Call Inc **solvable** whenever $\mathit{Sol}(Inc) \neq \emptyset$.

Definition 4.3.8: Define a **simplification** rewrite relation on support reduction problems by the rules in Figure 4.2.

The rules of Definition 4.3.8 split support reduction problems into either simpler problems, or solve them outright, eliminating them from the problem set (in the cases of ($\sqsubseteq\mathbf{a}$) and ($\sqsubseteq\mathbf{X}$)).

The following technical lemma is used heavily in the proof of Theorem 4.3.10.

$$\begin{array}{llcll}
(\sqsubseteq\mathbf{a}) & a \sqsubseteq S,\ Inc & \Longrightarrow & Inc & (a \in S) \\
(\sqsubseteq\mathsf{f}) & \mathsf{f}(r_1,\ldots,r_n) \sqsubseteq S,\ Inc & \Longrightarrow & r_1 \sqsubseteq S,\ldots,r_n \sqsubseteq S,\ Inc & \\
(\sqsubseteq[]) & [a]r \sqsubseteq S,\ Inc & \Longrightarrow & r \sqsubseteq S \cup \{a\},\ Inc & \\
(\sqsubseteq\mathbf{X}) & \pi\cdot X^S \sqsubseteq T,\ Inc & \Longrightarrow & X^S \sqsubseteq \pi^{-1}\cdot T,\ Inc & (S \not\subseteq \pi^{-1}\cdot T, \pi \neq id) \\
(\sqsubseteq\mathbf{X}') & \pi\cdot X^S \sqsubseteq T,\ Inc & \Longrightarrow & Inc & (S \subseteq \pi^{-1}\cdot T)
\end{array}$$

Figure 4.2 Support inclusion problem simplification

Lemma 4.3.9: We have:

1. If $a \in S$ then $fa(a\theta) \subseteq S$ always.
2. $fa(\mathsf{f}(r_1,\ldots,r_n)\theta) \subseteq S$ if and only if $fa(r_i\theta) \subseteq S$ for all $1 \leq i \leq n$.
3. $fa(([a]r)\theta) \subseteq S$ if and only if $fa(r\theta) \subseteq S \cup \{a\}$.
4. $fa((\pi\cdot X^S)\theta) \subseteq T$ if and only if $fa(X^S\theta) \subseteq \pi^{-1}\cdot T$.
5. If $S \subseteq \pi^{-1}\cdot T$ then $fa(\pi\cdot X^S) \subseteq T$ always.

Proof. We handle the claims individually:

- Claim One. Suppose $a \in S$. By Definition 4.2.29 we have $a\theta \equiv a$. By Definition 4.2.11 we have $fa(a\theta) = \{a\}$. It is a fact that $a \in S$ if and only if $\{a\} \subseteq S$. The result follows.
- Claim Two. We handle the two implications separately:
  - The left-to-right case. Suppose $fa(\mathsf{f}(r_1,\ldots,r_n)\theta) \subseteq S$. We have $\bigcup_{1\leq i\leq n} fa(r_i\theta) \subseteq S$, as:
$$\begin{array}{rcl}
fa(\mathsf{f}(r_1,\ldots,r_n)\theta) & \equiv & fa(\mathsf{f}(r_1\theta,\ldots,r_n\theta)) \\
 & & \text{by Definition 4.2.29} \\
 & \equiv & \bigcup_{1\leq i\leq n} fa(r_i\theta)
\end{array}$$
    Therefore $fa(r_i\theta) \subseteq S$ for $1 \leq i \leq n$. The result follows.
  - The right-to-left case. Suppose $fa(r_i\theta) \subseteq S$ for $1 \leq i \leq n$ therefore $\bigcup_{1\leq i\leq n} fa(r_i\theta) \subseteq S$. By Definition 4.2.11 we have $fa(\mathsf{f}(r_1\theta,\ldots,r_n\theta)) \subseteq S$. By Definition 4.2.29 we have $fa(\mathsf{f}(r_1,\ldots,r_n)\theta) \subseteq S$. The result follows.
- Claim Three. We handle the two implications separately:
  - The left-to-right case. Suppose $fa(([a]r)\theta) \subseteq S$. By Definition 4.2.29 we have $fa([a](r\theta)) \subseteq S$. By Definition 4.2.11 we have $fa(r\theta) \setminus \{a\} \subseteq S$, therefore $fa(r\theta) \subseteq S \cup \{a\}$. The result follows.
  - The right-to-left case. Suppose $fa(r\theta) \subseteq S \cup \{a\}$ therefore $fa(r\theta) \setminus \{a\} \subseteq S$. By Definition 4.2.11 we have $fa([a](r\theta)) \subseteq S$. By Definition 4.2.29 we have $fa(([a]r)\theta) \subseteq S$. The result follows.
- Claim Four. We handle the two implications separately:
  - The left-to-right case. Suppose $fa((\pi\cdot X^S)\theta) \subseteq T$. By Definition 4.2.29 we have $fa(\pi\cdot \theta(X^S)) \subseteq T$. By Lemma 4.2.21 we have $\pi\cdot fa(\theta(X^S)) \subseteq T$. It is a fact that $\subseteq$ is equivariant, so $(\pi^{-1}\circ\pi)\cdot fa(\theta(X^S)) \subseteq \pi^{-1}\cdot T$. It is a fact that $(\pi^{-1}\circ\pi) = id$, therefore $fa(\theta(X^S)) \subseteq \pi^{-1}\cdot T$. By Definition 4.2.29 we have $fa(X^S\theta) \subseteq \pi^{-1}\cdot T$. The result follows.

- The right-to-left case. Suppose $fa(X^S\theta) \subseteq \pi^{-1}\cdot T$. By Definition 4.2.29 we have $fa(\theta(X^S)) \subseteq \pi^{-1}\cdot T$. It is a fact that $\subseteq$ is equivariant, so $\pi\cdot fa(\theta(X^S)) \subseteq (\pi\circ\pi^{-1})\cdot T$. It is a fact that $(\pi\circ\pi^{-1}) = id$, therefore $\pi\cdot fa(\theta(X^S)) \subseteq T$. By Lemma 4.2.21 we have $fa(\pi\cdot\theta(X^S)) \subseteq T$. By Definition 4.2.29 we have $fa((\pi\cdot X^S)\theta) \subseteq T$. The result follows.

- Claim Five. Suppose $S \subseteq \pi^{-1}\cdot T$. It is a fact that $\subseteq$ is equivariant, so $\pi\cdot S \subseteq (\pi\circ\pi^{-1})\cdot T$. It is a fact that $(\pi\circ\pi^{-1}) = id$, therefore $\pi\cdot S \subseteq T$. By Definition 4.2.11 we have $fa(\pi\cdot X^S) \subseteq T$. The result follows.

⊠

Intuitively, Theorem 4.3.10 states that the support reduction relation does not throw away any solutions, as it simplifies a problem.

Theorem 4.3.10: If $Inc \Longrightarrow Inc'$ then $Sol(Inc) = Sol(Inc')$.

Proof. We proceed by case analysis on $Inc \Longrightarrow Inc'$:

- The case ($\sqsubseteq$**a**). Suppose $a \in S$ so that $a \sqsubseteq S,\ Inc \Longrightarrow Inc$ by ($\sqsubseteq$**a**), and suppose $\theta \in Sol(Inc)$. By Claim One of Lemma 4.3.9 we have $\theta \in Sol(a \sqsubseteq S,\ Inc)$. The result follows.
  Otherwise, suppose $\theta \in Sol(a \sqsubseteq S,\ Inc)$. By Definition 4.3.7 we have $\theta \in Sol(Inc)$. The result follows.

- The case ($\sqsubseteq\mathsf{f}$). Suppose $\mathsf{f}(r_1, \ldots, r_n),\ Inc \Longrightarrow r_1 \sqsubseteq S, \ldots, r_n \sqsubseteq S,\ Inc$ by ($\sqsubseteq\mathsf{f}$), and suppose $\theta \in Sol(r_1 \sqsubseteq S, \ldots, r_n \sqsubseteq S,\ Inc)$. By Claim Two of Lemma 4.3.9 we have $\theta \in Sol(\mathsf{f}(r_1, \ldots, r_n) \sqsubseteq S,\ Inc)$. The result follows.
  Otherwise, suppose $\theta \in Sol(\mathsf{f}(r_1, \ldots, r_n) \sqsubseteq S,\ Inc)$. By Claim Two of Lemma 4.3.9 we have $\theta \in Sol(r_1 \sqsubseteq S, \ldots, r_n \sqsubseteq S,\ Inc)$. The result follows.

- The case ($\sqsubseteq$[]). Suppose $[a]r \sqsubseteq S,\ Inc \Longrightarrow r \sqsubseteq S \cup \{a\},\ Inc$ by ($\sqsubseteq$[]), and suppose $\theta \in Sol(r \sqsubseteq S \cup \{a\},\ Inc)$. By Claim Three of Lemma 4.3.9 we have $\theta \in Sol([a]r \sqsubseteq S,\ Inc)$. The result follows.
  Otherwise, suppose $\theta \in Sol([a]r \sqsubseteq S,\ Inc)$. By Claim Three of Lemma 4.3.9 we have $\theta \in Sol(r \sqsubseteq S \cup \{a\},\ Inc)$. The result follows.

- The case ($\sqsubseteq$**X**). Suppose $\pi\cdot X^S \sqsubseteq T,\ Inc \Longrightarrow X^S \sqsubseteq \pi^{-1}\cdot T,\ Inc$ by ($\sqsubseteq$**X**), and suppose $\theta \in Sol(X^S \sqsubseteq \pi^{-1}\cdot T,\ Inc)$. By Claim Four of Lemma 4.3.9 we have $\theta \in Sol(\pi\cdot X^S \sqsubseteq T,\ Inc)$. The result follows.
  Otherwise, suppose $\theta \in Sol(\pi\cdot X^S \sqsubseteq T,\ Inc)$. By Claim Four of Lemma 4.3.9 we have $\theta \in Sol(X^S \sqsubseteq \pi^{-1}\cdot T,\ Inc)$. The result follows.

- The case ($\sqsubseteq$**X**′). Suppose $S \subseteq \pi^{-1}\cdot T$ so that $\pi\cdot X^S \sqsubseteq T,\ Inc \Longrightarrow Inc$ by ($\sqsubseteq$**X**′), and suppose $\theta \in Sol(Inc)$. By Claim Five of Lemma 4.3.9 we have $\theta \in Sol(\pi\cdot X^S \sqsubseteq T,\ Inc)$. The result follows.
  Otherwise, suppose $\theta \in Sol(\pi\cdot X^S \sqsubseteq T,\ Inc)$. By Definition 4.3.7 we have $\theta \in Sol(Inc)$. The result follows.

⊠

Definition 4.3.11: The **size** of a support inclusion problem $size(Inc)$ is a tuple (T, A, P, S) where:

- T is the number of term-formers appearing within terms in Inc,
- A is the number of abstractions appearing within terms in Inc,
- P is the number of suspended permutations, distinct from the identity permutation, appearing within terms in Inc,
- S is the number of support inclusions appearing within Inc.

Order tuples lexicographically.

Example 4.3.12: If $Inc = [a]b \sqsubseteq S, \mathsf{f}(\pi{\cdot}X^S, a) \sqsubseteq T$ then $size(Inc) = (1, 1, 1, 2)$. Similarly, if $Inc' = \mathsf{f}(\pi{\cdot}X^S, a)$ then $size(Inc') = (0, 0, 1, 1)$. By the lexicographic ordering we have $size(Inc') < size(Inc)$.

Theorem 4.3.13: The support inclusion simplification relation is strongly normalizing (that is, every reduction path eventually terminates).

Proof. By case analysis on $r \sqsubseteq S$, Inc showing that all rules reduce the size of a support inclusion problem. See Appendix B. ⊠

Definition 4.3.14: For every support inclusion problem Inc pick an arbitrary **normal form** $nf(Inc)$ guaranteed to exist by Theorem 4.3.13.

Definition 4.3.15: Call Inc **non-trivial** whenever $nf(Inc) \neq \emptyset$.

Definition 4.3.16: Call Inc **consistent** whenever $a \sqsubseteq S \notin nf(Inc)$ for all a and S.

Lemma 4.3.17: If $nf(Inc)$ is consistent then all $inc \in nf(Inc)$ have the form $Y^T \sqsubseteq S$ where $S \nsubseteq T$.

Proof. By inspection of the rules in Definition 4.3.8. ⊠

Definition 4.3.18: Define $fV(Inc)$ by $fV(Inc) = \bigcup\{fV(r) \mid \exists S.r \sqsubseteq S\}$.

Intuitively, $fV(Inc)$ is the set of unknowns appearing in Inc.

Definition 4.3.19: Let $\mathcal{V}$ range over sets of unknowns.

Definition 4.3.20: Suppose Inc is consistent. For every $X^S \in \mathcal{V}$ make a fixed but arbitrary choice of $X'^{S'}$ such that $X'^{S'} \notin \mathcal{V}$ and $S' = S \cap \bigcap\{T \mid X^S \sqsubseteq T \in nf(Inc)\}$.

We make our choice injectively; that is, for distinct X^S and $Y^T \in \mathcal{V}$ it is always the case that $X'^{S'}$ and $Y'^{T'}$ are distinct. It will be convenient to write $\bar{\mathcal{V}}_{Inc}$ for our choices $\{X'^{S'} \mid X^S \in Inc\}$.

Definition 4.3.21: Define a substitution $\rho^{\mathcal{V}}_{Inc}$ by:

$$\rho^{\mathcal{V}}_{Inc}(X^S) = id{\cdot}X'^{S'} \text{ if } X^S \in \mathcal{V} \text{ and } \rho^{\mathcal{V}}_{Inc}(X^S) = id{\cdot}X^S \text{ otherwise}$$

Lemma 4.3.22 checks that $\rho^{\mathcal{V}}_{Inc}$ is a substitution by our definition (Definition 4.3.7).

Lemma 4.3.22: $fa(\rho^{\mathcal{V}}_{Inc}(X^S)) \subseteq S$

Proof. Suppose $X^S \in \mathcal{V}$. By Definition 4.3.21 we have $\rho^{\mathcal{V}}_{Inc}(X^S) = id{\cdot}X'^{S'}$. By Definition 4.3.20 we have $S' \subseteq S$. By Definition 4.2.11 we have $fa(id{\cdot}X'^{S'}) = S'$. The result follows.

Otherwise we have $X^S \notin \mathcal{V}$. By Definition 4.3.21 we have $\rho^{\mathcal{V}}_{Inc}(X^S) = id{\cdot}X^S$. The result follows. ⊠

Lemma 4.3.23: If Inc is consistent then $\rho^{\mathcal{V}}_{Inc} \in Sol(Inc)$.

Proof. Suppose Inc is in normal-form. For every $X^S \sqsubseteq T \in Inc$ we have $\rho^{\mathcal{V}}_{Inc}(X^S) = id{\cdot}X'^{S'}$ for an S' that satisfies $S' \subseteq T$. The result follows.

Otherwise, if Inc is not a normal-form. By Theorem 4.3.10 we have $Sol(Inc) = Sol(nf(Inc))$, and we use the previous paragraph. ⊠

Theorem 4.3.24 states that the notions of consistency (Definition 4.3.16) and solvability (Definition 4.3.4) co-incide.

Theorem 4.3.24: *Inc is consistent if and only if Inc is solvable.*

Proof. By Theorem 4.3.10 we have $Sol(Inc) = Sol(nf(Inc))$. It therefore suffices to show the claim holds for the case where Inc is a normal-form.

Suppose Inc is inconsistent. By Definition 4.3.16 we have some a exists such that $a \sqsubseteq S \in Inc$ where $a \notin S$. By Definition 4.2.29 we have $a\theta \equiv a$ always. It is a fact that no substitution exists so that $\{a\theta\} \subseteq S$, and therefore Inc is not solvable. The result follows.

Suppose Inc is consistent. By Lemma 4.3.23 we have Inc is solvable. The result follows. ⊠

Definition 4.3.25: Suppose that Inc is consistent, $fV(Inc) \subseteq \mathcal{V}$ and $\theta \in Sol(Inc)$. Define a substitution $(\theta{-}\rho^{\mathcal{V}}_{Inc})$ by:

- $(\theta{-}\rho^{\mathcal{V}}_{Inc})(X'^{S'}) = \theta(X^S)$ if $X^S \in \mathcal{V}$ and $\rho^{\mathcal{V}}_{Inc}(X^S) = id{\cdot}X'^{S'}$,
- $(\theta{-}\rho^{\mathcal{V}}_{Inc})(X^S) = \theta(X^S)$ if $X^S \notin \mathcal{V}$.

It remains to check whether Definition 4.3.25 is well-defined:

Lemma 4.3.26: If $(\theta{-}\rho^{\mathcal{V}}_{Inc})$ exists then it is well-defined.

Proof. Suppose $(\theta{-}\rho^{\mathcal{V}}_{Inc})$ exists. There are four cases:

- The case $X^S \neq Y^T$, $X^S \notin \mathcal{V}$ and $Y^T \notin \mathcal{V}$. By Definition 4.3.25 we have $(\theta{-}\rho^{\mathcal{V}}_{Inc})(X^S) = \theta(X^S)$. By Definition 4.3.25 we have $(\theta{-}\rho^{\mathcal{V}}_{Inc})(Y^T) = \theta(Y^T)$. It is a fact that substitutions are well-defined. The result follows.
- The case $X'^{S'} \neq Y'^{T'}$, $\rho^{\mathcal{V}}_{Inc}(X^S) = id{\cdot}X'^{S'}$, $\rho^{\mathcal{V}}_{Inc}(Y^T) = id{\cdot}Y'^{T'}$, $X^S \notin \mathcal{V}$ and $Y^T \notin \mathcal{V}$. By Definition 4.3.25 we have $(\theta{-}\rho^{\mathcal{V}}_{Inc})(X'^{S'}) = \theta(X^S)$. By Definition 4.3.25 we have $(\theta{-}\rho^{\mathcal{V}}_{Inc})(Y'^{T'}) = \theta(Y^T)$. It is a fact that substitutions are well-defined. The result follows.

- The case $X'^{S'} \neq Y^T$, $\rho^{\mathcal{V}}_{Inc}(X^S) = id{\cdot}X'^{S'}$, $X^S \notin \mathcal{V}$ and $Y^T \in \mathcal{V}$. By Definition 4.3.25 we have $(\theta{-}\rho^{\mathcal{V}}_{Inc})(Y^T) = \theta(Y^T)$. By Definition 4.3.25 we have $(\theta{-}\rho^{\mathcal{V}}_{Inc})(X'^{S'}) = \theta(X^S)$. By Definition 4.3.21 we have $\rho^{\mathcal{V}}_{Inc}(X^S) \neq id{\cdot}Y^T$ as $Y^T \in \mathcal{V}$. It is a fact that substitutions are well-defined. The result follows.
- The case $X^S \neq Y'^{T'}$, $\rho^{\mathcal{V}}_{Inc}(Y^T) = id{\cdot}Y'^{T'}$, $Y^T \notin \mathcal{V}$ and $X^S \in \mathcal{V}$. This is similar to the previous case.

⊠

Lemma 4.3.27 and Lemma 4.3.28 are basic checks of correctness for the definition of $\rho^{\mathcal{V}}_{Inc}$.

Lemma 4.3.27: If $\theta \in Sol(Inc)$ then $\rho^{\mathcal{V}}_{Inc}$ exists.

Proof. By Definition 4.3.7 we have Inc is solvable. By Theorem 4.3.24 we have Inc is consistent. By Definition 4.3.21 we have $\rho^{\mathcal{V}}_{Inc}$ exists. The result follows. ⊠

Lemma 4.3.28: If $\rho^{\mathcal{V}}_{Inc}$ exists then it is well-defined.

Proof. Suppose $\rho^{\mathcal{V}}_{Inc}$ exists and that $X^S \neq Y^T$. Then:

- The case $X^S \in \mathcal{V}$ and $Y^T \in \mathcal{V}$. By Definition 4.3.21 we have $\rho^{\mathcal{V}}_{Inc}(X^S) = id{\cdot}X'^{S'}$ and by Definition 4.3.21 we have $\rho^{\mathcal{V}}_{Inc}(Y^T) = id{\cdot}Y'^{T'}$ where $X'^{S'}$ and $Y'^{T'}$ are chosen so that $X'^{S'} \neq Y'^{T'}$. The result follows.
- The case $X^S \notin \mathcal{V}$ and $Y^T \notin \mathcal{V}$. By Definition 4.3.21 we have $\rho^{\mathcal{V}}_{Inc}(X^S) = id{\cdot}X^S$ and $\rho^{\mathcal{V}}_{Inc}(Y^T) = id{\cdot}Y^T$. By assumption we have $X^S \neq Y^T$. The result follows.
- The case $X^S \in \mathcal{V}$ and $Y^T \notin \mathcal{V}$. By Definition 4.3.21 we have $\rho^{\mathcal{V}}_{Inc}(X^S) = id{\cdot}X'^{S'}$ and $\rho^{\mathcal{V}}_{Inc}(Y^T) = id{\cdot}Y^T$. By Definition 4.3.20 we have $X'^{S'} \notin \mathcal{V}$. The result follows.
- The case $X^S \notin \mathcal{V}$ and $Y^T \in \mathcal{V}$. This is similar to the previous case.

⊠

Lemma 4.3.29 checks that the support reduction simplification rewrite relation does not introduce extraneous unknowns into a simplified problem.

Lemma 4.3.29: If $Inc \Longrightarrow Inc'$ then $fV(Inc') \subseteq fV(Inc)$.

Proof. By case analysis of the simplification rules in Definition 4.3.8. See Appendix B. ⊠

Corollary 4.3.30: $fV(nf(Inc)) \subseteq fV(Inc)$

Proof. Immediate, from Lemma 4.3.29 and Definition 4.3.14. ⊠

Lemma 4.3.31: If $\theta \in Sol(Inc)$ and $fV(Inc) \subseteq \mathcal{V}$ then $(\theta{-}\rho^{\mathcal{V}}_{Inc})$ is a substitution.

Proof. By Lemma 4.3.27 we have that $\rho^{\mathcal{V}}_{Inc}$ exists. We show for all X^S that $fa((\theta{-}\rho^{\mathcal{V}}_{Inc})(X^S)) \subseteq S$ by cases:

- The case $id{\cdot}X'^{S'} \equiv \rho^{\mathcal{V}}_{Inc}(X^S)$ for some $X^S \in \mathcal{V}$. By Corollary 4.3.30 we have $fV(nf(Inc)) \subseteq fV(Inc)$. It is a fact that $fV(nf(Inc)) \subseteq \mathcal{V}$ as $fV(Inc) \subseteq \mathcal{V}$ by assumption. There are two cases:
  - The case $X^S \notin fV(nf(Inc))$. Then $S = S'$ and by Definition 4.3.25 we have $(\theta - \rho^{\mathcal{V}}_{Inc})(X'^{S'}) \equiv \theta(X^S)$. By Definition 4.2.28 we have $fa(\theta(X^S)) \subseteq S$. The result follows.
  - The case $X^S \in fV(nf(Inc))$. By assumption $\theta \in Sol(Inc)$. By Theorem 4.3.10 we have $\theta \in Sol(nf(Inc))$. By Definition 4.3.7 we have $fa(\theta(X^S)) \subseteq T$ for every $X^S \sqsubseteq T \in nf(Inc)$. By Definition 4.3.20 we have $S' = S \cap \bigcap\{T \mid \exists r.r \sqsubseteq T \in nf(Inc)\}$. It is a fact that $S' \subseteq S$. The result follows.
- Otherwise, $(\theta - \rho^{\mathcal{V}}_{Inc})(X^S) \equiv \theta(X^S)$, and by Definition 4.2.28 we have $fa(\theta(X^S)) \subseteq S$.

⊠

Intuitively, Theorem 4.3.32 claims that any substitution that is a solution to a support reduction problem can be expressed in terms of $\rho^{\mathcal{V}}_{Inc}$ and another, simpler substitution.

Theorem 4.3.32: If $\theta \in Sol(Inc)$ and $fV(Inc) \subseteq \mathcal{V}$ then $\theta(X^S) \equiv (\rho^{\mathcal{V}}_{Inc} \circ (\theta - \rho^{\mathcal{V}}_{Inc}))(X^S)$ for every $X^S \in \mathcal{V}$.

Proof. By Definition 4.2.35 we have $(\rho^{\mathcal{V}}_{Inc} \circ (\theta - \rho^{\mathcal{V}}_{Inc}))(X^S) \equiv (\rho^{\mathcal{V}}_{Inc}(X^S))(\theta - \rho^{\mathcal{V}}_{Inc})$. By Definition 4.3.21 and the fact that $X'^{S'} \notin \mathcal{V}$ we have $(\rho^{\mathcal{V}}_{Inc}(X^S))(\theta - \rho^{\mathcal{V}}_{Inc}) \equiv (id{\cdot}X'^{S'})(\theta - \rho^{\mathcal{V}}_{Inc})$. By Definition 4.2.29 we have $(id{\cdot}X'^{S'})(\theta - \rho^{\mathcal{V}}_{Inc}) \equiv id{\cdot}((\theta - \rho^{\mathcal{V}}_{Inc})(X'^{S'}))$. By Definition 4.3.25 we have $id{\cdot}((\theta - \rho^{\mathcal{V}}_{Inc})(X'^{S'})) \equiv id{\cdot}(\theta(X^S))$. By Lemma 4.2.19 we have $id{\cdot}(\theta(X^S)) \equiv \theta(X^S)$. The result follows. ⊠

4.3.2.2 Unification problem simplification

This Subsubsection introduces the unification algorithm proper, which calls the support reduction machinery of Subsubsection 4.3.2.1 as a subprocedure (see (**I3**) in Figure 4.3). The unification algorithm is given in the common simplification relation style (Definition 4.3.35).

The main result in this Subsubsection are Theorem 4.3.42, demonstrating that the simplification rules of Definition 4.3.35 always terminate.

Lemma 4.3.33: $(\theta \circ \theta') \in Sol(\mathcal{P})$ if and only if $\theta' \in Sol(\mathcal{P}\theta)$.

Proof. Suppose $\theta' \in Sol(\mathcal{P}\theta)$ and $r\theta \overset{?}{=} s\theta \in \mathcal{P}\theta$. By Theorem 4.2.36 we have $(r\theta)\theta' =_\alpha r(\theta \circ \theta')$. By assumption $r(\theta \circ \theta') \equiv s(\theta \circ \theta')$. By Theorem 4.2.36 we have $s(\theta \circ \theta') \equiv (s\theta)\theta'$. The result follows.

Otherwise, suppose $\theta \circ \theta' \in Sol(\mathcal{P})$. By Theorem 4.2.36 we have $r(\theta \circ \theta') \equiv (r\theta)\theta'$. By assumption $(r\theta)\theta' =_\alpha (s\theta)\theta'$. By Theorem 4.2.36 we have $(s\theta)\theta' \equiv s(\theta \circ \theta')$. The result follows.

⊠

Definition 4.3.34: If $\mathcal{P}$ is a unification problem, define its **related support inclusion problem** $\mathcal{P}_{\sqsubseteq}$ by:

$$\mathcal{P}_{\sqsubseteq} = \{r \sqsubseteq fa(s),\ s \sqsubseteq fa(r) \mid r \overset{?}{=} s \in \mathcal{P}\}$$

$$\begin{array}{lllll}
(\overset{?}{=}\mathbf{a}) & \mathcal{V}; a \overset{?}{=} a, \mathcal{P} & \Longrightarrow & \mathcal{V}; \mathcal{P} & \\
(\overset{?}{=}\mathsf{f}) & \mathcal{V}; \mathsf{f}(r_1, \ldots, r_n) \overset{?}{=} \mathsf{f}(s_1, \ldots, s_n), \mathcal{P} & \Longrightarrow & \mathcal{V}; r_i \overset{?}{=} s_i, \mathcal{P} & (1 \leq i \leq n) \\
(\overset{?}{=}[\mathbf{a}]) & \mathcal{V}; [a]r \overset{?}{=} [a]s, \mathcal{P} & \Longrightarrow & \mathcal{V}; r \overset{?}{=} s, \mathcal{P} & \\
(\overset{?}{=}[\mathbf{b}]) & \mathcal{V}; [a]r \overset{?}{=} [b]s, \mathcal{P} & \Longrightarrow & \mathcal{V}; (b\ a)\cdot r \overset{?}{=} s, \mathcal{P} & (b \notin fa(r)) \\
(\overset{?}{=}\mathbf{X}) & \mathcal{V}; \pi\cdot X^S \overset{?}{=} \pi\cdot X^S, \mathcal{P} & \Longrightarrow & \mathcal{V}; \mathcal{P} & \\
(\mathbf{I1}) & \mathcal{V}; \pi\cdot X^S \overset{?}{=} s, \mathcal{P} & \overset{[X^S:=\pi^{\text{-}1}\cdot s]}{\Longrightarrow} & \mathcal{V}; \mathcal{P}[X^S:=\pi^{\text{-}1}\cdot s] & \\
 & & & \qquad (X^S \notin fV(s), fa(s) \subseteq S) & \\
(\mathbf{I2}) & \mathcal{V}; r \overset{?}{=} \pi\cdot X^S, \mathcal{P} & \overset{[X^S:=\pi^{\text{-}1}\cdot r]}{\Longrightarrow} & \mathcal{V}; \mathcal{P}[X^S:=\pi^{\text{-}1}\cdot r] & \\
 & & & \qquad (X^S \notin fV(r), fa(r) \subseteq S) & \\
(\mathbf{I3}) & \mathcal{V}; \mathcal{P} & \overset{\rho^{\mathcal{V}}_{\mathcal{P}_{\sqsubseteq}}}{\Longrightarrow} & \mathcal{V} \cup \mathcal{V}'^{\mathcal{V}}_{\mathcal{P}_{\sqsubseteq}}; \mathcal{P}\rho^{\mathcal{V}}_{\mathcal{P}_{\sqsubseteq}} & \\
 & & & (\mathcal{P}_{\sqsubseteq} \text{ consistent and non-trivial}) &
\end{array}$$

Figure 4.3 Unification problem simplification rules

Definition 4.3.35: Define a **simplification** rewrite relation on unification problems by the rules in Figure 4.3.

Call $(\overset{?}{=}\mathbf{a})$, $(\overset{?}{=}\mathsf{f})$, $(\overset{?}{=}[\mathbf{a}])$, $(\overset{?}{=}[\mathbf{b}])$ and $(\overset{?}{=}\mathbf{X})$ **non-instantiating rules**. Call (**I1**), (**I2**) and (**I3**) **instantiating rules**. Write $\Longrightarrow^*$ for the reflexive transitive closure of $\Longrightarrow$.

The side condition $X^S \notin fV(s)$ on (**I1**) (and the similar side-condition on (**I2**)) may be thought of as akin to the familiar 'occurs-check' from first-order unification algorithms. Most rules in Definition 4.3.35 are straightforward; problems are either split into simpler problems, or solved outright. However, (**I3**) is potentially more difficult to understand. Here, $\rho^{\mathcal{V}}_{\mathcal{P}_{\sqsubseteq}}$ is the support reducing substitution ρ constructed from $\mathcal{P}_{\sqsubseteq}$. We will see later, when defining a unification algorithm, that (**I3**) is only applied when all other rules have been exhausted.

Lemma 4.3.36: If $\mathcal{V}; \mathcal{P} \Longrightarrow \mathcal{V}'; \mathcal{P}'$ by a non-instantiating rule then $Sol(\mathcal{P}) = Sol(\mathcal{P}')$.

Proof. By case analysis on the rules defined in Definition 4.3.35, using Lemma 4.2.31 and Theorems 4.2.30 and 4.2.27. See Appendix B. ⊠

Definition 4.3.37: Define $fV(\mathcal{P})$ by:

$$fV(\mathcal{P}) = \bigcup\{fV(r) \cup fV(s) \mid r \overset{?}{=} s \in \mathcal{P}\}$$

Definition 4.3.38: Suppose $\mathcal{V}$ is a set of unknowns. Define the **restriction** of a substitution θ to $\mathcal{V}$ by:

$$\theta|_{\mathcal{V}}(X^S) = \theta(X^S) \quad \text{if } X^S \in \mathcal{V} \text{ and} \quad \theta|_{\mathcal{V}}(X^S) = id\cdot X^S \quad \text{otherwise}$$

Definition 4.3.39: If $\mathcal{P}$ is a unification problem, define a **unification algorithm** by:

1. Rewrite $fV(\mathcal{P}); \mathcal{P}$ using the rules of Definition 4.3.35 as much as possible, with top-down precedence.

2. If we obtain $\mathcal{V}'; \emptyset$ then we succeed and return $\theta|_{\mathcal{V}'}$ where θ is the functional composition of all the substitution labeling rewrites. If there are no substitutions labeling rewrites, we take $\theta = id$. If we do not obtain $\mathcal{V}'; \emptyset$ then we fail.

Definition 4.3.40: Define the **size** of a unification problem $size(\mathcal{P})$ to be a tuple (T, E, A), where:

- T is the number of term-formers appearing within terms in the equalities of the unification problem,
- E is the number of equalities appearing in the unification problem,
- A is the number of abstractions appearing within terms in the equalities of the unification problem.

Order tuples lexicographically.

Example 4.3.41: If $\mathcal{P} = \mathsf{f}(a,b) \overset{?}{=} \mathsf{f}(a,b), [a]a \overset{?}{=} [b]b$ then $size(\mathcal{P}) = (2,2,2)$. Similarly, if $\mathcal{P}' = \mathsf{f}(\pi{\cdot}X^S, a) \overset{?}{=} \mathsf{f}([a]a, a)$ then $size(\mathcal{P}') = (2,1,1)$. By the lexicographic ordering we have $size(\mathcal{P}') < size(\mathcal{P})$.

Theorem 4.3.42: The unification algorithm of Definition 4.3.39 always terminates.

Proof. By case analysis on the rules in Definition 4.3.35, checking that all rules reduce the size of the unification problem. See Appendix B. ⊠

Lemma 4.3.43 states that solutions to unification problems are not perturbed by α-equivalence.

Lemma 4.3.43: Suppose $\theta(X^S) =_\alpha \theta'(X^S)$ for all $X^S \in fV(\mathcal{P})$. Then $\theta \in Sol(\mathcal{P})$ if and only if $\theta' \in Sol(\mathcal{P})$.

Proof. By Definition 4.3.3 it suffices to show $r\theta =_\alpha s\theta$ if and only if $r\theta' \overset{?}{=} s\theta'$ for all $r \overset{?}{=} s \in \mathcal{P}$.

Suppose $\theta \in Sol(\mathcal{P})$. By Definition 4.3.3 we have $r\theta =_\alpha s\theta$. By assumption $\theta(X^S) =_\alpha \theta'(X^S)$ for all $X^S \in fV(\mathcal{P})$. By Definition 4.3.37 we have $fV(r) \subseteq fV(\mathcal{P})$ and $fV(s) \subseteq fV(\mathcal{P})$. By Theorem 4.2.33 we have $r\theta =_\alpha r\theta'$ and $s\theta =_\alpha s\theta'$. By Theorem 4.2.27 we have $r\theta' =_\alpha s\theta'$. By Definition 4.3.3 we have $\theta' \in Sol(\mathcal{P})$. The result follows.

Otherwise, suppose $\theta' \in Sol(\mathcal{P})$. By Definition 4.3.3 we have $r\theta' =_\alpha s\theta'$. By assumption $\theta'(X^S) =_\alpha \theta(X^S)$ for all $X^S \in fV(\mathcal{P})$. By Definition 4.3.37 we have $fV(r) \subseteq fV(\mathcal{P})$ and $fV(s) \subseteq fV(\mathcal{P})$. By Theorem 4.2.33 we have $r\theta' =_\alpha r\theta$ and $s\theta' =_\alpha s\theta$. By Theorem 4.2.27 we have $r\theta =_\alpha s\theta$. By Definition 4.3.3 we have $\theta \in Sol(\mathcal{P})$. The result follows. ⊠

Definition 4.3.44: Suppose θ is a substitution. Define $(\theta{-}X^S)$ by:

$$(\theta{-}X^S)(X^S) = id{\cdot}X^S \quad \text{and} \quad (\theta{-}X^S)(Y^T) = \theta(Y^T) \quad \text{for all other } Y^T$$

Theorem 4.3.45 is a similar result to Theorem 4.3.32. Intuitively, the result states that any substitution making X^S and s α-equivalent can be refactored into a substitution which is *only* non-trivial on X^S and another substitution which is trivial on X^S.

Theorem 4.3.45: Suppose $X^S\theta =_\alpha s\theta$ and $X^S \notin fV(s)$. Then:

$$X^S\theta =_\alpha X^S([X^S{:=}s]\circ(\theta{-}X^S)) \quad \text{and} \quad Y^T\theta =_\alpha Y^T([X^S{:=}s]\circ(\theta{-}X^S))$$

Proof. We handle the two claims separately:

- The first claim. By Definition 4.2.29 we have $X^S([X^S{:=}s]\circ(\theta{-}X^S)) \equiv s(\theta{-}X^S)$. By Theorem 4.2.33 and as $X^S \notin fV(s)$ we have $s(\theta{-}X^S) \equiv s\theta$. By assumption $s\theta =_\alpha X^S\theta$. The result follows.
- The second claim. By Definition 4.2.29 we have $Y^T([X^S{:=}s]\circ(\theta{-}X^S)) \equiv Y^T(\theta{-}X^S)$. By Definition 4.3.44 we have $Y^T(\theta{-}X^S) \equiv Y^T\theta$. The result follows.

⊠

4.3.3 Principal solutions

This Subsection shows that the unification algorithm outlined in Subsection 4.3.2 generates principal (most general) solutions.

Definition 4.3.46 defines the important concept of the instantiation ordering, the measure by which we compare solutions to unification problems for principality. Theorem 4.3.51 shows that the unification algorithm generates solutions to unification problems. Theorem 4.3.56 goes further, and demonstrates that the solutions generated are principal. Finally, Theorem 4.3.58 is a basic check of correctness for the unification algorithm, demonstrating that, for a given unification problem, the algorithm will either produced a correct solution, or enter a failing state in a finite number of steps.

Definition 4.3.46: Define the **instantiation ordering**, by:

$$\theta \leq \theta' \text{ if there exists } \theta'' \text{ such that } \theta'(X^S) =_\alpha (\theta\circ\theta'')(X^S)$$

Example 4.3.47: We have $[X^S{:=}s] \leq [X^S{:=}id{\cdot}X^S]\circ[X^S{:=}s]$.

Lemma 4.3.48 and Lemma 4.3.49 demonstrate that the simplification rules defined in Definition 4.3.35 do not introduce any extraneous variables when simplifying a problem.

Lemma 4.3.48: If $fV(\mathcal{P}) \subseteq \mathcal{V}$ and $\mathcal{V};\mathcal{P} \Longrightarrow \mathcal{V}';\mathcal{P}'$ using a non-instantiating rule then $fV(\mathcal{P}') \subseteq \mathcal{V}'$.

Proof. By case analysis on the non-instantiating rules in Definition 4.3.35, using Lemma 4.2.22. See Appendix B. ⊠

Lemma 4.3.49: If $fV(\mathcal{P}) \subseteq \mathcal{V}$ and $\mathcal{V};\mathcal{P} \overset{\theta}{\Longrightarrow} \mathcal{V}';\mathcal{P}'\theta$ using an instantiating rule then $fV(\mathcal{P}'\theta) \subseteq \mathcal{V}$.

Proof. By case analysis on the instantiating rules in Definition 4.3.35, using Lemmas 4.2.32 and 4.3.29. See Appendix B. ⊠

Lemma 4.3.50: If $X^S \in \mathcal{V}$ then $([X^S{:=}s]\circ\theta)|_{\mathcal{V}} = [X^S{:=}s]\circ\theta|_{\mathcal{V}}$.

Proof. There are three cases:

- The case X^S with $X^S \in \mathcal{V}$. By Definition 4.3.38 we have $([X^S{:=}s]\circ\theta)|_{\mathcal{V}}(X^S) = ([X^S{:=}s]\circ\theta)(X^S)$. By Definition 4.2.35 we have $([X^S{:=}s]\circ\theta)(X^S) = ([X^S{:=}s](X^S))\theta$. By Definition 4.2.28 we have $([X^S{:=}s](X^S))\theta = s\theta$. By Definition 4.2.28 we have $s\theta = ([X^S{:=}s](X^S))\theta$. By Definition 4.3.38 we have $([X^S{:=}s](X^S))\theta = ([X^S{:=}s](X^S))\theta|_{\mathcal{V}}$. By Definition 4.2.35 we have $([X^S{:=}s](X^S))\theta|_{\mathcal{V}} = ([X^S{:=}s]\circ\theta|_{\mathcal{V}})(X^S)$. The result follows.
- The case Y^T with $Y^T \in \mathcal{V}$. By Definition 4.3.38 we have $([X^S{:=}s]\circ\theta)|_{\mathcal{V}}(Y^T) = ([X^S{:=}s]\circ\theta)(Y^T)$. By Definition 4.2.35 we have $([X^S{:=}s]\circ\theta)(Y^T) = ([X^S{:=}s](Y^T))\theta$. By Definition 4.2.28 we have $([X^S{:=}s](Y^T))\theta = Y^T\theta$. By Definition 4.3.38 we have $Y^T\theta = Y^T\theta|_{\mathcal{V}}$. By Definition 4.2.28 we have $Y^T\theta|_{\mathcal{V}} = ([X^S{:=}s](Y^T))\theta|_{\mathcal{V}}$. By Definition 4.2.35 we have $([X^S{:=}s](Y^T))\theta|_{\mathcal{V}} = ([X^S{:=}s]\circ\theta|_{\mathcal{V}})(Y^T)$. The result follows.
- The case Y^T with $Y^T \notin \mathcal{V}$. By Definition 4.3.38 we have $([X^S{:=}s]\circ\theta)|_{\mathcal{V}}(Y^T) = id\cdot Y^T$. By Definition 4.2.28 we have $id\cdot Y^T = (id\cdot([X^S{:=}s](Y^T)))$. By Definition 4.2.29 we have $(id\cdot([X^S{:=}s](Y^T))) = ((id\cdot Y^T)[X^S{:=}s])$. By Definition 4.3.38 we have $((id\cdot Y^T)[X^S{:=}s]) = ((id\cdot Y^T)[X^S{:=}s])\theta|_{\mathcal{V}}$. By Definition 4.2.29 we have $((id\cdot Y^T)[X^S{:=}s])\theta|_{\mathcal{V}} = id\cdot(([X^S{:=}s]Y^T)\theta|_{\mathcal{V}})$. By Definition 4.2.35 we have $id\cdot(([X^S{:=}s]Y^T)\theta|_{\mathcal{V}}) = id\cdot(([X^S{:=}s]\circ\theta|_{\mathcal{V}})(Y^T))$. By Lemma 4.2.19 we have $id\cdot(([X^S{:=}s]\circ\theta|_{\mathcal{V}})(Y^T)) = ([X^S{:=}s]\circ\theta|_{\mathcal{V}})(Y^T)$. The result follows.

⊠

Theorem 4.3.51 states that generated substitutions solve unification problems.

Theorem 4.3.51: If $fV(\mathcal{P}) \subseteq \mathcal{V}$ and $\mathcal{V};\mathcal{P} \overset{\theta}{\Longrightarrow}{}^* \mathcal{V}';\emptyset$ then $\theta|_{\mathcal{V}} \in Sol(\mathcal{P})$.

Proof. By induction on the path length of $\Longrightarrow^*$.

- Length 0. Then $\mathcal{P} = \emptyset$ and $\theta = id$. The result follows.
- Length $k{+}1$. There are four cases.
  - The non-instantiating case. Suppose:

$$\mathcal{V};\mathcal{P} \Longrightarrow \mathcal{V};\mathcal{P}' \overset{\theta}{\Longrightarrow}{}^* \mathcal{V}';\emptyset$$

By Lemma 4.3.48 we have $fV(\mathcal{P}') \subseteq \mathcal{V}$. By inductive hypothesis $\theta|_{\mathcal{V}} \in Sol(\mathcal{P}')$. By Lemma 4.3.36 we have $\theta|_{\mathcal{V}} \in Sol(\mathcal{P})$. The result follows.

  - The case (**I1**). Suppose:

$$\mathcal{V};\mathcal{P} \overset{[X^S:=\pi^{-1}\cdot s]}{\Longrightarrow} \mathcal{V};\mathcal{P}[X^S{:=}\pi^{-1}\cdot s] \overset{\theta'}{\Longrightarrow}{}^* \mathcal{V}';\emptyset$$

By Lemma 4.3.49 we have $fV(\mathcal{P}[X^S{:=}\pi^{-1}\cdot s]) \subseteq \mathcal{V}$. By inductive hypothesis $\theta|_{\mathcal{V}} \in Sol(\mathcal{P}[X^S{:=}\pi^{-1}\cdot s])$. By Lemma 4.3.33 we have $[X^S{:=}\pi^{-1}\cdot s]\circ\theta|_{\mathcal{V}} \in Sol(\mathcal{P})$. By Lemma 4.3.50 we have $[X^S{:=}\pi^{-1}\cdot s]\circ\theta|_{\mathcal{V}} = ([X^S{:=}\pi^{-1}\cdot s]\circ\theta)|_{\mathcal{V}}$. The result follows.

- The case (**I2**). Suppose:

$$\mathcal{V};\mathcal{P} \overset{[X^S:=\pi^{-1}\cdot r]}{\Longrightarrow} \mathcal{V};\mathcal{P}[X^S:=\pi^{-1}\cdot r] \overset{\theta}{\Longrightarrow}^{*} \mathcal{V}';\emptyset$$

  By Lemma 4.3.49 we have $fV(\mathcal{P}[X^S:=\pi^{-1}\cdot r]) \subseteq \mathcal{V}$. By inductive hypothesis $\theta|_{\mathcal{V}} \in Sol(\mathcal{P}[X^S:=\pi^{-1}\cdot r])$. By Lemma 4.3.33 we have $[X^S:=\pi^{-1}\cdot r]\circ\theta|_{\mathcal{V}} \in Sol(\mathcal{P})$. By Lemma 4.3.50 we have $[X^S:=\pi^{-1}\cdot r]\circ\theta|_{\mathcal{V}} = ([X^S:=\pi^{-1}\cdot r]\circ\theta)|_{\mathcal{V}}$. The result follows.

- The case (**I3**). Suppose:

$$\mathcal{V};\mathcal{P} \overset{\rho^{\mathcal{V}}_{\mathcal{P}_{\sqsubseteq}}}{\Longrightarrow} \mathcal{V}';\mathcal{P}\rho^{\mathcal{V}}_{\mathcal{P}_{\sqsubseteq}} \overset{\theta}{\Longrightarrow}^{*} \mathcal{V}'';\emptyset$$

  By Lemma 4.3.49 we have $fV(\mathcal{P}\rho^{\mathcal{V}}_{\mathcal{P}_{\sqsubseteq}}) \subseteq \mathcal{V}$. By inductive hypothesis $\theta|_{\mathcal{V}'} \in Sol(\mathcal{P}\rho^{\mathcal{V}}_{\mathcal{P}_{\sqsubseteq}})$. By Lemma 4.3.33 we have $\rho^{\mathcal{V}}_{\mathcal{P}_{\sqsubseteq}}\circ\theta|_{\mathcal{V}'} \in Sol(\mathcal{P})$. By Lemma 4.3.50 we have $\rho^{\mathcal{V}}_{\mathcal{P}_{\sqsubseteq}}\circ\theta|_{\mathcal{V}'} = \rho^{\mathcal{V}}_{\mathcal{P}_{\sqsubseteq}}\circ\theta|_{\mathcal{V}'}$. By Lemma 4.3.43 we have $\rho^{\mathcal{V}}_{\mathcal{P}_{\sqsubseteq}}\circ\theta|_{\mathcal{V}} \in Sol(\mathcal{P})$. The result follows.

⊠

Lemma 4.3.52: If $\theta' \leq \theta''$ then $\theta\circ\theta' \leq \theta\circ\theta''$.

Proof. Suppose $\theta' \leq \theta''$. By Definition 4.3.46 there exists a θ''' such that $X^S\theta'' =_\alpha X^S(\theta'\circ\theta''')$. By Definition 4.2.35 we have $X^S(\theta\circ\theta'') \equiv (\theta(X^S))\theta''$. By Theorem 4.2.33 we have $(\theta(X^S))\theta'' =_\alpha (\theta(X^S))(\theta'\circ\theta''')$. By Theorem 4.2.36 we have $(\theta(X^S))(\theta'\circ\theta''') \equiv X^S(\theta\circ(\theta'\circ\theta'''))$. The result follows. ⊠

Lemma 4.3.53 checks that the instantiation ordering of Definition 4.3.46 remains invariant under α-equivalence.

Lemma 4.3.53: Suppose $X^S\theta =_\alpha X^S\theta'$ always. Then $\theta'' \leq \theta$ implies $\theta'' \leq \theta'$.

Proof. Suppose $\theta'' \leq \theta$. By Definition 4.3.46 there exists a θ''' such that $X^S\theta =_\alpha X^S(\theta''\circ\theta''')$. By assumption $X^S\theta =_\alpha X^S\theta'$. By Theorem 4.2.27 we have $X^S\theta' =_\alpha X^S(\theta''\circ\theta''')$. By Definition 4.3.46 we have $\theta'' \leq \theta'$. The result follows. ⊠

Lemma 4.3.54 and Lemma 4.3.55 are technical results used in the proof of Theorem 4.3.56.

Lemma 4.3.54: If $\theta \in Sol(\mathcal{P})$ then $\theta \in Sol(\mathcal{P}_{\sqsubseteq})$.

Proof. Suppose $\theta \in Sol(\mathcal{P})$, and suppose $r \overset{?}{=} s \in \mathcal{P}$. By Definition 4.3.3 we have $r\theta =_\alpha s\theta$. By Lemma 4.2.24 we have $fa(r\theta) = fa(s\theta)$. By Definition 4.3.7 we have $\theta \in Sol(\mathcal{P}_{\sqsubseteq})$. The result follows. ⊠

Lemma 4.3.55: If $X^S \in \mathcal{V}$ then $(\theta - X^S)|_{\mathcal{V}} = \theta|_{\mathcal{V}} - X^S$.

Proof. There are three cases:

- The case X^S with $X^S \in \mathcal{V}$. By Definition 4.3.38 we have $(\theta - X^S)|_{\mathcal{V}}(X^S) = (\theta - X^S)(X^S)$. By Definition 4.3.44 we have $(\theta - X^S)(X^S) = id\cdot X^S$. By Definition 4.3.44 we have $id\cdot X^S = (\theta|_{\mathcal{V}} - X^S)(X^S)$. The result follows.

- The case Y^T with $Y^T \in \mathcal{V}$. By Definition 4.3.38 we have $(\theta{-}X^S)|_{\mathcal{V}}(Y^T) = (\theta{-}X^S)(Y^T)$. By Definition 4.3.44 we have $(\theta{-}X^S)(Y^T) = \theta(Y^T)$. By Definition 4.3.44 we have $\theta(Y^T) = (\theta|_{\mathcal{V}}{-}X^S)(Y^T)$. The result follows.
- The case Y^T with $Y^T \notin \mathcal{V}$. By Definition 4.3.38 we have $(\theta{-}X^S)|_{\mathcal{V}}(Y^T) = id{\cdot}Y^T$. By Definition 4.3.38 we have $id{\cdot}Y^T = \theta(Y^T)|_{\mathcal{V}}$. By Definition 4.3.44 we have $\theta(Y^T)|_{\mathcal{V}} = (\theta|_{\mathcal{V}}{-}X^S)(Y^T)$. The result follows.

⊠

Theorem 4.3.56 and Theorem 4.3.58 are the main results in Section 4.3. Theorem 4.3.56 demonstrates that substitutions generated by the simplification relation of Definition 4.3.35 are principal solutions to a given unification problem.

Theorem 4.3.56: If $fV(\mathcal{P}) \subseteq \mathcal{V}$ and $\mathcal{V};\mathcal{P} \overset{\theta}{\Longrightarrow}{}^{*} \mathcal{V}';\emptyset$ then $\theta|_{\mathcal{V}}$ is a principal solution to $\mathcal{P}$.

Proof. By Theorem 4.3.51 we have $\theta|_{\mathcal{V}} \in Sol(\mathcal{P})$. We proceed by induction on the path length of $\mathcal{V};\mathcal{P} \Longrightarrow^{*} \mathcal{V}';\emptyset$.

- The base case. Then $\mathcal{P} = \emptyset$ and $\theta = id|_{\mathcal{V}}$. By Definition 4.3.46 we have $id|_{\mathcal{V}} \leq \theta'|_{\mathcal{V}}$. The result follows.
- The step case. There are three cases:
  - The non-instantiating case. Suppose:

$$\mathcal{V}\mathcal{P} \Longrightarrow \mathcal{V}\mathcal{P}' \overset{\theta}{\Longrightarrow}{}^{*} \mathcal{V}';\emptyset$$

  By inductive hypothesis $\theta|_{\mathcal{V}}$ is a principal solution to $\mathcal{P}'$. By Lemma 4.3.36 we have $Sol(\mathcal{P}) = Sol(\mathcal{P}')$. It is a fact that $\theta|_{\mathcal{V}}$ is a principal solution to $\mathcal{P}$. The result follows.
  - The case (**I1**) or (**I2**). We consider only the first case, as the second is similar. Suppose $fa(s) \subseteq \pi{\cdot}S$ and $X^S \notin fV(s)$. We have:

$$\mathcal{V};X^S \overset{?}{=} s,\ \mathcal{P} \overset{[X^S:=\pi^{-1}\cdot s]}{\Longrightarrow} \mathcal{V};\mathcal{P}[X^S{:=}\pi^{-1}{\cdot}s] \overset{\theta}{\Longrightarrow}{}^{*} \mathcal{V}';\emptyset$$

  Suppose that $\theta'|_{\mathcal{V}} \in Sol(\mathcal{P})$.
  By Theorem 4.3.51 we have $\theta'|_{\mathcal{V}} \in Sol(\mathcal{P}[X^S{:=}\pi^{-1}{\cdot}s])$. By Lemma 4.3.49 we have $fV(\mathcal{P}[X^S{:=}\pi^{-1}{\cdot}s]) \subseteq \mathcal{V}$. By Theorem 4.3.45 we have $([X^S{:=}\pi^{-1}{\cdot}s]\circ(\theta'|_{\mathcal{V}}{-}X^S))(X^S) =_\alpha \theta'|_{\mathcal{V}}(X^S)$ for all $X^S \in \mathcal{V}$. By Lemma 4.3.43 we have $([X^S{:=}\pi^{-1}{\cdot}s]\circ(\theta'|_{\mathcal{V}}{-}X^S)) \in Sol(\mathcal{P})$. By Lemma 4.3.33 we have $\theta'|_{\mathcal{V}}{-}X^S \in Sol(\mathcal{P}[X^S{:=}\pi^{-1}{\cdot}s])$. By Lemma 4.3.55 we have $(\theta'{-}X^S)|_{\mathcal{V}} \in Sol(\mathcal{P}[X^S{:=}\pi^{-1}{\cdot}s])$.
  By inductive hypothesis we have $\theta|_{\mathcal{V}} \leq (\theta'{-}X^S)|_{\mathcal{V}}$. By Lemma 4.3.52 we have $[X^S{:=}\pi^{-1}{\cdot}s]\circ(\theta|_{\mathcal{V}}) \leq [X^S{:=}\pi^{-1}{\cdot}s]\circ(\theta'{-}X^S)|_{\mathcal{V}}$. By Lemma 4.3.50 we have $[X^S{:=}\pi^{-1}{\cdot}s]\circ(\theta|_{\mathcal{V}}) = ([X^S{:=}\pi^{-1}{\cdot}s]\circ\theta)|_{\mathcal{V}}$, therefore $[X^S{:=}\pi^{-1}{\cdot}s]\circ\theta|_{\mathcal{V}} \leq [X^S{:=}\pi^{-1}{\cdot}s]\circ(\theta'{-}X^S)|_{\mathcal{V}}$. By Lemma 4.3.55 we have $(\theta'{-}X^S)|_{\mathcal{V}} = \theta'{-}X^S|_{\mathcal{V}}$ therefore $[X^S{:=}\pi^{-1}{\cdot}s]\circ\theta|_{\mathcal{V}} \leq [X^S{:=}\pi^{-1}{\cdot}s]\circ\theta'{-}X^S|_{\mathcal{V}}$. By Theorem 4.3.45 and Lemma 4.3.53 we have $[X^S{:=}\pi^{-1}{\cdot}s]\circ\theta|_{\mathcal{V}} \leq \theta'|_{\mathcal{V}}$. The result follows.
  - The case (**I3**). Suppose $\mathcal{P}_{\sqsubseteq}$ is consistent and non-trivial. We have:

$$\mathcal{V};\mathcal{P} \overset{\rho^{\mathcal{V}}_{\mathcal{P}_{\sqsubseteq}}}{\Longrightarrow} \mathcal{V}';\mathcal{P}\rho^{\mathcal{V}}_{\mathcal{P}_{\sqsubseteq}} \overset{\theta}{\Longrightarrow}{}^{*} \mathcal{V}'';\emptyset$$

Further, suppose that $\theta'|_{\mathcal{V}} \in \mathit{Sol}(\mathcal{P})$.
By Theorem 4.3.51 we have $\theta'|_{\mathcal{V}'} \in \mathit{Sol}(\mathcal{P}\rho^{\mathcal{V}}_{\mathcal{P}_\sqsubseteq})$. It is a fact that $\mathcal{V}' = \mathcal{V} \cup \mathcal{V}''\rho^{\mathcal{V}}_{\mathcal{P}_\sqsubseteq}$ therefore we have $fV(\mathcal{P}\rho^{\mathcal{V}}_{\mathcal{P}_\sqsubseteq}) \subseteq \mathcal{V}'$. By Lemma 4.3.54 we have $\theta'|_{\mathcal{V}} \in \mathit{Sol}(\mathcal{P}_\sqsubseteq)$. By Theorem 4.3.32 and Lemma 4.3.43 we have $\rho^{\mathcal{V}}_{\mathcal{P}_\sqsubseteq} \circ (\theta'|_{\mathcal{V}} - \rho^{\mathcal{V}}_{\mathcal{P}_\sqsubseteq}) \in \mathit{Sol}(\mathcal{P})$. By Lemma 4.3.33 we have $\theta'|_{\mathcal{V}} - \rho^{\mathcal{V}}_{\mathcal{P}_\sqsubseteq} \in \mathit{Sol}(\mathcal{P}\rho^{\mathcal{V}}_{\mathcal{P}_\sqsubseteq})$. By inductive hypothesis $\theta|_{\mathcal{V}} \leq \theta'|_{\mathcal{V}} - \rho^{\mathcal{V}}_{\mathcal{P}_\sqsubseteq}$. By Lemma 4.3.52 we have $\rho^{\mathcal{V}}_{\mathcal{P}_\sqsubseteq} \circ \theta|_{\mathcal{V}} \leq \rho^{\mathcal{V}}_{\mathcal{P}_\sqsubseteq} \circ (\theta'|_{\mathcal{V}} - \rho^{\mathcal{V}}_{\mathcal{P}_\sqsubseteq})$. It is a fact that $\rho^{\mathcal{V}}_{\mathcal{P}_\sqsubseteq} \circ \theta|_{\mathcal{V}} = (\rho^{\mathcal{V}}_{\mathcal{P}_\sqsubseteq} \circ \theta)|_{\mathcal{V}}$. By Theorem 4.3.32 and Lemma 4.3.53 we have $(\rho^{\mathcal{V}}_{\mathcal{P}_\sqsubseteq} \circ \theta)|_{\mathcal{V}} \leq \theta'|_{\mathcal{V}}$. The result follows.

⊠

Lemma 4.3.57 is a technical result used in the proof of Theorem 4.3.58.

Lemma 4.3.57: We have:

1. Suppose $fa(s) \subseteq \pi \cdot S$ and $X^S \notin fV(s)$.
   If $\mathcal{V}; \mathcal{P} \overset{[X^S := \pi^{-1} \cdot s]}{\Longrightarrow} \mathcal{V}; \mathcal{P}'$ with (**I1**) or (**I2**) then $\theta \in \mathit{Sol}(\mathcal{P})$ implies $\theta - X^S \in \mathit{Sol}(\mathcal{P}')$.
2. If $\mathcal{V}; \mathcal{P} \overset{\rho^{\mathcal{V}}_{\mathcal{P}_\sqsubseteq}}{\Longrightarrow} \mathcal{V}'; \mathcal{P}'$ with (**I3**) then $\theta \in \mathit{Sol}(\mathcal{P})$ implies $\theta - \rho^{\mathcal{V}}_{\mathcal{P}_\sqsubseteq} \in \mathit{Sol}(\mathcal{P}')$.

Proof. We consider the claims separately:

- We consider only the first case, as the case of (**I2**) is similar. Suppose $\mathcal{V}; \pi \cdot X^S \overset{?}{=} s,\ \mathcal{P}' \overset{[X^S := \pi^{-1} \cdot s]}{\Longrightarrow} \mathcal{V}; \mathcal{P}'[X^S := \pi^{-1} \cdot s]$. Suppose also that $\theta \in \mathit{Sol}(\mathcal{P})$. By Lemma 4.3.43 and Theorem 4.3.45 we have $[X^S := \pi^{-1} \cdot s] \circ (\theta - X^S)) \in \mathit{Sol}(\mathcal{P})$. By Lemma 4.3.33 we have $\theta - X^S \in \mathit{Sol}(\mathcal{P}[X^S := \pi^{-1} \cdot s])$. It is a fact that $\theta - X^S \in \mathit{Sol}(\mathcal{P}'[X^S := \pi^{-1} \cdot s])$. The result follows.
- Suppose $\mathcal{P}_\sqsubseteq$ is consistent and non-trivial. Suppose $\mathcal{V}; \mathcal{P} \overset{\rho^{\mathcal{V}}_{\mathcal{P}_\sqsubseteq}}{\Longrightarrow} \mathcal{V}''; \mathcal{P}\rho^{\mathcal{V}}_{\mathcal{P}_\sqsubseteq}$. Suppose also that $\theta \in \mathit{Sol}(\mathcal{P})$. By Lemma 4.3.43 and Theorem 4.3.32 we have $\rho^{\mathcal{V}}_{\mathcal{P}_\sqsubseteq} \circ (\theta - \rho^{\mathcal{V}}_{\mathcal{P}_\sqsubseteq}) \in \mathit{Sol}(\mathcal{P})$. By Lemma 4.3.33 we have $\theta - \rho^{\mathcal{V}}_{\mathcal{P}_\sqsubseteq} \in \mathit{Sol}(\mathcal{P}\rho^{\mathcal{V}}_{\mathcal{P}_\sqsubseteq})$. The result follows.

⊠

Theorem 4.3.58 shows that the unification algorithm of Definition 4.3.39 is 'correct'. That is, if a solution to a given unification problem exists, then the unification algorithm will find a principal solution to that problem. On the other hand, is a given unification problem is unsolvable, then the unification algorithm will halt in a failing state.

Theorem 4.3.58: Given a unification problem $\mathcal{P}$, if the unification algorithm of Definition 4.3.39 succeeds then it returns a principal solution, and if it fails, there is no solution.

Proof. In the case that the algorithm succeeds, we use Theorem 4.3.56.

Otherwise, the algorithm generates equalities of the form $\mathsf{f}(r_1, \ldots, r_n) \overset{?}{=} \mathsf{f}(s_1, \ldots, s_m)$ where $n \neq m$, $\mathsf{f}(r_1, \ldots, r_n) \overset{?}{=} \mathsf{g}(s_1, \ldots, s_m)$, $\mathsf{f}(r_1, \ldots, r_n) \overset{?}{=} [a]r$, $\mathsf{f}(r_1, \ldots, r_n) \overset{?}{=} a$, $[a]r \overset{?}{=} a$, $[a]r \overset{?}{=} b$, $a \overset{?}{=} b$, a $\mathcal{P}$ where $\mathcal{P}_\sqsubseteq$ is inconsistent, or elements of the form $\pi \cdot X^S \overset{?}{=} s$ or $r \overset{?}{=} X^S$ where $X^S \in fV(r)$ or $X^S \in fV(s)$. It is a fact that none of these have a solution. By Lemma 4.3.57 we have $\mathcal{P}$ has no solution. The result follows. ⊠

4.3.4 The algorithm in action

This Subsection provides a number of examples of the unification algorithm in action. Example 4.3.59 demonstrates the support reduction algorithm. Examples 4.3.60 and 4.3.61 demonstrate successful attempts to unify two terms. Example 4.3.62 demonstrates two terms that the algorithm fails to unify. We place a coloured box around the equality currently under attention.

Example 4.3.59: Suppose $a, c \in comb$ and $b, d \notin comb$. Take $S = comb = T$, $U = comb \cup \{b\}$ and $V = comb \cup \{d\}$. We first run the support reduction algorithm (Definition 4.3.8) on $Inc = \{a \sqsubseteq S,\ \mathsf{f}([a]a, id\cdot Y^T) \sqsubseteq T,\ (c\ a)\cdot Z^U \sqsubseteq S,\ (b\ a)\cdot W^V \sqsubseteq T\}$:

Remaining Problem		Rule Used
$\boxed{a \sqsubseteq S}, \mathsf{f}([a]a, id\cdot Y^T) \sqsubseteq T, (c\ a)\cdot Z^U \sqsubseteq S, (b\ a)\cdot W^V \sqsubseteq T$	$\Longrightarrow$	$(\sqsubseteq\mathbf{a})$
$\boxed{\mathsf{f}([a]a, id\cdot Y^T) \sqsubseteq T}, (c\ a)\cdot Z^U \sqsubseteq S, (b\ a)\cdot W^V \sqsubseteq T$	$\Longrightarrow$	$(\sqsubseteq\mathsf{f})$
$\boxed{[a]a \sqsubseteq T}, id\cdot Y^T \sqsubseteq T, (c\ a)\cdot Z^U \sqsubseteq S, (b\ a)\cdot W^V \sqsubseteq T$	$\Longrightarrow$	$(\sqsubseteq[])$
$\boxed{id\cdot Y^T \sqsubseteq T}, (c\ a)\cdot Z^U \sqsubseteq S, (b\ a)\cdot W^V \sqsubseteq T$	$\Longrightarrow$	$(\sqsubseteq\mathbf{X}')$
$\boxed{(c\ a)\cdot Z^U \sqsubseteq S}, (b\ a)\cdot W^V \sqsubseteq T$	$\Longrightarrow$	$(\sqsubseteq\mathbf{X})$
$Z^U \sqsubseteq (c\ a)\cdot S, \boxed{(b\ a)\cdot W^V \sqsubseteq T}$	$\Longrightarrow$	$(\sqsubseteq\mathbf{X})$
$Z^U \sqsubseteq (c\ a)\cdot S, W^V \sqsubseteq (b\ a)\cdot T$		

We take $nf(Inc) = \{Z^U \sqsubseteq (c\ a)\cdot S,\ W^V \sqsubseteq (b\ a)\cdot T\}$. By Definition 4.3.14 we have $nf(Inc)$ is consistent, therefore by Theorem 4.3.24 a solution for Inc exists.

We now construct $\rho^{\mathcal{V}}_{Inc}$. Take W' and Z' as our injective choices of fresh unknowns. Take $U' = U \cap ((c\ a)\cdot S \cap (b\ a)\cdot T)$ and $V' = V \cap ((c\ a)\cdot S \cap (b\ a)\cdot T)$. An easy calculation shows that $U' = comb \setminus \{a\} = V'$. We define $\rho^{\mathcal{V}}_{Inc}$ (Definition 4.3.21) piecewise:

$$\begin{array}{lcll} \rho^{\mathcal{V}}_{Inc}(Z^U) & = & id\cdot {Z'}^{U'} & \\ \rho^{\mathcal{V}}_{Inc}(W^V) & = & id\cdot {W'}^{V'} & \\ \rho^{\mathcal{V}}_{Inc}(X^S) & = & id\cdot X^S & \text{(for all other } X^S\text{)} \end{array}$$

Example 4.3.60: Suppose $a, c \in comb$ and $b, d \notin comb$. Take $S = comb \cup \{b, d\}$, $T = comb \cup \{f\}$ and $U = comb$. Take $\mathcal{V} = \{X^S, Y^T\}$. Suppose a term-former f.

We apply the algorithm to $\{\mathsf{f}([a]b, id{\cdot}Z^U, id{\cdot}X^S) \overset{?}{=} \mathsf{f}([d]b, [a]a, id{\cdot}Y^T)\}$:

$$\begin{array}{rlcc}
 & \underline{\text{Remaining Problem}} & & \underline{\text{Rule Used}} \\
\mathcal{V}; & \mathsf{f}([a]b, id{\cdot}Z^U, id{\cdot}X^S) \overset{?}{=} \mathsf{f}([d]b, [a]a, id{\cdot}Y^T) & \Longrightarrow & (\overset{?}{=}\mathsf{f}) \\
\mathcal{V}; & [a]b \overset{?}{=} [d]b,\ id{\cdot}Z^U \overset{?}{=} [a]a,\ id{\cdot}X^S \overset{?}{=} id{\cdot}Y^T & \Longrightarrow & (\overset{?}{=}[\mathbf{b}]) \\
\mathcal{V}; & (d\ a){\cdot}b \overset{?}{=} b,\ id{\cdot}Z^U \overset{?}{=} [a]a,\ id{\cdot}X^S \overset{?}{=} id{\cdot}Y^T & \Longrightarrow & (\overset{?}{=}\mathbf{a}) \\
\mathcal{V}; & id{\cdot}Z^U \overset{?}{=} [a]a,\ id{\cdot}X^S \overset{?}{=} id{\cdot}Y^T & \overset{[Z^U:=[a]a]}{\Longrightarrow} & (\overset{?}{=}\mathbf{I1}) \\
\mathcal{V}; & id{\cdot}X^S \overset{?}{=} id{\cdot}Y^T & \overset{[X^S:=X'^{S'}][Y^T:=Y'^{T'}]}{\Longrightarrow} & (\mathbf{I3}) \\
\mathcal{V} \cup \{X'^{S'}, Y'^{T'}\}; & id{\cdot}X'^{S'} \overset{?}{=} id{\cdot}Y'^{T'} & \overset{[X'^{S'}:=Y'^{T'}]}{\Longrightarrow} & (\mathbf{I1}) \\
\mathcal{V} \cup \{X'^{S'}, Y'^{T'}\}; & id{\cdot}Y'^{T'} \overset{?}{=} id{\cdot}Y'^{T'} & \Longrightarrow & (\overset{?}{=}\mathbf{X}) \\
\mathcal{V} \cup \{X'^{S'}, Y'^{T'}\}; & \emptyset \quad \text{(Success!)} & &
\end{array}$$

Here X' and Y' are the choice of unknown made in Definition 4.3.20, with $S' = \mathit{comb} = T'$. The algorithm succeeds and returns the substitution

$$[X'^{S'}:=Y'^{T'}]\circ[X^S:=X'^{S'}]\circ[Y^T:=Y'^{T'}]\circ[Z^U:=[a]a]$$

Example 4.3.61: Suppose $a, c \in \mathit{comb}$ and $d \notin \mathit{comb}$. Take $S = \mathit{comb}$ and $\mathcal{V} = \{X^S\}$.

We apply the algorithm to $\{\mathsf{g}([a](id{\cdot}X^S), [a]a) \overset{?}{=} \mathsf{g}([d]c, [d]d)\}$:

$$\begin{array}{rlcc}
 & \underline{\text{Remaining Problem}} & & \underline{\text{Rule Used}} \\
\mathcal{V}; & \mathsf{g}([a](id{\cdot}X^S), [a]a) \overset{?}{=} \mathsf{g}([d]c, [d]d) & \Longrightarrow & (\overset{?}{=}\mathsf{f}) \\
\mathcal{V}; & [a](id{\cdot}X^S) \overset{?}{=} [d]c, [a]a \overset{?}{=} [d]d & \Longrightarrow & (\overset{?}{=}[\mathbf{b}]) \\
\mathcal{V}; & (d\ a){\cdot}X^S \overset{?}{=} a, [a]a \overset{?}{=} [d]d & \overset{[X^S:=c]}{\Longrightarrow} & (\mathbf{I1}) \\
\mathcal{V}; & [a]a \overset{?}{=} [d]d & \Longrightarrow & (\overset{?}{=}[\mathbf{b}]) \\
\mathcal{V}; & (d\ a){\cdot}a \overset{?}{=} d & \Longrightarrow & (\overset{?}{=}\mathbf{a}) \\
\mathcal{V}; & \emptyset \quad \text{(Success!)} & &
\end{array}$$

The algorithm succeeds and returns the substitution $[X^S:=c]$.

Example 4.3.62: An example that fails to unify. Take $S = \mathit{comb}$ and $\mathcal{V} = \{X^S\}$. We run the algorithm on $\{[a]([b](id{\cdot}X^S)) \overset{?}{=} [a](id{\cdot}X^S)\}$:

$$\begin{array}{rlcc}
 & \underline{\text{Remaining Problem}} & & \underline{\text{Rule Used}} \\
\mathcal{V}; & [a]([b](id{\cdot}X^S)) \overset{?}{=} [a](id{\cdot}X^S) & \Longrightarrow & (\overset{?}{=}[\mathbf{a}]) \\
\mathcal{V}; & [b](id{\cdot}X^S) \overset{?}{=} id{\cdot}X^S & & \text{(Failure!)}
\end{array}$$

The algorithm fails as the precondition of rule (**I2**), $X^S \notin fV([b](id{\cdot}X^S))$, the 'occurs check', fails to hold. By Theorem 4.3.58 there is no solution to the unification problem.

4.4 Relation with nominal terms

Following Section 4.2 we have two differing notions of nominal term. The rest of this Section aims to clarify the precise relation between the two.

Subsection 4.4.1 reintroduces nominal terms. Subsection 4.4.2 reintroduces the notions of derivable freshness and equality on nominal terms. Subsection 4.4.3 defines a translation between the two forms of nominal term. Finally, Subsection 4.4.4 demonstrates that the translation of Subsection 4.4.3 maps solutions to unification problems in either form of term to solutions to unification problems in the other.

4.4.1 Nominal terms

The following material should be familiar to those acquainted with the Urban, Pitts and Gabbay paper on nominal unification [UPG04]. We introduce definitions, and demonstrate basic properties of nominal terms, that are necessary for proving the correctness of the translation.

Definition 4.4.1: Fix a countably infinite set of **nominal atoms** $\dot{\mathbb{A}}$. Dotted lowercase letters from the beginning of the alphabet, $\dot{a}, \dot{b}, \dot{c}$, and so on, will range over nominal atoms. As usual, we employ the permutative convention: $\dot{a}$ and $\dot{b}$ denote two distinct nominal atoms.

Definition 4.4.2: Fix a bijection ι between nominal atoms and any permissions sort. For concreteness, we will assume that this permission sort is *comb*, but any infinite permission sort will do.

Definition 4.4.3: Fix a countably infinite set of **nominal unknowns**. Dotted uppercase letters from toward the end of the alphabet, $\dot{X}, \dot{Y}, \dot{Z}$, and so on, will range over nominal unknowns.

Definition 4.4.4: A **nominal permutation** $\dot{\pi}$ is a bijection on nominal atoms such that $nontriv(\dot{\pi})$ is finite. $\dot{\pi}, \dot{\pi}', \dot{\pi}''$, and so on, will range over nominal permutations.

As for the permissive case, we write $\dot{\pi}^{-1}$ for the **inverse** of a nominal permutation, and write the **composition** of two nominal permutation as $\dot{\pi}\circ\dot{\pi}'$ (so that $(\dot{\pi}\circ\dot{\pi}')(\dot{a}) = \dot{\pi}(\dot{\pi}'(\dot{a}))$).

We write $(\dot{b}\ \dot{a})$ for the nominal **swapping** permutation that maps $\dot{b}$ to $\dot{a}$ to $\dot{b}$, and all other $\dot{c}$ to themselves.

Swappings are self-inverse, so $(\dot{b}\ \dot{a})\circ(\dot{b}\ \dot{a}) = \dot{id}$ always. Further the order atoms appear within a swapping does not matter, so $(\dot{b}\ \dot{a}) = (\dot{a}\ \dot{b})$.

Definition 4.4.5: Define **nominal terms** by:

$$\dot{r}, \dot{s}, \dot{t} ::= \dot{a} \mid \dot{\pi}\cdot\dot{X} \mid [\dot{a}]\dot{r} \mid \mathsf{f}(\dot{r}_1, \ldots, \dot{r}_n)$$

To construct nominal terms, we use the same set of term-formers that we used when constructing permissive nominal terms.

We write $\equiv$ for syntactic identity between terms, using $\approx$ for derivable equality, which will be defined later, and reserve $=$ for setwise equality.

$$\frac{}{\Delta \vdash \dot{a}\#\dot{b}}\,(\#\dot{\mathbf{a}}) \qquad \frac{(\dot{\pi}^{-1}(\dot{a})\#\dot{X} \in \Delta)}{\Delta \vdash \dot{a}\#\dot{\pi}\cdot\dot{X}}\,(\#\dot{\mathbf{X}}) \qquad \frac{}{\Delta \vdash \dot{a}\#[\dot{a}]\dot{r}}\,(\#[\dot{\mathbf{a}}])$$

$$\frac{\Delta \vdash \dot{a}\#\dot{r}_i \quad (\text{for } 1 \leq i \leq n)}{\Delta \vdash \dot{a}\#\mathsf{f}(\dot{r}_1, \ldots, \dot{r}_n)}\,(\#\mathsf{f}) \qquad \frac{\Delta \vdash \dot{a}\#\dot{s}}{\Delta \vdash \dot{a}\#[\dot{b}]\dot{s}}\,(\#[\dot{\mathbf{b}}])$$

Figure 4.4 Rules for derivable freshness on nominal terms

$$\frac{}{\Delta \vdash \dot{a} \approx \dot{a}}\,(\approx\dot{\mathbf{a}}) \qquad \frac{\Delta \vdash \dot{r}_i \approx \dot{s}_i \quad (\text{for } 1 \leq i \leq n)}{\Delta \vdash \mathsf{f}(\dot{r}_1, \ldots, \dot{r}_n) \approx \mathsf{f}(\dot{s}_1, \ldots, \dot{s}_n)}\,(\approx\mathsf{f}) \qquad \frac{\Delta \vdash \dot{r} \approx \dot{s}}{\Delta \vdash [\dot{a}]\dot{r} \approx [\dot{a}]\dot{s}}\,(\approx[\dot{\mathbf{a}}])$$

$$\frac{\Delta \vdash (\dot{b}\ \dot{a})\cdot\dot{r} \approx \dot{s} \quad \Delta \vdash \dot{b}\#\dot{r}}{\Delta \vdash [\dot{a}]\dot{r} \approx [\dot{b}]\dot{s}}\,(\approx[\dot{\mathbf{b}}]) \qquad \frac{\dot{a}\#\dot{X} \in \Delta \quad (\text{for every } \dot{a} \text{ such that } \dot{\pi}(\dot{a}) \neq \dot{\pi}'(\dot{a}))}{\dot{\pi}\cdot\dot{X} \approx \dot{\pi}'\cdot\dot{X}}\,(\approx\dot{\mathbf{X}})$$

Figure 4.5 Rules for derivable equality on nominal terms

Definition 4.4.6: Define a **nominal permutation action** by:

$$\dot{\pi}\cdot\dot{a} \equiv \dot{\pi}(\dot{a}) \qquad \dot{\pi}\cdot(\dot{\pi}'\cdot\dot{X}) \equiv (\dot{\pi}\circ\dot{\pi}')\cdot\dot{X} \qquad \dot{\pi}\cdot[\dot{a}]\dot{r} \equiv [\dot{\pi}(\dot{a})](\dot{\pi}\cdot\dot{r})$$
$$\dot{\pi}\cdot\mathsf{f}(\dot{r}_1, \ldots, \dot{r}_n) \equiv \mathsf{f}(\dot{\pi}\cdot\dot{r}_1, \ldots, \dot{\pi}\cdot\dot{r}_n)$$

4.4.2 Derivable freshness and equality

In this section, we introduce derivable freshness, and derivable equality for nominal terms.

Definition 4.4.7: A **freshness** is a pair $\dot{a}\#\dot{r}$. A **primitive freshness** is a pair $\dot{a}\#\dot{X}$. Call a finite set of primitive freshnesses a **freshness context**. $\Delta, \Delta', \Delta''$, and so on, will range over freshness contexts.

Read $\dot{a}\#\dot{r}$ as '$\dot{a}$ is fresh for $\dot{r}$'.

Definition 4.4.8: Define **derivable freshness** on nominal terms by the rules in Figure 4.4.

We use the convenient shorthand $\Delta \vdash \dot{a}\#\dot{r}$ for '$\Delta \vdash \dot{a}\#\dot{r}$ is derivable using the rules in Figure 4.4'. We will employ this notation with the only other judgment form, derivable equality, that we will be considering in this section.

All rules in Figure 4.4 are syntax directed. A decision procedure that decides whether a nominal atom is fresh for a nominal term, with respect to a context of freshness assumptions, is obtained by reading each rule backward.

Definition 4.4.9: An **equality** is a pair $\dot{r} \approx \dot{s}$.

Definition 4.4.10: Define **derivable equality** on nominal terms by the rules in Figure 4.5.

4.4.3 The translation

In this section we define an interpretation function mapping nominal terms into permissive nominal terms.

The most important result in this section is Theorem 4.4.15, where we demonstrate that derivable equality in the nominal world is preserved and reflected by derivable equality in the permissive nominal world, under the interpretation.

Definition 4.4.11: Define a **mapping** between nominal permutations and permissive nominal permutations by:

$$[\![\dot\pi]\!](\iota(\dot a)) = \iota(\dot\pi(\dot a)) \quad \text{and} \quad [\![\dot\pi]\!](c) = c \quad \text{for all other } c \in \mathbb{A} \setminus \mathit{comb}$$

Definition 4.4.12: For each $\dot X$ make an arbitrary but fixed injective choice of X (so that the choices for $\dot X$ and $\dot Y$ are always distinct). Define an **interpretation** $[\![\dot r]\!]_\Delta$ by:

$$[\![\dot a]\!]_\Delta \equiv \iota(\dot a) \qquad [\![\dot\pi\cdot\dot X]\!] \equiv [\![\dot\pi]\!]\cdot X^S \quad \text{where } S = \mathit{comb} \setminus \{\iota(\dot a) \mid \dot a \# \dot X \in \Delta\}$$
$$[\![[\dot a]\dot r]\!]_\Delta \equiv [\iota(\dot a)][\![\dot r]\!]_\Delta \qquad [\![\mathsf{f}(\dot r_1, \ldots, \dot r_n)]\!]_\Delta \equiv \mathsf{f}([\![\dot r_1]\!]_\Delta, \ldots, [\![\dot r_n]\!]_\Delta)$$

The only interesting case in Definition 4.4.12 is that for $[\![\dot\pi\cdot\dot X]\!]_\Delta$. Here, we remove elements from our assumed fixed permission sort, *comb*, whose image under the mapping of Definition 4.4.11 was assumed fresh for $\dot X$ in Δ.

Lemma 4.4.13 states that the interpretation commutes with the permutation action on nominal terms.

Lemma 4.4.13: $[\![\dot\pi]\!]\cdot[\![\dot r]\!]_\Delta \equiv [\![\dot\pi\cdot\dot r]\!]_\Delta$

Proof. By induction on $\dot r$.

- The case $\dot a$. By Definition 4.4.12 we have $[\![\dot a]\!]_\Delta = \iota(\dot a)$. There are two cases:
  - The case $\iota(\dot a) \in \mathit{comb}$. By Definition 4.4.11 we have $[\![\dot\pi]\!](\iota(\dot a)) = \iota(\dot\pi(\dot a))$. By Definition 4.4.12 we have $\iota(\dot\pi(\dot a)) \equiv [\![\dot\pi(\dot a)]\!]_\Delta$. By Definition 4.4.6 we have $[\![\dot\pi(\dot a)]\!]_\Delta \equiv [\![\dot\pi\cdot\dot a]\!]_\Delta$. The result follows.
  - The case $\iota(\dot a) \in \mathit{comb} \setminus \mathbb{A}$. By Definition 4.4.2, this case is impossible.
- The case $\dot\pi'\cdot\dot X$. By Definition 4.4.6 we have $[\![\dot\pi\cdot(\dot\pi'\cdot\dot X)]\!]_\Delta \equiv [\![(\dot\pi\circ\dot\pi')\cdot\dot X]\!]_\Delta$. By Definition 4.4.12 we have $[\![(\dot\pi\circ\dot\pi')\cdot\dot X]\!]_\Delta \equiv [\![\dot\pi\circ\dot\pi']\!]\cdot X^S$, where $S = \mathit{comb} \setminus \{\iota(\dot a) \mid \dot a\#\dot X \in \Delta\}$. By Definition 4.4.12 we have $[\![\dot\pi\circ\dot\pi']\!]\cdot X^S \equiv [\![\dot\pi\circ\dot\pi']\!]\cdot[\![\dot{id}\cdot\dot X]\!]_\Delta$. The result follows.
- The case $[\dot a]\dot r$. By Definition 4.4.6 we have $[\![\dot\pi\cdot[\dot a]\dot r]\!]_\Delta \equiv [\![[\dot\pi(\dot a)](\dot\pi\cdot\dot r)]\!]_\Delta$. By Definition 4.4.12 we have $[\![[\dot\pi(\dot a)](\dot\pi\cdot\dot r)]\!]_\Delta \equiv [\iota(\dot\pi(\dot a))][\![\dot\pi\cdot\dot r]\!]_\Delta$. By inductive hypothesis $[\iota(\dot\pi(\dot a))][\![\dot\pi\cdot\dot r]\!]_\Delta \equiv [\iota(\dot\pi(\dot a))]([\![\pi]\!]\cdot [\![\dot r]\!]_\Delta)$. By Definition 4.4.11 we have $[\iota(\dot\pi(\dot a))]([\![\pi]\!]\cdot[\![\dot r]\!]_\Delta) \equiv [[\![\dot\pi]\!](\iota(\dot a)))]([\![\pi]\!]\cdot[\![\dot r]\!]_\Delta)$. By Definition 4.2.9 we have $[[\![\dot\pi]\!](\iota(\dot a)))]([\![\pi]\!]\cdot[\![\dot r]\!]_\Delta) \equiv [\![\dot\pi]\!]\cdot[\iota(\dot a)][\![\dot r]\!]_\Delta$. By Definition 4.4.12 we have $[\![\dot\pi]\!]\cdot[\iota(\dot a)][\![\dot r]\!]_\Delta \equiv [\![\dot\pi]\!]\cdot[\![[\dot a]\dot r]\!]_\Delta$. The result follows.
- The case $\mathsf{f}(\dot r_1, \ldots, \dot r_n)$. By Definition 4.4.6 we have $[\![\dot\pi\cdot\mathsf{f}(\dot r_1, \ldots, \dot r_n)]\!]_\Delta \equiv \mathsf{f}([\![\dot\pi\cdot\dot r_1, \ldots, [\![\dot\pi\cdot \dot r_n)]\!]_\Delta$. By inductive hypotheses $\mathsf{f}([\![\dot\pi\cdot\dot r_1]\!]_\Delta, \ldots, [\![\dot\pi\cdot\dot r_n]\!]_\Delta)]\!] \equiv \mathsf{f}([\![\dot\pi]\!]\cdot[\![\dot r_1]\!]_\Delta, \ldots, [\![\dot\pi]\!]\cdot[\![\dot r_n]\!]_\Delta)$. By Definition 4.2.9 we have $\mathsf{f}([\![\dot\pi]\!]\cdot[\![\dot r_1]\!]_\Delta, \ldots, [\![\dot\pi]\!]\cdot[\![\dot r_n]\!]_\Delta) \equiv [\![\dot\pi]\!]\cdot\mathsf{f}([\![\dot r_1]\!]_\Delta, \ldots, [\![\dot r_n]\!]_\Delta)$. The result follows.

⊠

Lemma 4.4.14 demonstrates that the interpretation also preserves and reflects freshness.

Lemma 4.4.14: $\iota(\dot{a}) \notin fa(\llbracket \dot{r} \rrbracket_\Delta)$ if and only if $\Delta \vdash \dot{a}\#\dot{r}$.

Proof. We handle the two cases separately:

- The left to right case. By induction on $\dot{r}$.
  - The case $\dot{b}$. Using $(\#\dot{\mathbf{a}})$ we obtain $\Delta \vdash \dot{a}\#\dot{b}$. The result follows.
  - The case $\dot{\pi}\cdot\dot{X}$. Suppose $\iota(a) \notin fa(\llbracket \dot{\pi}\cdot\dot{X} \rrbracket_\Delta)$. By Definition 4.4.12 we have $fa(\llbracket \dot{\pi}\cdot \dot{X} \rrbracket_\Delta) = fa(\llbracket \pi \rrbracket\cdot X^S)$ where $S = \mathit{comb} \setminus \{\iota(\dot{a}) \mid \dot{a}\#\dot{X}\}$. By Definition 4.2.11 we have $fa(\llbracket \pi \rrbracket\cdot X^S) = \llbracket \dot{\pi} \rrbracket\cdot S$. By Definition 4.4.11 we have $\dot{\pi}^{-1}(\dot{a})\#\dot{X} \in \Delta$. Using $(\#\dot{\mathbf{X}})$ we obtain $\Delta \vdash \dot{a}\#\dot{X}$. The result follows.
  - The case $[\dot{a}]\dot{r}$. Using $(\#[\dot{\mathbf{a}}])$ we obtain $\Delta \vdash \dot{a}\#[\dot{a}]\dot{r}$. The result follows.
  - The case $[\dot{b}]\dot{s}$. Suppose $\dot{a} \notin fa(\llbracket [\dot{b}]\dot{s} \rrbracket_\Delta)$. By Definition 4.4.12 we have $fa(\llbracket [\dot{b}]\dot{s} \rrbracket_\Delta) = fa(\llbracket \dot{s} \rrbracket_\Delta) \setminus \{\iota(\dot{b})\}$. By inductive hypothesis $\Delta \vdash \dot{a}\#\dot{s}$. Using $(\#[\dot{\mathbf{b}}])$ we obtain $\Delta \vdash \dot{a}\#[\dot{b}]\dot{s}$. The result follows.
  - The case $\mathsf{f}(\dot{r}_1, \ldots, \dot{r}_n)$. Suppose $\dot{a} \notin fa(\llbracket \mathsf{f}(\dot{r}_1, \ldots, \dot{r}_n) \rrbracket_\Delta)$. By Definition 4.4.12 we have $fa(\llbracket \mathsf{f}(\dot{r}_1, \ldots, \dot{r}_n) \rrbracket_\Delta) = fa(\mathsf{f}(\llbracket \dot{r}_1 \rrbracket_\Delta, \ldots, \llbracket \dot{r}_n \rrbracket_\Delta))$. By Definition 4.2.11 we have $fa(\mathsf{f}(\llbracket \dot{r}_1 \rrbracket_\Delta, \ldots,$
    $\llbracket \dot{r}_n \rrbracket_\Delta)) = \bigcup_{1 \leq i \leq n} fa(\llbracket \dot{r}_i \rrbracket_\Delta)$. By inductive hypothesis $\Delta \vdash \dot{a}\#\dot{r}_i$ for $1 \leq i \leq n$. Using $(\#\mathsf{f})$ we obtain $\Delta\dot{a}\#\mathsf{f}(\dot{r}_1, \ldots, \dot{r}_n)$. The result follows.
- The right to left case. By induction on the derivation of $\Delta \vdash \dot{a}\#\dot{r}$.
  - The case $(\#\dot{\mathbf{a}})$. Suppose $\Delta \vdash \dot{a}\#\dot{b}$. By Definition 4.4.12 we have $\llbracket \dot{b} \rrbracket_\Delta \equiv \iota(\dot{b})$. By Definition 4.2.11 we have $fa(\iota(\dot{b})) = \{\iota(\dot{b})\}$. The result follows, as ι is injective.
  - The case $(\#\dot{\mathbf{X}})$. Suppose $\dot{\pi}^{-1}(\dot{a})\#\dot{X} \in \Delta$ therefore $\Delta \vdash \dot{a}\#\dot{\pi}\cdot\dot{X}$. By Definition 4.4.12 we have $\llbracket \dot{\pi}\cdot\dot{X} \rrbracket_\Delta \equiv \llbracket \dot{\pi} \rrbracket\cdot X^S$ where $S = \mathit{comb} \setminus \{\iota(\dot{a}) \mid \dot{a}\#\dot{X}\}$. By Definition 4.2.11 we have $fa(\llbracket \dot{\pi} \rrbracket\cdot X^S) = \llbracket \dot{\pi} \rrbracket\cdot S$. The result follows.
  - The case $(\#[\dot{\mathbf{a}}])$. By Definition 4.4.12 we have $\llbracket [\dot{a}]\dot{r} \rrbracket_\Delta \equiv [\iota(\dot{a})]\llbracket \dot{r} \rrbracket_\Delta$. By Definition 4.2.11 we have $\iota(\dot{a}) \notin fa([\iota(\dot{a})]\dot{r})$. The result follows.
  - The case $(\#[\dot{\mathbf{b}}])$. Suppose $\Delta \vdash \dot{a}\#\dot{r}$. By inductive hypothesis $\iota(\dot{a}) \notin fa(\llbracket \dot{r} \rrbracket_\Delta)$. Using $(\#[\dot{\mathbf{b}}])$ we obtain $\Delta \vdash \dot{a}\#[\dot{b}]\dot{r}$. By Definition 4.4.12 we have $\llbracket [\dot{b}]\dot{r} \rrbracket_\Delta \equiv [\iota(\dot{b})]\llbracket \dot{r} \rrbracket_\Delta$. By Definition 4.2.11 we have $fa([\iota(\dot{b})]\llbracket \dot{r} \rrbracket_\Delta) = fa(\llbracket \dot{r} \rrbracket_\Delta) \setminus \{\iota(\dot{b})\}$. The result follows.
  - The case $(\#\mathsf{f})$. Suppose $\Delta \vdash \dot{a}\#\dot{r}_i$ for $1 \leq i \leq n$. By inductive hypotheses $\iota(\dot{a}) \notin fa(\llbracket \dot{r}_i \rrbracket_\Delta)$ for $1 \leq i \leq n$. Using $(\#\mathsf{f})$ we obtain $\Delta \vdash \dot{a}\#\mathsf{f}(\dot{r}_1, \ldots, \dot{r}_n)$. By Definition 4.4.12 we have $\llbracket \mathsf{f}(\dot{r}_1, \ldots, \dot{r}_n) \rrbracket_\Delta \equiv \mathsf{f}(\llbracket \dot{r}_1 \rrbracket_\Delta, \ldots, \llbracket \dot{r}_n \rrbracket_\Delta)$. By Definition 4.2.11 we have $fa(\mathsf{f}(\llbracket \dot{r}_1 \rrbracket_\Delta, \ldots, \llbracket \dot{r}_n \rrbracket_\Delta)) = \bigcup_{1 \leq i \leq n} fa(\llbracket \dot{r}_i \rrbracket_\Delta)$. The result follows.

⊠

Theorem 4.4.15 states that the interpretation preserves and reflects derivable equality between terms.

Theorem 4.4.15: $[\![\dot{r}]\!]_\Delta =_\alpha [\![\dot{s}]\!]_\Delta$ if and only if $\Delta \vdash \dot{r} \approx \dot{s}$.

Proof. We handle the two cases separately:

- The left to right case. By induction on the derivation of $[\![\dot{r}]\!]_\Delta =_\alpha [\![\dot{s}]\!]_\Delta$.
  - The case $(=_\alpha\mathbf{a})$. Suppose $[\![\dot{a}]\!]_\Delta =_\alpha [\![\dot{a}]\!]_\Delta$. Using $(\approx\dot{\mathbf{a}})$ we obtain $\Delta \vdash \dot{a} \approx \dot{a}$. The result follows.
  - The case $(=_\alpha\mathbf{X})$. Suppose $[\![\dot{\pi}\cdot\dot{X}]\!] =_\alpha [\![\dot{\pi}'\cdot\dot{X}]\!]$. By Definition 4.4.12 we have $[\![\dot{\pi}]\!]\cdot X^S =_\alpha [\![\dot{\pi}']\!]\cdot X^S$ where $S = \mathit{comb} \setminus \{\iota(\dot{a}) \mid \dot{a}\#\dot{X} \in \Delta\}$. Using $(=_\alpha\mathbf{X})$ we see $[\![\dot{\pi}]\!]|_S = [\![\dot{\pi}']\!]|_S$. As ι is injective, we have $\dot{a}\#\dot{X} \in \Delta$ for all $\dot{a}$ such that $\dot{\pi}(\dot{a}) \neq \dot{\pi}'(\dot{a})$. Using $(\approx\dot{\mathbf{X}})$ we obtain $\Delta \vdash \dot{\pi}\cdot\dot{X} \approx \dot{\pi}'\cdot\dot{X}$. The result follows.
  - The case $(=_\alpha[\mathbf{a}])$. Suppose $[\![\dot{r}]\!]_\Delta =_\alpha [\![\dot{s}]\!]_\Delta$. By inductive hypothesis $\Delta \vdash \dot{r} =_\alpha \dot{s}$. Using $(=_\alpha[\mathbf{a}])$ we obtain $[\iota(\dot{a})][\![\dot{r}]\!]_\Delta =_\alpha [\iota(\dot{a})][\![\dot{s}]\!]_\Delta$. Using $(\approx[\dot{\mathbf{a}}])$ we obtain $[\dot{a}]\dot{r} \approx [\dot{a}]\dot{s}$. By Definition 4.4.12 we have $[\![[\dot{a}]\dot{r}]\!]_\Delta \equiv [\iota(\dot{a})][\![\dot{r}]\!]_\Delta$. The result follows.
  - The case $(=_\alpha[\mathbf{b}])$. Suppose $[\![(\dot{b}\ \dot{a})\cdot\dot{r}]\!]_\Delta =_\alpha [\![\dot{s}]\!]_\Delta$ with $\iota(\dot{b}) \notin fa([\![\dot{r}]\!]_\Delta)$. By Definition 4.4.12 we have $[\![(\dot{b}\ \dot{a})]\!]\cdot[\![\dot{r}]\!]_\Delta =_\alpha [\![\dot{s}]\!]_\Delta$. By Definition 4.4.11 we have $(\iota(\dot{b})\ \iota(\dot{a}))\cdot [\![\dot{r}]\!]_\Delta =_\alpha [\![\dot{s}]\!]_\Delta$. By Lemma 4.4.14 and Definition 4.4.12 we have $\Delta \vdash \dot{b}\#\dot{r}$. By inductive hypothesis $\Delta \vdash (\dot{b}\ \dot{a})\cdot\dot{r} \approx \dot{s}$. Using $(=_\alpha[\mathbf{b}])$ we obtain $[\iota(\dot{a})][\![\dot{r}]\!]_\Delta =_\alpha [\iota(\dot{b})][\![\dot{s}]\!]_\Delta$. Using $(\approx[\dot{\mathbf{b}}])$ we obtain $\Delta \vdash [\dot{a}]\dot{r} \approx [\dot{b}]\dot{s}$. By Definition 4.4.12 we have $[\![\Delta \vdash [\dot{a}]\dot{r}]\!]_\Delta \equiv [\iota(\dot{a})][\![\dot{r}]\!]_\Delta$. The result follows.
  - The case $(=_\alpha\mathsf{f})$. Suppose $[\![\dot{r}_i]\!]_\Delta =_\alpha [\![\dot{s}_i]\!]_\Delta$ for $1 \leq i \leq n$. By inductive hypotheses $\Delta \vdash \dot{r}_i \approx \dot{s}_i$ for $1 \leq i \leq n$. Using $(=_\alpha\mathsf{f})$ we obtain $\mathsf{f}([\![\dot{r}_1]\!]_\Delta, \ldots, [\![\dot{r}_n]\!]_\Delta) =_\alpha \mathsf{f}([\![\dot{s}_1]\!]_\Delta, \ldots, [\![\dot{s}_n]\!]_\Delta)$. Using $(\approx\mathsf{f})$ we obtain $\mathsf{f}(\dot{r}_1, \ldots, \dot{r}_n) \approx \mathsf{f}(\dot{s}_1, \ldots, \dot{s}_n)$. By Definition 4.4.12 we have $[\![\mathsf{f}(\dot{r}_1, \ldots,$
    $\dot{r}_n)]\!]_\Delta =_\alpha [\![\mathsf{f}(\dot{s}_1, \ldots, \dot{s}_n)]\!]_\Delta$. The result follows.
- The right to left case. By induction on the derivation of $\Delta \vdash \dot{r} \approx \dot{s}$.
  - The case $(\approx\dot{\mathbf{a}})$. Suppose $\Delta \vdash \dot{a} \approx \dot{a}$. By Definition 4.4.12 we have $[\![\dot{a}]\!]_\Delta \equiv \iota(\dot{a})$. Using $(=_\alpha\mathbf{a})$ we obtain $\iota(\dot{a}) =_\alpha \iota(\dot{a})$. The result follows.
  - The case $(\approx\dot{\mathbf{X}})$. Suppose $\dot{a}\#\dot{X} \in \Delta$ for every $\dot{a}$ such that $\dot{\pi}(a) \neq \dot{\pi}'(\dot{a})$ therefore $\dot{\pi}\cdot \dot{X} \approx \dot{\pi}'\cdot\dot{X}$. By Definition 4.4.12 we have $[\![\dot{\pi}\cdot\dot{X}]\!]_\Delta \equiv [\![\pi]\!]\cdot X^S$ where $S = \mathit{comb} \setminus \{\iota(\dot{a}) \mid \dot{a}\#\dot{X} \in \Delta\}$. It is a fact that $[\![\pi]\!]|_S = [\![\pi]\!]'|_S$. Using $(=_\alpha\mathbf{X})$ we obtain $[\![\pi]\!]\cdot X^S =_\alpha [\![\pi]\!]'\cdot X^S$. The result follows.
  - The case $(\approx[\dot{\mathbf{a}}])$. Suppose $\Delta \vdash \dot{r} \approx \dot{s}$. By inductive hypothesis $[\![\dot{r}]\!]_\Delta =_\alpha [\![\dot{s}]\!]_\Delta$. Using $(=_\alpha[\mathbf{a}])$ we obtain $[\iota(\dot{a})][\![\dot{r}]\!]_\Delta =_\alpha [\iota(\dot{a})][\![\dot{s}]\!]_\Delta$. By Definition 4.4.12 we have $[\![[\dot{a}]\dot{r}]\!]_\Delta =_\alpha [\![[\dot{a}]\dot{s}]\!]_\Delta$. The result follows.
  - The case $(\approx[\dot{\mathbf{b}}])$. Suppose $\Delta \vdash (\dot{b}\ \dot{a})\cdot\dot{r} \approx \dot{s}$ with $\Delta \vdash \dot{b}\#\dot{r}$. By Lemma 4.4.14 and Definition 4.4.12 we have $[\![\dot{b}]\!]_\Delta \notin fa([\![\dot{r}]\!]_\Delta)$. By Definition 4.4.12 we have $\iota(\dot{b}) \notin fa([\![\dot{r}]\!]_\Delta)$. By inductive hypothesis $[\![(\dot{b}\ \dot{a})\cdot\dot{r}]\!] =_\alpha [\![\dot{s}]\!]$. By Definition 4.4.12 we have $[\![(\dot{b}\ \dot{a})]\!]\cdot[\![\dot{r}]\!]_\Delta \approx [\![\dot{s}]\!]_\Delta$. By Definition 4.4.11 we have $[\![(\dot{b}\ \dot{a})]\!]\cdot[\![\dot{r}]\!]_\Delta \equiv (\iota(\dot{b})\ \iota(\dot{a}))\cdot[\![\dot{r}]\!]_\Delta$. Using $(=_\alpha[\mathbf{b}])$ we obtain $[\iota(\dot{a})][\![\dot{r}]\!]_\Delta =_\alpha [\iota(\dot{b})][\![\dot{s}]\!]_\Delta$. By Definition 4.4.12 we have $[\![[\dot{a}]\dot{r}]\!]_\Delta =_\alpha [\![[\dot{b}]\dot{s}]\!]_\Delta$. The result follows.

- The case ($\approx$f). Suppose $\Delta \vdash \dot{r}_i \approx \dot{s}_i$ for $1 \leq i \leq n$. By inductive hypotheses $[\![\dot{r}_i]\!]_\Delta =_\alpha [\![\dot{s}_i]\!]_\Delta$ for $1 \leq i \leq n$. Using ($=_\alpha$f) we obtain $\mathsf{f}([\![\dot{r}_1]\!]_\Delta, \ldots, [\![\dot{r}_n]\!]_\Delta) =_\alpha \mathsf{f}([\![\dot{s}_1]\!]_\Delta, \ldots, [\![\dot{s}_n]\!]_\Delta)$. By Definition 4.4.12 we have $[\![\mathsf{f}(\dot{r}_1, \ldots, \dot{r}_n)]\!]_\Delta =_\alpha [\![\mathsf{f}(\dot{s}_1, \ldots, \dot{s}_n)]\!]_\Delta$. The result follows.

⊠

4.4.4 Substitutions and solutions to unification problems

This Subsection reintroduces the notion of a substitution, and substitution action, on nominal terms, as well as nominal unification problems and their solutions. Definition 4.4.19 extends the translation function of Definition 4.4.12 to solutions of nominal unification problems, and Definition 4.4.21 defines a translation function for nominal unification problems. The Subsection concludes with Theorem 4.4.24 which states that the translations proferred preserve and reflect unification solutions.

Definition 4.4.16: A **nominal substitution** $\dot{\theta}$ is a function from nominal unknowns to nominal terms such that the set $\{\dot{X} \mid \dot{\theta}(\dot{X}) \neq \dot{id}\cdot\dot{X}\}$ is finite. $\dot{\theta}, \dot{\theta}', \dot{\theta}'', \ldots$ will range over nominal substitutions. Write $\dot{id}$ for the **identity** mapping $\dot{X}$ to $\dot{id}\cdot\dot{X}$.

Definition 4.4.17: Define a **nominal substitution action** by:

$$\dot{a}\dot{\theta} \equiv \dot{a} \quad (\dot{\pi}\cdot\dot{X})\dot{\theta} \equiv \dot{\pi}\cdot\dot{\theta}(\dot{X}) \quad ([\dot{a}]\dot{r})\dot{\theta} \equiv [\dot{a}](\dot{r}\dot{\theta}) \quad \mathsf{f}(\dot{r}_1, \ldots, \dot{r}_n)\dot{\theta} \equiv \mathsf{f}(\dot{r}_1\dot{\theta}, \ldots, \dot{r}_n\dot{\theta})$$

Definition 4.4.18: A **nominal unification problem** $\dot{\mathcal{P}}$ is a finite multiset of freshnesses and equalities. A **solution** to a unification problem is a pair $(\Delta, \dot{\theta})$ such that $\Delta \vdash \dot{a}\#\dot{r}\dot{\theta}$ for every $\dot{a}\#\dot{r} \in \dot{\mathcal{P}}$ and $\Delta \vdash \dot{r}\dot{\theta} = \dot{s}\dot{\theta}$ for every $\dot{r} = \dot{s} \in \dot{\mathcal{P}}$.

Definition 4.4.19: Extend the interpretation of Definition 4.4.12 to solutions of nominal unification problems by:

$$[\![(\dot{\Delta}, \dot{\theta})]\!](X^S) \equiv [\![\dot{\theta}(\dot{X})]\!]_\Delta \quad \text{if} \quad id\cdot X^S \equiv [\![\dot{X}]\!]_\Delta$$
$$\text{and}$$
$$[\![(\dot{\Delta}, \dot{\theta})]\!](Y^T) \equiv id\cdot Y^T \quad \text{otherwise}$$

Lemma 4.4.20 states that the interpretation of nominal unification solutions in Definition 4.4.19 commutes with the interpretation of nominal terms given in Definition 4.4.12.

Lemma 4.4.20: $[\![\dot{r}]\!]_\Delta[\![(\Delta, \dot{\theta})]\!] \equiv [\![\dot{r}\dot{\theta}]\!]_\Delta$

Proof. By induction on $\dot{r}$.

- The case $\dot{a}$. By Definition 4.4.17 we have $[\![\dot{a}\dot{\theta}]\!]_\Delta \equiv [\![\dot{a}]\!]_\Delta$. By Definition 4.4.12 we have $[\![\dot{a}]\!]_\Delta = \iota(\dot{a})$. By Definition 4.2.29 we have $\iota(\dot{a}) \equiv \iota(\dot{a})[\![(\Delta, \dot{\theta})]\!]$. By Definition 4.4.12 we have $\iota(\dot{a})[\![(\Delta, \dot{\theta})]\!] \equiv [\![\dot{a}]\!]_\Delta[\![(\Delta, \dot{\theta})]\!]$. The result follows.

- The case $\dot\pi\cdot\dot X$. By Definition 4.4.12 we have $[\![\dot\pi\cdot\dot X]\!]_\Delta[\![(\Delta,\dot\theta)]\!] \equiv ([\![\dot\pi]\!]\cdot X^S)[\![(\Delta,\dot\theta)]\!]$ where $S = \mathit{comb}\setminus\{\iota(\dot a) \mid \dot a\#\dot X \in \Delta\}$. By Definition 4.2.29 we have $([\![\dot\pi]\!]\cdot X^S)[\![(\Delta,\dot\theta)]\!] \equiv [\![\dot\pi]\!]\cdot [\![(\Delta,\dot\theta)]\!](X^S)$. By Definition 4.4.19 we have $[\![\dot\pi]\!]\cdot[\![(\Delta,\dot\theta)]\!](X^S) \equiv [\![\dot\pi]\!]\cdot[\![\dot\theta(\dot X)]\!]_\Delta$. By Definition 4.4.12 we have $[\![\dot\pi]\!]\cdot[\![\dot\theta(\dot X)]\!]_\Delta \equiv [\![\dot\pi\cdot\dot\theta(\dot X)]\!]_\Delta$. By Definition 4.4.17 we have $[\![\dot\pi\cdot\dot\theta(\dot X)]\!]_\Delta \equiv [\![(\dot\pi\cdot\dot X)\dot\theta]\!]_\Delta$. The result follows.
- The case $[\dot a]\dot r$. By Definition 4.4.17 we have $[\![([\dot a]\dot r)\dot\theta]\!]_\Delta \equiv [\![[\dot a](\dot r\dot\theta)]\!]_\Delta$. By Definition 4.4.12 we have $[\![[\dot a](\dot r\dot\theta)]\!]_\Delta \equiv [\iota(\dot a)][\![\dot r\dot\theta]\!]_\Delta$. By inductive hypothesis $[\iota(\dot a)][\![\dot r\dot\theta]\!]_\Delta \equiv [\iota(\dot a)]([\![\dot r]\!]_\Delta[\![(\Delta,\dot\theta)]\!])$. By Definition 4.2.29 we have:

  $$[\iota(\dot a)]([\![\dot r]\!]_\Delta[\![(\Delta,\dot\theta)]\!]) \equiv ([\iota(\dot a)][\![\dot r]\!]_\Delta)[\![(\Delta,\dot\theta)]\!]$$

  By Definition 4.4.12 we have $([\iota(\dot a)][\![\dot r]\!]_\Delta)[\![(\Delta,\dot\theta)]\!] \equiv [\![[\dot a]\dot r]\!]_\Delta[\![(\Delta,\dot\theta)]\!]$. The result follows.
- The case $\mathsf{f}(\dot r_1,\dots,\dot r_n)$. By Definition 4.4.17 we have $[\![\mathsf{f}(\dot r_1,\dots,\dot r_n)\dot\theta]\!]_\Delta \equiv [\![\mathsf{f}(\dot r_1\dot\theta,\dots,\dot r_n\dot\theta)]\!]_\Delta$. By Definition 4.4.12 we have

  $$[\![\mathsf{f}(\dot r_1\dot\theta,\dots,\dot r_n\dot\theta)]\!]_\Delta \equiv \mathsf{f}([\![\dot r_1\dot\theta]\!]_\Delta,\dots,[\![\dot r_1\dot\theta]\!]_\Delta)$$

  and by inductive hypotheses:

  $$\mathsf{f}([\![\dot r_1\dot\theta]\!]_\Delta,\dots,[\![\dot r_1\dot\theta]\!]_\Delta) \equiv \mathsf{f}([\![\dot r_1]\!]_\Delta[\![(\Delta,\dot\theta)]\!],\dots,[\![\dot r_1]\!]_\Delta[\![(\Delta,\dot\theta)]\!])$$

  By Definition 4.2.29 we have

  $$\mathsf{f}([\![\dot r_1]\!]_\Delta[\![(\Delta,\dot\theta)]\!],\dots,[\![\dot r_1]\!]_\Delta[\![(\Delta,\dot\theta)]\!]) \equiv \mathsf{f}([\![\dot r_1]\!]_\Delta,\dots,[\![\dot r_1]\!]_\Delta)[\![(\Delta,\dot\theta)]\!]$$

  and by Definition 4.4.12 we have:

  $$\mathsf{f}([\![\dot r_1]\!]_\Delta,\dots,[\![\dot r_1]\!]_\Delta)[\![(\Delta,\dot\theta)]\!] \equiv [\![\mathsf{f}(\dot r_1,\dots,\dot r_1)]\!]_\Delta[\![(\Delta,\dot\theta)]\!]$$

  The result follows.

⊠

Definition 4.4.21: Define $[\![\dot{\mathcal{P}}]\!]_\Delta$ by mapping $\dot r = \dot s$ to $[\![\dot r]\!]_\Delta \overset{?}{=} [\![\dot s]\!]_\Delta$ and mapping $\dot a\#\dot r$ to $(b\ \iota(\dot a))\cdot[\![\dot r]\!]_\Delta \overset{?}{=} [\![\dot r]\!]_\Delta$. Here b is chosen fresh (i.e. $b \notin fa([\![\dot r]\!]_\Delta)$).

Lemma 4.4.22 is a technical result used in the proof of Lemma 4.4.23.

Lemma 4.4.22: If $a, b \notin \pi\cdot S$ then $((b\ a)\circ\pi)|_S = \pi|_S$.

Proof. By case analysis:

- The cases a and b. We consider only the first case, as the second is similar. Suppose $a \in S$ then $a \in \mathit{nontriv}(\pi)$ and $b \neq \pi(a)$. By Definition 4.2.12 we have $((b\ a)\circ\pi)|_S(a) = ((b\ a)\circ\pi)(a)$. By definition we have $((b\ a)\circ\pi)(a) = (b\ a)(\pi(a))$. By assumption $(b\ a)(\pi(a)) = \pi(a)$. By Definition 4.2.12 we have $\pi(a) = \pi|_S(a)$. The result follows.
  Otherwise, suppose $a \notin S$. By Definition 4.2.12 we have that both permutations are undefined. The result follows.
- The case c. Suppose $c \in S$. By Definition 4.2.12 we have $((b\ a)\circ\pi)|_S(c) = ((b\ a)\circ\pi)(c)$. By definition we have $((b\ a)\circ\pi)(c) = (b\ a)(\pi(c))$. There are three cases:

- The case $a = \pi(c)$ or $b = \pi(c)$. We handle only the first case, as the second is similar. By assumption we have $a \notin \pi{\cdot}S$ and $c \in S$. There is nothing to prove.
- The case $a \neq \pi(c)$ and $b \neq \pi(c)$. By definition we have $(b\ a)(\pi(c)) = \pi(c)$. By Definition 4.2.12 we have $\pi(c) = \pi|_S(c)$. The result follows.

Otherwise, suppose $c \notin S$. By Definition 4.2.12 we have that both permutations are undefined. The result follows.

⊠

Lemma 4.4.23: If $b \notin fa(r)$ then $a \notin fa(r)$ if and only if $(b\ a){\cdot}r =_\alpha r$.

Proof. We handle the two implications separately:

- The left to right case. By induction on r.
  - The case c. By Definition 4.2.9 we have $(b\ a){\cdot}c \equiv c$. Using $(=_\alpha\mathbf{a})$ we obtain $c =_\alpha c$. The result follows.
  - The case $\pi{\cdot}X^S$. Suppose $a \notin fa(\pi{\cdot}X^S)$ and $b \notin fa(\pi{\cdot}X^S)$. By Definition 4.2.11 we have $a \notin \pi{\cdot}p(X)$ and $b \notin \pi{\cdot}p(X)$. By Definition 4.2.9 we have $(b\ a){\cdot}(\pi{\cdot}X^S) \equiv ((b\ a)\circ\pi){\cdot}X^S$. By Lemma 4.4.22 we have $((b\ a)\circ\pi)|_S = \pi|_S$. Using $(=_\alpha\mathbf{X})$ we obtain $((b\ a)\circ\pi){\cdot}X^S =_\alpha \pi{\cdot}X^S$. By Definition 4.2.9 we have $((b\ a)\circ\pi){\cdot}X^S \equiv (b\ a){\cdot}(\pi{\cdot}X)$. The result follows.
  - The cases $[a]r$ and $[b]s$. We handle only the first case, as the second is similar. Suppose $a, b \notin fa(r)$. By inductive hypothesis we have $(b\ a){\cdot}r =_\alpha r$. Using $(=_\alpha[\mathbf{b}])$ we obtain $[b]((b\ a){\cdot}r) =_\alpha [a]r$. By Definition 4.2.9 we have $[b]((b\ a){\cdot}r) \equiv (b\ a){\cdot}[a]r$. The result follows.
  - The case $[c]t$. Suppose $b \notin fa([c]t)$ and $a \notin fa([c]t)$. By Definition 4.2.11 we have $a \notin fa(t)\setminus\{c\}$ and $b \notin fa(t)\setminus\{c\}$. By inductive hypothesis $(b\ a){\cdot}t =_\alpha t$. Using $(=_\alpha[\mathbf{a}])$ we obtain $[c]((b\ a){\cdot}t) =_\alpha [c]t$. By Definition 4.2.9 we have $[c]((b\ a){\cdot}t) \equiv (b\ a){\cdot}[c]t$. The result follows.
  - The case $\mathsf{f}(r_1, \ldots, r_n)$. Suppose $b \notin fa(\mathsf{f}(r_1, \ldots, r_n))$ and $a \notin fa(\mathsf{f}(r_1, \ldots, r_n))$. By Definition 4.2.11 we have $b \notin \bigcup_{1\leq i\leq n} fa(r_i)$ and $a \notin \bigcup_{1\leq i\leq n} fa(r_i)$. By inductive hypotheses $(b\ a){\cdot}r_i =_\alpha r_i$ for $1 \leq i \leq n$. Using $(=_\alpha\mathsf{f})$ we obtain $\mathsf{f}((b\ a){\cdot}r_1, \ldots, (b\ a){\cdot}r_n) =_\alpha \mathsf{f}(r_1, \ldots, r_n)$. By Definition 4.2.9 we have $\mathsf{f}((b\ a){\cdot}r_1, \ldots, (b\ a){\cdot}r_n) \equiv (b\ a){\cdot}\mathsf{f}(r_1, \ldots, r_n)$. The result follows.
- The right to left case. By induction on r.
  - The case c. By Definition 4.2.9 we have $(b\ a){\cdot}c \equiv c$. The result follows.
  - The case $\pi{\cdot}X^S$. Suppose $b \notin fa(\pi{\cdot}X^S)$. By Definition 4.2.11 we have $b \notin \pi{\cdot}S$. By Definition 4.2.9 we have $(b\ a){\cdot}(\pi{\cdot}X^S) =_\alpha ((b\ a)\circ\pi){\cdot}X^S$. Suppose $((b\ a)\circ\pi){\cdot}X^S =_\alpha \pi{\cdot}X^S$. Using $(=_\alpha\mathbf{X})$ we obtain $((b\ a)\circ\pi){\cdot}X^S =_\alpha \pi{\cdot}X^S$ whenever $((b\ a)\circ\pi)|_S = \pi|_S$. Suppose $a \in \pi{\cdot}S$ therefore $\pi^{-1}(a) \in S$. By Definition 4.2.12 we have $\pi|_S(\pi^{-1}(a)) = \pi(\pi^{-1}(a)) = a$. By Definition 4.2.12 we have $((b\ a)\circ\pi)|_S(\pi^{-1}(a)) = (b\ a)(\pi(\pi^{-1}(a))) = b$. Therefore if $a \in \pi{\cdot}S$ we have $((b\ a)\circ\pi)|_S \neq \pi|_S$, a contradiction. By Definition 4.2.11 we have $a \notin fa(\pi{\cdot}X^S)$. The result follows.
  - The case $[a]r$. By Definition 4.2.11 we have $a \notin fa([a]r)$. The result follows.

- The case $[b]s$. By Definition 4.2.9 we have $(b\ a){\cdot}[b]s \equiv [a]((b\ a){\cdot}s)$. Using $(=_\alpha[\mathbf{b}])$ we obtain $[b]s \equiv [a]((b\ a){\cdot}s)$ whenever $a \notin fa(s)$. By Definition 4.2.11 we have $[b]s \equiv [a]((b\ a){\cdot}s)$ whenever $a \notin fa([b]s)$. The result follows.
- The case $[c]t$. Suppose $b \notin fa([c]t)$. By Definition 4.2.11 we have $b \notin fa(t)$. By Definition 4.2.9 we have $(b\ a){\cdot}[c]t \equiv [c]((b\ a){\cdot}t)$. By inductive hypothesis $(b\ a){\cdot}t =_\alpha t$ implies $a \notin fa(t)$. By Definition 4.2.11 we have $(b\ a){\cdot}t =_\alpha t$ implies $a \notin fa([c]t)$. Using $(=_\alpha[\mathbf{a}])$ we obtain $[c]((b\ a){\cdot}t) =_\alpha [c]t$. By Definition 4.2.9 we have $[c]((b\ a){\cdot}t) \equiv (b\ a){\cdot}[c]t$. The result follows.
- The case $(=_\alpha \mathsf{f})$. Suppose $b \notin fa(\mathsf{f}(r_1,\dots,r_n))$. By Definition 4.2.11 we have $b \notin fa(r_i)$ for $1 \leq i \leq n$. By Definition 4.2.9 we have $(b\ a){\cdot}\mathsf{f}(r_1,\dots,r_n) \equiv \mathsf{f}((b\ a){\cdot}r_1,\dots,(b\ a){\cdot}r_n)$. By inductive hypotheses $(b\ a){\cdot}r_i =_\alpha r_i$ implies $a \notin fa(r_i)$ for $1 \leq i \leq n$. By Definition 4.2.11 and $(=_\alpha\mathsf{f})$ we obtain $\mathsf{f}((b\ a){\cdot}r_1,\dots,(b\ a){\cdot}r_n) =_\alpha \mathsf{f}(r_1,\dots,r_n)$ implies $a \notin fa(\mathsf{f}(r_1,\dots,r_n))$. By Definition 4.2.9 we have $(b\ a){\cdot}\mathsf{f}(r_1,\dots,r_n) =_\alpha \mathsf{f}(r_1,\dots,r_n)$ implies $a \notin fa(\mathsf{f}(r_1,\dots,r_n))$. The result follows.

⊠

Theorem 4.4.24 is the main result in Section 4.4. Intuitively, it states that the translation of Definition 4.4.12 and Definition 4.4.19 are sufficient to preserve and reflect nominal unification solutions. That is, if there's a solution to a unification problem using one form of nominal term, then the translation of that solution is a solution of the translated unification problem.

Theorem 4.4.24: $(\Delta, \dot\theta)$ solves $\dot{\mathcal{P}}$ if and only if $[\![(\Delta,\dot\theta)]\!]$ solves $[\![\dot{\mathcal{P}}]\!]_\Delta$.

Proof. We handle the two implications separately:

- The left to right case. Suppose $\Delta \vdash \dot r\dot\theta \approx \dot s\dot\theta$. By Lemma 4.4.20 and Theorem 4.4.15 we have $[\![\dot r]\!]_\Delta[\![(\Delta,\dot\theta)]\!] =_\alpha [\![\dot s]\!]_\Delta[\![(\Delta,\dot\theta)]\!]$.
  Suppose $\Delta \vdash \dot a\#\dot r\dot\theta$. By Lemma 4.4.14 we have $\iota(\dot a) \notin fa([\![\dot r\dot\theta]\!]_\Delta)$. By Lemma 4.4.20 we have $\iota(\dot a) \notin fa([\![\dot r]\!]_\Delta[\![(\Delta,\dot\theta)]\!])$. By Lemma 4.4.23 we have $(b\ \iota(\dot a)){\cdot}[\![\dot r]\!]_\Delta[\![(\Delta,\dot\theta)]\!] =_\alpha [\![\dot r]\!]_\Delta[\![(\Delta,\dot\theta)]\!]$ where b is fresh. By Lemma 4.2.31 we have $((b\ \iota(\dot a)){\cdot}[\![\dot r]\!]_\Delta)[\![(\Delta,\dot\theta)]\!]$. The result follows.
- The right to left case. Suppose $[\![\dot r]\!]_\Delta[\![(\Delta,\dot\theta)]\!] =_\alpha [\![\dot s]\!]_\Delta[\![(\Delta,\dot\theta)]\!]$. By Theorem 4.4.15 we have $\Delta \vdash \dot r\dot\theta \approx \dot s\dot\theta$.
  Suppose $((b\ \iota(\dot a)){\cdot}[\![\dot r]\!]_\Delta)[\![(\Delta,\dot\theta)]\!]$. By Lemma 4.2.31 we have $(b\ \iota(\dot a)){\cdot}[\![\dot r]\!]_\Delta[\![(\Delta,\dot\theta)]\!] =_\alpha [\![\dot r]\!]_\Delta[\![(\Delta,\dot\theta)]\!]$. By Lemma 4.4.23 we have $\iota(\dot a) \notin fa([\![\dot r]\!]_\Delta[\![(\Delta,\dot\theta)]\!])$. By Lemma 4.4.20 we have $\iota(\dot a) \notin fa([\![\dot r\dot\theta]\!]_\Delta)$. By Lemma 4.4.14 we have $\Delta \vdash \dot a\#\dot r\dot\theta$. The result follows.

⊠

4.5 Conclusions

This Chapter has introduced a new form of nominal term, *permissive nominal terms*. We view Corollaries 4.2.17 and 4.2.18 (the 'always fresh' and 'always rename' properties) as significant improvements over nominal terms, and it is our opinion that these two properties make working with permissive nominal terms more attractive than working with UPG nominal terms. For instance, permissive nominal terms can be quotiented by syntax; nominal terms cannot. We view this as a real improvement on the state-of-the-art. Another improvement, the simplification of permissive nominal unification solutions, compared to UPG nominal unification solutions, also brings permissive nominal unification into line with existing unification algorithms.

Ongoing research is already using permissive nominal terms. For instance, Gabbay and Mulligan [GM09e] used permissive nominal techniques relating lambda-algebras and (permissive) nominal algebra. Similarly, Dowek and Gabbay [DGM10, second half of paper] used permissive nominal terms to relate the solutions of higher-order pattern unification problems and permissive nominal unification problems. Finally, Dowek and Gabbay introduced Permissive Nominal Logic [DG10], a foundational logic intended for specifying other logics and calculi, based on first-order logic with permissive nominal terms as the term language. We believe in all cases that the use of permissive nominal techniques vastly simplified the work.

Although it may appear that the use of an infinite and coinfinite sorting system a bar to implementation, Chapter 5 will present an implementation of the permissive nominal unification algorithm in the functional language Haskell, proving that permission sorts are no barrier to implementation (and in fact, the unification algorithm presented in this Chapter is straightforward to implement). The reader should note here that implementation in a lazy language such as Haskell is not necessary, and the work presented in Chapter 5 could well have been implemented in Standard ML, or another strict language. The relationship between an existing implementation of nominal techniques, Calvès' HNT [Cal09], and the permissive implementation, will also be discussed in Chapter 5.

CHAPTER 5

Implementing permissive nominal terms

Abstract

Permissive nominal terms label nominal unknowns with infinite and coinfinite sets of atoms called a permission sort. This Chapter provides a constructive ‘proof’ that these infinite and coinfinite sets do not preclude a straightforward implementation of permissive nominal terms, nor their unification algorithm. We provide this ‘proof’ by way of a prototype implementation of permissive nominal terms and their unification algorithm, which we call PNT, in the programming language Haskell. Moreover, we do not depend on the lazy evaluation semantics of Haskell to represent permission sorts; all data structures in the implementation admit a finite representation.

For testing and debugging purposes we include the PNT Frontend that accepts a small domain specific language for specifying permission sorts, terms and unification problems for solving. We conclude the Chapter with a demonstration of PNT in action, unifying two permissive nominal terms via the PNT Frontend.

5.1 Introduction

Chapter 4 introduced permissive nominal terms. Permissive nominal terms eliminate explicit freshness contexts. Instead unknowns X are labeled with a permission sort, and the infinite and coinfinite form of these permission sorts allows us to recover the ‘always fresh’ and ‘always rename’ properties that nominal terms lack.

Reasons for implementing permissive nominal terms are twofold.

First, this Chapter proves that the infinite and coinfinite nature of permission sorts is no barrier to a straightforward implementation. We do this by providing a ‘constructive proof’, by way of a prototype implementation in Haskell, called PNT, of permissive nominal terms and their unification algorithm.

Despite being infinite in size, permission sorts are uniquely characterised by two finite sets of atoms, and we use this fact to implement permissive nominal terms in a straightforward manner.

Second, we believe that implementing computational mathematics, in of itself, is a good idea. An implementation provides a concrete framework for exploring permissive nominal terms as an idea that complements the abstract mathematical description in Chapter 4.

We implement PNT in Haskell [PJ02]. Haskell is a lazy pure functional programming language, but we do not rely on Haskell’s purity, nor its laziness, to implement permissive nominal terms; we could easily have obtained the same results in an eager, impure language like Standard ML [MTHM97] or Java [GJSB05].

PNT is provided as a Haskell library, i.e. a collection of independent modules, which can in principle be used in other applications that require a notion of ‘term with binding’. PNT provides the following features:

- An abstract data type implementing permutations, including functions for computing difference sets between two permutations, and composing and inverting permutations (Subsection 5.2.2).
- An abstract data type implementing permission sorts, including functions for computing intersections and unions of permission sorts, for adding and subtracting atoms from permission sorts, and testing equality (Subsection 5.2.3).
- An abstract data type implementing permissive nominal terms, including an α-equivalence test, permutation and substitution actions, functions for the calculation of free atoms, free variables and calculating a fresh atom for a term (Subsection 5.2.5).
- A module implementing the permissive nominal unification algorithm, including an implementation of the support-reduction algorithm (Definition 4.3.8), and unification algorithm (Definition 4.3.39) proper (Section 5.3).

In addition, we also provide the PNT Frontend—a mini language—which allows a user to easily define permission sorts and terms, and amongst other operations, unify two terms and test for α-equivalence. The PNT Frontend operates from the command line. The design of the PNT Frontend, and a sample user interaction, is briefly described in Section 5.4. The complete BNF grammar of the language accepted by the PNT Frontend is available in Chapter C.

The PNT implementation is a prototype, and not a 'full' implementation of (permissive) nominal terms. We draw a distinction between the work presented in this Chapter, and the work of Calvès on his implementation of the Haskell Nominal Toolkit (HNT) [Cal09]. Aside from the obvious difference between HNT and PNT—the difference between standard nominal terms and permissive nominal terms described in the introductory sections of Chapter 4—the most striking difference between the two implementations is the choice of unification algorithm implemented. In particular, Calvès and Fernández have expended significant research effort investigating efficient (polynomial space and time) nominal unification and matching algorithms. This is in contrast with the exponential asymptotic running time of the naïve, recursive descent, permissive unification algorithm.[1] The HNT implements efficient unification and matching algorithms, whereas PNT implements the less efficient recursive descent algorithm.

We see the difference in implemented unification algorithms as a difference in research focus. The HNT was part of a wider research context, with the aim of finding the most efficient nominal unification algorithm possible. In contrast, the goals of PNT are more humble: we aim to show that permissive nominal terms can be implemented in a straightforward manner, as well as providing a concrete implementation of permissive nominal terms that people can use to familiarise themselves with permissive nominal technology.

The HNT also goes further than the prototype permissive implementation in implementing a nominal rewriting algorithm. As a direct consequence, functions and ancillary data structures for navigating efficiently through terms (i.e. an implementation of Huet's zipper [Hue97]) are also provided.

[1]The permissive nominal unification algorithm of Chapter 4 has exponential worst case running time. This is for the same reasons as for the naïve recursive descent first-order unification algorithm. Namely, consider trying to solve the following unification problem: $\mathsf{f}(X_1, X_1) \overset{?}{=} X_2, \mathsf{f}(X_2, X_2) \overset{?}{=} X_3, \mathsf{f}(X_3, X_3) \overset{?}{=} X_4, \ldots$.

In contrast, PNT implements no rewriting algorithms. Again, this represents a difference of research focus between the two implementations. Rewriting algorithms can easily be implemented on top of the PNT, and the modular design of the implementation should make this straightforward. In addition, the recursive descent permissive unification algorithm could also be replaced with minimal changes to the rest of the code base, with the more efficient Calvès-Fernández algorithm.

In particular, the modular design of the implementation is aided through extensive use of Haskell's type classes. This makes it easy to write generic functions that assume as little as possible about the particular implementation of the data types which they work on. In addition, we make use of view patterns [Wad87], an extension to the Haskell 98 provided by the Glasgow Haskell Compiler [Tea10]. These allow us to keep abstract data types truly abstract, thus preserving important invariants, whilst still retaining the ability to pattern match against the data type's constructors.

All source code for this Chapter is available online, at the following address: `http://www2.macs.hw.ac.uk/~dpm8/permissive`. Sections within this Chapter, describing particular functionality of the PNT Frontend, make reference to directories in the source code distribution, where the relevant Haskell code can be found. A web-based version of the PNT Frontend is also available online at the same location. An example script is preloaded: clicking the button marked *Solve obligations* will process the input.

The work in this Chapter is individual work and does not currently appear in published form.

5.1.1 Conventions

We typeset Haskell source mostly 'as is'. However, to improve the typography of Haskell excerpts, and also for reasons of legibility, we make a few changes. These are summarised as follows:

Convention	Description	Standard Haskell
$\circ$	Function composition	`.`
λ	Anonymous function abstraction	`\`
$\wedge$	Boolean conjunction	`&&`
$\vee$	Boolean disjunction	`||`
$==$	Boolean equality	`==`
$\neq$	Boolean inequality	`/=`
$+\!\!+$	List append	`++`

Type variables in function type ascriptions and data type declarations are denoted by Greek lower case letters, α, β, γ, and so on, as opposed to `a`, `b`, `c`, as is standard in Haskell. When referring to Haskell functions or data types in the narrative, we use an *italicised* font face (*map*, *foldr*, and so on).

5.2 Building terms

In this Section, we provide a brief overview of the design decisions taken whilst implementing the permissive nominal terms subsystem.

5.2.1 Atoms

The excerpts described in this Subsection can be found in the file `Terms/Permissive.hs`.

Atoms are one of the fundamental building blocks of nominal terms. Formally, we assume that atoms have no implicit structure; we may test for equality and inequality between two atoms, and that is it. Fortunately, Haskell has an elegant way of capturing this idea: the *Eq* type class; types that are instances of *Eq* have equality and inequality tests defined.

Ideally what we would like is a single type class that captures *all* the reasoning that one may need to perform on atoms. We can then write generic unification functions that operate over elements of this type class.

However, consideration must be given to the role of permission sorts in permissive nominal terms. In Definition 4.2.2 we fixed a single permission sort called *comb*. All other permission sorts are taken to vary finitely from this permission sort. Similarly, in PNT, we also assume a single, fixed permission sort *comb*, and we find it convenient to assume a function *inComb* that returns a Boolean value, stating whether an atom lies inside *comb* or not.

In addition, we'd also like to be able to generate 'fresh' atoms based on a list of previously used atoms, with the obvious property that the generated fresh atom must not appear in the input list.

```
class Eq α ⇒ Permissive α
  where
    inComb :: α → Bool
    fresh  :: [α] → a
```

Permissive is the most fundamental type class used in the implementation. Nearly all functions operating on terms, and many operations in the unification algorithm, make frequent use of it.

One possible weakness in the design of the type class is the coupling of two seemingly separate concepts. That is, *Permissive* combines operations that arise naturally on atoms—equality and inequality tests—with operations that seem to deal with permission sorts. Would it not be a better idea to factor *Permissive* into two type classes: keeping atoms as elements of *Eq*, and introducing a new *Sort* type class that does not extend *Eq*?

We don't see this as a weakness of our design for two reasons. Firstly, we have a single fixed permission sort, from which all others differ finitely; the case for admitting multiple fixed permission sorts (i.e. $comb_i$ for $i = 1, 2, \ldots, \omega$) has not been made. For this reason, testing whether an atom is a member of this fixed permission sort seems to be just as much a property of the atoms themselves, as a property of the permission sort. Secondly, the use of a single *Permissive* type class seems neater.

For the sake of the PNT Frontend, and to allow us to test our development, we provide a single instance of *Permissive*, for the Haskell *Int* type. Note that any user of the library is entirely free to use their own implementation of atoms as long as it satisfies the constraints of the *Permissive* type class. Though we provide an implementation entirely in terms of integers, the unification algorithm is ambivalent about the underlying representation of atoms, and this representation can be changed by the user as needed.

We assume that odd integers are outside *comb*, whilst even integers lie within it. We assume that only positive integers are used. This allows us to provide particularly simple functions that satisfy *Permissive* for the *Int* type.

Generating a fresh *Int* is as simple as adding all the previously used integers together. Depending on the value of this addition, we then add a constant—1, if the result is even, 2 if the result is odd—to obtain a final *Int* that lies outside *comb*.

5.2.2 Permutations

The excerpts described in this Subsection can be found in the file `Terms/Permutations.hs`.

Permutations are the means through which α-equivalence in permissive nominal terms are handled. They hold a prominent place in nominal techniques. We choose to implement a *Permutation* data type with a simple representation—lists of pairs of atoms:

```
data Permutation α = Permutation [(α, α)]
```

Each pair (a, b) represents a swapping of b with a with b, and fixing on all other atoms c. This choice of representation is a standard construction. For instance, it was taken as the definition of a permutation in the paper introducing nominal unification [UPG04]. It has the attractive property that composing two permutations is as straightforward as appending their underlying association lists. Similarly, inverting a permutation is as straightforward as reversing its underlying association list.

There is, however, one subtlety. In several places in the unification algorithm (notably ($\sqsubseteq\mathbf{X}$) in Definition 4.3.8 we need to test whether a particular permutation is the identity. Unfortunately, representing permutations as lists means that there is no single ‘canonical’ representation of the identity permutation.

For example, the empty list [ ] may serve as the identity: it acts on no atoms, therefore every atom is fixed by it. However, the lists $[(b, a), (a, b)]$ and $[(c, a), (a, c)]$ also serve as the identity permutation. It is a rather obvious fact that the identity permutation, under this representation scheme, has an infinity of different representations!

To solve this problem we define an auxiliary function *atomsIn*. This can be thought of collecting all the atoms that appear anywhere within the *Permutation*'s underlying lists into another list. These atoms are exactly the atoms that possibly could be mapped to some other atom by the permutation. *atomsIn* is defined in a straightforward manner, as follows:

```
atomsIn :: Permissive α ⇒ Permutation α → [α]
atomsIn (Permutation [ ]) = [ ]
atomsIn (Permutation ((a, b):t)) = nub $ [a, b] ++ atomsIn (Permutation t)
```

Here *nub* is a Haskell Prelude function that removes duplicated items from a list. Using *atomsIn*, we define a test for permutations, *isIdentity*:

```
isIdentity :: Permissive α ⇒ Permutation α → Bool
isIdentity (Permutation [ ]) = True
isIdentity prm = map (act prm) (atomsIn prm) == atomsIn prm
```

Here *act* is a function that applies a permutation to an atom. *isIdentity* first computes all the candidate atoms that could be mapped non-trivially by the input *Permutation*, using *atomsIn*. It then applies, successively, the *Permutation* to each candidate. If we get the same list of atoms back, after mapping the *Permutation*, then the *Permutation* does indeed represent the identity.

Clearly, *isIdentity* has poor computational properties as it traverses lists representing permutations multiple times, both in building the lists of candidate atoms, and then mapping the *Permutation* across the candidates. If the prototype is expanded into a full implementation of permissive nominal terms, a more effective representation of permutations, ideally facilitating a canonical representation, would improve efficiency matters. However, the identity test performs satisfactorily for our purposes.

5.2.3 Permission sorts

The excerpts described in this Subsection can be found in the file `Terms/Sorts.hs`.

In Chapter 4 we characterise permission sorts as sets of atoms of the form $comb \cup A \setminus B$. Here *comb* is a single fixed coinfinite and infinite set of atoms, whilst A and B are both finite sets of atoms.

However, what really matters are the sets A and B, which uniquely determine any particular permission sort (as *comb* is fixed). Conveniently, A and B are also assumed to be finite, and hence by recording A and B we have a way of uniquely and finitely identifying any permission sort.

There is, however, one subtlety.

A basic operation on terms is the calculation of free atoms. Free atoms calculations underpin α-equivalence and unification. Now consider the following two clauses, taken from Definition 4.2.11:

$$fa(a) = \{a\} \qquad fa(\pi \cdot X^S) = \pi \cdot S$$

The free atoms of an atom is a singleton set. However, the free atoms of an unknown is an infinite and coinfinite set—a permission sort.

Definition 4.2.11 seems to demand that free atom calculations on terms returns a permission sort as the result. This is convenient, as we can take the union of the free atoms of terms by unioning permission sorts, and so on. However, this slightly complicates the design of our data type representing permission sorts, as we must also take into account finite sets of atoms. This propagates through the implementation of all functions we wish to define on permission sorts:

union, intersection, and so on. Our *Sort* data type is therefore a union type with constructors for finite sets of atoms and infinite permission sorts:

```
data Sort α = Finite [α]
            | Infinite [α] [α]
```

The *Infinite* constructor carries two lists. The first corresponds to atoms added to the permission sort that lie outside *comb*. The second corresponds to atoms removed from the permission sort, that lie outside *comb*. We maintain the invariant that only atoms outside of *comb* are members of the first list, and only atoms inside *comb* are members of the second list. Using this representation, *comb* itself has a particularly simple definition:

```
comb :: Permissive α ⇒ Sort α
comb = Infinite [ ] [ ]
```

That is, a permission sort where no atoms are added nor any atoms are removed from the implicit fixed sort comb.

5.2.3.1 Operations on permission sorts

Sort is our data type representing permission sorts. We'd like to use *Sort* to write functions that compute the union of two sorts; intersection of two sorts; remove atoms from sorts; compute whether one sort is a subset of another; permute sorts with an *Permutation.* Fortunately, most of these operations are entirely straightforward to implement, for instance, calculating whether one permission sort is a subset of another:

```
subset :: Permissive α ⇒ Sort α → Sort α → Bool
subset (Finite [ ]) _ = True
subset (Finite (h:t)) (Finite fsrt) =
  elem h fsrt ∧ subset (Finite t) (Finite fsrt)
subset (Finite (h:t)) (Infinite add sub)
  | inComb h =
  if elem h sub then
    False
  else
    subset (Finite t) (Infinite add sub)
  | elem h add = subset (Finite t) (Infinite add sub)
  | otherwise = False
subset (Infinite add sub) (Infinite add′ sub′) =
  let
    subset′ :: Eq α ⇒ [α] → [α] → Bool
    subset′ [ ] _ = True
    subset′ (h:t) lst = elem h lst ∧ subset′ t lst
  in
    subset′ add add′ ∧ subset′ sub′ sub
subset i@(Infinite _ _) f@(Finite _) = False
```

Here *elem* is a Haskell Prelude function that tests an element for membership of a set. The cases for finite *Sorts* are straightforward: the empty set is a subset of all other permission sorts, and for non-empty finite sorts, a finite set is a subset of another if all its members also belong to the other set.

Similarly, comparing a finite set of atoms with an infinite permission sort is easy. For each atom in the finite set, if the atom is in *comb* we need only check that we haven't removed that atom from the permission sort. If the atom isn't in *comb*, then we check that we have added it to the permission sort.

The only complicated test is between two infinite sorts. Suppose we have two *Sorts* representing the following permission sorts: $S = comb \cup A \setminus B$ and $S' = comb \cup A' \setminus B'$. To check whether $S \subseteq S'$, we must check that all elements of A are also elements of A' and all elements of B' are elements of B.

The remaining functions operating on permission sorts are also as straightforward. For instance, removing an atom from a *Sort* is just list manipulation, checking whether the atom to remove lies inside *comb* or not:

```
remove' :: Permissive α ⇒ Sort α → α → Sort α
remove' (Finite fsrt) elem =
  Finite $ delete elem fsrt
remove' (Infinite add sub) elem =
   if inComb elem then then
     Infinite add $ nub $ elem:sub
   else
     (Infinite $ delete elem add) sub
remove :: Permissive α ⇒ [α] → Sort α → Sort α
remove atms srt =
  foldl remove' srt atms
```

Here *delete* is a Haskell Prelude function that deletes an element from a list. The Prelude function *foldl* performs a left-fold over a list. We provide two functions for removing atoms from *Sorts*: *remove'* removes single atoms, whilst *remove* removes multiple atoms (in the form of a list) at once.

However, there is one remarkable exception to the ease through which *Sort* permits straightforward implementations of operations on permission sorts—the pointwise action of a *Permutation* on a *Sort*. Whilst we can define this operation in a single line, mathematically (Definition 4.2.10), implementing *pointwise* in Haskell leads to a 100-line function involving dozens of case splits.

To understand why Definition 4.2.10 blows up like it does, consider the following. Suppose you have a single swapping $\pi = (b\ a)$. There are multiple ways that, under our *Sort* representation, atoms can be acted on:

- π acts on c, in which case nothing changes.
- π acts on b, b lies outside *comb*, but a is inside *comb*, so the *Sort*'s underlying lists have to be modified, swapping a and b in to and out of the *Sort*.

- π acts on b, b lies inside *comb*, but a is also inside *comb*, in which case nothing changes.
- π acts on b, b lies inside *comb*, but b has been removed from the particular *Sort* we are operating on, in which case if a lies inside the *Sort*, the underlying lists have to be modified, swapping a and b in to and out of the *Sort*.

...and many more cases thereafter. For this reason, we elide a Haskell excerpt for *pointwise* here, as it is simply too large. We refer the reader to the definition of *pointwise* in the file `Terms/Sorts.hs`.

5.2.4 Unknowns

The excerpts described in this Subsection can be found in the file `Terms/Unknowns.hs`.

Unknowns are another fundamental building block of nominal terms. We provide a dedicated data type for representing unknowns, called *Unknown*, as during the course of the unification algorithm's execution, we need to be able to manipulate sets of unknowns (see for instance the rules in Definition 4.3.35).

We introduce an *Unknown* data type that carries an *Identifier* combined with a *Sort*. *Identifier* is a minor data type that handles objects with names throughout the system:

```
data Unknown α β = Unknown (Identifier α) (Sort β)
  deriving (Eq, Ord)
```

We automatically derive an instance of *Ord* for *Unknown*; this is necessary to make use of the Haskell Prelude *Set* collection data type.

5.2.5 Terms

The excerpts described in this Subsection can be found in the file `Terms/Terms.hs`.

We now have all the necessary components for defining a data type that adequately represents permissive nominal terms:

```
data Term α β = Atm β
              | Unk (Permutation β) (Unknown α β)
              | Abs β (Term α β)
              | TFr (Identifier α) [Term α β]
```

Term has a constructor for each possible element of a permissive nominal term: atoms, unknowns with suspended permutation, abstractions and term formers. The *Term* data type admits straightforward implementations for basic operations on permissive nominal terms. For instance, computing the free atoms of a term is particularly straightforward, and corresponds naturally to Definition 4.2.11:

```
fa :: (Eq α, Permissive β) ⇒ Term α β → Sort β
fa (Atm atm) = finiteSort [atm]
fa (Unk prm unk) = pointwise prm (srt unk)
fa (Abs abs bdy) = remove [abs] (fa bdy)
fa (TFr nm args) = foldr (union ∘ fa) empty args
```

Free atom calculations in permissive nominal terms return a permission sort, as discussed in Subsection 5.2.3. Calculating the free atoms of a single atom entails returning a finite sort (here, *finiteSort* is a function that takes a list of atoms and returns a *Sort*). Calculating the free atoms of an unknown involves permuting the unknown's sort with its suspended permutation.

The case for calculating the free atoms of a term former may seem a little opaque. We calculate the free atoms of each argument of the term former, and then union them together with *foldr*, a Haskell Prelude function, that performs a right-fold over a list.

Similarly, the implementation of α-equivalence is remarkably close to the mathematical definition (Definition 4.2.13):

```
aeq :: (Eq α, Permissive β) ⇒ Term α β → Term α β → Bool
aeq (Atm atm) (Atm atm') = atm == atm'
aeq (Unk prm unk) (Unk prm' unk') =
  srt unk == srt unk' ∧ (finiteSort (ds prm prm') `intersection` srt unk) == empty
aeq (Abs abs bdy) (Abs abs' bdy')
  | abs == abs' = aeq bdy bdy'
  | member abs' (fa bdy) = False
  | otherwise = permutationAction (permute abs abs') bdy `aeq` bdy'
aeq (TFr nm args) (TFr nm' args') =
  length args == length args' ∧ (and ∘ zipWith aeq args args')
aeq _ _ = False
```

Here *length* (returns the length of a list), *and* (returns the logical conjunction of a list of boolean values) and *zipWith* (combines two lists into one pointwise using a given function) are Haskell Prelude functions.

The α-equivalence test proceeds by induction on the structure of the two terms. Two atoms are α-equivalent if they are identical. The only remarkable difference between the Haskell implementation of α-equivalence, and the mathematical definition previously referenced, lies in the α-equivalence test for unknowns. In Definition 4.2.13, the following rule is used for unknowns:

$$\frac{(\pi|_S = \pi'|_S)}{\pi \cdot X^S =_\alpha \pi' \cdot X^S}\,(=_\alpha\mathbf{X})$$

Here $\pi|_S$ denotes the permutation π *restricted* to S, an infinite set. To implement $(=_\alpha\mathbf{X})$, we would have to find some way of restricting permutations to *Sort*s, a feat that doesn't immediately have an obvious remedy.

Fortunately, there is a way around this problem. We write $ds(\pi, \pi') = \{a \mid \pi(a) \neq \pi'(a)\}$ and call $ds(\pi, \pi')$ the **difference set** of π and π'. Intuitively, $ds(\pi, \pi')$ are all the atoms that π and π', when acting, 'disagree on'. Using $ds(\pi, \pi')$ we can formulate an alternative rule to $(=_\alpha\mathbf{X})$:

$$\frac{(ds(\pi, \pi') \cap S = \emptyset)}{\pi \cdot X^S =_\alpha \pi' \cdot X^S}\,(=_\alpha\mathbf{X}')$$

It is a fact that $(=_\alpha\mathbf{X})$ and $(=_\alpha\mathbf{X}')$ are equivalent: if we can derive two terms as being α-equivalent with $(=_\alpha\mathbf{X})$, then we can do the same with $(=_\alpha\mathbf{X}')$, and vice-versa. Indeed, early Definitions of α-equivalence for permissive nominal terms used $(=_\alpha\mathbf{X}')$ instead of $(=_\alpha\mathbf{X})$ (see for instance [DGM09b, Definition 9]). However, $(=_\alpha \mathbf{X}')$ is much more straightforward to implement than $(=_\alpha \mathbf{X})$, as it relies on 'standard' operations on the *Sort* data type, like the *intersection* function, and *Sort* equality tests.

We implement difference sets for our *Permutation* data type using a function, *ds*. This is used in the case for unknowns in *aeq*.

The only other subtlety to *aeq*'s definition is the catch-all pattern matching case. This catches any other case not previously matched. Clearly, if this case is 'fired', the two input terms are not α-equivalent.

5.2.6 Implementing substitutions

The implementation of permissive nominal terms uses two notions of substitution. First, there is a simple *Substitution* data type, corresponding simply to an association list of unknowns and terms.

```
data Substitution α β = Substitution [((Unknown α β), (Term α β))]
```

We provide a single mechanism for constructing simple unknowns, through a smart constructor, which checks our constructed substitution is in fact support-reducing in the sense of Definition 4.2.28:

```
substitution :: (Eq α, Permissive β) ⇒ Unknown α β
                                      → Term α β → Maybe (Substitution α β)
substitution unk trm
  | fa trm ‘subseteq‘ srt unk = Just (Substitution [(unk, trm)])
  | otherwise = Nothing
```

Here *Maybe* is the standard Haskell monad for denoting some computation that could possibly fail. *Nothing* is returned when *substitution* detects that the constructed substitution is not support reducing.

Composing two *Substitutions* is as simple as appending their lists. Similarly, computing the domain and range of the substitution is straightforward: simply map first or second projection functions across the underlying list representation. We define a substitution action on *Terms* with:

```
substitutionAction :: (Eq α, Permissive β) ⇒ Substitution α β → Term α β → Term α β
substitutionAction (Substitution [ ]) trm = trm
substitutionAction subs (Atm atm) = Atm atm
substitutionAction (Substitution ((unk, trm):t)) (Unk prm unk′)
  | unk == unk′ = substitutionAction (Substitution t) $ permutationAction prm trm
  | otherwise = substitutionAction (Substitution t) $ Unk prm unk′
substitutionAction subs (Abs abs bdy) =
  Abs abs $ substitutionAction subs bdy
substitutionAction subs (TFr nm args) =
  TFr nm $ (map ∘ substitutionAction) subs args
```

Applying a *Substitution* is a case of descending through the *Term* structure, until we hit the unknowns. As permissive nominal term substitutions are not capture avoiding, we obtain a straightforward implementation. The most interesting case is when we descend through the term, and hit an unknown. If the unknown in question matches an unknown in the underlying *Unknown-Term* list carried by the inputted *Substitution*, we perform a replacement, then proceed recursively once more on the new *Term*.

The simple list of *Unknown-Term* pairs representation for substitutions is fine for most applications, where we simply wish to descend into a term and replace all instances of some unknown with a term. However, the unification algorithm of Chapter 4 also makes heavy use of another type of substitution, where this representation no longer works as well. Notably, the 'support reducing substitution' ρ of Definition 4.3.21 requires a notion of state. We repeat the most relevant line of the definition here, for convenience:

$$\rho^{\mathcal{V}}_{Inc}(X^S) = id{\cdot}X'^{S'} \quad \text{if } X^S \in \mathcal{V} \text{ and } \quad \rho^{\mathcal{V}}_{Inc}(X^S) = id{\cdot}X^S \quad \text{otherwise}$$

The particular trappings of the definition are irrelevant for our discussion, but can be inferred from the context of Definition 4.3.21, and surrounding definitions. What matters is the fact that ρ carries an 'internal state', so to speak: how ρ acts cannot be described using a simple list of *Unknown-Term* pairs. Instead, we must introduce another type, corresponding to these substitutions with state, called *ConditionalSubstitution*:

$$\boxed{\quad \textbf{type}\ \mathit{ConditionalSubstitution}\ \alpha\ \beta\ =\ (\mathit{Unknown}\ \alpha\ \beta\ \rightarrow\ \mathit{Substitution}\ \alpha\ \beta) \quad}$$

Intuitively, *ConditionalSubstitution* represents a specific kind of closure, which takes an unknown as input and returns a straightforward substitution. Like their *Substitution* counterparts, we define an action on terms for *ConditionalSubstitutions*, in roughly an analogous fashion.

However, unlike *Substitution*s, defining substitution composition now becomes tricky. Fortunately, the algorithm of Chapter 4 only uses substitutions with state in a small number of places, most prominently in rule (**I3**) in Definition 4.3.35. Notably, (**I3**) is applied *after* every other simplification rule has been applied, and therefore we do not need to be able to explicitly compose *ConditionalSubstitution*s to implement the permissive unification algorithm.

5.3 Unification of terms

The previous section introduced functions and data types for working with and manipulating permissive nominal terms. In this Section, we describe the implementation of the permissive unification algorithm of Chapter 4. The unification algorithm is split into two submodules: support reduction (described in Subsection 5.3.1) and the unification algorithm proper (described in Subsection 5.3.2).

5.3.1 Support reduction

The excerpts described in this Subsection can be found in the file `Unification/SupportReduction.hs`.

Trying to unify $id{\cdot}X^S$ and $id{\cdot}Y^T$ where $S \neq T$ requires that we find a new permission sort U where $U \subseteq S$ and $U \subseteq T$. This 'joining' of permission sorts is called support reduction (Subsubsection 4.3.2.1). Accordingly, a support reduction problem is a list of pairs of *Term*s and *Sort*s, and is easily representable as a data type:

```
data SupportReduction α β = SupportReduction [(Term α β, Sort β)]
```

Using this list-based representation, like the *Substitution* and *Permutation* data types, we implement the union of two *SupportReduction*s by simply concatenating their underlying list representations. We also calculate the free variables of a support reduction problem by simply mapping a function for calculating the free variables of terms across the underlying list representation:

```
fV :: (Ord α, Ord β, Eq α, Permissive β) ⇒ SupportReduction α β → Set (Unknown α β)
fV (SupportReduction [ ]) = Set.empty
fV (SupportReduction ((trm, srt):t)) =
  Terms.Terms.fV trm `Set.union` fV (SupportReduction t)
```

In Definition 4.3.14 we defined a notion of normal form for a support reduction problem, and provided a characterisation of normal forms in Lemma 4.3.17. Moreover, we proved that we can calculate this normal form using a reduction relation, defined in Definition 4.3.8. This relation always halts, on any input, a result proved in Theorem 4.3.13. We implement the support reduction simplification relation, using the following function, *nf*:

```
nf :: (Eq α, Permissive β) ⇒ SupportReduction α β → SupportReduction α β
nf (SupportReduction [ ]) = SupportReduction [ ]
nf (SupportReduction (a@(Terms.view → AtmView atm, srt):t)) =
  if member atm srt then
    nf (SupportReduction t)
  else
    union (SupportReduction [a]) (nf (SupportReduction t))
nf (SupportReduction ((Terms.view → TFrView nm args, srt):t)) =
  let
    argsSupp = foldr union empty (map (λx → (SupportReduction [(x, srt)])) args)
  in
    nf $ union argsSupp $ SupportReduction t
nf (SupportReduction ((Terms.view → AbsView abs bdy, srt):t)) =
  let
    newsrt = Sorts.union srt $ finiteSort [abs]
  in
    nf $ SupportReduction $ (bdy, newsrt):t
nf (SupportReduction (u@(Terms.view → UnkView prm unk, srt'):t)) =
  let
    invSrt = pointwise (inverse prm) srt'
  in
    if not (isIdentity prm) ∧ not (srt unk `subseteq` invSrt) then
      nf $ SupportReduction $ (mkUnk identity unk, invSrt):t
    else
      if srt unk `subseteq` invSrt then
        nf $ SupportReduction t
      else
        union (SupportReduction [u]) $ nf $ SupportReduction t
```

Here *inverse* is the function for inverting a *Permutation*, by reversing its underlying list representation.

The empty *SupportReduction* is already in normal form. Otherwise, suppose we have a *Term-Sort* pair at the head of our *SupportReduction*, representing the support reduction $a \sqsubseteq S$. If $a \in S$ then we 'discharge', removing the head, and simplifying the rest of the problem. If $a \notin S$, we keep the associated *Term-Sort* pair, and return it with the remainder of the simplified problem.

Simplifying a support reduction with a termformer is straightforward, as we simply traverse into the termformer and simplify its arguments. Similarly, simplifying a *SupportReduction* with a *Term-Sort* pair at the head, representing the support reduction $[a]r \sqsubseteq S$, is as straightforward as simplifying the *Term-Sort* pair representing $r \sqsubseteq S \cup \{a\}$.

The hardest case comes when simplifying a *SupportReduction* with a *Term-Sort* pair at the head, representing the support reduction $\pi{\cdot}X^S \sqsubseteq T$. In this case we have a case split, corresponding to the two side conditions on $(\sqsubseteq\mathbf{X})$ and $(\sqsubseteq\mathbf{X}')$ from Definition 4.3.8. In particular, we first check whether π is the identity and then whether $S \not\subseteq \pi^{-1}{\cdot}T$. If so, we attempt to simplify a *SupportReduction* with a *Term-Sort* pair corresponding to $X^S \sqsubseteq \pi^{-1}{\cdot}T$ at the head. Otherwise, we check whether $S \subseteq \pi^{-1}{\cdot}T$, in which case we discharge, and simplify the rest of the *SupportReduction*. Any other case is 'stuck'.

Using *nf* we may test whether a support reduction problem is inconsistent. A *SupportReduction* in normal form is inconsistent if it contains a *Term-Sort* pair representing a support reduction of the form $a \sqsubseteq S$ (Definition 4.3.14):

```
isConsistent :: (Eq α, Permissive β) ⇒ SupportReduction α β → Bool
isConsistent suppinc =
  let
    isConsistent′ :: (Eq α, Permissive β) ⇒ SupportReduction α β → Bool
    isConsistent′ (SupportReduction [ ]) = True
    isConsistent′ (SupportReduction ((Terms.view → AtmView atm, srt):t)) = False
    isConsistent′ (SupportReduction (suppinc:t)) = isConsistent′ $ SupportReduction t

    nfrm = nf suppinc
  in
    isConsistent′ nfrm
```

Similarly, using *nf* we may also check whether a *SupportReduction* is trivial. Following Definition 4.3.15, a *SupportReduction* is trivial if its associated normal form is empty.

```
isTrivial :: (Eq α, Permissive β) ⇒ SupportInclusion α β → Bool
isTrivial suppinc =
  case nf suppinc of
    SupportInclusion [ ] → True
    _ → False
```

The following function corresponds to an important building block in the support reduction algorithm. Namely, *injectiveChoice* is a function that corresponds to Definition 4.3.20. The function takes two arguments: an *Unknown* and a *SupportReduction*. As output, the function returns a fresh unknown, injectively picked based on the input *Unknown*. The output *Unknown*'s *Sort* is the intersection of the *Sorts* of *Unknowns* appearing in the input *SupportReduction*, after being placed into normal form. *injectiveChoice* produces the fresh unknowns necessary for joining distinct unknowns with differing permission sorts in the unification algorithm.

```
injectiveChoice :: (Injective α, Permissive β) ⇒ Unknown α β → Set (Unknown α β) →
                                                   SupportReduction α β → Unknown α β
injectiveChoice unk unks suppinc =
  case nf suppinc of
    SupportInclusion [ ] → unk
    SupportInclusion supps →
     let
       injectiveChoice′ unk unks s =
       let
         name = (id ∘ nm) unk
         sort = srt unk
         name′ = identifier $ injective name unks
         toIntersect = map snd s

         injectiveChoice″ :: Permissive a ⇒ [Sort α] → Sort α
         injectiveChoice″ [ ] = error "injectiveChoice : the impossible happened"
         injectiveChoice″ [srt] = srt
         injectiveChoice″ (h:t) = intersection h $ injectiveChoice″ t

         sort′ = injectiveChoice″ $ sort:toIntersect
         unk′ = unknown name′ sort′
       in
         unk′
     in
       injectiveChoice′ unk supps
```

We see that *injectiveChoice* makes use of the *Injective* type class. This type class is reproduced below:

```
class Eq α ⇒ Injective α
  where
    injective :: α → [α] → α
```

Injective allows us to make an injective choice of fresh object. That is, we enforce the condition that if x and y are distinct, then *injective* x s and *injective* y s are also distinct, where s is a list of atoms, and the fresh choice of object does not appear within the second argument of *injective*. Moreover, we provide a simple instance of the *Injective* type class for the Haskell *String* type, which we use as the underlying implementation of identifiers for unknowns in the implementation.

There is the possibility of *injectiveChoice* raising a runtime exception. This occurs if the *SupportInclusion* passed to *injectiveChoice* happens to be trivial. Before calling *injectiveChoice*, we ensure that this is not the case.

The operation of *injectiveChoice* is largely straightforward. We first place the input *SupportReduction* into normal form, though we ensure that this is already the case before calling *injectiveChoice*. Then, we collect all the permission sorts from the normal form *SupportReduction* into a list, and collectively intersect them with each other, and set this as the permission sort of a freshly picked unknown.

injectiveChoice is used as a subroutine in the definition of *rho*, a function capturing the support reducing substitution from Definition 4.3.21. *rho* is an example of a *ConditionalSubstitution*—a substitution with state, or a closure. We remind the reader that a *ConditionalSubstitution* is a function from *Unknowns* to a *Substitution*. *rho* takes as input a *Set* of *Unknowns* and the behaviour of the resulting *ConditionalSubstitution* is determined by whether the substitution's input *Unknown* is a member of this *Set*.

```
rho :: (Ord α, Ord β, Injective α, Permissive β) ⇒ Set (Unknown α β) →
                                                    SupportReduction α β →
                                                    ConditionalSubstitution α β
rho v suppinc x =
  if Set.member x v then
    let
      inj = injectiveChoice x suppinc
      subst = substitution x $ mkUnk Terms.Permutations.identity inj
    in
      case subst of
        Just s → s
        Nothing → error "rho : substitution has incorrect support"
  else
    case substitution x $ mkUnk Permutations.identity x of
      Just s → s
      Nothing → error "rho : substitution has incorrect support"
```

Once more, there is the possibility that *rho* will raise a runtime exception. This occurs if the substitution constructed has incorrect support (i.e. takes the form $[X^S{:=}Y^T]$ where $T \not\subseteq S$). If the permission sort intersection calculation carried out in *injectiveChoice* is correct, this should never occur.

rho solves a given support reduction problem. As a result, it plays a fundamental part in the operation of the unification algorithm.

5.3.2 Unification algorithm

The excerpts described in this Subsection can be found in the file `Unification/Unification.hs`.

The unification problem described in this section builds on all the Haskell excerpts so-far described. In particular, it uses the support reduction algorithm of Subsection 5.3.1 as a sub-procedure for 'joining' unknowns of differing permission sorts. We start by introducing a data type, *Unification*, that represents unification problems:

```
data Unification α β = Unification [(Term α β, Term α β)]
```

We encode unification problems as lists of *Term*-*Term* pairs. Once more, this enables us to easily represent the empty unification problem (an empty list) and facilitates an easy implementation of union (list concatenation).

It is often necessary to apply a substitution to each term appearing in a *Unification*. This is especially true when computing the unification problem simplification procedure of Definition 4.3.35. Notably, the instantiation rules, (**I1**) through to (**I3**), require that we implement a function for applying *Substitution*s pointwise to *Unification*s, and also a function for applying *ConditionalSubstitution*s to *Unification*s. These two functions are provided below:

```
substitutionAction :: (Eq α, Permissive β) ⇒ Substitution α β → Unification α β → Unification α β
substitutionAction subst (Unification uni) =
  Unification $ map (Terms.substitutionAction subst *** Terms.substitutionAction subst) uni

condSubstitutionAction :: (Eq α, Permissive β) ⇒ ConditionalSubstitution α β →
Unification α β →
(Unification α β, Substitution α β)
condSubstitutionAction subst (Unification uni) =
  let
    temp = map (Terms.condSubstitutionAction subst *** Terms.condSubstitutionAction subst) uni
    uni′ = map (fst *** fst) temp
    subst′ = foldr compose identity (map (uncurry compose . (snd *** snd)) temp)
  in
    (Unification uni′, subst′)
```

Here, ***** is a Haskell function in the *Control.Arrow* module, which takes two functions and a pair as input, then applies one function to the left element of the pair, and the other function to the right element of the pair.

substitutionAction is largely straightforward, simply mapping across the underlying list of the *Unification*, applying the *substitutionAction* function to each *Term*. More subtle is the *condSubstitutionAction*, especially its return type. First, we apply the *ConditionalSubstitution* to each *Term* in the *Unification*, just as in the case of *substitutionAction*.

Note that a *ConditionalSubstitution* is really a function from *Unknown*s to *Substitution*s. That is, a *ConditionalSubstitution* is morally a 'hidden' substitution, which is only revealed once we apply the *ConditionalSubstitution* to an *Unknown*. However, in order to eventually present the unifier, computed by the unification algorithm, to the user as feedback, we need to be able to record these hidden substitutions. We therefore collect the 'unhidden' *Substitutions* by composing them into a single *Substitution*, *subst'*, and return this *Substitution* as part of the output of the function.

Only one other function of note is included in the unification module, *solve*, an implementation of the unification algorithm defined in Definition 4.3.39. Notably, if properly implemented, this function is guaranteed to always terminate (Theorem 4.3.42) and always return a most general unifier (Theorem 4.3.56). The implementation of *solve* is rather lengthy, we therefore present only a few cases and refer the reader to `Unification.hs` for the full function definition.

The function *solve* takes two inputs. The first argument is a *Set* of *Unknowns* at least as large as the set of free unknowns of the unification problem. The second argument is the unification algorithm to solve.

We note that *solve* returns an object of type *Either Unification Substitution*. Here, *Either* is the standard Haskell sum type. *solve* returns either a *Unification* which cannot be further simplified, therefore no solution exists, or a *Substitution*, which solves the input *Unification* problem.

Simplifying a *Unification* problem with a *Term-Term* pair at the head, corresponding to $a \overset{?}{=} a$ is straightforward:

```
solve unks (Unification ((Terms.view → AtmView atm, Terms.view → AtmView atm′):t)) =
  if atm == atm′ then
    solve unks (Unification t)
  else
    inject (mkAtm atm, mkAtm atm′) (solve unks (Unification t))
```

That is, if the two *Terms* at the head of the *Unification* represent identical atoms, we discharge, and simplify the rest of the *Unification*.

The alternative case, the two *Terms* at the head of the *Unification* represent differing atoms, represented by the else-branch in the previous excerpt, is slightly more complicated. Here, *inject* is a small helper function used to make the implementation of *solve* clearer. Intuitively, *inject* is called when the unification problem currently under consideration cannot be solved. In this case, we call *inject* to simplify the rest of the *Unification*, before ‘consing’ the unsolvable obligation onto the front of the *Unification*. The definition of *inject* is slightly complicated by the fact that simplifying the rest of the *Unification* problem returns either another *Unification* or a *Substitution*.

Simplifying a *Unification* with a *Term-Term* pair at the head corresponding to two abstractions involves a case split. If the two abstracted atoms are identical, then we simplify their bodies. Otherwise, we check whether a renaming is possible to make the abstracted atom identical, without sacrificing α-equivalence, perform the renaming, and then simplify the bodies:

```
solve unks (Unification ((Terms.view → AbsView abs bdy, Terms.view → AbsView abs′ bdy′):t))
  | abs == abs′ = solve unks (Unification ((bdy, bdy′):t))
  | not (abs′ ‘member‘ fa bdy) =
  let
    newBdy = permutationAction (permute abs abs′) bdy
  in
    solve unks (Unification ((newBdy, bdy′):t))
  | otherwise = inject (mkAbs abs bdy, mkAbs abs′ bdy′) (solve unks (Unification t))
```

To simplify a *Unification* with two termformers at the head we simply traverse through the termformer and simplify its arguments.

The hardest case to implement is the case where two *Unknowns* must be unified: Suppose our two *Unknowns* have identical *Permutations* suspended upon them, and identical *Sorts*. Then we discharge, and simplify the rest of the *Unification*:

```
solve unks (Unification ((Terms.view → UnkView prm unk, Terms.view → UnkView prm' unk'):t))
    | prm == prm' ∧ unk == unk' = solve unks (Unification t)
```

Otherwise, suppose our *Unknowns* are different, but the free atoms of one is a subset of the other. In this case, we may directly build a *Substitution* that unifies these two *Unknowns*. This step corresponds to an application of either (**I1**) or (**I2**). We show only the Haskell excerpt implementing (**I1**), as the case for (**I2**) is similar:

```
| unk ≠ unk' ∧ (fa (mkUnk prm' unk') `subseteq` fa (mkUnk prm unk)) =
  let
    subst = (substitution unk (mkUnk (Permutations.compose (inverse prm) prm') unk'))
  in
    case subst of
      Nothing → error "solve : error constructing substitution"
      Just subst' → let
                      newProb = substitutionAction subst' (Unification t)
                      tail = solve unks newProb
                    in
                      case tail of
                        Right (subst'') → (Right (subst' `compose` subst''))
                        _ → tail
```

Finally, we implement (**I3**). Note that (**I3**) is the last attempt at unifying two unknowns, per Definition 4.3.39. We first construct our support reducing substitution *rhoSubst* and apply it across the *Unification*. As noted, *condSubstitutionAction* returns a new *Unification* and also a *Substitution*. If the rest of the *Unification* can be further simplified, into a solved state, by recursively calling *solve*, then we return this *Substitution* as part of the answer, otherwise we return the rest of the unsolvable *Unification*. Note, we expand the set of *Unknowns* that were passed to *solve* as input, to include the set of freshly generated *Unknowns* picked injectively:

```
| otherwise =
  let
    suppIncL = map (second fa) ((mkUnk prm unk, mkUnk prm' unk'):t)
    suppIncR = map (second fa) ((mkUnk prm unk, mkUnk prm' unk'):t)
    suppInc = supportInclusionL (List.nub (suppIncL ++ suppIncR))
  in
    if isConsistent suppInc ∧ not (isTrivial suppInc) then
      let
        maxInjectiveChoices = Set.map (λx → injectiveChoice x suppInc)
          (fV (Unification ((mkUnk prm unk, mkUnk prm' unk'):t))
        rhoSubst = rho unks (nf suppInc)
        (newUni, subs) = condSubstitutionAction rhoSubst
          (Unification ((mkUnk prm unk, mkUnk prm' unk'):t))
        tail = solve (maxInjectiveChoices `Set.union` unks) newUni
      in
        case tail of
          Right subs' → Right (subs `compose` subs')
          _ → tail
    else
      inject (mkUnk prm unk, mkUnk prm' unk') (solve unks (Unification t))
```

All other cases are routine.

5.4 The PNT Frontend

The excerpts described in this Section can be found in the directory `Frontend`.

We built the PNT Frontend to provide a simple text-based interaction layer between the underlying PNT library and the user. The PNT Frontend serves as a simple means of using the functionality of the PNT library, and was also instrumental in testing and debugging the implementation.

The PNT Frontend accepts a language describing sorts, unknowns, terms and ‘obligations’. Here obligations are goals to solve, and fall into five classes:

1. *Unification* obligations take two terms and attempt to find a substitution unifying the two. If one does not exist, a message saying so is printed as output, otherwise the required substitution is pretty-printed.
2. *Fresh name* obligations take a term and print an atom fresh for that term as output.
3. *α-equivalence* obligations take two terms and test them for α-equivalence. A Boolean value is printed as output.
4. *Permutation* obligations take a term and a permutation and return and pretty-print the permuted term as output.
5. *Free atoms* obligations take a term and print the free atoms of the term as output.

A BNF grammar of the language supported by the PNT Frontend is provided in Appendix C. However, we provide the following table to allow the reader to quickly translate between a permissive nominal term (and associated definitions) and a possible representation in the PNT Frontend:

Permissive nominal term	PNT Frontend representation
a, b, c, ...	`1, 2, 3, ...`
comb	`comb`
$\mathit{comb} \setminus \{a\} \cup \{b\}$	`comb union { 1 } minus { 2 }`
X^S	`X^S`
$(b\ a){\cdot}X$	`(1 2) .  X`
$[a]a$	`[1]1`
$\mathsf{f}([a]a, c)$	`f([1]1, 3)`

An example script, complete with the PNT Frontend response, is provided in Subsection 5.4.1.

PNT Frontend interaction with the user either proceeds by the executable loading a file, or the user pasting a 'module' as the command line arguments of the executable. Results are returned to the user on the command line, pretty printed.

5.4.1 Example interaction

In this Subsection we present an example interactions between the user and the PNT Frontend.[2].

Example 5.4.1: We wish to unify and otherwise manipulate the two permissive terms $\mathsf{Tst}([a]a, id{\cdot}X^S, [a](id{\cdot}Y^T))$ and $\mathsf{Tst}([b]b, id{\cdot}Y^T, [a]a)$, and manipulate the permissive term $[e]f$. We refer to these terms as terms one, two and three, respectively, and define $S = \mathit{comb}$ and $T = \mathit{comb} \setminus \{b, d\} \cup \{a, c\}$. We work under the convention that a, c, e lie outside *comb* whereas b, d, f lie inside *comb*. Tst is an arbitrary termformer of arity 3.

Concretely, in terms of working with the PNT implementation, we take a, c, e to be the integers $1, 3, 5$, respectively, and b, d, f to be the integers $2, 4, 6$, respectively. Note the fact that the underlying representation of atoms is visible to the user is a failure in the PNT Frontend. A more sophisticated layer built on top of the PNT library would keep the implementation of atoms completely abstract, allowing the user to write that $a \notin \mathit{comb}$ and $b \in \mathit{comb}$, as needed.

The corresponding input to the PNT Frontend is listed in Figure 5.1. Note, in the syntax of the PNT Frontend, the ascription of permission sorts to unknowns occurs once, after which we only ever refer to an unknown by its name (i.e. we do not write X^S, but only X). We did this to remove syntactic clutter.

The PNT Frontend enforces the convention that even positive integers are in *comb*, whilst odd positive integers are outside *comb*. We aim to unify terms one and two (Obligation o1), calculate a fresh name for term one (Obligation o2), test terms one and two for α-equivalence (Obligation o3), permute term three with the swapping $(e\ f)$ (Obligation o5) and finally calculate the free atoms of term four (Obligation o6).

Figure 5.2 reproduces in abbreviated form the output produced by the PNT Frontend after processing the obligations. Formatted and translated, this output reads:

[2]As previously mentioned, a web-based version of the PNT Frontend is available at `http://www2.macs.hw.ac.uk/~dpm8/permissive`

```
module Test begin
  sorts begin
    sort S is comb
    sort T is comb minus {2, 4} union {1, 3}
  end
  unknowns begin
    unknown X has sort S
    unknown Y has sort T
  end
  termformers begin
    termformer Tst
  end
  terms begin
    term t1 is Tst([1](1), id . X, [1](id . Y))
    term t2 is Tst([2](2), id . Y, [1](1))
    term t3 is [5](6)
  end
  obligations begin
    obligation o1 is unify t1 and t2
    obligation o3 is fresh name for t1
    obligation o4 is aeq t1 and t2
    obligation o5 is permute t3 with (5 6)
    obligation o6 is free atoms of t1
  end
end
```

Figure 5.1 Example interaction: PNT Frontend input script

```
"o1" has solution ["X’"<comb - {4 2}> := 1] o
                  ["Y’"<comb - {4 2} + {3 1}> := Id."X’"<comb - {4 2}>] o
                  ["Y"<comb - {4 2} + {3 1}> := Id."Y’"<comb - {4 2} + {3 1}>] o
                  ["X"<comb> := Id."X’"<comb - { 4 2 }>]
"o3" has solution 5
"o4" is False
"o5" has solution [6](5)
"o6" has solution comb + {3}
```

Figure 5.2 Output from the PNT Frontend

Obligation	Solution
1	$[X'{:=}a]{\circ}[Y'^{T'}{:=}id{\cdot}X'^{S'}]{\circ}[Y^{T}{:=}id{\cdot}Y'^{T'}]{\circ}[X^{S}{:=}id{\cdot}X'^{S'}]$
2	e
3	false
4	$[f]e$
5	$comb \cup \{c\}$

Here, $T' = comb\backslash\{b,d\}$ and $S' = comb\backslash\{b,d\}$. A simple check confirms that $[X'{:=}a]{\circ}[Y'^{T'}{:=}id{\cdot}X'^{S'}]{\circ}[Y^{T}{:=}id{\cdot}Y'^{T'}]{\circ}[X^{S}{:=}id{\cdot}X'^{S'}]$ indeed unifies terms one and two.

5.5 Conclusions

This Chapter provided an overview of a prototype implementation of permissive nominal terms and their associated unification algorithm, called PNT. We also provide the text-based PNT Frontend, for easily defining permission sorts and terms, and computing unifiers. PNT is written in Haskell, though neither its design nor implementation depend on Haskell's laziness, as all data types are finite.

One other implementation of nominal terms exists, Calvès' Haskell Nominal Toolkit exists, though we are careful to draw a distinction between the HNT, a 'full' implementation of nominal terms, with a focus on efficiency, and our implementation of permissive nominal terms. In contrast to Calvès, we aimed not to provide a feature full and efficient implementation, but to show that the infinite and coinfinite nature of permissive nominal terms' permission sorts are no barrier to a straightforward implementation.

Also of note is Cheney's FreshLib [Che05b]. This is an implementation of a library for generic programming, which aims to reduce the amount of *nameplate* source code—fresh name generation, capture avoiding substitution etc.—that Haskell programmers must write when implementing data types representing abstract syntax with binding. FreshLib uses many nominal ideas, but is *not* an implementation of nominal terms.

Aside from the HNT, to the best of our knowledge, no other stand-alone implementation of nominal terms exists.

CHAPTER 6

Conclusions

Section 6.1 provides a brief summary of the work presented in this thesis. Section 6.2 summarises the broader body of doctoral research, not necessarily included in this thesis, and attempts to draw some unifying themes.

6.1 Summary of thesis

Nominal terms were proposed as a metalanguage for embedding syntax with binding [UPG04]. They have excellent computational properties, and a concrete semantics in nominal sets.

A recent body of work by Gabbay and others [Gab05, GL08, GM09a, FG07b] has examined nominal terms as a metalanguage for embedding object languages into. As opposed to pure syntax, where equality is taken to be α-equivalence, object languages may have their own, more complex, forms of equality, such as β-equivalence for the λ-calculus. These forms of equivalence are captured by equalities in the case of nominal algebra, or rewrite rules in the case of nominal rewriting. This is in contrast to the narrow focus of nominal terms as a system for embedding syntax, as exemplified by the work of Cheney and Urban on αProlog, for instance [CU08, pg. 14]:

> As reflected by our choice of examples, at present we view αProlog as rather narrowly focused on the domain of prototyping and experimenting with logics, operational semantics for programming and concurrency calculi, and type systems and other program analyses.

The work presented in this thesis continues in the tradition of treating nominal terms as a metalanguage. Chapters 1 and 2 introduce the thesis. However, the main body of work in Chapters 3 through 5 implicitly ask: how can this metalanguage be extended, and can we import ideas from other metalanguages for use with nominal terms?

In Chapter 3, we extended nominal terms with λ-abstraction for atoms and unknowns. Associated notions of β-reduction were provided for both λ-bound atoms and unknowns. λ-abstraction is a prominent feature in many metalanguages, but something nominal terms lack.

A novel context calculus—useful research tools for investigating dynamic linking, module systems, novel programming language designs, amongst many other uses—called the **two-level λ-calculus** was the result of extending nominal terms with λ-bound atoms and unknowns. Both an explicit and implicit nominal influence on the two-level λ-calculus can be detected. The explicit influence—the use of swappings and freshness side-conditions—led to a straightforward theory of α-equivalence whilst retaining a 'nameful' syntax. The implicit influence—the clear conceptual separation of two levels of variable—led to a simple solution to the well-known failure of commutation and hole filling in context-calculi.

The two-level λ-calculus is confluent (Theorem 3.4.42). Proving that this is so was non-trivial. The confluence proof is split into two parts: a confluence proof for the level one fragment

(Subsection 3.4.1), and a confluence proof for the level two fragment (Subsection 3.4.2). Finally, these two proofs are stitched together to form a proof of confluence for the whole calculus (Subsection 3.4.2).

The proof of confluence for the level two fragment uses a standard proof technique, due to Tait. However, confluence for the level one fragment uses a novel proof technique, using a notion of canonical forms (Definition 3.4.13) that all terms must eventually reduce to. This technique appears to be quite general; we have used the same proof technique in confluence proofs for one-and-a-half-level terms [GM08b, GM09b] and the permissive two-level λ-calculus [GM10b].

The confluence proof for the two-level λ-calculus also taught us something important. Namely, though the tacit slogan of nominal techniques has always been 'ϵ-away from informal practice', in actual fact, nominal terms do not possess some important properties that informal terms do. Two properties, 'always fresh' and 'always rename', mean that it is always possible to rename a bound variable to some name chosen fresh when working with informal syntax. These properties do not hold for nominal terms.

The confluence proof of the two-level λ-calculus showed that these properties matter. Often we would like to 'just rename' a bound atom to something fresh, in order to push a substitution under a binder. But, depending on the context we are working in, this may or may not be possible. Our solution to this was to introduce a 'freshening' operation, extending a freshness context with some fresh atoms (Definition 3.4.10). Yet this freshening operation had the effect of obscuring the technical meat of the confluence proof and complicating the statements of various lemmas and theorems.

Chapter 4 introduced **permissive nominal terms**, a variant of nominal terms which elide explicit freshness contexts, and therefore recover the 'always fresh' and 'always rename' properties that nominal terms lack (Corollaries 4.2.17 and 4.2.18). Instead of explicit freshness contexts, unknowns are labeled with an infinite and coinfinite set of atoms, called their permission sort (Definition 4.2.2). Intuitively, a permission sort controls how an unknown is instantiated, and its special infinite and coinfinite form means it is always possible to find a fresh atom for any term. With permissive nominal terms, it is *always* possible to 'just rename'.

Permissive nominal terms handle freshness differently to nominal terms. Freshness now becomes a structural property of the term itself, similar to the notion of 'free variables of'; we define a notion of the 'free atoms of' a term by structural induction on the term itself (Definition 4.2.11), not via a derivable freshness relation.

Nominal terms enjoy attractive computational properties: unification is decidable, and most general unifiers exist. Do permissive nominal terms retain these same computational properties? The answer is 'yes': permissive nominal unification is decidable (though the algorithm presented in this thesis computes unifiers in worst-case exponential time), and most general unifiers exist (Theorem 4.3.58) modulo the reasonable assumptions regarding the decidability of certain relations on permission sorts that we made in Chapter 4.

There remains a close correspondence between permissive nominal terms and nominal terms. We make this correspondence formal with a non-trivial translation between the permissive and the nominal worlds (Definition 4.4.12), and we show that unifiers are preserved under this translation (Theorem 4.4.24).

Finally, Chapter 5 demonstrates that the infinite and coinfinite nature of permission sorts is no barrier to straightforward implementation of permissive nominal terms. We demonstrate this constructively, by way of a prototype implementation of permissive nominal terms and their unification algorithm, called **PNT**. An implementation is also considerably more 'tactile' than the abstract mathematical description of permissive nominal terms of Chapter 4. Users can explore permissive nominal terms, and obtain a feel for how they differ from nominal terms.

PNT is implemented in Haskell, a pure lazy functional language, and all data structures within the implementation are finite. PNT comes equipped with a frontend (PNT Frontend) accepting as input a domain specific language for defining permission sorts, terms, and unification obligations (described in Section 5.4). We demonstrate PNT in action with an example session, unifying two permissive nominal terms (Subsection 5.4.1).

6.2 Unifying themes of doctoral research

This thesis does not present all work that I carried out as a doctoral student (in order to form a coherent 'story' with this thesis). This extra work is now summarised. We can group the publications presented in Section 1.3 into several strands of related material, as follows:

Extensions of nominal terms The two-level λ-calculus [GM09d, GM10c], one-and-a-half level terms [GM08b, GM09b] and the permissive two-level λ-calculus [GM10b] have a great deal in common. All three systems may be seen as taking an underlying notion of nominal term, and providing λ-abstraction for atoms (in the case of one-and-a-half level terms), or for both atoms and unknowns (in the case of the two-level λ-calculus and permissive two-level λ-calculus).

The two-level λ-calculus, as Chapter 3 made clear, is a 'vanilla' context-calculus, and motivation for investigating pure context-calculi was discussed within that chapter.

One-and-a-half-level terms [GM08b, GM09b], endowed with a type-system corresponding to first-order logic, serve as proof terms for incomplete derivations. The Curry-Howard correspondence allows one to construct a 'witnessing λ-term' for any Natural Deduction proof. However, the correspondence has a 'forward' bias: it is easier to construct proof terms corresponding to a derivation from axioms towards goals ('forwards proof'), rather than a derivation from goal towards axioms ('backwards proof'). However, backwards proof is often the predominant form of proof in proof assistants like Isabelle, where at any stage the derivation may be incomplete.

Proof terms for incomplete derivations require some notion of metavariable, modeling a currently unknown branch of the derivation which will hopefully be refined at some later point. We therefore extend nominal terms with the ability to λ-bind an atom, and use nominal unknowns to model incomplete branches of a derivation tree.

Soundness and completeness for closed terms (with no free unknowns) is proved with respect to Natural Deduction [GM09b, Theorems 41 and 42]. The associated notion of β-reduction on terms corresponds to a proof normalisation procedure for incomplete derivations [GM09b, Definition 43]. Nominal technology allows us to manage α-equivalence in the presence of unknown terms.

The permissive two-level λ-calculus [GM10b] is another context-calculus in the mould of the two-level λ-calculus of Chapter 4. However, the permissive two-level λ-calculus uses a different notion of freshness and α-equivalence, making use of the permissive technology introduced in Chapter 4.

Further, the focus of the two-level and permissive two-level λ-calculi differ. The permissive two-level λ-calculus has a case statement for atoms, allowing the level two fragment of the calculus to inspect the level one fragment. As a result, the permissive two-level λ-calculus is not a vanilla context-calculus, rather designed specifically with the goal of meta-programming in mind. Again, the use of nominal technology allows us to handle α-equivalence in the presence of unknowns.

Although these three calculi—the two-level λ-calculus, one-and-a-half-order terms and the permissive two-level λ-calculus—share many traits, their focus is remarkably different. The author and M. J. Gabbay have employed extensions of nominal terms in the interdisciplinary study of context-calculi [GM09d, GM10c], proof terms for incomplete derivations [GM08b, GM09b] and meta-programming [GM10b]. We believe that this wide range of applications validates the informal thesis that two-levels of variable, capturing substitution for unknowns, binding atoms in unknowns, freshness side-conditions are useful, and the nominal terms syntax, which internalises all of these features, is of independent research interest.

Different notions of nominal term A second thread of research has been the investigation of different notions of pure nominal terms. This thread is exemplified by investigation into permissive nominal terms [DGM09a, DGM10] and semantic nominal terms [GM09c].

Chapter 4 introduced permissive nominal terms, and the reasons for introducing these were discussed in detail within that chapter.

Semantic nominal terms are an attempt to correct an asymmetry in the semantics of nominal terms. Whilst atoms have an independent denotational existence in the cumulative hierarchy of nominal sets, unknowns do not, and a denotation is provided using a valuation, as with first-order variables.

Semantic nominal terms were proposed to give unknowns an independent denotation, in line with atoms, and now α-equivalence and syntactic identity coincide. Unknowns are taken to be infinite lists of atoms, and permutations no longer suspend on semantic unknowns, but directly act pointwise on the elements of these representative lists. This yields a syntax where open terms become members of an inductively defined nominal abstract datatype.

Just as permissive nominal terms may be seen as a refinement of nominal terms, semantic nominal terms may be seen as a further evolution of permissive nominal terms.

Other work The final thread of research concerns the investigation of the relation between nominal techniques and higher-order techniques. This thread is exemplified by the work relating universal algebra over nominal terms and λ-terms [GM09e].

There exists a close relationship between nominal syntax and higher-order abstract syntax. Specifically, the relation between nominal and higher-order pattern unification has been extensively studied, first by Cheney [Che05d], and finally independently by Levy and Villaret, and Dowek and Gabbay [LV08, DGM10], who conclusively established the relationship between nominal unification and higher-order pattern unification.

This relationship was further studied by relating nominal algebra theories with λ-theories—equational logic over λ-terms [GM09e]. Non-trivial translations were presented between theories and derivations, and this translation was proved sound and complete in a suitable sense.

6.3 Future work

We survey some ideas for further work, based on the work presented in this thesis.

6.3.1 The two-level λ-calculus

Denotations. Section 3.4 presented an operational semantics for the two-level λ-calculus. However, the task of identifying a suitable denotation for the two-level λ-calculus remains. A suitable denotation could possibly be constructed using nominal sets. Another potential source of denotations would be a suitably enriched notion of Scott domain, following their use in the denotation of the λ-calculus.

An expressive higher-order logic. Through imposing Church-style simple types on the λ-calculus we obtain higher-order logic. Working analogously, imposing Church-style simple types on the two-level λ-calculus, we obtain a highly expressive higher-order logic. For instance, consider the following example axiom, the introduction rule for the universal quantifier taken from Natural Deduction, encoded in the logic:

$$\Lambda X_o.(a_\iota \# X_o \supset X_o \supset \forall a_\iota . X_o)$$

Here o is the type of propositions and ι is the type of individuals, familiar from higher-order logic. Further, Λ is the metalevel universal quantifier, belonging to the two-level higher-order logic. $\#$, short for $\lambda\#$, and $\forall$, short for $\lambda\forall$ are typed constants, corresponding to an internalisation of the nominal freshness judgment, and object level universal quantification, respectively. The axiom models 'if $a \notin fv(\phi)$ then $\phi \supset \forall a.\phi$'.

Extending the calculus with Church-style types should be straightforward. Rules for calculating freshness will become slightly different, corresponding to the additional ways two atoms of differing types can be fresh for each other. Further, the notion of congruence from Definition 3.4.2 will also need to be slightly modified, in particular the rule $(\triangleright\alpha)$:

$$\frac{\Delta \vdash r \triangleright s \quad \Delta \vdash a\#r \quad \Delta \vdash b\#r}{\Delta \vdash (b\ a)\cdot r \triangleright s}\,(\triangleright\alpha)$$

In particular, consider the swapping $(b\ a)$ which is completely unconstrained. In order to preserve subject reduction, we need to ensure that both b and a are of the same type, and to do this we need to parameterise reduction rules with a typing context. The congruence rule $(\triangleright\alpha)$ then becomes:

$$\frac{\Gamma; \Delta \vdash r \triangleright s \quad \Gamma; \Delta \vdash a\#r \quad \Gamma; \Delta \vdash b\#r \quad \Gamma \vdash \pi}{\Gamma; \Delta \vdash (b\ a)\cdot r \triangleright s}\,(\triangleright\alpha)$$

Here Γ is a typing context. The assertion $\Gamma \vdash \pi$ ensures that π is 'well behaved', mapping atoms of type ϕ to other atoms of type ϕ, only.

As a result of these changes, being fully formal about the matter, the reduction relation is sufficiently different that a new proof of confluence for the Church-style simply typed system will have to be written. This is expected to cause no real difficulties, and will essentially follow the structure of the confluence proof presented earlier, in Chapter 3.

A proof of strong normalisation is also not expected to cause any additional difficulty compared to a similar proof for the simply typed λ-calculus.

Formal relationship with other context-calculi. As highlighted in Subsection 3.5.1, a host of other context-calculi have been developed. Formally relating the two-level λ-calculus with other context-calculi, or environment calculi in the style of Sato et al's $\lambda\epsilon$, is an interesting avenue for future work.

6.3.2 Permissive nominal terms and their unification

Efficiency A large body of research by Calvès and Fernández has been undertaken investigating the efficiency of nominal unification (for instance, see [CF07, CF08a, CF08b]). A polynomial time algorithm, co-opting standard techniques from first-order unification, is currently the most efficient nominal unification algorithm [CF08b].

The algorithm of Section 4.3 is optimised for ease of mathematical reasoning and exposition, not for efficiency. However, we believe that many of the techniques pioneered by Cálves and Fernández could be reused in a permissive setting to obtain a faster permissive algorithm, hopefully with a similar asymptotic running time.

This will *not* be a trivial translation of Cálves and Fernández' work. For instance, the naïve nominal and permissive nominal unification algorithms significantly differ in their treatment of freshness constraints. Nominal unification simply returns a series of freshness constraints, and foregoes the support reduction procedure of Subsubsection 4.3.2.1. Nevertheless, we believe that the algorithms are close enough, and obviously 'morally' do the same thing, that an expectation of obtaining a polynomial time algorithm is justified.

Further, the asymptotic time complexity of nominal unification is *still* an open problem, and with it, the overall efficiency of permissive nominal unification. As mentioned, Dowek and Gabbay shows that higher-order pattern unification and nominal unification are 'essentially' the same thing (using permissive nominal techniques). A linear time algorithm for higher-order pattern unification exists [Qia93], and we believe that this, coupled with Dowek and Gabbay's work, is evidence that a linear time (permissive) nominal unification algorithm is attainable. Cálves and Fernández state that obtaining such a linear time algorithm will likely be 'significant work'. As the Dowek-Gabbay translation forces an exponential blow up in the size of a translated term, and hence simply translating permissive nominal terms into higher-order patterns for unifying, and then back again, cannot be used, Cálves and Fernández appear to be correct in their prediction.

Permissive nominal antiunification Antiunification is dual to unification: wheres unification attempts to find most general unifiers, antiunification attempts to find least general generalisers [AEMO09]. For example, given two permissive nominal terms, $\mathsf{f}(a, \mathsf{g}(b))$ and $\mathsf{f}(c, \mathsf{g}(d))$, the least general generaliser of these two terms is the term $\mathsf{f}(X^S, \mathsf{g}(Y^T))$, where X^S and Y^T are fresh variables of level two, with $a, c \in S$ and $b, d \in T$. Intuitively, the previous example demonstrates that antiunification attempts to find the 'skeleton' of a pair of terms, that is, the structure of a term common to both. If correctly computed, the inputs to an antiunification problem should be substitutative instances of the returned 'skeleton'. Indeed, a quick check will convince the reader that through applying the substitution $[X^S{:=}a] \circ [Y^T{:=}b]$ to $\mathsf{f}(X^S, \mathsf{g}(Y^T))$ we recover $\mathsf{f}(a, \mathsf{g}(b))$.

Antiunification was first introduced in the 1970s by Plotkin [Plo70] and Reynolds [Rey70] for inductive reasoning in logic programming. Until recently, antiunification gained little attention, other than occasionally being used in applications as wide-ranging as supercompilation [SG95] and setting mathematics in Maple [OSW05]. However, a recent body of research has demonstrated that antiunification is also useful for finding invariants and refactoring programs [BM08, BM09, BKZ09] (the Code Digger tool uses antiunification at the AST level to find duplicated code [Bul08]).

Nominal antiunification hasn't previously been studied, and could form an interesting topic for future work. In particular, αProlog extends Prolog's term language with nominal terms, and antiunification could form a fundamental component of a refactoring tool for that language.

There is good reason to believe that a nominal antiunification enjoys good computational properties. Research by Pfenning on antiunification in the Calculus of Constructions demonstrates that, if we restrict terms to higher-order pattern form, antiunification is decidable, and computes least general generalisers [Pfe91]. Given the recent body of work relating higher-order patterns to nominal terms, and their respective unification algorithms, this is strong evidence that the as-of-yet undiscovered nominal antiunification is also decidable, and computes least general generalisers.

Resolution for permissive nominal logic Permissive Nominal Logic (PNL) is an extension of first-order predicate logic where function symbols can bind names in their arguments [DG10]. To a first degree of approximation, PNL is first-order logic with the simple term language replaced with permissive nominal terms.

PNL was introduced as a 'foundational logic', a framework for encoding other logics. In particular, the use of permissive nominal terms allows straight forward encodings of logics and languages that feature name binding constructs. PNL *could* serve as the meta-logic for a generic proof assistant, similar to the role that higher-order logic plays in Isabelle.

A significant first step toward using PNL as a meta-logic would be the implementation of a resolution procedure. One of Isabelle's advances over the LCF line of proof assistants was the implementation of Natural Deduction via resolution [Pau85]. Whereas LCF implemented each inference rule in the logic by a pair of functions: one for forward reasoning and another for backwards reasoning, Isabelle implements inference rules as Horn clauses. Applying an inference rule implemented as a Horn Clause involves unifying the current open goal with the head of the clause, and opening the premises of the unified clause as new subgoals. This process allows Isabelle to encode a wide range of logics, when compared to the LCF approach.

A proof assistant based on PNL would presumably work in a similar fashion. Any implementation could use the unification code presented in Chapter 5 as a base.

6.3.3 Implementing permissive nominal terms

A library for structural operational semantics Chapter 5 covers a prototype implementation of permissive nominal unification. However, we believe that permissive nominal techniques could be used to develop an ergonomic library for animating structural operational semantics (SOS), embedded in a language such as Haskell. Alternatively, permissive nominal techniques could be incorporated into an existing tool, such as Maude [CELM00], for exploring structural operational semantics.

The first step toward this goal would be the investigation of permissive nominal rewriting. Gabbay and Fernández have already performed a large amount of research into nominal rewriting. We believe that this research will transfer quite readily to the permissive nominal world, yet benefit from the improved reasoning properties that permissive nominal terms enjoy over their traditional counterparts.

Further, a type system, similar to the one developed by Gabbay and Fernández for nominal terms could be added to permissive nominal terms. This will add a certain level of type safety to the encodings of operational rules.

Finally, a method for embedding reduction rules into Haskell, such as the following, will need to be investigated:

$$\frac{\langle a_0, \sigma\rangle \to n_0 \quad \langle a_1, \sigma\rangle \to n_1 \quad (n = n_0 \cdot n_1)}{\langle a_0 \times a_1, \sigma\rangle \to n}$$

Rule taken from Winskel's book [Win93, pg 14].

Here, we envision a translation of the rule above into the proposed tool representing $\times$ as a permissive nominal term former (possibly typed) taking two arguments, a_0 and a_1 are permissive unknowns (meta-variables). Integer literals are wrapped in 'silent' term formers, and functions are provided to extract integers from these wrappers, and inject them back in, allowing the user to perform operations on computed values (for instance, the multiplication $n = n_0 \cdot n_1$). Permissive nominal rewriting, employing permissive nominal unification, is used to rewrite the current state using encoded operational rules. We envision using Haskell's type system, particularly polymorphism, to carry around state, here represented by σ, within rules. What really sets the nominal approach apart is the promise of handling SOS rules involving name binders seamlessly. For instance, handling the following rule could potentially be simplified using a (permissive) nominal approach:

$$\frac{}{\mathbf{rec}\ f.(\lambda x.t) \to \lambda x.t[\mathbf{rec} f.\lambda x.t/f]}$$

Rule taken from Winskel's book [Win93, pg 257].

The advantage of providing a Haskell library, versus writing a special purpose language (such as Lakin's MLSOS/αML [LP08]) is the ability to use Haskell's predefined data structures for the complex book keeping sometimes needed in SOS. Further, users will be able to reuse their existing Haskell knowledge[1], as opposed to having to learn a new language and associated set of libraries. In particular, we view the improved reasoning properties of permissive nominal terms as what would make such a project viable. Every rule needn't explicitly manipulate freshness contexts; book keeping of this sort is kept hidden from the programmer.

Fernández-Gabbay polymorphic type system Further possible ideas are an implementation of the Fernández-Gabbay polymorphic typing system for nominal terms [FG07a]. Existing 'hooks' into the PNT Frontend for defining types already exist; the `termformer` ascription block, for instance, is intended as a block for assigning types to terms.

[1]Providing, of course, that they have any in the first place.

APPENDIX A

Additional proofs: the two-level λ-calculus

Proof of Lemma 3.2.5.

Proof. By induction on r.

- The case a. By Definition 3.2.4 we have $a[X{:=}\pi{\cdot}X] \equiv a$. The result follows.
- The case c. By Definition 3.2.4 we have $\mathsf{c}[X{:=}\pi{\cdot}X] \equiv \mathsf{c}$. The result follows.
- The case $\pi'{\cdot}X$. By Definition 3.2.3 we have $fV(\pi'{\cdot}X) = \{X\}$. By Definition 3.2.4 we have $fV((\pi'{\cdot}X)[X{:=}\pi{\cdot}X]) = fV((\pi'{\circ}\pi){\cdot}X)$. By Definition 3.2.3 we have $fV((\pi'{\circ}\pi){\cdot}X) = \{X\}$. The result follows.
- The case rs. By Definition 3.2.4 we have $fV((rs)[X{:=}\pi{\cdot}X]) = fV((r[X{:=}\pi{\cdot}X])(s[X{:=}\pi{\cdot}X]))$. By Definition 3.2.3 we have $fV((r[X{:=}\pi{\cdot}X])(s[X{:=}\pi{\cdot}X])) = fV(r[X{:=}\pi{\cdot}X]) \cup fV(s[X{:=}\pi{\cdot}X])$. By inductive hypotheses $fV(r[X{:=}\pi{\cdot}X]) \cup fV(s[X{:=}\pi{\cdot}X]) = fV(r) \cup fV(s)$. By Definition 3.2.3 we have $fV(r) \cup fV(s) = fV(rs)$. The result follows.
- The case $\lambda a.r$. By Definition 3.2.4 we have $fV((\lambda a.r)[X{:=}\pi{\cdot}X]) = fV(\lambda a.(r[X{:=}\pi{\cdot}X]))$. By Definition 3.2.3 we have $fV(\lambda a.(r[X{:=}\pi{\cdot}X])) = fV(r[X{:=}\pi{\cdot}X])$. By inductive hypothesis $fV(r[X{:=}\pi{\cdot}X]) = fV(r)$. By Definition 3.2.3 we have $fV(r) = fV(\lambda a.r)$. The result follows.
- The case $\lambda X.r$. By Definition 3.2.4 we have $(\lambda X.r)[X{:=}\pi{\cdot}X] \equiv \lambda X.r$. The result follows.
- The case $\lambda Y.s$. By Definition 3.2.4 we have $fV((\lambda Y.s)[X{:=}\pi{\cdot}X]) = fV(\lambda Y.(s[X{:=}\pi{\cdot}X]))$. By Definition 3.2.3 we have $fV(\lambda Y.(s[X{:=}\pi{\cdot}X])) = fV(s[X{:=}\pi{\cdot}X]) \setminus \{Y\}$. By inductive hypothesis $fV(s[X{:=}\pi{\cdot}X]) \setminus \{Y\} = fV(s) \setminus \{Y\}$. By Definition 3.2.3 we have $fV(s) \setminus \{Y\} = fV(\lambda Y.s)$. The result follows.

⊠

Proof of Lemma 3.2.6.

Proof. By induction on r.

- The case a. By Definition 3.2.3 we have $fV(a) = \emptyset$. The result follows.
- The case c. By Definition 3.2.4 we have $\pi{\cdot}\mathsf{c} \equiv \mathsf{c}$. By Definition 3.2.3 we have $fV(\mathsf{c}) = \emptyset$. The result follows.
- The case $\pi'{\cdot}X$. By Definition 3.2.3 we have $fV(\pi'{\cdot}X) = \{X\}$. By Definition 3.2.4 we have $fV(\pi{\cdot}(\pi'{\cdot}X)) = fV((\pi{\circ}\pi'){\cdot}X)$. By Definition 3.2.3 we have $fV((\pi{\circ}\pi'){\cdot}X) = \{X\}$. The result follows.
- The case rs. By Definition 3.2.4 we have $fV(\pi{\cdot}rs) \equiv fV((\pi{\cdot}r)(\pi{\cdot}s))$. By Definition 3.2.3 we have $fV((\pi{\cdot}r)(\pi{\cdot}s)) = fV(\pi{\cdot}r) \cup fV(\pi{\cdot}s)$. By inductive hypotheses $fV(\pi{\cdot}r) \cup fV(\pi{\cdot}s) = fV(r) \cup fV(s)$. By Definition 3.2.3 we have $fV(r) \cup fV(s) = fV(rs)$. The result follows.

- The case $\lambda a.r$. By Definition 3.2.4 we have $fV(\pi{\cdot}\lambda a.r) = fV(\lambda\pi(a).(\pi{\cdot}r))$. By Definition 3.2.3 we have $fV(\lambda\pi(a).(\pi{\cdot}r)) = fV(\pi{\cdot}r)$. By inductive hypothesis $fV(\pi{\cdot}r) = fV(r)$. By Definition 3.2.3 we have $fV(r) = fV(\lambda a.r)$. The result follows.
- The case $\lambda X.r$. By Definition 3.2.4 we have $fV(\pi{\cdot}\lambda X.r) = fV(\lambda X.\pi{\cdot}r[X{:=}\pi^{-1}{\cdot}X])$. By Definition 3.2.3 we have $fV(\lambda X.\pi{\cdot}r[X{:=}\pi^{-1}{\cdot}X]) = fV(\pi{\cdot}r[X{:=}\pi^{-1}{\cdot}X])\setminus\{X\}$. By Lemma 3.2.5 we have $fV(\pi{\cdot}r[X{:=}\pi^{-1}{\cdot}X]) \setminus \{X\} = fV(\pi{\cdot}r) \setminus \{X\}$. By inductive hypothesis $fV(\pi{\cdot}r) \setminus \{X\} = fV(r) \setminus \{X\}$. By Definition 3.2.3 we have $fV(r) \setminus \{X\} = fV(\lambda X.r)$. The result follows.

⊠

Proof of Lemma 3.2.8.

Proof. By induction on r.

- The case a. By Definition 3.2.4 we have $a[X{:=}\pi{\cdot}Y] \equiv a$. The result follows.
- The case c. By Definition 3.2.4 we have $\mathsf{c}[X{:=}\pi{\cdot}Y] \equiv \mathsf{c}$. The result follows.
- The case $\pi'{\cdot}X$. By Definition 3.2.7 we have $\mathit{depth}(\pi'{\cdot}X) = 1$. By Definition 3.2.4 we have $\mathit{depth}((\pi'{\cdot}X)[X{:=}\pi{\cdot}Y]) = \mathit{depth}((\pi'{\circ}\pi){\cdot}Y)$. By Definition 3.2.7 we have $\mathit{depth}((\pi{\circ}\pi'){\cdot}Y) = 1$. The result follows.
- The case $\pi'{\cdot}Z$. By Definition 3.2.4 we have $\mathit{depth}((\pi'{\cdot}Z)[X{:=}\pi{\cdot}Y]) \equiv \mathit{depth}(\pi'{\cdot}Z)$. The result follows.
- The case rs. By Definition 3.2.7 we have $\mathit{depth}(rs) = \mathit{depth}(r) + \mathit{depth}(s)$. By inductive hypotheses we have $\mathit{depth}(r[X{:=}\pi{\cdot}Y]) + \mathit{depth}(s[X{:=}\pi{\cdot}Y])$. By Definition 3.2.7 we have $\mathit{depth}(r[X{:=}\pi{\cdot}Y]) + \mathit{depth}(s[X{:=}\pi{\cdot}Y]) = \mathit{depth}((r[X{:=}\pi{\cdot}Y])(s[X{:=}\pi{\cdot}Y]))$. By Definition 3.2.4 we have $\mathit{depth}((r[X{:=}\pi{\cdot}Y])(s[X{:=}\pi{\cdot}Y])) = \mathit{depth}((rs)[X{:=}\pi{\cdot}Y])$. The result follows.
- The case $\lambda a.r$. By Definition 3.2.7 we have $\mathit{depth}(\lambda a.r) = 1 + \mathit{depth}(r)$. By inductive hypothesis $1+\mathit{depth}(r) = 1+\mathit{depth}(r[X{:=}\pi{\cdot}Y])$. By Definition 3.2.7 we have $1+\mathit{depth}(r[X{:=}\pi{\cdot}Y]) = \mathit{depth}(\lambda a.(r[X{:=}\pi{\cdot}Y]))$. By Definition 3.2.4 we have $\mathit{depth}(\lambda a.(r[X{:=}\pi{\cdot}Y])) = \mathit{depth}((\lambda a.r)[X{:=}\pi{\cdot}Y])$. The result follows.
- The case $\lambda X.r$. By Definition 3.2.4 we have $\mathit{depth}((\lambda X.r)[X{:=}\pi{\cdot}Y]) \equiv \mathit{depth}(\lambda X.r)$. The result follows.
- The case $\lambda Z.t$. Suppose $Z \neq Y$ and $Z \neq X$, which can always be guaranteed by renaming. By Definition 3.2.7 we have $\mathit{depth}(\lambda Z.t) = 1 + \mathit{depth}(t)$. By inductive hypothesis $1 + \mathit{depth}(t) = 1 + \mathit{depth}(t[X{:=}\pi{\cdot}Y])$. By Definition 3.2.7 we have $1 + \mathit{depth}(t[X{:=}\pi{\cdot}Y]) = \mathit{depth}(\lambda Z.(t[X{:=}\pi{\cdot}Y]))$. By Definition 3.2.4 we have $\mathit{depth}(\lambda Z.(t[X{:=}\pi{\cdot}Y])) = \mathit{depth}((\lambda Z.t)[X{:=}\pi{\cdot}Y])$. The result follows.

⊠

Proof of Lemma 3.2.9.

Proof. By induction on r.

- The case a. By Definition 3.2.7 we have $\mathit{depth}(a) = 1$. By Definition 3.2.4 we have $\mathit{depth}(\pi\cdot a) = \mathit{depth}(\pi(a))$. By Definition 3.2.7 we have $\mathit{depth}(\pi(a)) = 1$. The result follows.
- The case c. By Definition 3.2.4 we have $\mathit{depth}(\pi\cdot\mathsf{c}) \equiv \mathit{depth}(\mathsf{c})$. The result follows.
- The case $\pi'\cdot X$. By Definition 3.2.7 we have $\mathit{depth}(\pi'\cdot X) = 1$. By Definition 3.2.4 we have $\mathit{depth}(\pi\cdot(\pi'\cdot X)) = \mathit{depth}((\pi\circ\pi')\cdot X)$. By Definition 3.2.7 we have $\mathit{depth}((\pi\circ\pi')\cdot X) = 1$. The result follows.
- The case rs. By Definition 3.2.7 we have $\mathit{depth}(rs) = \mathit{depth}(r) + \mathit{depth}(s)$. By inductive hypothesis $\mathit{depth}(r) + \mathit{depth}(s) = \mathit{depth}(\pi\cdot r) + \mathit{depth}(\pi\cdot s)$. By Definition 3.2.7 we have $\mathit{depth}(\pi\cdot r) + \mathit{depth}(\pi\cdot s) = \mathit{depth}((\pi\cdot r)(\pi\cdot s))$. By Definition 3.2.4 we have $\mathit{depth}((\pi\cdot r)(\pi\cdot s)) = \mathit{depth}(\pi\cdot rs)$. The result follows.
- The case $\lambda a.r$. By Definition 3.2.7 we have $\mathit{depth}(\lambda a.r) = 1 + \mathit{depth}(r)$. By inductive hypothesis $1 + \mathit{depth}(r) = 1 + \mathit{depth}(\pi\cdot r)$. By Definition 3.2.7 we have $1 + \mathit{depth}(\pi\cdot r) = \mathit{depth}(\lambda\pi(a).(\pi\cdot r))$. By Definition 3.2.4 we have $\mathit{depth}(\lambda\pi(a).(\pi\cdot r)) = \mathit{depth}(\pi\cdot\lambda a.r)$. The result follows.
- The case $\lambda X.r$. By Definition 3.2.7 we have $\mathit{depth}(\lambda X.r) = 1 + \mathit{depth}(r)$. By inductive hypothesis $1 + \mathit{depth}(r) = 1 + \mathit{depth}(\pi\cdot r)$. By Lemma 3.2.8 $1 + \mathit{depth}(\pi\cdot r) = 1 + \mathit{depth}(\pi\cdot r[X{:=}\pi^{-1}\cdot X])$. By Definition 3.2.7 $1 + \mathit{depth}(\pi\cdot r[X{:=}\pi^{-1}\cdot X]) = \mathit{depth}(\lambda X.(\pi\cdot r[X{:=}\pi^{-1}\cdot X]))$. By Definition 3.2.4 $\mathit{depth}(\lambda X.(\pi\cdot r[X{:=}\pi^{-1}\cdot X])) = \mathit{depth}(\pi\cdot\lambda.r)$. The result follows.

⊠

Proof of Lemma 3.2.10.

Proof. By induction on r.

- The case a. By Definition 3.2.4 we have $a[X{:=}\pi\cdot X][X{:=}\pi'\cdot X] \equiv a$. By Definition 3.2.4 we have $a[X{:=}(\pi\circ\pi')\cdot X] \equiv a$. The result follows.
- The case c. By Definition 3.2.4 we have $\mathsf{c}[X{:=}\pi\cdot X][X{:=}\pi'\cdot X] \equiv \mathsf{c}$. By Definition 3.2.4 we have $\mathsf{c}[X{:=}(\pi\circ\pi')\cdot X] \equiv \mathsf{c}$. The result follows.
- The case $\pi''\cdot X$. By Definition 3.2.4 we have $(\pi''\cdot X)[X{:=}(\pi\circ\pi')\cdot X] \equiv \pi''\cdot((\pi\circ\pi')\cdot X)$. By Definition 3.2.4 we have $\pi''\cdot((\pi\circ\pi')\cdot X) \equiv \pi''\cdot(\pi\cdot(\pi'\cdot X))$. By Definition 3.2.4 we have $\pi''\cdot(\pi\cdot(\pi'\cdot X)) \equiv (\pi''\cdot(\pi\cdot X))[X{:=}\pi'\cdot X]$. By Definition 3.2.4 we have $(\pi''\cdot(\pi\cdot X))[X{:=}\pi'\cdot X] \equiv (\pi''\cdot X)[X{:=}\pi\cdot X][X{:=}\pi'\cdot X]$. The result follows.
- The case $\pi''\cdot Y$. By Definition 3.2.4 we have $(\pi''\cdot Y)[X{:=}\pi\cdot X][X{:=}\pi'\cdot X] \equiv \pi''\cdot Y$. By Definition 3.2.4 we have $(\pi''\cdot Y)[X{:=}(\pi\circ\pi')\cdot X] \equiv \pi''\cdot Y$. The result follows.
- The case rs. By Definition 3.2.4 we have $(rs)[X{:=}\pi\circ\pi'\cdot X] \equiv (r[X{:=}\pi\circ\pi'\cdot X])(s[X{:=}\pi\circ\pi'\cdot X])$. By inductive hypothesis $(r[X{:=}\pi\circ\pi'\cdot X])(s[X{:=}\pi\circ\pi'\cdot X]) \equiv (r[X{:=}\pi\cdot X][X{:=}\pi'\cdot X])(s[X{:=}\pi\cdot X][X{:=}\pi'\cdot X])$. By Definition 3.2.4 $(r[X{:=}\pi\cdot X][X{:=}\pi'\cdot X])(s[X{:=}\pi\cdot X][X{:=}\pi'\cdot X]) \equiv ((r[X{:=}\pi\cdot X])(s[X{:=}\pi\cdot X]))[X{:=}\pi'\cdot X])$. By Definition 3.2.4 we have $((r[X{:=}\pi\cdot X])(s[X{:=}\pi\cdot X]))[X{:=}\pi'\cdot X]) \equiv (rs)[X{:=}\pi\cdot X][X{:=}\pi'\cdot X]$. The result follows.
- The case $\lambda a.r$. By Definition 3.2.4 we have $(\lambda a.r)[X{:=}\pi\circ\pi'\cdot X] \equiv (\lambda a.r[X{:=}\pi\circ\pi'\cdot X])$. By inductive hypothesis $(\lambda a.r[X{:=}\pi\circ\pi'\cdot X]) \equiv \lambda a.(r[X{:=}\pi\cdot X][X{:=}\pi'\cdot X])$. By Definition 3.2.4 we have $\lambda a.(r[X{:=}\pi\cdot X][X{:=}\pi'\cdot X]) \equiv (\lambda a.(r[X{:=}\pi\cdot X]))[X{:=}\pi'\cdot X]$. By Definition 3.2.4 we have $(\lambda a.(r[X{:=}\pi\cdot X]))[X{:=}\pi'\cdot X] \equiv (\lambda a.r)[X{:=}\pi\cdot X][X{:=}\pi'\cdot X]$. The result follows.

- The case $\lambda X.r$. By Definition 3.2.4 we have $(\lambda X.r)[X{:=}\pi\cdot X][X{:=}\pi'\cdot X] \equiv \lambda X.r$. By Definition 3.2.4 we have $(\lambda X.r)[X{:=}(\pi\circ\pi')\cdot X] \equiv \lambda X.r$. The result follows.
- The case $\lambda Y.s$. Suppose $Y \notin fV(\pi\cdot X) \cup fV(\pi'\cdot X) \cup fV((\pi\circ\pi')\cdot X)$. This side-condition can always be guaranteed by renaming. By Definition 3.2.4 $(\lambda Y.s)[X{:=}(\pi\circ\pi')\cdot X] \equiv \lambda Y.(s[X{:=}(\pi\circ\pi')\cdot X])$. By inductive hypothesis $\lambda Y.(s[X{:=}(\pi\circ\pi')\cdot X]) \equiv \lambda Y.(s[X{:=}\pi\cdot X][X{:=}\pi'\cdot X])$. By Definition 3.2.4 we have $\lambda Y.(s[X{:=}\pi\cdot X][X{:=}\pi'\cdot X]) \equiv (\lambda Y.(s[X{:=}\pi\cdot X]))[X{:=}\pi'\cdot X]$. By Definition 3.2.4 we have $(\lambda Y.(s[X{:=}\pi\cdot X]))[X{:=}\pi'\cdot X] \equiv (\lambda Y.s)[X{:=}\pi\cdot X][X{:=}\pi'\cdot X]$. The result follows.

⊠

Proof of Lemma 3.2.11.

Proof. By induction on r.

- The case a. By Definition 3.2.4 we have $a[X{:=}id\cdot X] \equiv a$. The result follows.
- The case c. By Definition 3.2.4 we have $\mathsf{c}[X{:=}id\cdot X] \equiv \mathsf{c}$. The result follows.
- The case $\pi\cdot X$. By Definition 3.2.4 we have $(\pi\cdot X)[X{:=}id\cdot X] \equiv (\pi\circ id)\cdot X$. It is a fact that $\pi\circ id = id$. The result follows.
- The case $\pi\cdot Y$. By Definition 3.2.4 we have $(\pi\cdot Y)[X{:=}id\cdot X] \equiv \pi\cdot Y$. The result follows.
- The case rs. By Definition 3.2.4 we have $(rs)[X{:=}id\cdot X] \equiv (r[X{:=}id\cdot X])(s[X{:=}id\cdot X])$. By inductive hypotheses $(r[X{:=}id\cdot X])(s[X{:=}id\cdot X]) \equiv rs$. The result follows.
- The case $\lambda a.r$. By Definition 3.2.4 we have $(\lambda a.r)[X{:=}id\cdot X] \equiv \lambda a.(r[X{:=}id\cdot X])$. By inductive hypothesis $\lambda a.(r[X{:=}id\cdot X]) \equiv \lambda a.r$. The result follows.
- The case $\lambda X.r$. By Definition 3.2.4 we have $(\lambda X.r)[X{:=}id\cdot X] \equiv \lambda X.r$. The result follows.
- The case $\lambda Y.s$. By Definition 3.2.4 we have $(\lambda Y.s)[X{:=}id\cdot X] \equiv \lambda Y.(s[X{:=}id\cdot X])$. By inductive hypothesis $\lambda Y.(s[X{:=}id\cdot X]) \equiv \lambda Y.s$. The result follows.

⊠

Proof of Lemma 3.2.12.

Proof. By induction on r.

- The case a. By Definition 3.2.4 we have $id\cdot a \equiv id(a)$. By definition $id(a) = a$. The result follows.
- The case c. By Definition 3.2.4 we have $id\cdot\mathsf{c} \equiv \mathsf{c}$. The result follows.
- The case $\pi\cdot X$. By Definition 3.2.4 we have $id\cdot(\pi\cdot X) \equiv (id\circ\pi)\cdot X$. It is a fact that $id\circ\pi = \pi$. The result follows.
- The case rs. By Definition 3.2.4 we have $id\cdot rs \equiv (id\cdot r)(id\cdot s)$. By inductive hypotheses $(id\cdot r)(id\cdot s) \equiv rs$. The result follows.
- The case $\lambda a.r$. By Definition 3.2.4 we have $id\cdot\lambda a.r \equiv \lambda id(a).(id\cdot r)$. By inductive hypothesis $\lambda id(a).(id\cdot r) \equiv \lambda id(a).r$. By definition $id(a) = a$. The result follows.

- The case $\lambda X.r$. By Definition 3.2.4 we have $id{\cdot}\lambda X.r \equiv \lambda X.((id{\cdot}r)[X{:=}id^{-1}{\cdot}X])$. It is a fact that $id^{-1} = id$. By Lemma 3.2.11 we have $\lambda X.((id{\cdot}r)[X{:=}id{\cdot}X]) \equiv \lambda X.(id{\cdot}r)$. By inductive hypothesis $\lambda X.(id{\cdot}r) \equiv \lambda X.r$. The result follows.

⊠

Proof of Lemma 3.3.10.

Proof. By induction on the derivation of $\Delta \vdash a\#r$.

- The case $(\#\mathbf{b})$. Using $(\#\mathbf{b})$ we obtain $\Delta, b\#Y \vdash a\#c$. The result follows.
- The case $(\#\mathsf{c})$. Using $(\#\mathsf{c})$ we obtain $\Delta, b\#Y \vdash a\#\mathsf{c}$. The result follows.
- The case $(\#\mathbf{X})$. Suppose $\pi^{-1}(a)\#X \in \Delta$ therefore $\Delta \vdash a\#\pi{\cdot}X$ by $(\#\mathbf{X})$. It is a fact that $\pi^{-1}(a)\#X \in \Delta, b\#Y$. Using $(\#\mathbf{X})$ we obtain $\Delta, b\#Y \vdash a\#\pi{\cdot}X$. The result follows.
- The case $(\#\mathbf{rs})$. Suppose $\Delta \vdash a\#r$ and $\Delta \vdash a\#s$. By inductive hypotheses we have $\Delta, b\#Y \vdash a\#r$ and $\Delta, b\#Y \vdash a\#s$. Using $(\#\mathbf{rs})$ we obtain $\Delta, b\#Y \vdash a\#rs$. The result follows.
- The case $(\#\lambda\mathbf{a})$. Using $(\#\lambda\mathbf{a})$ we obtain $\Delta, b\#Y \vdash a\#\lambda a.r$. The result follows.
- The case $(\#\lambda\mathbf{b})$. Suppose $\Delta \vdash a\#t$. By inductive hypothesis $\Delta, b\#Y \vdash a\#t$. Using $(\#\lambda\mathbf{b})$ we obtain $\Delta, b\#Y \vdash a\#\lambda c.t$. The result follows.
- The case $(\#\lambda\mathbf{X})$. Suppose $\Delta, a\#X \vdash \pi(a)\#\pi{\cdot}r$. By inductive hypothesis $\Delta, a\#X, b\#Y \vdash \pi(a)\#\pi{\cdot}r$. Using $(\#\lambda\mathbf{X})$ we obtain $\Delta, b\#Y \vdash a\#\lambda X.r$. The result follows.

⊠

Proof of Lemma 3.4.5.

Proof. By induction on r.

- The case a. By Definition 3.2.4 we have $a\theta \equiv a$. The result follows.
- The case c. By Definition 3.2.4 we have $\mathsf{c}\theta \equiv \mathsf{c}$. The result follows.
- The case $\pi{\cdot}X$. By Definition 3.4.1 we have $level(\pi{\cdot}X) = 2$. There is nothing to prove.
- The case rs. By Definition 3.4.1 we have $level(rs) = 1$ implies $level(r) = 1$ and $level(s) = 1$. By Definition 3.4.1 we have $level(rs) = \max(level(r), level(s))$. By inductive hypothesis $\max(level(r), level(s)) = \max(level(r\theta), level(s\theta))$. By Definition 3.4.1 we have $\max(level(r\theta),$
$level(s\theta)) = level((r\theta)(s\theta))$. By Definition 3.2.4 we have $level((r\theta)(s\theta)) = level((rs)\theta)$. The result follows.
- The case $\lambda a.r$. By Definition 3.4.1 we have $level(\lambda a.r) = 1$ implies $level(r) = 1$. By Definition 3.4.1 we have $level(\lambda a.r) = level(r)$. By inductive hypothesis $level(r) = level(r\theta)$. By Definition 3.4.1 we have $level(r\theta) = level(\lambda a.(r\theta))$. By Definition 3.2.4 we have $level(\lambda a.(r\theta)) = level((\lambda a.r)\theta)$. The result follows.
- The case $\lambda X.r$. By Definition 3.4.1 we have $level(\lambda X.r) = 2$. There is nothing to prove.

⊠

Proof of Lemma 3.4.7.

Proof. By induction on r.

- The case a. By Definition 3.4.1 we have $level(a) = 1$. By Definition 3.2.4 we have $level(\pi\cdot a) = level(\pi(a))$. By Definition 3.4.1 we have $level(\pi(a)) = 1$. The result follows.
- The case c. By Definition 3.2.4 we have $\pi\cdot\mathsf{c} \equiv \mathsf{c}$. The result follows.
- The case $\pi'\cdot X$. By Definition 3.4.1 we have $level(\pi'\cdot X) = 1$. By Lemma 3.2.13 we have $level(\pi\cdot(\pi'\cdot X)) = level((\pi\circ\pi')\cdot X)$. By Definition 3.4.1 we have $level(\pi'\cdot X) = level((\pi\circ\pi')\cdot X)$. The result follows.
- The case rs. By Definition 3.4.1 we have $level(rs) = \max(level(r), level(s))$. By inductive hypothesis $\max(level(r), level(s)) = \max(level(\pi\cdot r), level(\pi\cdot s))$. By Definition 3.4.1 we have $\max(level(\pi\cdot r), level(\pi\cdot s)) = level((\pi\cdot r)(\pi\cdot s))$. By Definition 3.2.4 we have $level((\pi\cdot r)(\pi\cdot s)) = level(\pi\cdot rs)$. The result follows.
- The case $\lambda a.r$. By Definition 3.4.1 we have $level(\lambda a.r) = level(r)$. By inductive hypothesis $level(r) = level(\pi\cdot r)$. By Definition 3.4.1 we have $level(\pi\cdot r) = level(\lambda\pi(a).(\pi\cdot r))$. By Definition 3.2.4 we have $level(\lambda\pi(a).(\pi\cdot r)) = level(\pi\cdot\lambda a.r)$. The result follows.
- The case $\lambda X.r$. By Definition 3.4.1 we have $level(\lambda X.r) = 2$. By Definition 3.2.4 we have $level(\pi\cdot\lambda X.r) = level(\lambda X.(\pi\cdot r[X{:=}\pi^{-1}\cdot X]))$. By Definition 3.4.1 we have $level(\lambda X.((\pi\cdot r)[X{:=}\pi^{-1}\cdot X])) = 2$. The result follows.

⊠

Proof of Lemma 3.4.17.

Proof. By induction on $depth(r)$.

- The case a. By Definition 3.4.12 we have $a[a{:=}t] \equiv t$. By assumption $\Delta \vdash a\#t$. The result follows.
- The case b. By Definition 3.4.12 we have $b[a{:=}t] \equiv b$. Using (**#b**) we obtain $\Delta \vdash a\#b$. The result follows.
- The case c. By Definition 3.4.12 we have $\mathsf{c}[a{:=}t] \equiv \mathsf{c}$. Using (#c) we obtain $\Delta \vdash a\#\mathsf{c}$. The result follows.
- The case $\pi\cdot X$. There are two cases:
  - The case $\Delta \vdash a\#\pi\cdot X$. By Definition 3.4.12 we have $(\pi\cdot X)[a{:=}t] \equiv \pi\cdot X$. The result follows.
  - The case $\Delta \nvdash a\#\pi\cdot X$. By Definition 3.4.12 we have $(\pi\cdot X)[a{:=}t] \equiv (\pi\cdot X)[a{\mapsto}t]$. By definition $(\pi\cdot X)[a{\mapsto}t] \equiv (\lambda a.(\pi\cdot X))t$. Using (**#rs**) and (**#λa**) we obtain $\Delta \vdash a\#(\lambda a.(\pi\cdot X))t$. The result follows.
- The case rs. There are multiple cases:
  - The case $level(r) = 1$. By Definition 3.4.12 we have $(rs)[a{:=}t] \equiv (r[a{:=}t])(s[a{:=}t])$. By inductive hypotheses $\Delta \vdash a\#r[a{:=}t]$ and $\Delta \vdash a\#s[a{:=}t]$. Using (**#rs**) we obtain $\Delta \vdash a\#(r[a{:=}t])(s[a{:=}t])$. The result follows.

  - The case $\Delta \vdash a\#s$. By Definition 3.4.12 we have $(rs)[a{:=}t] \equiv r(s[a{:=}t])$. By inductive hypothesis $\Delta \vdash a\#s[a{:=}t]$. Using $(\#\mathbf{rs})$ we obtain $\Delta \vdash a\#r(s[a{:=}t])$. The result follows.
  - The case $level(r) = 2$ and $\Delta \nvdash a\#s$. There is nothing to prove as $\Delta \vdash a\#s$ by assumption.
- The case $\lambda a.r$. By Definition 3.4.12 we have $(\lambda a.r)[a{:=}t] \equiv \lambda a.r$. Using $(\#\lambda\mathbf{a})$ we obtain $\Delta \vdash a\#\lambda a.r$. The result follows.
- The case $\lambda b.s$. There are three cases:
  - The case $\Delta \vdash b\#t$. By Definition 3.4.12 we have $(\lambda b.s)[a{:=}t] \equiv \lambda b.(s[a{:=}t])$. By inductive hypothesis $\Delta \vdash a\#s[a{:=}t]$. Using $(\#\lambda\mathbf{b})$ we obtain $\Delta \vdash a\#\lambda b.(s[a{:=}t])$. The result follows.
  - The case $\Delta \nvdash b\#t$ with Δ containing sufficient freshness. Suppose $\Delta \nvdash b\#t$. By Definition 3.4.12 we have $(\lambda b.s)[a{:=}t] \equiv \lambda c.(((c\ b){\cdot}s)[a{:=}t])$ where c is a fresh atom, distinct from a and b, chosen so that $\Delta \vdash c\#s$ and $\Delta \vdash c\#t$. By Lemma 3.2.9 we have $depth((c\ b){\cdot}s) = depth(s)$. By inductive hypothesis $\Delta \vdash a\#((c\ b){\cdot}s)[a{:=}t]$. Using $(\#\lambda\mathbf{b})$ we obtain $\Delta \vdash a\#\lambda c.(((c\ b){\cdot}s)[a{:=}t])$. The result follows.
  - The case $\Delta \nvdash b\#t$ with Δ not containing sufficient freshness. By Definition 3.4.12 we have $(\lambda b.s)[a{:=}t] \equiv (\lambda b.s)[a{\mapsto}t]$. By definition $(\lambda b.s)[a{\mapsto}t] \equiv (\lambda a.(\lambda b.s))t$. Using $(\#\mathbf{rs})$ and $(\#\lambda\mathbf{a})$ we obtain $\Delta \vdash a\#(\lambda a.(\lambda b.s))t$. The result follows.
- The case $\lambda X.r$. Suppose $X \notin fV(t)$ which can be guaranteed. By Definition 3.4.12 we have $(\lambda X.r)[a{:=}t] \equiv \lambda X.(r[a{:=}t])$. By inductive hypothesis $\Delta, a\#X \vdash a\#r[a{:=}t]$. Using $(\#\lambda\mathbf{X})$ we obtain $\Delta \vdash a\#\lambda X.(r[a{:=}t])$. The result follows.

⊠

Proof of Lemma 3.4.18.

Proof. By induction on $depth(r)$.

- The case b. By Definition 3.4.12 we have $b[b{:=}u] \equiv u$. The result follows.
- The case c. By Definition 3.4.12 we have $c[b{:=}u] \equiv c$. The result follows.
- The case c. By Definition 3.4.12 we have $\mathsf{c}[b{:=}u] \equiv \mathsf{c}$. The result follows.
- The case $\pi{\cdot}X$. There are two cases:
  - The case $\Delta \vdash b\#\pi{\cdot}X$. By Definition 3.4.12 we have $(\pi{\cdot}X)[b{:=}u] \equiv \pi{\cdot}X$. The result follows.
  - The case $\Delta \nvdash b\#\pi{\cdot}X$. By Definition 3.4.12 we have $(\pi{\cdot}X)[b{:=}u] \equiv (\pi{\cdot}X)[b{\mapsto}u]$. By definition $(\pi{\cdot}X)[b{\mapsto}u] \equiv (\lambda b.(\pi{\cdot}X))u$. Using $(\#\lambda\mathbf{a})$ and $(\#\mathbf{rs})$ we obtain $\Delta \vdash a\#(\lambda b.(\pi{\cdot}X))u$. The result follows.
- The case rs. There are multiple cases:
  - The case $level(r) = 1$. Suppose $\Delta \vdash a\#rs$ therefore $\Delta \vdash a\#r$ and $\Delta \vdash a\#s$. By Definition 3.4.12 we have $(rs)[b{:=}u] \equiv (r[b{:=}u])(s[b{:=}u])$. By inductive hypotheses $\Delta \vdash a\#r[b{:=}u]$ and $\Delta \vdash a\#s[b{:=}u]$. Using $(\#\mathbf{rs})$ we obtain $\Delta \vdash a\#((r[b{:=}u])(s[b{:=}u]))$. The result follows.

- The case $\Delta \vdash a\#s$. Suppose $\Delta \vdash a\#rs$ so that $\Delta \vdash a\#r$. By Definition 3.4.12 we have $(rs)[b{:=}u] \equiv r(s[b{:=}u]])$. By inductive hypothesis $\Delta \vdash a\#s[b{:=}u]$. Using $(\#\mathbf{rs})$ we obtain $\Delta \vdash a\#r(s[b{:=}u])$. The result follows.
- The case $level(r) = 2$ and $\Delta \nvdash a\#s$. There is nothing to prove, as $\Delta \vdash a\#s$ by assumption.

- The case $\lambda a.r$. There are three cases:
  - The case $\Delta \vdash a\#u$. By Definition 3.4.12 we have $(\lambda a.r)[b{:=}u] \equiv \lambda a.(r[b{:=}u])$. Using $(\#\lambda\mathbf{a})$ we obtain $\Delta \vdash a\#\lambda a.(r[b{:=}u])$. The result follows.
  - The case $\Delta \nvdash a\#u$ with Δ containing sufficient freshness. By Definition 3.4.12 we have $(\lambda a.r)[b{:=}u] \equiv \lambda c.(((c\ a)\cdot r)[b{:=}u])$, where c is a fresh atom, distinct from a and b, chosen so that $\Delta \vdash c\#r$ and $\Delta \vdash c\#t$. By Lemma 3.2.9 we have $depth((c\ a)\cdot r) = depth(r)$. By inductive hypothesis $\Delta \vdash a\#((c\ a)\cdot r)[b{:=}u]$. Using $(\#\lambda\mathbf{b})$ we obtain $\Delta \vdash a\#\lambda c.(((c\ a)\cdot r)[b{:=}u])$. By Definition 3.4.12 we have $\Delta \vdash a\#(\lambda a.r)[b{:=}u]$. The result follows.
  - The case $\Delta \nvdash a\#u$ with Δ not containing sufficient freshness. By Definition 3.4.12 we have $(\lambda a.r)[b{:=}u] \equiv (\lambda a.r)[b{\mapsto}u]$. By definition $(\lambda a.r)[b{\mapsto}u] \equiv (\lambda b.(\lambda a.r))u$. Using $(\#\mathbf{rs})$, $(\#\lambda\mathbf{b})$ and $(\#\lambda\mathbf{a})$ we obtain $\Delta \vdash a\#(\lambda b.(\lambda a.r))u$. The result follows.
- The case $\lambda c.t$. There are three cases:
  - The case $\Delta \vdash c\#u$. Suppose $\Delta \vdash a\#\lambda c.t$ so that $\Delta \vdash a\#t$. By Definition 3.4.12 we have $(\lambda c.t)[b{:=}u] \equiv \lambda c.(t[b{:=}u])$. By inductive hypothesis $\Delta \vdash a\#t[b{:=}u]$. Using $(\#\lambda\mathbf{b})$ we obtain $\Delta \vdash a\#\lambda c.(t[b{:=}u])$. By Definition 3.4.12 we have $\Delta \vdash a\#(\lambda c.t)[b{:=}u]$. The result follows.
  - The case $\Delta \nvdash c\#u$ with Δ containing sufficient freshness. Suppose $\Delta \vdash a\#\lambda c.t$ so that $\Delta \vdash a\#t$. By Definition 3.4.12 we have $(\lambda c.t)[b{:=}u] \equiv \lambda d.(((d\ c)\cdot t)[b{:=}u])$, where d is a fresh atom, distinct from a, c and b, chosen so that $\Delta \vdash d\#t$ and $\Delta \vdash d\#u$. By Lemma 3.2.9 we have $depth((d\ c)\cdot t) = depth(t)$. By inductive hypothesis $\Delta \vdash a\#((d\ c)\cdot t)[b{:=}u]$. Using $(\#\lambda\mathbf{b})$ we obtain $\Delta \vdash a\#\lambda d.(((d\ c)\cdot t)[b{:=}u])$. By Definition 3.4.12 we have $\Delta \vdash a\#(\lambda c.t)[b{:=}u]$. The result follows.
  - The case $\Delta \nvdash c\#u$ with Δ not containing sufficient freshness. By Definition 3.4.12 we have $(\lambda c.t)[b{:=}u] \equiv (\lambda c.t)[b{\mapsto}u]$. By definition $(\lambda c.t)[b{\mapsto}u] \equiv (\lambda b.(\lambda c.t))u$. Using $(\#\mathbf{rs})$ and $(\#\lambda\mathbf{b})$ we obtain $\Delta \vdash a\#(\lambda b.(\lambda c.t))u$. The result follows.
- The case $\lambda b.s$. By Definition 3.4.12 we have $(\lambda b.s)[b{:=}u] \equiv \lambda b.s$. The result follows.
- The case $\lambda X.r$. Suppose $\Delta \vdash a\#\lambda X.r$ and $\Delta \vdash a\#u$ with $X \notin fV(u)$, which can be guaranteed. By Definition 3.4.12 we have $(\lambda X.r)[b{:=}u] \equiv \lambda X.(r[b{:=}u])$. By inductive hypothesis $\Delta, a\#X \vdash a\#r[b{:=}u]$. Using $(\#\lambda\mathbf{X})$ we obtain $\Delta \vdash a\#\lambda X.(r[b{:=}u])$. The result follows.

⊠

Proof of Lemma 3.4.20.

Proof. By induction on r.

- The case a. By Definition 3.4.13 we have $a^\star \equiv a$. The result follows.

- The case c. By Definition 3.4.13 we have $\mathsf{c}^\star \equiv \mathsf{c}$. The result follows.
- The case $\pi{\cdot}X$. By Definition 3.4.13 we have $(\pi{\cdot}X)^\star \equiv \pi{\cdot}X$. The result follows.
- The case rs. By Definition 3.4.13 we have $rs^\star \equiv r^\star s^\star$. By Definition 3.2.3 we have $fV(r^\star s^\star) = fV(r^\star) \cup fV(s^\star)$. By inductive hypotheses $fV(r^\star) \subseteq fV(r)$ and $fV(s^\star) \subseteq fV(s)$ therefore $fV(r^\star) \cup fV(s^\star) \subseteq fV(r) \cup fV(s)$. By Definition 3.2.3 we have $fV(r) \cup fV(s) = fV(rs)$. The result follows.
- The case $\lambda a.r$. By Definition 3.4.13 we have $(\lambda a.r)^\star \equiv \lambda a.(r^\star)$. By Definition 3.2.3 we have $fV(\lambda a.(r^\star)) = fV(r^\star)$. By inductive hypothesis $fV(r^\star) \subseteq fV(r)$. By Definition 3.2.3 we have $fV(r) = fV(\lambda a.r)$. The result follows.
- The case $\lambda X.r$. By Definition 3.4.13 we have $(\lambda X.r)^\star = \lambda X.(r^\star)$. By Definition 3.2.3 we have $fV(\lambda X.(r^\star)) = fV(r^\star) \setminus \{X\}$. By inductive hypothesis $fV(r^\star) \subseteq fV(r)$ therefore $fV(r^\star) \setminus \{X\} \subseteq fV(r) \setminus \{X\}$. By Definition 3.2.3 we have $fV(r) \setminus \{X\} = fV(\lambda X.r)$. The result follows.

⊠

Proof of Lemma 3.4.28.

Proof. By induction on r.

- The case a. Using $(\Rightarrow\mathbf{a})$ we obtain $\Delta \vdash a \Rightarrow a$. The result follows.
- The case $\pi{\cdot}X$. Using $(\Rightarrow\mathbf{X})$ we obtain $\Delta \vdash \pi{\cdot}X \Rightarrow \pi{\cdot}X$. The result follows.
- The case c. Using $(\Rightarrow\mathsf{c})$ we obtain $\Delta \vdash \mathsf{c} \Rightarrow \mathsf{c}$. The result follows.
- The case rs. By inductive hypotheses $\Delta \vdash r \Rightarrow r$ and $\Delta \vdash s \Rightarrow s$. Using $(\Rightarrow\mathbf{rs})$ we obtain $\Delta \vdash rs \Rightarrow rs$. The result follows.
- The case $\lambda a.r$. By inductive hypothesis $\Delta \vdash r \Rightarrow r$. Using $(\Rightarrow\lambda\mathbf{a})$ we obtain $\Delta \vdash \lambda a.r \Rightarrow \lambda a.r$. The result follows.
- The case $\lambda X.r$. By inductive hypothesis $\Delta \vdash r \Rightarrow r$. Using $(\Rightarrow\lambda\mathbf{X})$ we obtain $\Delta \vdash \lambda X.r \Rightarrow \lambda X.r$. The result follows.

⊠

Proof of Lemma 3.4.32.

Proof. By induction on the derivation of $\Delta \vdash r \Rightarrow s$.

- The case $(\Rightarrow\mathbf{a})$. Suppose $\Delta \vdash a \Rightarrow a$. By Definition 3.2.4 we have $a[X{:=}\pi{\cdot}X] \equiv a$. Using $(\Rightarrow\mathbf{a})$ we obtain $\Delta' \vdash a[X{:=}\pi{\cdot}X] \Rightarrow a[X{:=}\pi{\cdot}X]$. The result follows.
- The case $(\Rightarrow\mathbf{X})$. There are two cases:
  - The case $\pi'{\cdot}X$. Suppose $\Delta \vdash \pi'{\cdot}X \Rightarrow \pi'{\cdot}X$. By Definition 3.2.4 we have $(\pi'{\cdot}X)[X{:=}\pi{\cdot}X] \equiv (\pi'{\circ}\pi){\cdot}X$. Using $(\Rightarrow\mathbf{X})$ we obtain $\Delta' \vdash (\pi'{\circ}\pi){\cdot}X \Rightarrow (\pi'{\circ}\pi){\cdot}X$. The result follows.
  - The case $\pi'{\cdot}Y$. Suppose $\Delta \vdash \pi'{\cdot}Y \Rightarrow \pi'{\cdot}Y$. By Definition 3.2.4 we have $(\pi'{\cdot}Y)[X{:=}\pi{\cdot}X] \equiv \pi'{\cdot}Y$. Using $(\Rightarrow\mathbf{X})$ we obtain $\Delta' \vdash (\pi'{\cdot}Y)[X{:=}\pi{\cdot}X] \Rightarrow (\pi'{\cdot}Y)[X{:=}\pi{\cdot}X]$. The result follows.

- The case ($\Rightarrow\mathsf{c}$). Suppose $\Delta \vdash \mathsf{c} \Rightarrow \mathsf{c}$. By Definition 3.2.4 we have $\mathsf{c}[X{:=}\pi{\cdot}X] \equiv \mathsf{c}$. Using ($\Rightarrow\mathsf{c}$) we obtain $\Delta' \vdash \mathsf{c}[X{:=}\pi{\cdot}X] \Rightarrow \mathsf{c}[X{:=}\pi{\cdot}X]$. The result follows.
- The case ($\Rightarrow\mathbf{rs}$). Suppose $\Delta \vdash r \Rightarrow t$ and $\Delta \vdash s \Rightarrow u$. By inductive hypothesis $\Delta' \vdash r[X{:=}\pi{\cdot}X] \Rightarrow t[X{:=}\pi{\cdot}X]$ and $\Delta \vdash s[X{:=}\pi{\cdot}X] \Rightarrow u[X{:=}\pi{\cdot}X]$. Using ($\Rightarrow\mathbf{rs}$) we obtain $\Delta' \vdash (r[X{:=}\pi{\cdot}X])(s[X{:=}\pi{\cdot}X]) \Rightarrow (t[X{:=}\pi{\cdot}X])(u[X{:=}\pi{\cdot}X])$. By Definition 3.2.4 we have $(t[X{:=}\pi{\cdot}X])(u[X{:=}\pi{\cdot}X]) \equiv (tu)[X{:=}\pi{\cdot}X]$. The result follows.
- The case ($\Rightarrow\lambda\mathbf{a}$). Suppose $\Delta \vdash r \Rightarrow s$. By inductive hypothesis $\Delta' \vdash r[X{:=}\pi{\cdot}X] \Rightarrow s[X{:=}\pi{\cdot}X]$. Using ($\Rightarrow\lambda\mathbf{a}$) we obtain $\Delta' \vdash \lambda a.(r[X{:=}\pi{\cdot}X]) \Rightarrow \lambda a.(s[X{:=}\pi{\cdot}X])$. By Definition 3.2.4 we have $\lambda a.(s[X{:=}\pi{\cdot}X]) \equiv (\lambda a.s)[X{:=}\pi{\cdot}X]$. The result follows.
- The case ($\Rightarrow\lambda\mathbf{X}$). Suppose $\Delta \vdash r \Rightarrow s$. By inductive hypothesis $\Delta' \vdash r[X{:=}\pi{\cdot}X] \Rightarrow s[X{:=}\pi{\cdot}X]$. Using ($\Rightarrow\lambda\mathbf{X}$) we obtain $\Delta' \vdash \lambda Y.(r[X{:=}\pi{\cdot}X]) \Rightarrow \lambda Y.(s[X{:=}\pi{\cdot}X])$ where $Y \neq X$. By Definition 3.2.4 we have $\lambda Y.(s[X{:=}\pi{\cdot}X]) \equiv (\lambda Y.s)[X{:=}\pi{\cdot}X]$. The result follows.
- The case ($\Rightarrow\epsilon$). Suppose $\Delta \vdash r \Rightarrow t$, $\Delta \vdash s \Rightarrow u$ and $\Delta \vdash tu \overset{(\mathbf{level2})}{\rightarrow} v$. By inductive hypotheses $\Delta' \vdash r[X{:=}\pi{\cdot}X] \Rightarrow t[X{:=}\pi{\cdot}X]$ and $\Delta' \vdash s[X{:=}\pi{\cdot}X] \Rightarrow u[X{:=}\pi{\cdot}X]$. By Lemma 3.4.6 we have $\Delta' \vdash (t[X{:=}\pi{\cdot}X])(u[X{:=}\pi{\cdot}X]) \overset{(\mathbf{level2})}{\rightarrow} v[X{:=}\pi{\cdot}X]$. Using ($\Rightarrow\epsilon$) we obtain $\Delta' \vdash (r[X{:=}\pi{\cdot}X])(s[X{:=}\pi{\cdot}X]) \Rightarrow v[X{:=}\pi{\cdot}X]$. By Definition 3.2.4 we have $(r[X{:=}\pi{\cdot}X])(s[X{:=}\pi{\cdot}X]) \equiv (rs)[X{:=}\pi{\cdot}X]$. The result follows.
- The case ($\Rightarrow\alpha$). Suppose $\Delta \vdash r \Rightarrow s$, $\Delta \vdash a\#r$ and $\Delta \vdash b\#r$. By inductive hypothesis $\Delta' \vdash r[X{:=}\pi{\cdot}X] \Rightarrow s[X{:=}\pi{\cdot}X]$. By Lemma 3.3.8 we have $\Delta' \vdash a\#r[X{:=}\pi{\cdot}X]$ and $\Delta' \vdash b\#r[X{:=}\pi{\cdot}X]$. Using ($\Rightarrow\alpha$) we obtain $\Delta' \vdash (b\ a){\cdot}(r[X{:=}\pi{\cdot}X]) \Rightarrow s$. By Lemma 3.2.16 we have $(b\ a){\cdot}(r[X{:=}\pi{\cdot}X]) \equiv ((b\ a){\cdot}r)[X{:=}\pi{\cdot}X]$. The result follows.

⊠

Proof of Lemma 3.4.34.

Proof. By induction on the derivation of $\Delta \vdash r \Rightarrow s$.

- The case ($\Rightarrow\mathbf{a}$). By Definition 3.2.4 we have $a[X{:=}t] \equiv a$ and $a[X{:=}u] \equiv a$. Using ($\Rightarrow\mathbf{a}$) we obtain $\Delta \vdash a \rightarrow a$. The result follows.
- The case ($\Rightarrow\mathbf{X}$). There are two cases:
  - The case $\pi{\cdot}X$. By Definition 3.2.4 we have $(\pi{\cdot}X)[X{:=}t] \equiv \pi{\cdot}t$ and $(\pi{\cdot}X)[X{:=}u] \equiv \pi{\cdot}u$. By Lemma 3.4.33 we have $\Delta \vdash \pi{\cdot}t \Rightarrow \pi{\cdot}u$. The result follows.
  - The case $\pi{\cdot}Y$. By Definition 3.2.4 we have $(\pi{\cdot}Y)[X{:=}t] \equiv \pi{\cdot}Y$ and $(\pi{\cdot}Y)[X{:=}u] \equiv \pi{\cdot}Y$. Using ($\Rightarrow\mathbf{X}$) we obtain $\Delta \vdash \pi{\cdot}Y \Rightarrow \pi{\cdot}Y$. The result follows.
- The case ($\Rightarrow\mathsf{c}$). By Definition 3.2.4 we have $\mathsf{c}[X{:=}t] \equiv \mathsf{c}$ and $\mathsf{c}[X{:=}u] \equiv \mathsf{c}$. Using ($\Rightarrow\mathsf{c}$) we obtain $\Delta \vdash \mathsf{c} \rightarrow \mathsf{c}$. The result follows.
- The case ($\Rightarrow\mathbf{rs}$). Suppose $\Delta \vdash r \Rightarrow r'$ and $\Delta \vdash s \Rightarrow s'$. By inductive hypotheses $\Delta \vdash r[X{:=}t] \Rightarrow r'[X{:=}u]$ and $\Delta \vdash s[X{:=}t] \Rightarrow s'[X{:=}u]$. Using ($\Rightarrow\mathbf{rs}$) we obtain $\Delta \vdash (r[X{:=}t])(s[X{:=}t]) \Rightarrow (r'[X{:=}u])(s'[X{:=}u])$. By Definition 3.2.4 we have $(r'[X{:=}u])(s'[X{:=}u]) \equiv (r's')[X{:=}u]$. The result follows.
- The case ($\Rightarrow\lambda\mathbf{a}$). Suppose $\Delta \vdash r \Rightarrow s$. By inductive hypothesis $\Delta \vdash r[X{:=}t] \Rightarrow s[X{:=}u]$. Using ($\Rightarrow\lambda\mathbf{a}$) we obtain $\Delta \vdash \lambda a.(r[X{:=}t]) \Rightarrow \lambda a.(s[X{:=}u])$. By Definition 3.2.4 we have $\lambda a.(s[X{:=}u]) \equiv (\lambda a.r)[X{:=}u]$. The result follows.

- The case ($\Rightarrow\lambda\mathbf{X}$). Suppose $\Delta \vdash r \Rightarrow s$. By inductive hypothesis $\Delta \vdash r[X{:=}t] \Rightarrow s[X{:=}u]$. Using ($\Rightarrow\lambda\mathbf{X}$) we obtain $\Delta \vdash \lambda Y.(r[X{:=}t]) \Rightarrow \lambda Y.(s[X{:=}u])$. By Definition 3.2.4 we have $\lambda Y.(s[X{:=}u]) \equiv (\lambda Y.r)[X{:=}u]$, where $Y \notin fV(u) \cup fV(t)$. The result follows.
- The case ($\Rightarrow\epsilon$). Suppose $\Delta \vdash r \Rightarrow r'$, $\Delta \vdash\Rightarrow s \Rightarrow s'$ and $\Delta \vdash r's' \overset{\textbf{(level2)}}{\rightarrow} v$. By inductive hypotheses $\Delta \vdash r[X{:=}t] \Rightarrow r'[X{:=}u]$ and $\Delta \vdash s[X{:=}t] \Rightarrow s'[X{:=}u]$. By Lemma 3.4.6 we have $\Delta \vdash (r'[X{:=}u])(s'[X{:=}u]) \overset{\textbf{(level2)}}{\rightarrow} v[X{:=}u]$. Using ($\Rightarrow\epsilon$) we obtain $\Delta \vdash (r[X{:=}t])(s[X{:=}t]) \Rightarrow v[X{:=}u]$. By Definition 3.2.4 we have $(r[X{:=}t])(s[X{:=}t]) \equiv (rs)[X{:=}t]$. The result follows.
- The case ($\Rightarrow\alpha$). Suppose $\Delta \vdash r \Rightarrow s$, $\Delta \vdash a\#r$ and $\Delta \vdash a\#s$. By inductive hypothesis $\Delta \vdash r[X{:=}t] \Rightarrow s[X{:=}u]$. By Lemma 3.3.8 we have $\Delta \vdash a\#r[X{:=}t]$ and $\Delta \vdash b\#r[X{:=}t]$. Using ($\Rightarrow\alpha$) we obtain $\Delta \vdash (b\ a){\cdot}(b\ a){\cdot}(r[X{:=}t]) \Rightarrow s[X{:=}u]$. By Lemma 3.2.16 we have $(b\ a){\cdot}(r[X{:=}t]) \equiv ((b\ a){\cdot}r)[X{:=}t]$. The result follows.

⊠

Proof of Lemma 3.4.37.

Proof. By induction on the derivation of $\Delta \vdash r \rightarrow s$.

- The case ($\rightarrow\mathbf{a}$). Suppose $\Delta \vdash a[a{\mapsto}t] \rightarrow t$. By Definition 3.4.1 we have $level(a[a{\mapsto}t]) = level(t)$. The result follows.
- The case ($\rightarrow\#$). Suppose $\Delta \vdash a\#r$ so that $\Delta \vdash r[a{\mapsto}t] \rightarrow r$. By Definition 3.4.1 we have $level(r[a{\mapsto}t]) = \max(level(r),\ level(t))$. If $\max(level(r),\ level(t)) = level(r)$ then the result follows. If $\max(level(r),\ level(t)) = level(t)$ then $level(r) \leq level(t)$. The result follows.
- The case ($\rightarrow\mathbf{rs1}$). Suppose $\Delta \vdash a\#s$ so that $\Delta \vdash (rs)[a{\mapsto}t] \rightarrow (r[a{\mapsto}t])s$. By Definition 3.4.1 we have $level((rs)[a{\mapsto}t]) = \max(level(rs),\ level(t))$. By Definition 3.4.1 we have $\max(level(rs),\ level(t)) = \max(level(r),\ level(s),\ level(t))$. By Definition 3.4.1 we have $\max(level(r),\ level(s),\ level(t)) = level((r[a{\mapsto}t])s)$. The result follows.
- The case ($\rightarrow\mathbf{rs2}$). Suppose $level(r) = 1$ so that $\Delta \vdash (rs)[a{\mapsto}t] \rightarrow (r[a{\mapsto}t])(s[a{\mapsto}t])$. By Definition 3.4.1 we have $level((rs)[a{\mapsto}t]) = \max(level(rs),\ level(t))$. By Definition 3.4.1 we have $\max(level(rs),\ level(t)) = \max(level(r),\ level(s),\ level(t))$. By Definition 3.4.1 we have $\max(level(r),\ level(s),\ level(t)) = level((r[a{\mapsto}t])(s[a{\mapsto}t]))$. The result follows.
- The case ($\rightarrow\lambda\mathbf{b}$). Suppose $\Delta \vdash b\#t$ so that $\Delta \vdash (\lambda b.s)[a{\mapsto}t] \rightarrow \lambda b.(s[a{\mapsto}t])$. By Definition 3.4.1 we have $level((\lambda b.s)[a{\mapsto}t]) = \max(level(s),\ level(t))$. By Definition 3.4.1 we have $level(\lambda b.(s[a{\mapsto}t])) = \max(level(s),\ level(t))$. The result follows.
- The case ($\rightarrow\lambda\mathbf{X}$). Suppose $X \notin fV(t)$ so that $\Delta \vdash (\lambda X.r)[a{\mapsto}t] \rightarrow \lambda X.(r[a{\mapsto}t])$. By Definition 3.4.1 we have $level((\lambda X.r)[a{\mapsto}t]) = 2$. By Definition 3.4.1 we have $\lambda X.(r[a{\mapsto}t]) = 2$. The result follows.
- The case ($\rightarrow\beta$). Suppose $\Delta \vdash (\lambda X.r)t \rightarrow r[X{:=}t]$. By Definition 3.4.1 we have $level((\lambda X.r)t) = 2$. The result follows.
- The case ($\triangleright\lambda\mathbf{a}$). Suppose $\Delta \vdash r \rightarrow s$. By inductive hypothesis $level(s) \leq level(r)$. Using ($\triangleright\lambda\mathbf{a}$) we obtain $\Delta \vdash \lambda a.r \rightarrow \lambda a.s$. By Definition 3.4.1 we have $level(s) = level(\lambda a.s)$. The result follows.

- The case ($\triangleright$**rs1**). Suppose $\Delta \vdash r \to t$. By inductive hypothesis $level(t) \leq level(r)$. Using ($\triangleright$**rs1**) we obtain $\Delta \vdash rs \to ts$. It is a fact that $\max(level(t), level(s)) \leq \max(level(r), level(s))$. By Definition 3.4.1 we have $\max(level(t), level(s)) = level(ts)$. The result follows.
- The case ($\triangleright$**rs2**). Suppose $\Delta \vdash s \to u$. By inductive hypothesis $level(u) \leq level(s)$. Using ($\triangleright$**rs2**) we obtain $\Delta \vdash rs \to ru$. It is a fact that $\max(level(r), level(u)) \leq \max(level(r), level(s))$. By Definition 3.4.1 we have $\max(level(r), level(u)) = level(ru)$. The result follows.
- The case ($\triangleright\lambda$**X**). Suppose $\Delta \vdash \lambda X.r \to \lambda X.s$. By Definition 3.4.1 we have $level(\lambda X.r) = 2$. The result follows.
- The case ($\triangleright\alpha$). Suppose $\Delta \vdash a\#r$, $\Delta \vdash b\#r$ and $\Delta \vdash r \to s$. By inductive hypothesis $level(s) \leq level(r)$. Using ($\triangleright\alpha$) we obtain $\Delta \vdash (b\ a)\cdot r \to s$. By Lemma 3.4.7 we have $level((b\ a)\cdot r) = level(r)$. The result follows.

⊠

Proof of Lemma 3.4.38.

Proof. By considering all possible non-trivial divergences (where $s \not\equiv t$). We assume $\Delta \vdash t \Rightarrow t'$, $\Delta \vdash s \Rightarrow s'$, and so on.

- The case $a[a{\mapsto}t]$ and ($\Rightarrow\epsilon$) with ($\to$**a**). Suppose that $\Delta \vdash a[a{\mapsto}t] \Rightarrow t$ and $\Delta \vdash a[a{\mapsto}t] \overset{\text{(level1)}}{\to} a[a{\mapsto}t']$. By inductive hypothesis there exists t'' such that $\Delta \vdash t \overset{\text{(level1)}}{\to^*} t''$ and $\Delta \vdash t' \Rightarrow t''$. Using ($\Rightarrow\epsilon$) with ($\to$**a**) and various congruence rules we obtain $\Delta \vdash a[a{\mapsto}t'] \Rightarrow t''$ and $\Delta \vdash t \overset{\text{(level1)}}{\to^*} t''$. The result follows.
- The case $b[a{\mapsto}t]$ and ($\Rightarrow\epsilon$) with ($\to\#$). Suppose that $\Delta \vdash b[a{\mapsto}t] \Rightarrow b$ and $\Delta \vdash b[a{\mapsto}t] \overset{\text{(level1)}}{\to} b[a{\mapsto}t']$. Using ($\Rightarrow\epsilon$) with ($\to\#$) we obtain $\Delta \vdash b[a{\mapsto}t'] \Rightarrow b$ and $\Delta \vdash b \overset{\text{(level1)}}{\to^*} b$. The result follows.
- The case $\mathsf{c}[a{\mapsto}t]$ and ($\Rightarrow\epsilon$) with ($\to\#$). Suppose that $\Delta \vdash \mathsf{c}[a{\mapsto}t] \Rightarrow \mathsf{c}$ and $\Delta \vdash \mathsf{c}[a{\mapsto}t] \overset{\text{(level1)}}{\to} \mathsf{c}[a{\mapsto}t']$. Using ($\Rightarrow\epsilon$) with ($\to\#$) we obtain $\Delta \vdash \mathsf{c}[a{\mapsto}t'] \Rightarrow \mathsf{c}$ and $\Delta \vdash \mathsf{c} \overset{\text{(level1)}}{\to^*} \mathsf{c}$. The result follows.
- The case $(\pi\cdot X)[a{\mapsto}t]$ and ($\Rightarrow\epsilon$) with ($\to\#$). Suppose $\Delta \vdash a\#\pi\cdot X$ so that $\Delta \vdash (\pi\cdot X)[a{\mapsto}t] \Rightarrow \pi\cdot X$ and $\Delta \vdash (\pi\cdot X)[a{\mapsto}t] \overset{\text{(level1)}}{\to} (\pi\cdot X)[a{\mapsto}t']$. Using ($\Rightarrow\epsilon$) with ($\to\#$) we obtain $\Delta \vdash (\pi\cdot X)[a{\mapsto}t'] \Rightarrow \pi\cdot X$ and $\Delta \vdash \pi\cdot X \overset{\text{(level1)}}{\to^*} \pi\cdot X$. The result follows.
- The case $(rs)[a{\mapsto}t]$ and ($\to\#$). Suppose that $\Delta \vdash a\#rs$ so that $\Delta \vdash (rs)[a{\mapsto}t] \Rightarrow rs$ and $\Delta \vdash (rs)[a{\mapsto}t] \overset{\text{(level1)}}{\to} (r's')[a{\mapsto}t']$. By inductive hypotheses there exists r'' and s'' such that $\Delta \vdash r \overset{\text{(level1)}}{\to^*} r''$ and $\Delta \vdash r' \Rightarrow r''$, and $\Delta \vdash s \overset{\text{(level1)}}{\to^*} s''$ and $\Delta \vdash s' \Rightarrow s''$. By Lemma 3.4.4 we have $\Delta \vdash a\#r's'$. Using ($\Rightarrow\epsilon$) with ($\to\#$) and various congruence rules we obtain $\Delta \vdash (r's')[a{\mapsto}t'] \Rightarrow r''s''$ and $\Delta \vdash rs \overset{\text{(level1)}}{\to^*} r''s''$. The result follows.
- The case $(rs)[a{\mapsto}t]$ and ($\Rightarrow\epsilon$) with ($\to$**rs1**). Suppose $\Delta \vdash a\#s$ so that $\Delta \vdash (rs)[a{\mapsto}t] \Rightarrow (r'[a{\mapsto}t'])s'$ and $\Delta \vdash (rs)[a{\mapsto}t] \overset{\text{(level1)}}{\to} (r''s'')[a{\mapsto}t'']$. By inductive hypotheses there exists r''', s''' and t''' such that $\Delta \vdash r' \overset{\text{(level1)}}{\to^*} r'''$ and $\Delta \vdash r'' \Rightarrow r'''$, $\Delta \vdash s' \overset{\text{(level1)}}{\to^*} s'''$ and $\Delta \vdash s'' \Rightarrow s'''$, and $\Delta \vdash t' \overset{\text{(level1)}}{\to^*} t'''$ and $\Delta \vdash t'' \Rightarrow t'''$. By Lemma 3.4.4 we have $\Delta \vdash a\#s''$. Using ($\Rightarrow\epsilon$) with ($\to$**rs1**) and various congruence rules we obtain $\Delta \vdash (r''s'')[a{\mapsto}t''] \Rightarrow (r'''[a{\mapsto}t'''])s'''$ and $\Delta \vdash (r'[a{\mapsto}t'])s' \overset{\text{(level1)}}{\to^*} (r'''[a{\mapsto}t'''])s'''$. The result follows.

- The case $(rs)[a\mapsto t]$ and $(\Rightarrow\epsilon)$ with $(\rightarrow\mathbf{rs2})$. Suppose $level(r) = 1$ so that $\Delta \vdash (rs)[a\mapsto t] \Rightarrow (r'[a\mapsto t'])(s'[a\mapsto t'])$ and $\Delta \vdash (rs)[a\mapsto t] \overset{(\mathbf{level1})}{\rightarrow} (r''s'')[a\mapsto t'']$. By inductive hypotheses there exists r''', s''' and t''' such that $\Delta \vdash r' \overset{(\mathbf{level1})}{\rightarrow^*} r'''$ and $\Delta \vdash r'' \Rightarrow r'''$, $\Delta \vdash s' \overset{(\mathbf{level1})}{\rightarrow^*} s'''$ and $\Delta \vdash s'' \Rightarrow s'''$, and $\Delta \vdash t' \overset{(\mathbf{level1})}{\rightarrow^*} t'''$ and $\Delta \vdash t'' \Rightarrow t'''$. By Lemma 3.4.37 we have $level(r'') = 1$. Using $(\Rightarrow\epsilon)$ with $(\rightarrow\mathbf{rs2})$ and various congruence rules we obtain $\Delta \vdash (r''s'')[a\mapsto t''] \Rightarrow (r'''[a\mapsto t'''])(s'''[a\mapsto t'''])$ and $\Delta \vdash (r'[a\mapsto t'])(s'[a\mapsto t']) \overset{(\mathbf{level1})}{\rightarrow^*} (r'''[a\mapsto t'''])(s'''[a\mapsto t'''])$. The result follows.
- The case $(\lambda a.r)[a\mapsto t]$ and $(\Rightarrow\epsilon)$ with $(\rightarrow\#)$. Suppose $\Delta \vdash (\lambda a.r)[a\mapsto t] \Rightarrow \lambda a.r'$ and $\Delta \vdash (\lambda a.r)[a\mapsto t] \overset{(\mathbf{level1})}{\rightarrow} (\lambda a.r'')[a\mapsto t'']$. By inductive hypothesis there exists r''' such that $\Delta \vdash r' \overset{(\mathbf{level1})}{\rightarrow^*} r'''$ and $\Delta \vdash r'' \Rightarrow r'''$. Using $(\Rightarrow\epsilon)$ with $(\rightarrow\#)$ and various congruence rules we obtain $\Delta \vdash (\lambda a.r'')[a\mapsto t''] \Rightarrow \lambda a.r'''$ and $\Delta \vdash \lambda a.r' \overset{(\mathbf{level1})}{\rightarrow^*} \lambda a.r'''$. The result follows.
- The case $(\lambda a.r)[b\mapsto u]$ and $(\Rightarrow\epsilon)$ with $(\rightarrow\lambda\mathbf{b})$. Suppose $\Delta \vdash a\#u$ so that $\Delta \vdash (\lambda a.r)[b\mapsto u] \Rightarrow \lambda a.(r'[b\mapsto u'])$ and $\Delta \vdash (\lambda a.r)[b\mapsto u] \overset{(\mathbf{level1})}{\rightarrow} (\lambda a.r'')[b\mapsto u'']$. By inductive hypotheses there exists r''' and u''' such that $\Delta \vdash r' \overset{(\mathbf{level1})}{\rightarrow^*} r'''$ and $\Delta \vdash r'' \Rightarrow r'''$. and $\Delta \vdash u' \overset{(\mathbf{level1})}{\rightarrow^*} u'''$ and $\Delta \vdash u'' \Rightarrow u'''$. By Lemma 3.4.4 we have $\Delta \vdash a\#u''$. Using $(\Rightarrow\epsilon)$ with $(\rightarrow\lambda\mathbf{b})$ and various congruence rules we obtain $\Delta \vdash (\lambda a.r'')[b\mapsto u''] \Rightarrow \lambda a.(r'''[b\mapsto u'''])$ and $\Delta \vdash \lambda a.(r'[b\mapsto u']) \overset{(\mathbf{level1})}{\rightarrow^*} \lambda a.(r'''[b\mapsto u'''])$. The result follows.
- The case $(\lambda X.r)[a\mapsto t]$ and $(\Rightarrow\epsilon)$ with $(\rightarrow\#)$. Suppose $\Delta \vdash a\#\lambda X.r$ so that $\Delta \vdash (\lambda X.r)[a\mapsto t] \Rightarrow \lambda X.r'$ and $\Delta \vdash (\lambda X.r)[a\mapsto t] \overset{(\mathbf{level1})}{\rightarrow} (\lambda X.r'')[a\mapsto t'']$. By inductive hypothesis there exists r''' such that $\Delta \vdash r' \overset{(\mathbf{level1})}{\rightarrow^*} r'''$ and $\Delta \vdash r'' \Rightarrow r'''$. By Lemma 3.4.4 we have $\Delta \vdash a\#\lambda X.r''$. Using $(\Rightarrow\epsilon)$ with $(\rightarrow\#)$ and various congruence rules we obtain $\Delta \vdash (\lambda X.r'')[a\mapsto t''] \Rightarrow \lambda X.r'''$ and $\Delta \vdash \lambda X.r' \overset{(\mathbf{level1})}{\rightarrow^*} \lambda X.r'''$. The result follows.
- The case $(\lambda X.r)[a\mapsto t]$ and $(\Rightarrow\epsilon)$ with $(\rightarrow\lambda\mathbf{X})$. Suppose $X \notin fV(t')$ which can be guaranteed by renaming, so that $\Delta \vdash (\lambda X.r)[a\mapsto t] \Rightarrow \lambda X.(r'[a\mapsto t'])$ and $\Delta \vdash (\lambda X.r)[a\mapsto t] \overset{(\mathbf{level1})}{\rightarrow} (\lambda X.r'')[a\mapsto t'']$. By inductive hypotheses there exists r''' and t''' such that $\Delta \vdash r' \overset{(\mathbf{level1})}{\rightarrow^*} r'''$ and $\Delta \vdash r'' \Rightarrow r'''$, and $\Delta \vdash t' \overset{(\mathbf{level1})}{\rightarrow^*} t'''$ and $\Delta \vdash t'' \Rightarrow t'''$. Using $(\Rightarrow\epsilon)$ with $(\rightarrow\lambda\mathbf{X})$ and various congruence rules we obtain $\Delta \vdash (\lambda X.r'')[a\mapsto t''] \Rightarrow \lambda X.(r'''[a\mapsto t'''])$ and $\Delta \vdash \lambda X.(r'[a\mapsto t']) \overset{(\mathbf{level1})}{\rightarrow^*} \lambda X.(r'''[a\mapsto t'''])$. The result follows.
- The case $((\lambda X.r)t)[b\mapsto u]$ and $(\Rightarrow\epsilon)$ with $(\rightarrow\beta)$. Suppose $\Delta \vdash ((\lambda X.r)t)[b\mapsto u] \Rightarrow r'[X{:=}t'][b\mapsto u']$ and $\Delta \vdash ((\lambda X.r)t)[b\mapsto u] \overset{(\mathbf{level1})}{\rightarrow} ((\lambda X.r'')t'')[b\mapsto u'']$. By inductive hypotheses there exists r''', t''' and u''' such that $\Delta \vdash r' \overset{(\mathbf{level1})}{\rightarrow^*} r'''$ and $\Delta \vdash r'' \Rightarrow r'''$, $\Delta \vdash u' \overset{(\mathbf{level1})}{\rightarrow^*} u'''$ and $\Delta \vdash u'' \Rightarrow u'''$, and $\Delta \vdash t' \overset{(\mathbf{level1})}{\rightarrow^*} t'''$ and $\Delta \vdash t'' \Rightarrow t'''$. Using $(\Rightarrow\epsilon)$ with $(\rightarrow\beta)$ and various congruence rules we obtain $\Delta \vdash ((\lambda X.r'')t'')[b\mapsto u''] \Rightarrow r'''[X{:=}t'''][b\mapsto u''']$ and $\Delta \vdash r''[X{:=}t''][b\mapsto u''] \overset{(\mathbf{level1})}{\rightarrow^*} r'''[X{:=}t'''][b\mapsto u''']$. The result follows.
- The case $((\lambda X.r)t)[b\mapsto u]$ and $(\Rightarrow\epsilon)$ with $(\rightarrow\mathbf{rs1})$. Suppose $\Delta \vdash b\#t$, so that $\Delta \vdash ((\lambda X.r)t) \Rightarrow ((\lambda X.r')[b\mapsto u'])t'$ and $\Delta \vdash ((\lambda X.r)t)[b\mapsto u] \overset{(\mathbf{level1})}{\rightarrow} ((\lambda X.r'')t'')[b\mapsto u'']$. By inductive hypotheses there exists r''', t''' and u''' such that $\Delta \vdash r' \overset{(\mathbf{level1})}{\rightarrow^*} r'''$ and $\Delta \vdash r'' \Rightarrow r'''$, $\Delta \vdash u' \overset{(\mathbf{level1})}{\rightarrow^*} u'''$, and $\Delta \vdash u'' \Rightarrow u'''$, and $\Delta \vdash t' \overset{(\mathbf{level1})}{\rightarrow^*} t'''$ and $\Delta \vdash t'' \Rightarrow t'''$. By Lemma 3.4.4 we have $\Delta \vdash b\#t''$. Using $(\Rightarrow\epsilon)$ with $(\rightarrow\mathbf{rs1})$ and various congruence rules we obtain $\Delta \vdash ((\lambda X.r'')t'')[b\mapsto u''] \Rightarrow ((\lambda X.r''')[b\mapsto u'''])t'''$ and $\Delta \vdash ((\lambda X.r'')[b\mapsto u''])t'' \overset{(\mathbf{level1})}{\rightarrow^*}$

$((\lambda X.r''')[b{\mapsto}u'''])t'''$. The result follows.

- The case $((\lambda X.r)t)[b{\mapsto}u]$ and $(\Rightarrow\epsilon)$ with $(\rightarrow\beta)$. Suppose $X \notin fV(u)$, which can be guaranteed by renaming. Suppose $\Delta \vdash b\#t$, so that $\Delta \vdash ((\lambda X.r)t)[b{\mapsto}u] \Rightarrow r'[X{:=}t'][b{\mapsto}u']$ and $\Delta \vdash ((\lambda X.r)t)[b{\mapsto}u] \overset{\text{(level1)}}{\rightarrow} ((\lambda X.r'')[b{\mapsto}u''])t''$. By inductive hypotheses there exists r''', t''' and u''' such that $\Delta \vdash r' \overset{\text{(level1)}}{\rightarrow^*} r'''$ and $\Delta \vdash r'' \Rightarrow r'''$, $\Delta \vdash u' \overset{\text{(level1)}}{\rightarrow^*} u'''$, and $\Delta \vdash u'' \Rightarrow u'''$, and $\Delta \vdash t' \overset{\text{(level1)}}{\rightarrow^*} t'''$ and $\Delta \vdash t'' \Rightarrow t'''$. Using $(\Rightarrow\epsilon)$ with $(\rightarrow\lambda\mathbf{X})$ and $(\rightarrow\beta)$ we obtain $\Delta \vdash r'[X{:=}t'][b{\mapsto}u'] \overset{\text{(level1)}}{\rightarrow^*} r'''[X{:=}t'''][b{\mapsto}u''']$ and $\Delta \vdash ((\lambda X.r'')[b{\mapsto}u''])t'' \Rightarrow r'''[X{:=}t'''][b{\mapsto}u''']$. The result follows.

⊠

APPENDIX B

Additional proofs: permissive nominal terms

Proof of Lemma 4.2.19.

Proof. By induction on r.

- The case a. Since $id(a) = a$.
- The case $\pi{\cdot}X^s$. Since $id{\circ}\pi = \pi$.
- The case $\mathsf{f}(r_1, \dots, r_n)$. By Definition 4.2.9 $id{\cdot}\mathsf{f}(r_1, \dots, r_n) \equiv \mathsf{f}(id{\cdot}r_1, \dots, id{\cdot}r_n)$. By inductive hypothesis $\mathsf{f}(id{\cdot}r_1, \dots, id{\cdot}r_n) \equiv \mathsf{f}(r_1, \dots, r_n)$. The result follows.
- The case $[a]r$. By Definition 4.2.9 $id{\cdot}[a]r \equiv [id(a)](id{\cdot}r)$. By definition $id(a) = a$ for all a therefore $[id(a)](id{\cdot}r) \equiv [a](id{\cdot}r)$. By inductive hypothesis $[a](id{\cdot}r) \equiv [a]r$. The result follows.

⊠

Proof of Lemma 4.2.20.

Proof. By induction on r.

- The case a. Since $\pi{\cdot}(\pi'{\cdot}a) \equiv \pi(\pi'(a)) \equiv (\pi{\circ}\pi'){\cdot}a$.
- The case $\pi''{\cdot}X^s$. By Definition 4.2.9 we have $\pi{\cdot}(\pi'{\cdot}(\pi''{\cdot}X^s)) \equiv \pi{\cdot}((\pi'{\circ}\pi''){\cdot}X^s)$. By Definition 4.2.9 we have $\pi{\cdot}((\pi'{\circ}\pi''){\cdot}X^s) \equiv (\pi{\circ}(\pi'{\circ}\pi'')){\cdot}X^s$. It is a fact that $(\pi{\circ}(\pi'{\circ}\pi'')){\cdot}X^s \equiv ((\pi{\circ}\pi'){\circ}\pi''){\cdot}X^s$. By Definition 4.2.9 we have $((\pi{\circ}\pi'){\circ}\pi''){\cdot}X^s \equiv (\pi{\circ}\pi'){\cdot}(\pi''{\cdot}X^s)$. The result follows.
- The case $\mathsf{f}(r_1, \dots, r_n)$. By Definition 4.2.9 we have $\pi{\cdot}(\pi'{\cdot}\mathsf{f}(r_1, \dots, r_n)) \equiv \pi{\cdot}\mathsf{f}(\pi'{\cdot}r_1, \dots, \pi'{\cdot}r_n)$. By Definition 4.2.9 we have $\pi{\cdot}\mathsf{f}(\pi'{\cdot}r_1, \dots, \pi'{\cdot}r_n) \equiv \mathsf{f}((\pi{\cdot}(\pi'{\cdot}r_1), \dots, \pi{\cdot}(\pi'{\cdot}r_n))$. By inductive hypotheses $\mathsf{f}((\pi{\cdot}(\pi'{\cdot}r_1), \dots, \pi{\cdot}(\pi'{\cdot}r_n)) \equiv \mathsf{f}((\pi{\circ}\pi'){\cdot}r_1, \dots, (\pi{\circ}\pi'){\cdot}r_n)$. By Definition 4.2.9 we have $\mathsf{f}((\pi{\circ}\pi'){\cdot}r_1, \dots, (\pi{\circ}\pi'){\cdot}r_n) \equiv (\pi{\circ}\pi'){\cdot}\mathsf{f}(r_1, \dots, r_n)$. The result follows.
- The case $[a]r$. By Definition 4.2.9 we have $\pi{\cdot}(\pi'{\cdot}[a]r) \equiv \pi{\cdot}[\pi'(a)](\pi'{\cdot}r)$. By Definition 4.2.9 we have $\pi{\cdot}[\pi'(a)](\pi'{\cdot}r) \equiv [\pi(\pi'(a))](\pi{\cdot}(\pi'{\cdot}r))$. It is a fact that $[\pi(\pi'(a))](\pi{\cdot}(\pi'{\cdot}r)) \equiv [(\pi{\circ}\pi')(a)](\pi{\cdot}(\pi'{\cdot}r))$. By inductive hypothesis $[(\pi{\circ}\pi')(a)](\pi{\cdot}(\pi'{\cdot}r)) \equiv [(\pi{\circ}\pi')(a)]((\pi{\circ}\pi'){\cdot}r)$. By Definition 4.2.9 we have $[(\pi{\circ}\pi')(a)]((\pi{\circ}\pi'){\cdot}r) \equiv (\pi{\circ}\pi'){\cdot}[a]r$. The result follows.

⊠

Proof of Lemma 4.2.21.

Proof. By induction on r.

- The case a. By Definition 4.2.11 we have $\pi{\cdot}fa(a) = \pi{\cdot}\{a\}$. By Definition 4.2.10 we have $\pi{\cdot}\{a\} = \{\pi(a)\}$. By Definition 4.2.11 we have $\{\pi(a)\} = fa(\pi(a))$. By Definition 4.2.9 we have $fa(\pi(a)) = fa(\pi{\cdot}a)$. The result follows.

- The case $\pi'{\cdot}X^S$. By Definition 4.2.11 we have $\pi{\cdot}fa(\pi'{\cdot}X^S) = \pi{\cdot}(\pi'{\cdot}S)$. By Definition 4.2.10 we have $\pi{\cdot}(\pi'{\cdot}S) = (\pi\circ\pi'){\cdot}S$. By Definition 4.2.11 we have $(\pi\circ\pi'){\cdot}S = fa((\pi\circ\pi'){\cdot}X^S)$. By Definition 4.2.9 we have $fa((\pi\circ\pi'){\cdot}X^S) = fa(\pi{\cdot}(\pi'{\cdot}X^S))$. The result follows.
- The case $\mathsf{f}(r_1,\dots,r_n)$. By Definition 4.2.11 we have $\pi{\cdot}fa(\mathsf{f}(r_1,\dots,r_n)) = \pi{\cdot}\bigcup_{1\leq i\leq n} fa(r_i)$. By Definition 4.2.10 we have $\pi{\cdot}\bigcup_{1\leq i\leq n} fa(r_i) = \bigcup_{1\leq i\leq n}(\pi{\cdot}fa(r_i))$. By inductive hypotheses $\bigcup_{1\leq i\leq n}(\pi{\cdot}fa(r_i)) = \bigcup_{1\leq i\leq n} fa(\pi{\cdot}r_i)$. By Definition 4.2.11 we have $\bigcup_{1\leq i\leq n} fa(\pi{\cdot} r_i) = fa(\mathsf{f}(\pi{\cdot}r_1,\dots,\pi{\cdot}r_n))$. By Definition 4.2.9 we have $fa(\mathsf{f}(\pi{\cdot}r_1,\dots,\pi{\cdot}r_n)) = fa(\pi{\cdot} \mathsf{f}(r_1,\dots,r_n))$. The result follows.
- The case $[a]r$. By Definition 4.2.11 we have $\pi{\cdot}fa([a]r) = \pi{\cdot}(fa(r)\setminus\{a\})$. By Definition 4.2.10 we have $\pi{\cdot}(fa(r)\setminus\{a\}) = (\pi{\cdot}fa(r))\setminus(\pi{\cdot}\{a\})$. By Definition 4.2.10 we have $(\pi{\cdot} fa(r))\setminus(\pi{\cdot}\{a\}) = (\pi{\cdot}fa(r))\setminus\{\pi(a)\}$. By inductive hypothesis $(\pi{\cdot}fa(r))\setminus\{\pi(a)\} = fa(\pi{\cdot} r)\setminus\{\pi(a)\}$. By Definition 4.2.11 we have $fa(\pi{\cdot}r)\setminus\{\pi(a)\} = fa([\pi(a)](\pi{\cdot}r))$. By Definition 4.2.9 we have $fa([\pi(a)](\pi{\cdot}r)) = fa(\pi{\cdot}[a]r)$. The result follows.

⊠

Proof of Lemma 4.2.22.

Proof. By induction on r.

- The case a. Since $fV(a) = \emptyset = fV(\pi(a))$.
- The case $\pi'{\cdot}X^S$. By Definition 4.2.9 we have $fV(\pi{\cdot}(\pi'{\cdot}X^S)) = fV((\pi\circ\pi'){\cdot}X^S)$. By Definition 4.2.14 we have $fV((\pi\circ\pi'){\cdot}X^S) = \{X^S\}$. By Definition 4.2.14 we have $\{X^S\} = fV(\pi'{\cdot} X^S)$. The result follows.
- The case $\mathsf{f}(r_1,\dots,r_n)$. By Definition 4.2.9 we have $fV(\pi{\cdot}\mathsf{f}(r_1,\dots,r_n)) = fV(\mathsf{f}(\pi{\cdot}r_1,\dots,\pi{\cdot} r_n))$. By Definition 4.2.14 we have $fV(\mathsf{f}(\pi{\cdot}r_1,\dots,\pi{\cdot}r_n)) = \bigcup_{1\leq i\leq n} fV(\pi{\cdot}r_i)$. By inductive hypotheses $\bigcup_{1\leq i\leq n} fV(\pi{\cdot}r_i) = \bigcup_{1\leq i\leq n} fV(r_i)$. By Definition 4.2.14 we have $\bigcup_{1\leq i\leq n} fV(r_i) = fV(\mathsf{f}(r_1,\dots,r_n))$. The result follows.
- The case $[a]r$. By Definition 4.2.9 we have $fV(\pi{\cdot}[a]r) = fV([\pi(a)](\pi{\cdot}r))$. By Definition 4.2.14 we have $fV([\pi(a)](\pi{\cdot}r)) = fV(\pi{\cdot}r)$. By inductive hypothesis $fV(\pi{\cdot}r) = fV(r)$. By Definition 4.2.14 we have $fV(r) = fV([a]r)$. The result follows.

⊠

Proof of Lemma 4.2.26.

Proof. By induction on r.

- The case a. Since $size(a) = 1 = size(\pi(a))$.
- The case $\pi'{\cdot}X^S$. Since $size(\pi'{\cdot}X^S) = 1 = size((\pi\circ\pi'){\cdot}X^S)$.
- The case $\mathsf{f}(r_1,\dots,r_n)$. By Definition 4.2.9 we have $size(\pi{\cdot}\mathsf{f}(r_1,\dots,r_n)) = size(\mathsf{f}(\pi{\cdot}r_1,\dots,\pi{\cdot} r_n))$. By Definition 4.2.25 we have $size(\mathsf{f}(\pi{\cdot}r_1,\dots,\pi{\cdot}r_n)) = \sum_{1\leq i\leq n} size(\pi{\cdot}r_i)$. By inductive hypotheses $\sum_{1\leq i\leq n} size(\pi{\cdot}r_i) = \sum_{1\leq i\leq n} size(r_i)$. By Definition 4.2.25 we have $\sum_{1\leq i\leq n} size(r_i) = size(\mathsf{f}(r_1,\dots,r_n))$. The result follows.

- The case $[a]r$. By Definition 4.2.9 we have $\mathit{size}(\pi\cdot[a]r) = \mathit{size}([\pi(a)](\pi\cdot r))$. By Definition 4.2.25 we have $\mathit{size}([\pi(a)](\pi\cdot r)) = 1 + \mathit{size}(\pi\cdot r)$. By inductive hypothesis $1 + \mathit{size}(\pi\cdot r) = 1 + \mathit{size}(r)$. By Definition 4.2.25 we have $1 + \mathit{size}(r) = \mathit{size}([a]r)$. The result follows.

⊠

Proof of Lemma 4.2.32.

Proof. By induction on r.

- The case a. By Definition 4.2.29 we have $a[X^S{:=}t] \equiv a$. By Definition 4.2.14 we have $fV(a) = \emptyset$. The result follows.
- The case $\pi\cdot X^S$. By Definition 4.2.29 we have $fV((\pi\cdot X^S)[X^S{:=}t]) = fV(\pi\cdot t)$. By Lemma 4.2.22 we have $fV(\pi\cdot t) = fV(t)$. It is a fact that $fV(t) \subseteq \{X^S\} \cup fV(t)$. By Definition 4.2.14 we have $\{X^S\} \cup fV(t) = fV(\pi\cdot X^S) \cup fV(t)$. The result follows.
- The case $\pi\cdot Y^T$. By Definition 4.2.29 we have $fV((\pi\cdot Y^T)[X^S{:=}t]) = fV(\pi\cdot Y^T)$. By Definition 4.2.14 we have $fV(\pi\cdot Y^T) = \{Y^T\}$. It is a fact that $\{Y^T\} \subseteq \{Y^T\} \cup fV(t)$. By Definition 4.2.14 we have $\{Y^T\} \cup fV(t) = fV(Y^T) \cup fV(t)$. By Lemma 4.2.22 we have $fV(Y^T) \cup fV(t) = fV(\pi\cdot Y^T) \cup fV(t)$. The result follows.
- The case $[a]r$. By Definition 4.2.29 we have $fV(([a]r)[X^S{:=}t]) = fV([a](r[X^S{:=}t]))$. By Definition 4.2.14 we have $fV([a](r[X^S{:=}t])) = fV(r[X^S{:=}t])$. By inductive hypothesis $fV(r[X^S{:=}t]) \subseteq fV(r) \cup fV(t)$. By Definition 4.2.14 we have $fV(r) \cup fV(t) = fV([a]r) \cup fV(t)$. The result follows.
- The case $\mathsf{f}(r_1, \ldots, r_n)$. By Definition 4.2.29 we have $fV(\mathsf{f}(r_1, \ldots, r_n)[X^S{:=}t]) = fV(\mathsf{f}(r_1[X^S{:=}t], \ldots, r_n[X^S{:=}t]))$. By Definition 4.2.14 we have $fV(\mathsf{f}(r_1[X^S{:=}t], \ldots, r_n[X^S{:=}t])) = \bigcup_{1\leq i\leq n} fV(r_i[X^S{:=}t])$. By inductive hypotheses $\bigcup_{1\leq i\leq n} fV(r_i[X^S{:=}t]) \subseteq \bigcup_{1\leq i\leq n} fV(r_i) \cup fV(t)$. By Definition 4.2.14 we have $\bigcup_{1\leq i\leq n} fV(r_i) \cup fV(t) = fV(\mathsf{f}(r_1, \ldots, r_n)) \cup fV(t)$. The result follows.

⊠

Proof of Theorem 4.3.13.

Proof. By case analysis on $r \sqsubseteq S,\ \mathit{Inc}$ showing that all rules reduce the size of a support inclusion problem.

- The case $a \sqsubseteq S,\ \mathit{Inc}$. Suppose $\mathit{size}(a \sqsubseteq S,\ \mathit{Inc}) = (T, A, P, S)$. Suppose also that $a \sqsubseteq S,\ \mathit{Inc} \Longrightarrow \mathit{Inc}$. Then $\mathit{size}(\mathit{Inc}) = (T, A, P, S{-}1)$. The result follows.
- The case $\mathsf{f}(r_1, \ldots, r_n) \sqsubseteq S,\ \mathit{Inc}$. Suppose $\mathit{size}(\mathsf{f}(r_1, \ldots, r_n) \sqsubseteq S,\ \mathit{Inc}) = (T, A, P, S)$. Suppose also that $\mathsf{f}(r_1, \ldots, r_n) \sqsubseteq S,\ \mathit{Inc} \Longrightarrow r_1 \sqsubseteq S, \ldots, r_n \sqsubseteq S,\ \mathit{Inc}$. Then $\mathit{size}(r_1 \sqsubseteq S, \ldots, r_n \sqsubseteq S,\ \mathit{Inc}) = (T{-}1, A, P, S{+}n{-}1)$. The result follows.
- The case $[a]r \sqsubseteq S,\ \mathit{Inc}$. Suppose $\mathit{size}([a]r \sqsubseteq S,\ \mathit{Inc}) = (T, A, P, S)$. Suppose also that $[a]r \sqsubseteq S,\ \mathit{Inc} \Longrightarrow r \sqsubseteq S \cup \{a\},\ \mathit{Inc}$. Then $\mathit{size}(r \sqsubseteq S \cup \{a\},\ \mathit{Inc}) = (T, A{-}1, P, S)$. The result follows.
- The case $\pi\cdot X^S \sqsubseteq T,\ \mathit{Inc}$. There are two cases:

- The ($\sqsubseteq$**X**) case. Suppose $size(\pi{\cdot}X^S \sqsubseteq T,\ Inc) = (T, A, P, S)$. Suppose also that $\pi{\cdot}X^S \sqsubseteq T,\ Inc \Longrightarrow X^S \sqsubseteq \pi^{-1}{\cdot}T,\ Inc$. Then $size(X^S \sqsubseteq \pi^{-1}{\cdot}T,\ Inc) = (T, A, P{-}1, S)$. The result follows.
- The ($\sqsubseteq$**X**$'$) case. Suppose $size(\pi{\cdot}X^S \sqsubseteq T,\ Inc) = (T, A, P, S)$. Suppose also that $\pi{\cdot}X^S \sqsubseteq T,\ Inc \Longrightarrow Inc$. Then $size(Inc) = (T, A, P{-}1, S{-}1)$. The result follows.

⊠

Proof of Lemma 4.3.29.

Proof. By case analysis of the simplification rules in Definition 4.3.8.

- The case ($\sqsubseteq$**a**). Suppose $a \sqsubseteq S,\ Inc \Longrightarrow Inc$ by ($\sqsubseteq$**a**). By Definition 4.3.18 we have $fV(a \sqsubseteq S,\ Inc) = fV(Inc)$. The result follows.
- The case ($\sqsubseteq\mathsf{f}$). Suppose $\mathsf{f}(r_1, \ldots, r_n) \sqsubseteq T,\ Inc \Longrightarrow r_1 \sqsubseteq T, \ldots, r_n \sqsubseteq T,\ Inc$ by ($\sqsubseteq\mathsf{f}$). By Definition 4.3.18 we have $fV(\mathsf{f}(r_1, \ldots, r_n)) = \bigcup_{1\leq i\leq n}(fV(r_i))$. It is a fact that $fV(\mathsf{f}(r_1, \ldots, r_n) \sqsubseteq T,\ Inc) = fV(r_1 \sqsubseteq T, \ldots, r_n \sqsubseteq T,\ Inc)$. The result follows.
- The case ($\sqsubseteq$[]). Suppose $[a]r \sqsubseteq T,\ Inc \Longrightarrow r \sqsubseteq T \cup \{a\},\ Inc$ by ($\sqsubseteq$[]). By Definition 4.3.18 we have $fV([a]r \sqsubseteq T,\ Inc) = fV(r \sqsubseteq T \cup \{a\},\ Inc)$. The result follows.
- The case ($\sqsubseteq$**X**). Suppose $\pi{\cdot}X^S \sqsubseteq T,\ Inc \Longrightarrow X^S \sqsubseteq \pi^{-1}{\cdot}T,\ Inc$ by ($\sqsubseteq$**X**). By Definition 4.3.18 we have $fV(\pi{\cdot}X^S \sqsubseteq T,\ Inc) = fV(X^S \sqsubseteq \pi^{-1}{\cdot}T,\ Inc)$. The result follows.
- The case ($\sqsubseteq$**X**$'$). Suppose $\pi{\cdot}X^S \sqsubseteq T,\ Inc \Longrightarrow Inc$ by ($\sqsubseteq$**X**$'$). By Definition 4.3.18 we have $fV(\pi{\cdot}X^S \sqsubseteq T,\ Inc) = \{X^S\} \cup fV(Inc)$. The result follows.

⊠

Proof of Lemma 4.3.36.

Proof. By case analysis on the rules defined in Definition 4.3.35.

- The case ($\overset{?}{=}$**a**). Suppose $\mathcal{V}; a \overset{?}{=} a,\ \mathcal{P} \Longrightarrow \mathcal{V}; \mathcal{P}$ by ($\overset{?}{=}$**a**). Suppose also that $\theta \in Sol(\mathcal{P})$. By Definition 4.2.29 we have $a\theta \equiv a$. Using ($=_\alpha$**a**) we obtain $a\theta =_\alpha a\theta$. It is a fact that $\theta \in Sol(a \overset{?}{=} a,\ \mathcal{P})$. The result follows.
  Otherwise, suppose $\theta \in Sol(a \overset{?}{=} a, \mathcal{P})$. By Definition 4.3.3 we have $\theta \in Sol(\mathcal{P})$. The result follows.
- The case ($\overset{?}{=}\mathsf{f}$). Suppose $\mathcal{V}; \mathsf{f}(r_1, \ldots, r_n) \overset{?}{=} \mathsf{f}(s_1, \ldots, s_n),\ \mathcal{P} \Longrightarrow \mathcal{V}; r_i \overset{?}{=} s_i,\ \mathcal{P}$ for $1 \leq i \leq n$ by ($\overset{?}{=}\mathsf{f}$). Suppose also that $\theta \in Sol(r_i \overset{?}{=} s_i,\ \mathcal{P})$ for $1 \leq i \leq n$. By Definition 4.3.3 we have $r_i\theta =_\alpha s_i\theta$ for $1 \leq i \leq n$. Using ($=_\alpha\mathsf{f}$) we obtain $\mathsf{f}(r_1\theta, \ldots, r_n\theta) =_\alpha \mathsf{f}(s_1\theta, \ldots, s_n\theta)$. By Definition 4.2.29 we have $\mathsf{f}(r_1, \ldots, r_n)\theta =_\alpha \mathsf{f}(s_1, \ldots, s_n)\theta$. By Definition 4.3.3 we have $\theta \in Sol(\mathsf{f}(r_1, \ldots, r_n) \overset{?}{=} \mathsf{f}(s_1, \ldots, s_n),\ \mathcal{P})$. The result follows.
  Otherwise, suppose $\theta \in Sol(\mathsf{f}(r_1, \ldots, r_n) \overset{?}{=} \mathsf{f}(s_1, \ldots, s_n),\ \mathcal{P})$. By Definition 4.3.3 we have $\mathsf{f}(r_1, \ldots, r_n)\theta =_\alpha \mathsf{f}(s_1, \ldots, s_n)\theta$. By Definition 4.2.29 we have $\mathsf{f}(r_1\theta, \ldots, r_n\theta) =_\alpha \mathsf{f}(s_1\theta, \ldots, s_n\theta)$. Using ($=_\alpha\mathsf{f}$) we obtain $r_i\theta =_\alpha s_i\theta$ for $1 \leq i \leq n$. By Definition 4.3.3 we have $\theta \in Sol(r_i \overset{?}{=} s_i,\ \mathcal{P})$ for $1 \leq i \leq n$. The result follows.

- The case ($\overset{?}{=}$[**a**]). Suppose $\mathcal{V}; [a]r \overset{?}{=} [a]s,\ \mathcal{P} \Longrightarrow \mathcal{V}; r \overset{?}{=} s,\ \mathcal{P}$ by ($\overset{?}{=}$[**a**]). Suppose also that $\theta \in \mathit{Sol}(r \overset{?}{=} s,\ \mathcal{P})$. By Definition 4.3.3 we have $r\theta =_\alpha s\theta$. Using ($=_\alpha$[**a**]) we obtain $[a](r\theta) =_\alpha [a](s\theta)$. By Definition 4.2.29 we have $([a]r)\theta =_\alpha ([a]s)\theta$. By Definition 4.3.3 we have $\theta \in \mathit{Sol}([a]r \overset{?}{=} [a]s,\ \mathcal{P})$. The result follows.
  Otherwise, suppose $\theta \in \mathit{Sol}([a]r \overset{?}{=} [a]s,\ \mathcal{P})$. By Definition 4.3.3 we have $([a]r)\theta =_\alpha ([a]s)\theta$. By Definition 4.2.29 we have $[a](r\theta) =_\alpha [a](s\theta)$. Using ($=_\alpha$[**a**]) we obtain $r\theta =_\alpha s\theta$. By Definition 4.3.3 we have $\theta \in \mathit{Sol}(r \overset{?}{=} s,\ \mathcal{P})$. The result follows.
- The case ($\overset{?}{=}$[**b**]). Suppose $\mathcal{V}; [a]r \overset{?}{=} [b]s,\ \mathcal{P} \Longrightarrow \mathcal{V}; (b\ a)\cdot r \overset{?}{=} s,\ \mathcal{P}$ as $b \notin fa(r)$ by ($\overset{?}{=}$[**b**]). Suppose also that $\theta \in \mathit{Sol}((b\ a)\cdot r \overset{?}{=} s,\ \mathcal{P})$. By Definition 4.3.3 we have $((b\ a)\cdot r)\theta =_\alpha s\theta$. By Lemma 4.2.31 we have $(b\ a)\cdot(r\theta) =_\alpha s\theta$. By Theorem 4.2.30 we have $b \notin fa(r\theta)$. Using ($=_\alpha$[**b**]) we obtain $[a](r\theta) =_\alpha [b](s\theta)$. By Definition 4.2.29 we have $([a]r)\theta =_\alpha ([b]s)\theta$. By Definition 4.3.3 we have $\theta \in \mathit{Sol}([a]r \overset{?}{=} [b]s,\ \mathcal{P})$. The result follows.
  Otherwise, suppose $\theta \in \mathit{Sol}([a]r \overset{?}{=} [b]s,\ \mathcal{P})$. By Definition 4.3.3 we have $([a]r)\theta =_\alpha ([b]s)\theta$. By Definition 4.2.29 we have $[a](r\theta) =_\alpha [b](s\theta)$. Using ($=_\alpha$[**b**]) we obtain $(b\ a)\cdot(r\theta) =_\alpha s\theta$. By Lemma 4.2.31 we have $((b\ a)\cdot r)\theta =_\alpha s\theta$. By Definition 4.3.3 we have $\theta \in \mathit{Sol}((b\ a)\cdot r \overset{?}{=} s,\ \mathcal{P})$. The result follows.
- The case ($\overset{?}{=}$**X**). Suppose $\mathcal{V}; \pi\cdot X^S \overset{?}{=} \pi\cdot X^S,\ \mathcal{P} \Longrightarrow \mathcal{V}; \mathcal{P}$ by ($\overset{?}{=}$**X**). Suppose also that $\theta \in \mathit{Sol}(\mathcal{P})$. By Theorem 4.2.27 we have $\theta \in \mathit{Sol}(\pi\cdot X^S \overset{?}{=} \pi\cdot X^S,\ \mathcal{P})$. The result follows.
  Otherwise, suppose $\theta \in \mathit{Sol}(\pi\cdot X^S \overset{?}{=} \pi\cdot X^S,\ \mathcal{P})$. By Definition 4.3.3 we have $\theta \in \mathit{Sol}(\mathcal{P})$. The result follows.

⊠

Proof of Theorem 4.3.42.

Proof. By case analysis on the rules in Definition 4.3.35, checking that all rules reduce the size of the unification problem.

- The case ($\overset{?}{=}$**a**). Suppose $\mathit{size}(a \overset{?}{=} a,\ \mathcal{P}) = (T, E, A)$. Suppose also that $\mathcal{V}; a \overset{?}{=} a,\ \mathcal{P} \Longrightarrow \mathcal{V}; \mathcal{P}$ by ($\overset{?}{=}$**a**). It is a fact that $\mathit{size}(\mathcal{P}) = (T, E{-}1, A)$. The result follows.
- The case ($\overset{?}{=}$f). Suppose $\mathit{size}(\mathsf{f}(r_1, \ldots, r_n) \overset{?}{=} \mathsf{f}(s_1, \ldots, s_n),\ \mathcal{P}) = (T, E, A)$. Suppose also that $\mathcal{V}; \mathsf{f}(r_1, \ldots, r_n) \overset{?}{=} \mathsf{f}(s_1, \ldots, s_n),\ \mathcal{P} \Longrightarrow \mathcal{V}; r_i \overset{?}{=} s_i,\ \mathcal{P}$ for $1 \le i \le n$ by ($\overset{?}{=}$f). It is a fact that $\mathit{size}(r_i \overset{?}{=} s_i,\ \mathcal{P}) = (T{-}1, E{+}n{-}1, A)$. The result follows.
- The case ($\overset{?}{=}$[**a**]). Suppose $\mathit{size}([a]r \overset{?}{=} [a]s,\ \mathcal{P}) = (T, E, A)$. Suppose also that $\mathcal{V}; [a]r \overset{?}{=} [a]s,\ \mathcal{P} \Longrightarrow \mathcal{V}; r \overset{?}{=} s,\ \mathcal{P}$ by ($\overset{?}{=}$[**a**]). It is a fact that $\mathit{size}(r \overset{?}{=} s,\ \mathcal{P}) = (T, E, A{-}1)$. The result follows.
- The case ($\overset{?}{=}$[**b**]). Suppose $\mathit{size}([a]r \overset{?}{=} [b]s,\ \mathcal{P}) = (T, E, A)$. Suppose also that $b \notin fa(r)$ so that $\mathcal{V}; [a]r \overset{?}{=} [b]s,\ \mathcal{P} \Longrightarrow \mathcal{V}; (b\ a)\cdot r \overset{?}{=} s,\ \mathcal{P}$ by ($\overset{?}{=}$[**b**]). It is a fact that $\mathit{size}((b\ a)\cdot r \overset{?}{=} s,\ \mathcal{P}) = (T, E, A{-}1)$. The result follows.
- The case ($\overset{?}{=}$**X**). Suppose $\mathit{size}(\pi\cdot X^S \overset{?}{=} \pi\cdot X^S,\ \mathcal{P}) = (T, E, A)$. Suppose also that $\mathcal{V}; \pi\cdot X^S \overset{?}{=} \pi\cdot X^S,\ \mathcal{P} \Longrightarrow \mathcal{V}; \mathcal{P}$ by ($\overset{?}{=}$**X**). It is a fact that $\mathit{size}(\mathcal{V}) = (T, E{-}1, A)$. The result follows.
- The case (**I1**). Suppose $\mathit{size}(\pi\cdot X^S \overset{?}{=} s,\ \mathcal{P}) = (T, E, A)$. Suppose also that $X^S \notin fV(s)$ and $\pi \neq id$, so that $\mathcal{V}; \pi\cdot X^S \overset{?}{=} s,\ \mathcal{P} \overset{[X^S := \pi^{-1}\cdot s]}{\Longrightarrow} \mathcal{V}; \mathcal{P}[X^S{:=}\pi^{-1}\cdot s]$ by (**I1**). It is a fact that $\mathit{size}(\mathcal{P}[X^S{:=}\pi^{-1}\cdot s]) = (T, E{-}1, A)$. The result follows.

- The case (**I2**). Suppose $size(r \overset{?}{=} \pi{\cdot}X^S,\ \mathcal{P}) = (T, E, A)$. Suppose also that $X^S \notin fV(r)$ and $\pi \neq id$, so that $\mathcal{V}; r \overset{?}{=} \pi{\cdot}X^S,\ \mathcal{P} \overset{[X^S:=\pi^{-1}\cdot r]}{\Longrightarrow} \mathcal{V}; \mathcal{P}[X^S{:=}\pi^{-1}{\cdot}r]$ by (**I2**). It is a fact that $size(\mathcal{P}[X^S{:=}\pi^{-1}{\cdot}r]) = (T, E{-}1, A)$. The result follows.
- The case (**I3**). As $\mathcal{P}_{\sqsubseteq}$ is non-trivial, (**I3**) always terminates.

⊠

Proof of Lemma 4.3.48.

Proof. By case analysis on the non-instantiating rules in Definition 4.3.35.

- The case ($\overset{?}{=}$**a**). Suppose $\mathcal{V}; a \overset{?}{=} a,\ \mathcal{P} \Longrightarrow \mathcal{V}; \mathcal{P}$. By Definition 4.3.37 we have $fV(a \overset{?}{=} a,\ \mathcal{P}) = fV(\mathcal{P})$. The result follows.
- The case ($\overset{?}{=}$f). Suppose $\mathcal{V}; \mathsf{f}(r_1, \ldots, r_n) \overset{?}{=} \mathsf{f}(s_1, \ldots, s_n) \mathcal{P} \Longrightarrow \mathcal{V}; r_i \overset{?}{=} s_i,\ \mathcal{P}$ for $1 \leq i \leq n$. By Definition 4.2.14 we have $fV(\mathsf{f}(r_1, \ldots, r_n)) = \bigcup_{1 \leq i \leq n} fV(r_i)$. By Definition 4.3.37 we have $fV(\mathsf{f}(r_1, \ldots, r_n) \overset{?}{=} \mathsf{f}(s_1, \ldots, s_n) \mathcal{P}) = fV(r_i \overset{?}{=} s_i,\ \mathcal{P})$. The result follows.
- The case ($\overset{?}{=}$[**a**]). Suppose $\mathcal{V}; [a]r \overset{?}{=} [a]s,\ \mathcal{P} \Longrightarrow \mathcal{V}; r \overset{?}{=} s,\ \mathcal{P}$. By Definition 4.2.14 we have $fV([a]r) = fV(r)$. By Definition 4.3.37 we have $fV([a]r \overset{?}{=} [a]s,\ \mathcal{P}) = fV(r \overset{?}{=} s,\ \mathcal{P})$. The result follows.
- The case ($\overset{?}{=}$[**b**]). Suppose $\mathcal{V}; [a]r \overset{?}{=} [b]s \mathcal{P} \Longrightarrow \mathcal{V}; (b\ a){\cdot}r \overset{?}{=} s,\ \mathcal{P}$. By Definition 4.2.14 we have $fV([a]r) = fV(r)$. By Lemma 4.2.22 we have $fV(r) = fV((b\ a){\cdot}r)$. By Definition 4.3.37 we have $fV([a]r \overset{?}{=} [b]s,\ \mathcal{P}) = fV((b\ a){\cdot}r \overset{?}{=} s,\ \mathcal{P})$. The result follows.
- The case ($\overset{?}{=}$**X**). Suppose $\mathcal{V}; \pi{\cdot}X^S \overset{?}{=} \pi{\cdot}X^S,\ \mathcal{P} \Longrightarrow \mathcal{V}; \mathcal{P}$. By Definition 4.3.37 we have $fV(\pi{\cdot}X^S \overset{?}{=} \pi{\cdot}X^S,\ \mathcal{P}) = \{X^S\} \cup fV(\mathcal{P})$. The result follows.

⊠

Proof of Lemma 4.3.49.

Proof. By case analysis on the instantiating rules in Definition 4.3.35.

- The case (**I1**). Suppose $fV(\pi{\cdot}X^S \overset{?}{=} s,\ \mathcal{P}) \subseteq \mathcal{V}$. Suppose also that $\mathcal{V}; \pi{\cdot}X^S \overset{?}{=} s,\ \mathcal{P} \overset{[X^S:=\pi^{-1}\cdot s]}{\Longrightarrow} \mathcal{V}; \mathcal{P}[X^S{:=}\pi^{-1}{\cdot}s]$ by (**I1**). By Definition 4.3.37 we have $fV(\pi{\cdot}X^S \overset{?}{=} s,\ \mathcal{P}) = \{X^S\} \cup fV(s) \cup fV(\mathcal{P})$. By Lemma 4.2.32 we have $\{X^S\} \cup fV(s) \cup fV(\mathcal{P}) \supseteq fV(\mathcal{P}[X^S{:=}\pi^{-1}{\cdot}s])$. The result follows.
- The case (**I2**). Suppose $fV(r \overset{?}{=} \pi{\cdot}X^S,\ \mathcal{P}) \subseteq \mathcal{V}$. Suppose also that $\mathcal{V}; r \overset{?}{=} \pi{\cdot}X^S,\ \mathcal{P} \overset{[X^S:=\pi^{-1}\cdot r]}{\Longrightarrow} \mathcal{V}; \mathcal{P}[X^S{:=}\pi^{-1}{\cdot}r]$ by (**I1**). By Definition 4.3.37 we have $fV(r \overset{?}{=} \pi{\cdot}X^S,\ \mathcal{P}) = \{X^S\} \cup fV(r) \cup fV(\mathcal{P})$. By Lemma 4.2.32 we have $\{X^S\} \cup fV(r) \cup fV(\mathcal{P}) \supseteq fV(\mathcal{P}[X^S{:=}\pi^{-1}{\cdot}r])$. The result follows.
- The case (**I3**). This is an immediate corollary of Lemma 4.3.29.

⊠

APPENDIX C

BNF Grammar for PNT Frontend

```
<digit>           ::=  0 | 1 . . .
<upper>           ::=  A | B . . .
<lower>           ::=  a | b . . .
<alphanum>        ::=  <digit> | <upper> | <lower>
<atom>            ::=  <digit>+
<atomlst>         ::=  <atom> | <atom>, <atoms>
<swapping>        ::=  ( <atom> <atom> )
<swappings>       ::=  ϵ | <swapping><swappings>
<idpermutation>   ::=  id
<permutation>     ::=  <idpermutation> | <swappings>
<tfident>         ::=  <upper>+<alphanum>*
<unkident>        ::=  <upper>+
<oblident>        ::=  <lower>+<alphanum>*
<srtident>        ::=  <upper>+
<termident>       ::=  <lower>+<alphanum>*
<modident>        ::=  <upper>+<alphanum>*
<term>            ::=  <atom> | <permutation> . <unkident> |
                       [<atom>]<term> | <tfident>(<termlst>)
<termlst>         ::=  ϵ | <terms'>
<termlst'>        ::=  <term> | <term>,<terms'>
<terms>           ::=  <term>*
<termasc>         ::=  term <termident> is <term>
<termascs>        ::=  <termasc>*
<termblk>         ::=  terms begin <termascs> end
<unifobl>         ::=  obligation <oblident> is unify <termident> and <termident>
<aeqobl>          ::=  obligation <oblident> is aeq <termident> and <termident>
<faobl>           ::=  obligation <oblident> is free atoms of <termident>
<permobl>         ::=  obligation <oblident> is permute <termident> with <permutation>
<freshobl>        ::=  obligation <oblident> is fresh name for <termident>
<oblasc>          ::=  <unifobl> | <aeqobl> | <faobl> | <permobl> | <freshobl>
<oblascs>         ::=  <oblasc>*
<oblblk>          ::=  obligations begin <oblascs> end
<implst>          ::=  import <modident>
<atomset>         ::=  { <atomlst> }
<psort>           ::=  comb <psort'>
<psort'>          ::=  union <atomset> | minus <atomset> | (<psort'>)
<srtasc>          ::=  sort <srtident> is <psort>
<srtascs>         ::=  <srtasc>*
<srtblk>          ::=  sorts begin <srtascs> end
<unkasc>          ::=  unknown <unkident> has sort <srtident>
<unkascs>         ::=  <unkasc>*
<unkblk>          ::=  unknowns begin <unkascs> end
<tfasc>           ::=  termformer <tfident>
<tfascs>          ::=  <tfasc>*
<tfblk>           ::=  termformers begin <tfascs> end
<module>          ::=  module <modident> begin
                       <implst> <srtascs> <unkascs> <tfascs> <trmascs> <oblascs>
                       end
```

Bibliography

[ABW07] Brian Aydemir, Aaron Bohannon, and Stephanie Weirich. Nominal reasoning techniques in Coq. *Electronic Notes in Theoretical Computer Science*, 174(5):69–77, 2007.

[ACP+08] Brian Aydemir, Arthur Charguéraud, Benjamin C. Pierce, Randy Pollack, and Stephanie Weirich. Engineering formal metatheory. In *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2008)*, pages 3–15, 2008.

[ACTZ07] Andrea Asperti, Claudo Sacerdoti Coen, Enrico Tassi, and Stefano Zacchiroli. *Crafting a Proof Assistant*, pages 18–32. Lecture Notes in Computer Science. Springer-Verlag, 2007.

[AEMO09] María Alpuente, Santiago Escobar, José Meseguer, and Pedro Ojeda. Order-sorted generalization. *Electronic Notes in Theoretical Computer Science*, 246:27–38, 2009.

[AGM+04] Samson Abramsky, Dan R. Ghica, Andrzej Murawski, Chih-Hao Luke Ong, and Iain D. B. Stark. Nominal games and full abstraction for the ν-calculus. In *Proceedings of the 19th Annual IEEE Symposium on Logic in Computer Science: (LICS 2004)*, pages 150–159, 2004.

[AKG95] Hassan Aït-Kaci and Jacques Garrigue. Label selective λ-calculus: syntax and confluence. *Theoretical Computer Science*, 151:353–383, 1995.

[Atk09] Robert Atkey. Syntax for free: representing syntax with binding with parametricity. In *Proceedings of the 9th International Conference on Typed Lambda Calculi and Applications (TLCA 2009)*, volume 5608 of *Lecture Notes in Computer Science*, pages 35–49, 2009.

[Aug06] Lennart Augustsson. Lambda calculus cooked four ways. Unpublished, 2006.

[AW10] Brian Aydemir and Stephanie Weirich. LNGen: Tool support for locally nameless representations. Draft, 2010.

[Bal87] Herman Balsters. Lambda calculus extended with segments. *Mathematical Logic and Theoretical Computer Science*, pages 15–27, 1987.

[Bal94] Herman Balsters. *Lambda calculus extended with segments*, pages 339–367. North Holland, 1994.

[Bar84] Henk Barendregt. *The λ-calculus: Its Syntax and Semantics.* Studies in Logic. Elsevier, 1984.

[BC97] Alexandre Boudet and Evelyne Countejean. AC-unification of higher-order patterns. In *Proceedings of the 3*rd *International Conference on Principles and Practice of Constraint Programming (CP 1997)*, pages 267–281, 1997.

[BC01] Alexandre Boudet and Evelyne Countejean. Combining pattern E-unification algorithms. In *Proceeedings of the International Conference on Rewriting Techniques and Applications (RTA 2001)*, pages 63–76, 2001.

[BC04] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development Coq'Art: The Calculus of Inductive Constructions.* Springer, 2004.

[BdV01] Mirna Bognar and Roel de Vrijer. A calculus of lambda calculus contexts. *Journal of Automated Reasoning*, 27:29–59, 2001.

[Ber76] Klaus J. Berkling. A symmetric complement to the λ-calculus. Technical Report ISF-76-7, Gesellschaft Fur Mathematik Und Datenverarbeitung MbH, 1976.

[BF07] William E. Byrd and Daniel Friedman. αKanren: A fresh name in Nominal Logic programming. In *Prooceedings of the 2007 Workshop on Scheme and Functional Programming*, pages 79–90, 2007.

[BKBH07] Nick Benton, Andrew Kennedy, Lennart Beringer, and Martin Hofmann. Relational semantics for effect-based program transformations with dynamic allocation. In *Proceedings of the 9*th *ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP 2007)*, pages 87–96, 2007.

[BKZ09] Peter Bulychev, Egor V. Kostylev, and Vladimir A. Zakharov. Antiunification algorithms and their applications in program analysis. In *Proceedings of the* 7th *International Andrei Ershov Memorial Conference, Perspective of System Informatics (PSI 2009)*, 2009.

[BM08] Peter Bulychev and Marius Minea. Duplicate code detection using antiunification. In *Proceedings of the Spring/Summer Young Researchers Colloquium on Software Engineering (SyRCoSE 2008)*, pages 51–54, 2008.

[BM09] Peter Bulychev and Marius Minea. An evaluation of duplicate code detection using antiunification. In *Proceedings of the* 3rd *International Workshop on Software Clones (IWSC 2009)*, 2009.

[Bog02] Mirna Bognar. *Contexts in Lambda Calculus.* PhD thesis, Vrije Universiteit Amsterdam, 2002.

[Bou00] Alexandre Boudet. Unification of higher-order patterns modulo simple syntactic equational theories. *Journal of Discrete Mathematics and Theoretical Computer Science*, 4:11–30, 2000.

[BP07] Jesper Bengtson and Joachim Parrow. A completeness proof for bisimulation in the pi-calculus using Isabelle. *Electronic Notes in Theoretical Computer Science*, 192(1):61–75, 2007.

[BP09a] Jesper Bengtson and Joachim Parrow. Formalising the π-calculus using Nominal Logic. *Logical Methods in Computer Science*, 5:16, 2009.

[BP09b] Jesper Bengtson and Joachim Parrow. Psi-calculi in Isabelle. In *Proceedings of the 22*$^{\text{nd}}$ *International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2009)*, pages 99–114, 2009.

[BS81] Stanley Burris and H. P. Sankappanavar. *A Course in Universal Algebra*. Springer, 1981.

[BS01] Franz Baader and Wayne Snyder. *Unification Theory*, pages 445–533. Elsevier and MIT Press, 2001.

[BU06] Stefan Berghofer and Christian Urban. A head-to-head comparison of de Bruijn indices and names. In *Proceedings of the* 1$^{\text{st}}$ *International Workshop on Logical Frameworks and Meta-Languages : Theory and Practice (LFMTP 2006)*, volume 147(5) of *Electronic Notes in Theoretical Computer Science*, pages 25–32, 2006.

[BU08] Stefan Berghofer and Christian Urban. Nominal inversion principles. In *21st International Conference on Theorem Proving in Higher Order Logics*, 2008. To appear.

[Bul08] Peter Bulychev. Duplicate code detection using Code Digger. Python Magazine, 2008.

[Bur91] Rod Burstall. Computer assisted proof for mathematics: an introduction using the LEGO proof system. Technical Report ECS-LFCS-91-132, LFCS, Department of Informatics, University of Edinburgh, 1991.

[Cal09] Christophe Calvès. A Haskell nominal toolkit. In *Informal Proceedings of the 2nd International Workshop on Theory and Applications of Abstraction, Substitution and Naming (TAASN 2009)*, page Available Online, 2009.

[Cal10] Christophe Calvès. *Complexity and Implementation of Nominal Algorithms*. PhD thesis, Department of Computer Science, King's College London, 2010.

[CC02] Luís Caires and Luca Cardelli. A spatial logic for concurrency (part ii). In *13th International Conference on Concurrency (CONCUR 2002)*, pages 209–225, 2002.

[CELM00] Manuel Clavel, Steven Eker, Patrick Lincoln, and Joseph Meseguer. Principles of Maude. In *Electronic Notes in Theoretical Computer Science*, volume 4, 2000.

[CF07] Christophe Calvès and Maribel Fernández. Implementing nominal unification. *Electronic Notes in Theoretical Computer Science*, 176(1):25–37, 2007.

[CF08a] Christophe Calvès and Maribel Fernández. Nominal matching and α-equivalence (extended abstract). In *Workshop on Logic, Language and Information in Computation (WoLLIC) 2008*, pages 111–122, 2008.

[CF08b] Christophe Calvès and Maribel Fernández. A polynominal nominal unification algorithm. *Theoretical Computer Science*, 403:285–306, 2008.

[CH85] Thierry Coquand and Gerard Huet. Constructions: a higher-order proof system for mechanising mathematics. In *Proceedings of the European Conference on Computer Algebra (EUROCAL 1985)*, volume 203 of *Lecture Notes in Computer Science*, pages 151–184, 1985.

[CH06] Karl Crary and Robert Harper. *Logic Column 16: Higher-Order Abstract Syntax—Setting the Record Straight*, volume 37, pages 93–96. Association of Computing Machinery, 2006.

[Che04a] James Cheney. *The Complexity of Equivariant Unification*, pages 332–344. Lecture Notes in Computer Science. Springer, 2004.

[Che04b] James Cheney. *Nominal Logic Programming*. PhD thesis, Cornell University, 2004.

[Che05a] James Cheney. Equivariant unification. In *Rewriting Techniques and Applications (RTA 2005)*, pages 74–89. Springer, 2005.

[Che05b] James Cheney. Functional pearl – scrap your nameplate. In *Proceedings of the Tenth ACM SIGPLAN International Conference on Functional Programming*, pages 180–191, 2005.

[Che05c] James Cheney. *Logic Column 14: Nominal Logic and Abstract Syntax*, volume 36, pages 47–69. Association of Computing Machinery, 2005.

[Che05d] James Cheney. Relating higher order pattern unification and nominal unification. In *Proceedings of the 19th International Conference on Unification (UNIF 2005)*, pages 104–119, 2005.

[Che05e] James Cheney. A simpler proof theory for Nominal Logic. In *Foundations of Software Science and Computational Structures*, pages 379–394, 2005.

[Che06a] James Cheney. Completeness and Herbrand results for Nominal Logic. *Journal of Symbolic Logic*, 81(1):299–320, 2006.

[Che06b] James Cheney. The semantics of Nominal Logic programs. In *International Conference on Logic Programming 2006*, pages 361–375, 2006.

[Che08] James Cheney. Simple nominal type theory. In *Proceedings of the 3$^{\text{rd}}$ International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice (LFMTP 2008)*, 2008.

[Chl08] Adam Chlipala. Parametric higher-order abstract syntax for mechanized semantics. In *13th ACM SIGPLAN International Conference on Functional Programming 2008 (ICFP 2008)*, 2008. To appear.

[Chu36] Alonzo Church. A note on the Entscheidungsproblem. *The Journal of Symbolic Logic*, 1(1):40–41, 1936.

[Chu40] Alonzo Church. A formulation of the simple theory of types. *The Journal of Symbolic Logic*, 5:56–68, 1940.

[CM07] James Cheney and Alberto Momigliano. Mechanized metatheory model-checking. In *Proceedings of the 9th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP 2007)*, pages 75–86, 2007.

[CMU08] Peter Chapman, James MacKinna, and Christian Urban. Mechanising a proof of Craig's interpolation theorem for intuitionistic logic in Nominal Isabelle. In *Proceedings of the 9th AISC International Conference, the 15th Calculemas Symposium, and the 7th International MKM Conference on Intelligent Computer Mathematics*, pages 38–52, 2008.

[Coh63] Paul J. Cohen. The independence of the continuum hypothesis. *Proceedings of the National Academy of Sciences of the United States of America*, 50:1143–1148, 1963.

[CP07] Ranald A. Clouston and Andrew M. Pitts. Nominal equational logic. In *Computation, Meaning and Logic, Articles Dedicated to Gordon Plotkin*, volume 1496 of *Electronic Notes in Theoretical Computer Science*, pages 223–257. Elsevier, 2007.

[CU03] James Cheney and Christian Urban. System description: α-Prolog, a fresh approach to logic programming modulo α-equivalence. In *Proceedings of the 17th International Workshop on Unification*, pages 15–19, 2003.

[CU04] James Cheney and Christian Urban. αProlog: A logic programming language with names, binding and α-equivalence. In *Proceedings of the 20th International Conference on Logic Programming (ICLP 2004)*, number 3132 in Lecture Notes in Computer Science, pages 269–283. Springer-Verlag, 2004.

[CU08] James Cheney and Christian Urban. Nominal Logic programming. *ACM Transactions on Programming Language Systems*, 30(5):1–47, 2008.

[Dam98] Laurent Dami. A lambda-calculus with dynamic binding. *Theoretical Computer Science*, 192:201–231, 1998.

[dB72] Nicolaas G. de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation. *Indagationes Mathematicae*, 34:381–392, 1972.

[dB78] Nicholaas G. de Bruijn. A namefree λ-calculus with facilities for internal definition of expressions and segments. Technical Report 78-WSK-03, Technische Universiteit Eindhoven, 1978.

[dB80] Nicolas G. de Bruijn. A survey of the project AUTOMATH. In *To Haskell B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*. Academic Press, 1980.

[DFH95] Joëlle Despeyroux, Amy Felty, and André Hirschowitz. Higher-order abstract syntax in Coq. In *Proceedings of the International Conference on Typed Lambda Calculi and Applications*, volume 902 of *Lecture Notes in Computer Science*, pages 124–138. Springer-Verlag, 1995.

[DG10] Gilles Dowek and Murdoch J. Gabbay. Permissive Nominal Logic. In *Proceedings of the 12th International ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming (PPDP 2010)*, pages 165–176, 2010.

[DGM09a] Gilles Dowek, Murdoch J. Gabbay, and Dominic P. Mulligan. Permissive nominal terms. In *Proceedings of the 24th Convegno Italiano di Logica Computazionale (CILC 2009)*, 2009.

[DGM09b] Gilles Dowek, Murdoch J. Gabbay, and Dominic P. Mulligan. Permissive nominal terms and their unification. Technical Report 6682, INRIA, 2009.

[DGM10] Gilles Dowek, Murdoch J. Gabbay, and Dominic P. Mulligan. Permissive nominal terms and their unification: An infinite, coinfinite approach to nominal techniques. *Logic Journal of the Interest Group in Pure and Applied Logic*, 18:769–822, 2010.

[DHK95] Gilles Dowek, Thérèse Hardin, and Claude Kirchner. Higher-order unification via explicit substitutions. In *Proceedings of the 10th Annual IEEE Symposium on Logic in Computer Science (LICS 1995)*, pages 366–374, 1995.

[DM04] Davide D'Ancona and Eugenio Moggi. A fresh calculus for names management. In *Proceedings of the 3rd International Conference on Generative Programming and Component Engineering (GPCE 2004)*, pages 206–224, 2004.

[Dow01] Gilles Dowek. *Higher-order unification and matching*, pages 1009–1062. Elsevier, 2001.

[DP08] Joshua Dunfield and Brigitte Pientka. Case analysis of higher-order data. In *International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice (LFMTP'08)*, Electronic Notes in Theoretical Computer Science, 2008.

[DST09] Lucas Dixon, Alan Smaill, and Tracy Tsang. Plans, actions and dialogues using linear logic. *Journal of Logic, Language and Information*, 18(2):251–289, 2009.

[FG05] Maribel Fernández and Murdoch J. Gabbay. Nominal rewriting with name generation: Abstraction vs. locality. In *PPDP '05: Proceedings of the 7th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*, pages 47–58, 2005.

[FG07a] Maribel Fernández and Murdoch J. Gabbay. Curry-style types for nominal terms. In *Proceedings of TYPES'06*, Lecture Notes in Computer Science, 2007.

[FG07b] Maribel Fernández and Murdoch J. Gabbay. Nominal rewriting. *Information and Computation*, 205:917–965, 2007.

[FG10] Maribel Fernández and Murdoch J. Gabbay. Closed nominal rewriting and efficiently computable nominal algebra equality. In *Proceedings of the 5th International Workshop on Logical Frameworks and Meta-languages: Theory and Practice (LFMTP 2010)*, 2010.

[FGM04] Maribel Fernández, Murdoch J. Gabbay, and Ian Mackie. Nominal rewriting systems. In *PPDP '04: Proceedings of the 6th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*, pages 108–119, 2004.

[FP10] Amy P. Felty and Brigitte Pientka. Reasoning with higher-order abstract syntax and contexts. In *Proceedings of the International Conference on Theorem Proving (ITP 2010)*, Lecture Notes in Computer Science, 2010.

[FR10] Maribel Fernández and Albert Rubio. Reduction orderings and completion for rewrite systems with binding. In *Proceedings of the 5th International Workshop on Higher-Order Rewriting (HOR 2010)*, 2010.

[Fra22] Abraham Fraenkel. Der begriff 'definit' und die unabhngigkeit des auswahlsaxioms. *Sitzungsberichte der Preussischen Akademie der Wissenschaften, Physikalisch-mathematische Klasse*, pages 253–257, 1922.

[FS96] Leonidas Fegaras and Timothy Sheard. Revisiting catamorphisms over datatypes with embedded functions. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 1996)*, pages 284–294, 1996.

[FS09] Marcelo Fiore and Sam Staton. A congruence rule format for name-passing process calculi. *Information and Computation*, 207(2):209–236, 2009.

[Gab00] Murdoch J. Gabbay. *A Theory of Inductive Definitions with α-equivalence.* PhD thesis, DPMMS, University of Cambridge, 2000.

[Gab02a] Murdoch J. Gabbay. Automating Fraenkel-Mostowski syntax. In *Proceedings of the 15th International Conference on Theorem Proving in Higher Order Logics (TPHOLS 2002)*, pages 60–70, 2002.

[Gab02b] Murdoch J. Gabbay. FM-HOL, a higher-order theory of names. In *Workshop on Thirty Five Years of Automath*, 2002.

[Gab03] Murdoch J. Gabbay. The π-calculus in FM. In Fairouz Kamareddine, editor, *35 Yes of AUTOMATH*, Applied Logic Series. Kluwer, 2003.

[Gab05] Murdoch J. Gabbay. A NEW calculus of contexts. In *PPDP '05: Proceedings of the 7th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*, pages 94–105, 2005.

[Gab07a] Murdoch J. Gabbay. Fresh Logic. *Journal of Applied Logic*, 2007. In Press.

[Gab07b] Murdoch J. Gabbay. A general mathematics of names. *Information and Computation*, 205:982–1011, 2007.

[Gab09] Murdoch J. Gabbay. Nominal algebra and the HSP theorem. *Journal of Logic and Computation*, 19:341–367, 2009.

[Gac08] Andrew Gacek. The Abella interactive theorem prover (system description). In *Proceedings of the 4th International Joint Conference on Automated Reasoning (IJCAR 2008)*, volume 5195 of *Lecture Notes in Artificial Intelligence*, pages 154–161, 2008.

[GC04] Murdoch J. Gabbay and James Cheney. A sequent calculus for Nominal Logic. In *19th Annual IEEE Symposium on Logic in Computer Science (LICS 2004)*, pages 139–148, 2004.

[GG08] Murdoch J. Gabbay and Michael Gabbay. Substitution for Fraenkel-Mostowski foundations. In *Proceedings of the 2008 AISB Symposium on Computing and Philosophy*, 2008.

[GJ02] Herman Geuvers and Gueorgui I. Jojgov. Open proofs and open terms: A basis for interactive logic. In *CSL '02: Proceedings of the 16th International Workshop and 11th Annual Conference of the EACSL on Computer Science Logic*, pages 537–552, 2002.

[GJSB05] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Definition.* Addison-Wesley, 3rd edition, 2005.

[GK96a] Claire Gardent and Michael Kohlasse. Focus and higher-order unification. In *Proceedings of the 16th Conference on Computational Linguistics (ICCL 1996)*, pages 430–435, 1996.

[GK96b] Claire Gardent and Michael Kohlasse. Higher-order coloured unification and natural language semantics. In *Proceedings of the 34th Annual Meeting of the Association for Computational Linguistics (ACL 1996)*, pages 1–9, 1996.

[GL08] Murdoch J. Gabbay and Stéphane Lengrand. The lambda context calculus. *Electronic Notes in Theoretical Computer Science*, 196:19–35, 2008.

[GM06] Murdoch J. Gabbay and Aad Mathijssen. Nominal algebra. In *Proceedings of the 18th Nordic Workshop on Programming Theory (NWPT'06)*, page Talk Abstract, 2006.

[GM07a] Murdoch J. Gabbay and Aad Mathijssen. *Festschrift in Honour of Peter B. Andrews on his 70th Birthday*, chapter The Lambda-calculus is Nominal Algebraic. Studies in Logic and the Foundations of Mathematics. IFCoLog, 2007.

[GM07b] Murdoch J. Gabbay and Aad Mathijssen. A formal calculus for informal equality with binding. In *Proceedings of the 14th Workshop on Logic, Language, Information and Computation 2007, (WoLLIC 2007)*, Lecture Notes in Computer Science, pages 162–176, 2007.

[GM07c] Murdoch J. Gabbay and Aad Mathijssen. Nominal algebra and the HSP theorem. Technical Report HW-MACS-TR-0057, Heriot-Watt University, DSG, 2007.

[GM07d] Murdoch J. Gabbay and Aad Mathijssen. One-and-a-halfth order logic. *Journal of Logic and Computation*, 2007.

[GM08a] Murdoch J. Gabbay and Aad Mathijssen. Capture avoiding substitution as a nominal algebra. *Formal Aspects of Computing*, 20:451–479, 2008.

[GM08b] Murdoch J. Gabbay and Dominic P. Mulligan. One-and-a-halfth order terms: Curry-Howard for incomplete derivations. In *Proceedings of 2008 Workshop on Logic, Language and Information in Computation (WoLLIC 2008)*, volume 5110 of *Lecture Notes in Artificial Intelligence*, pages 180–194, 2008.

[GM09a] Murdoch J. Gabbay and Aad Mathijssen. Nominal universal algebra: Equational logic with names and binding. *Journal of Logic and Computation*, 19:1455–1580, 2009.

[GM09b] Murdoch J. Gabbay and Dominic P. Mulligan. One-and-a-half-order terms: Curry-Howard for incomplete derivations. *Journal of Information and Computation*, 208:230–258, 2009.

[GM09c] Murdoch J. Gabbay and Dominic P. Mulligan. Semantic nominal terms. In *Informal proceedings of the 2$^{\text{nd}}$ International Workshop on the Theory and Applications of Abstraction, Substitution and Naming (TAASN 2009)*, 2009.

[GM09d] Murdoch J. Gabbay and Dominic P. Mulligan. The two-level λ-calculus. In *Proceedings of the 17$^{\text{th}}$ Workshop on Functional and Logic Programming (WFLP 2008)*, volume 246 of *Electronic Notes in Theoretical Computer Science*, pages 107–129, 2009.

[GM09e] Murdoch J. Gabbay and Dominic P. Mulligan. Universal algebra over λ-terms and nominal terms: the connection in logic between nominal techniques and higher-order variables. In *Informal Proceedings of the 4$^{\text{th}}$ International Workshop on Logical Frameworks and Meta-languages: Theory and Practice (LFMTP 2009)*, 2009.

[GM10a] Murdoch J. Gabbay and Aad Mathijssen. A nominal axiomatisation of the lambda-calculus. *Information and Computation*, 20:501–531, 2010.

[GM10b] Murdoch J. Gabbay and Dominic P. Mulligan. The permissive two-level λ-calculus. To be submitted, 2010.

[GM10c] Murdoch J. Gabbay and Dominic P. Mulligan. The two-level λ-calculus, part one. *Journal of Logic and Computation*, 2010. Submitted.

[Gol81] Warren D. Goldfarb. The undecidability of the second-order unification problem. *Theoretical Computer Science*, 13:225–230, 1981.

[Gor93] Andrew D. Gordon. A mechanisation of name carrying syntax up to alpha conversion. In *Proceedings of Higher Order Logic Theorem Proving and its Applications*, Lecture Notes in Computer Science, pages 414–426, 1993.

[GP99] Murdoch J. Gabbay and Andrew M. Pitts. A NEW approach to abstract syntax involving binders. In *14th Annual Symposium on Logic in Computer Science*, pages 214–224, 1999.

[Has98] Masatomo Hashimoto. First-class contexts in ML. In *Proceedings of Advances in Computing Science (ASIAN 1998)*, pages 206–223, 1998.

[HDKP98] Thérèse Hardin, Gilles Dowek, Claude Kirchner, and Frank Pfenning. Unification via explicit substitutions: The case of higher-order patterns. In *Proceedings of Joint International Conference and Symposium on Logic Programming (JICSLP 1996)*, pages 259–273, 1998.

[Hid00] Sakurada Hideki. An interpretation of a context calculus in an environment calculus. *Transactions of Information Processing Society of Japan*, 41:8–24, 2000.

[HLZ09] Robert Harper, Daniel R. Licata, and Noam Zeilberger. A pronominal approach to binding and computation. In *Proceedings of the 9th International Conference on Typed Lambda Calculi and Applications (TLCA 2009)*, pages 3–4, 2009.

[HO01] Masatomo Hashimoto and Atsushi Ohori. A typed context calculus. *Theoretical Computer Science*, 266(1–2):249–272, 2001.

[HU10] Brian Huffman and Christian Urban. Proof Pearl: A new foundation for Nominal Isabelle. In *Proceedings of the International Conference on Interactive Theorem Proving (ITP 2010)*, 2010.

[Hue75] Gerard Huet. A unification algorithm for typed lambda calculus. *Theoretical Computer Science*, 1:27–57, 1975.

[Hue97] Gérard Huet. Functional pearl: The zipper. *Journal of Functional Programming*, 7(5), 1997.

[Joj03] Gueorgui I. Jojgov. *Holes with Binding Power*, volume 2646/2003 of *Lecture Notes in Computer Science*, page 617. Springer, 2003.

[JW95] Winfried Just and Martin Weese. *Discovering Modern Set Theory I: The Basics.* American Mathematical Society, 1995.

[KC93] Jean Louis Krivine and Rene Cori. *Lambda calculus, types and models.* Ellis Horwood, 1993.

[KM08] Temesghen Kahsai and Marino Miculan. Implementing spi calculus using nominal techniques. In *Proceedings of the 4th conference on Computability in Europe (CiE 2008)*, pages 294–305, 2008.

[KN10] Ramana Kumar and Michael Norrish. (Nominal) Unification by recursive descent with triangular substitutions. In *International Conference on Interactive Theorem Proving (ITP 2010)*, 2010.

[KP09] Alexander Kurtz and Daniela Petrişan. Towards universal algebra over nominal sets. In *Informal Proceedings of Topology, Algebra and Categories in Logic (TACL 2009)*, 2009.

[KP10] Alexander Kurz and Daniela Petrişan. On universal algebra over nominal sets. To appear, 2010.

[KPV10] Alexander Kurz, Daniela Petrişan, and Jiří Velebil. Algebraic theories over nominal sets. Submitted, 2010.

[Lak09] Matthew R. Lakin. Representing names with variables in nominal abstract syntax. In *Informal Proceedings of the 2nd International Workshop on Theory and Applications of Abstraction, Substitution and Naming (TAASN 2009)*, 2009.

[Lam88] John Lamping. A unified system of parameterization for programming languages. In *Proceedings of the ACM Conference on Lisp and functional programming (LFP 1988)*, pages 316–326, 1988.

[Ler07] Xavier Leroy. A locally nameless solution to the POPLmark challenge. Technical Report 6098, INRIA, 2007.

[LF96] Shinn-Der Lee and Daniel P. Friedman. Enriching the λ-calculus with contexts: Toward a theory of incremental program construction. In *Proceedings of the 1st ACM SIGPLAN International Conference on Functional Programming (ICFP 1996)*, pages 239–250, 1996.

[LP08] Matthew Lakin and Andrew M. Pitts. A metalanguage for structural operational semantics. In *Trends in Functional Programming Volume 8*, pages 19–35. Intellect, 2008.

[LP09] Matthew R. Lakin and Andrew M. Pitts. Resolving inductive definitions with binders in higher-order typed functional programming. In *Proceedings of the 18th European Symposium on Programming (ESOP 2009)*, volume 5502 of *Lecture Notes in Computer Science*, pages 47–61, 2009.

[LP10] Matthew R. Lakin and Andrew M. Pitts. Encoding abstract syntax without fresh names. *Journal of Automated Reasoning*, 2010. To appear.

[Lug94] Denis Lugiez. Higher order disunification: Some decidable cases. In *Proceedings of the 1st International Conference on Constraints in Computational Logics (CCL 1994)*, pages 121–135, 1994.

[LV08] Jordi Levy and Mateu Villaret. Nominal unification from a higher-order perspective. In *Proceedings of the 19th international conference on Rewriting Techniques and Applications (RTA 2008)*, pages 246–260, 2008.

[LV10] Jordi Levy and Mateu Villaret. An efficient nominal unification algorithm. In *Proceedings of the 21st International Conference on Rewriting Techniques and Applications (RTA 2010)*, 2010.

[LZH08] Daniel R. Licata, Noam Zeilberger, and Robert Harper. Focusing on binding and computation. In *Proceedings of the 23rd Annual IEEE Symposium on Logic in Computer Science (LICS 2008)*, pages 241–252, 2008.

[Mas99] Ian A. Mason. Computing with contexts. *Higher-order and Symbolic Computation*, 12:171–201, 1999.

[Mat07] Aad Mathijssen. *Logical Calculi for Reasoning with Binding.* PhD thesis, Technische Universiteit Eindhoven, 2007.

[McB04] Conor McBride. The Epigram tutorial. Association of Functional Programming Summer School, 2004.

[MGR06] MohammadReza Mousavi, Murdoch J. Gabbay, and Michel A. Reniers. Nominal SOS. In *Proceedings of the 18th Nordic Workshop on Programming Theory (NWPT'06)*, page Talk Abstract, 2006.

[Mil91a] Dale Miller. A logic programming language with λ-abstraction, function variables and simple unification. *Journal of Logic and Computation*, 1(4):497–536, 1991.

[Mil91b] Dale Miller. Unification of simply-typed λ-terms as logic programming. In *Proceedings of the 8th International Conference on Logic Programming (LP 1991)*, pages 255–269, 1991.

[MM04] Conor McBride and James McKinna. Functional pearl: I am not a number—I am a free variable. In *Haskell '04: Proceedings of the 2004 ACM SIGPLAN Workshop on Haskell*, pages 1–9, 2004.

[MP97] James Mckinna and Randy Pollack. Some lambda calculus and type theory formalized. *Journal of Automated Reasoning*, 23:373–409, 1997.

[MP99] Alberto Momigliano and Frank Pfenning. The relative complement problem for higher-order patterns. In *Proceedings of the International Conference on Logic Programming (ICLP 1999)*, pages 380–394, 1999.

[MP03] Alberto Momigliano and Frank Pfenning. Higher-order pattern complement and the strict λ-calculus. *ACM Transaction on Computational Logic*, 4:493–529, 2003.

[MSH05] Marino Miculan, Ivan Scagnetto, and Furio Honsell. Translating specifications from Nominal Logic to CIC with the theory of contexts. In *Proceedings of the 3rd ACM SIGPLAN Workshop on Mechanized Reasoning about Languages with Variable Binding (MERLIN 2005)*, pages 41–49, 2005.

[MTHM97] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML.* MIT Press, 1997.

[Muñ97] César Muñoz. *A Calculus of Explicit Substitutions for Incomplete Proof Representation in Type Theory.* PhD thesis, INRIA Rocqencourt, Projet Coq, 1997.

[NBF09] Joseph P. Near, William E. Byrd, and Daniel P. Friedmann. αLeanTAP: A declarative theorem prover for first-order classical logic. In *Proceedings of the 25th International Conference on Logic Programming (ICLP 2009)*, Lecture Notes in Computer Science, pages 238–252, 2009.

[Nip91] Tobias Nipkow. Higher-order critical pairs. In *Proceedings of the 6$^{\text{th}}$ IEEE Symposium on Logic in Computer Science (LICS 1991)*, pages 342–349, 1991.

[Nip93a] Tobias Nipkow. Functional unification of higher-order patterns. In *Proceedings of the 8$^{\text{th}}$ IEEE Symposium on Logic in Computer Science (LICS 1993)*, pages 64–74, 1993.

[Nip93b] Tobias Nipkow. Orthogonal higher-order rewrite systems are confluent. In *Proceedings of the International Conference on Typed Lambda Calculi and Applications (TLCA 1993)*, volume 664 of *Lecture Notes in Computer Science*, pages 306–317, 1993.

[Nip98] Tobias Nipkow. *Term Rewriting and All That.* Cambridge University Press, 1998.

[NM88] Gopalan Nadathur and Dale Miller. An overview of λ-Prolog. In *Fifth International Logic Programming Conference*, pages 810–827, 1988.

[Nor06] Michael Norrish. Mechanising λ-calculus using a classical first order theory of terms with permutations. *Higher Order Symbolic Computation*, 19(2–3):169–195, 2006.

[Nor09] Ulf Norell. A brief overview of Agda—a functional language with dependent types. In *Proceedings of the 22$^{\text{nd}}$ International Conference on Theorem Proving in Higher-order Logics (TPHOLS 2009)*, pages 73–78, 2009.

[NU08] Julien Narboux and Christian Urban. Formalising in Nominal Isabelle Crary's completeness proof for equivalence checking. *Electronic Notes in Theoretical Computer Science*, 196:3–18, 2008.

[NU09] Julian Narboux and Christian Urban. Nominal verification of typical SOS proofs. In *Proceedings of the 3$^{\text{rd}}$ Workshop on Logical and Semantic Frameworks with Applications (LFSA 2008)*, Electronic Notes in Theoretical Computer Science, pages 139–155, 2009.

[NV07] Michael Norrish and René Vestergaard. Proof pearl: de Bruijn terms really do work. In *Proceedings of the 20th International Conference on Theorem Proving in Higher Order Logics (TPHOLS 2007)*, pages 207–222, 2007.

[OSW05] Cosmin Oancea, Claire So, and Steven M. Watt. Generalization in Maple. Maple Conference, 2005.

[Pau85] Lawrence Paulson. Natural deduction as higher-order resolution (revised edition). Technical Report UCAM-CL-TR-83, University of Cambridge, Computer Laboratory, 1985.

[Pau88] Lawrence C. Paulson. Isabelle: The next 700 theorem provers (UCAM-CL-TR-143). Technical report, University of Cambridge, Computer Laboratory, 1988.

[Pau98] Lawrence C. Paulson. Strategic principles in the design of Isabelle. In *Workshop on Strategies in Automated Deduction*, pages 11–17, 1998.

[Pau10] Lawrence C. Paulson. *Isabelle's Logics*, 2010.

[PD08] Brigitte Pientka and Joshua Dunfield. Programming with proofs and explicit contexts. In *ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming (PPDP '08)*, 2008.

[PD10] Brigitte Pientka and Joshua Dunfield. Beluga: A framework for programming and resoning with deductive systems (system description). In *Proceedings of the International 5th Joint Conference on Automated Reasoning (IJCAR 2010)*, 2010.

[PE88] Frank Pfenning and Conal Elliot. Higher-order abstract syntax. In *PLDI '88: Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*, pages 199–208, 1988.

[Pfe88] Frank Pfenning. Partial polymorphic type inference and higher-order unification. In *Proceedings of the 1988 ACM Conference on Lisp and Functional Programming*, 1988.

[Pfe91] Frank Pfenning. Unification and anti-unification in the Calculus of Constructions. In *Proceedings of the* 6th *Annual IEEE Symposium on Logic in Computer Science (LICS 1991)*, pages 74–85, 1991.

[PG00] Andrew M. Pitts and Murdoch J. Gabbay. A metalanguage for programming with bound names modulo renaming. In *Proceedings of 5th International Conference on the Mathematics of Program Construction. (MPC2000)*, volume 1837 of *Lecture Notes in Computer Science*, pages 230–255, 2000.

[Pie06] Brigitte Pientka. Eliminating redundancy in higher-order unification: a lightweight approach. In *Proceedings of the 3rd International Joint Conference on Automated Reasoning (IJCAR 2006)*, Lecture Notes in Computer Science, pages 362–376, 2006.

[Pit94] Andrew M. Pitts. Some notes on inductive and co-inductive techniques in the semantics of functional programs. Technical Report NS-94-5, BRICS, Department of Computer Science, University of Aarhus, 1994.

[Pit97] Andrew M. Pitts. Operationally-based theories of program equivalence. In *Semantics and Logics of Computation*, pages 241–298, 1997.

[Pit03] Andrew M. Pitts. Nominal Logic, a first order theory of names and binding. *Information and Computation*, 186:165–193, 2003.

[Pit10] Andrew M. Pitts. Nominal System T. In *Proceedings of the 37th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL 2010)*, 2010. To appear.

[PJ02] Simon Peyton-Jones, editor. *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, 2002.

[Plo70] Gordon Plotkin. A note on inductive generalisation. *Machine Intelligence*, 5:153–163, 1970.

[Pol93] Randy Pollack. Closure under alpha-conversion. In *Informal Proceedings of the Workshop on Types for Proofs and Programs (TYPES 1993)*, pages 313–332, 1993.

[Pot07] François Pottier. Static name control for FreshML. In *LICS '07: Proceedings of the 22nd Annual IEEE Symposium on Logic in Computer Science*, pages 356–365, 2007.

[PP03] Brigitte Pientka and Frank Pfenning. Optimizing higher-order pattern unification. In *Proceedings of the 19th International Conference on Automated Deduction (CADE 2003)*, pages 473–487, 2003.

[PP10] Nicolas Pouillard and François Pottier. A fresh look at programming with names and binders. Submitted, 2010.

[PS08a] Andrew M. Pitts and Mark Shinwell. Generative unbinding of names. *Logical Methods in Computer Science*, 4(1:4):1–33, 2008.

[PS08b] Adam Poswolsky and Carsten Schürmann. Practical programming with higher-order encodings and dependent types. In *Proceedings of the 17th European Conference on Programming Languages and Systems (EAPLS 2008)*, pages 93–107, 2008.

[Qia93] Zhenyu Qian. Linear unification of higher-order patterns. In *Proceedings of the International Joint Conference CAAP/FASE on Theory and Practice of Software Development (TAPSOFT 1993)*, pages 391–405, 1993.

[Rey70] John C. Reynolds. Transformational systems and the algebraic structure of atomic formulas. *Machine Intelligence*, 5:135–151, 1970.

[San98] David Sands. Computing with contexts: A simple approach. *Electronic Notes in Theoretical Computer Science*, 10:134–149, 1998.

[Sat08] Masahiko Sato. External and internal syntax of the λ-calculus. In *Symbolic Computation in Software Science Austrian-Japanese Workshop (SCSS 2008)*, 2008.

[Sch06] Ulrich Schöpp. *Names and Binding in Type Theory.* PhD thesis, Department of Informatics, University of Edinburgh, 2006.

[Sch07] Ulrich Schöpp. Modelling generic judgements. *Electronic Notes in Theoretical Computer Science*, 174(5):19–35, 2007.

[SG95] Morten Heine Sørensen and Robert Glück. An algorithm of generalization in positive supercompilation. In *Proceedings of the International Logic Programming Symposium (ILPS 1995)*, pages 465–479, 1995.

[Shi03] Mark R. Shinwell. Swapping the atom: Programming with binders in FreshO'Caml. In *Proceedings of the 2nd ACM SIGPLAN Workshop on Mechanized Reasoning about Languages with Variable Binding (MERLIN 2003)*, 2003.

[Shi05] Mark R. Shinwell. *The Fresh Approach: Functional Programming with Names and Binders.* PhD thesis, Computer Laboratory, University of Cambridge, 2005.

[Smu95] Raymond Smullyan. *First-order logic.* Dover, 1995.

[SNO$^+$10] Peter Sewell, Francesco Zappa Nardelli, Scott Owens, Gilles Peskine, Thomas Ridge, Susmit Sarkar, and Rok Strniša. Ott: Effective tool support for the working semanticist. *Journal of Functional Programming*, 20(1):71–122, 2010.

[SP05] Mark R. Shinwell and Andrew M. Pitts. On a monadic semantics for freshness. *Theoretical Computer Science*, 342:28–55, 2005.

[SP10] Masahijo Sato and Randy Pollack. A canonical locally named representation of binding. *Journal of Automated Reasoning*, 2010. Under review.

[SPG03] Mark R. Shinwell, Andrew M. Pitts, and Murdoch J. Gabbay. FreshML: Programming with binders made simple. In *Eighth ACM SIGPLAN International Conference on Functional Programming (ICFP 2003), Uppsala, Sweden*, pages 263–274, 2003.

[SPS05] Carsten Schürmann, Adam Poswolsky, and Jeffrey Sarnat. The ∇-calculus: functional programming with higher-order encodings. In *Proceedings of the* 7th *International Conference on Typed Lambda Calculi and Applications (TLCA 2005)*, pages 339–353, 2005.

[SS04] Ulrich Schöpp and Iain Stark. A dependent type theory with names and binding. In *Computer Science Logic*, volume 3210 of *Lecture Notes in Computer Science*, pages 235–249, 2004.

[SSK02] Masahiko Sato, Takafumi Sakurai, and Yukiyoshi Kameyama. A simply typed context calculus with first-class environments. *Journal of Functional and Logic Programming*, 2002:359–374, 2002.

[SSKI03] Masahiko Sato, Takafumi Sakurai, Yukiyoshi Kameyama, and Atsushi Igarashi. Calculi of meta-variables. In *Computer Science Logic*, volume 2803 of *Lecture Notes in Computer Science*, pages 484–497, 2003.

[Sta06] Sam Staton. *Name-Passing Process Calculi: Operational Models and Structural Operational Semantics.* PhD thesis, Computer Laboratory, University of Cambridge, 2006.

[Ste00] Mark E. Stehr. CINNI—a generic calculus of explicit contexts, and it's application to λ-, π- and ς-calculi. *Electronic Notes in Theoretical Computer Science*, 36:70–92, 2000.

[Tak95] Masahako Takahashi. Parallel reduction in the λ-calculus. *Information and Computation*, 118(1):120–127, 1995.

[Tea10] The Glasgow Haskell Compiler Team. User guide: Glasgow Haskell Compiler, version 6.12.1, 2010.

[Tiu08] Alwen Tiu. On the role of names in reasoning about λ-tree syntax. In *Proceedings of the International Workshop on Logical Frameworks and Metalanguages: Theory and Practice (LFMTP 2008)*, 2008.

[TW09] David Turner and Glyn Winskel. Nominal domain theory for concurrency. In *Proceedings of the 23rd International Conference on Computer Science Logic (CSL 2009) and the 18th European Annual Conference on Computer Science Logic (EACSL 2009)*, Lecture Notes in Computer Science, pages 546–560, 2009.

[Tze07] Nikos Tzevelekos. Full abstraction for nominal general references. In *Proceedings of the 22nd Annual IEEE Symposium on Logic in Computer Science (LICS 2007)*, pages 399–410, 2007.

[UB06] Christian Urban and Stefan Berghofer. A recursion operator for nominal datatypes implemented in Isabelle/HOL. In *Proceedings of the 3rd International Joint Conference on Automated Deduction (IJCAR 2006)*, pages 498–512, 2006.

[UBN07] Christian Urban, Stefan Berghofer, and Michael Norrish. Barendregt's variable convention in rule inductions. In *Proceedings of the 21st International Conference on Automated Deduction (CADE-21)*, pages 35–50, 2007.

[UC05] Christian Urban and James Cheney. Avoiding equivariance in α-Prolog. In *Proceedings of the 7th International Conference on Typed Lambda Calculi and Applications (TLCA 2005)*, pages 401–416, 2005.

[UCB08] Christian Urban, James Cheney, and Stefan Berghofer. Mechanizing the metatheory of LF. In *Logic in Computer Science (LICS 2008)*, pages 45–46, 2008.

[UN05] Christian Urban and Michael Norrish. A formal treatment of the Barendregt convention in rule inductions (extended abstract). In *Proceedings of the ACM Workshop on Mechanized Reasoning about Languages with Variable Binding and Names (MERLIN 2005)*, pages 25–32, 2005.

[UN09] Christian Urban and Tobias Nipkow. Nominal verification of algorithm W. In Gerard Huet, Jean-Jacques Lévy, and Gordon Plotkin, editors, *From Semantics to Computer Science: Essays in Honour of Gilles Kahn*, pages 363–382. Cambridge University Press, 2009.

[UNB07] Christian Urban, Julien Narboux, and Stefan Berghofer. *The Nominal Datatype Package (Preliminary Manual)*, 2007.

[UPG04] Christian Urban, Andrew M. Pitts, and Murdoch J. Gabbay. Nominal unification. *Theoretical Computer Science*, 323(1–3):473–497, 2004.

[Urb08] Christian Urban. How to prove false using the variable convention. Poster, Tools and Techniques for Verification of System Infrastructure, 2008.

[US06] Pawel Urzyczyn and Morten Sørensen. *Lectures on the Curry-Howard Isomorphism*, volume 149 of *Studies in Logic*. Elsevier, 2006.

[UT05] Christian Urban and Christine Tasson. Nominal techniques in Isabelle/HOL. In *Proceedings of the 20th Conference on Automated Deduction (CADE)*, volume 3632 of *Lecture Notes in Artificial Intelligence*, pages 38–53, 2005.

[UZ08] Christian Urban and Bozhi Zhu. Revisiting cut-elimination: One difficult proof is really a proof. In *Proceedings of the 19*th *International Conference on Rewriting Techniques and Applications (RTA 2008)*, pages 409–424, 2008.

[Wad87] Phil Wadler. Views: A way for pattern matching to cohabit with data abstraction. In *Proceedings of the 14*th *ACM Symposium on Principles of Programming Languages (POPL 1987)*, pages 307–313, 1987.

[Win93] Glynn Winskel. *The Formal Semantics of Programming Languages (An Introduction).* Foundations of Computing. The MIT Press, 1993.

[WSA09] Edwin Westbrook, Aaron Stump, and Evan Austin. The calculus of nominal inductive constructions: An intensional approach to encoding name-bindings. In *Proceedings of the 4*th *International Workshop on Logical Frameworks and Meta-Languages (LFMTP 2009)*, pages 74–83, 2009.

[WW03] Geoffrey Washburn and Stephanie Weirich. Boxes go bananas: Encoding higher order abstract syntax with parametric polymorphism. In *Proceedings of the 8th ACM SIGPLAN International Conference on Functional Programming (ICFP 2003)*, pages 249–262, 2003.

[YG08] Ayesha Yasmeen and Elsa L. Gunter. Implementing secure broadcast ambients in Isabelle using Nominal Logic. In *Emerging Trends Report of the 21*st *International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2008)*, pages 123–134, 2008.

[YHT04] Tetsuo Yokoyama, Zhenjiang Hu, and Masato Takeichi. Deterministic higher-order patterns for program transformation. In *Proceedings of the 13*th *International Symposium on Logic Based Program Synthesis and Transformation (LOPSTR 2004)*, volume 3018 of *Lecture Notes in Computer Science*, pages 128–142, 2004.