Saarland University
Faculty of Natural Sciences and Technology I
Department of Computer Science

Master's Thesis

Formalizing $\top\top$-lifting in Isabelle/HOL-Nominal

submitted by

Christian Doczkal

on June 16, 2009

Supervisor

Prof. Dr. Gert Smolka

Advisor

Dr. Jan Schwinghammer

Reviewers

Prof. Dr. Gert Smolka

Dr. Jan Schwinghammer

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides Statt, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Statement under Oath

I confirm under oath that I have written this thesis on my own and that I have not used any other media or materials than the ones referred to in this thesis.

Einverständniserklärung

Ich bin damit einverstanden, dass meine (bestandene) Arbeit in beiden Versionen in die Bibliothek der Informatik aufgenommen und damit veröffentlicht wird.

Declaration of Consent

I agree to make both versions of my thesis (with a passing grade) accessible to the public by having them added to the library of the Computer Science Department.

Saarbrücken, _____ _____

(Datum/Date) (Unterschrift/Signature)

Acknowledgements

I am profoundly in debt to my advisor, Jan Schwinghammer for his continuous support during the last months. Through countless hours of discussion Jan greatly deepened my understanding of the material. I also thank Jan for his invaluable feedback while I was writing this thesis.

I thank my supervisor, Gert Smolka, for giving me the opportunity to work on this interesting topic. The knowledge I obtained in his lectures on logics and semantics proved to be very valuable for this work.

I also thank the people on the Isabelle mailing list for time and again explaining Isabelle's sometimes unexpected behavior.

Last but not least, I want to thank all my family and friends, who have supported me during my entire studies, especially at times where progress was slow and tedious.

Abstract

Handling variable binding is one of the main difficulties in formal proofs. In this context, Moggi's computational metalanguage serves as an interesting case study. It features monadic types and a commuting conversion rule that rearranges the binding structure. Lindley and Stark have given an elegant proof of strong normalization for this calculus. The key construction in their proof is a notion of relational $\top\top$-lifting, using stacks of elimination contexts, to obtain a Girard-Tait style logical relation.

We give a formalization of their proof in Isabelle/HOL-Nominal with a particular emphasis on the treatment of bound variables. Using the Isar structured proof language and the Isabelle document preparation system, we obtain a formal proof document that is suitable for human consumption.

CONTENTS

CHAPTER 1

Introduction

Proving theorems about languages with binding is still one of the major challenges when working with proof assistants [ABF+05]. This thesis contains a formalization of a normalization proof using $\top\top$-lifting, as introduced by Lindley and Stark [LS05]. It uses the Nominal package of the Isabelle proof assistant, to deal with α-equivalence classes of terms.

Proofs of (strong) normalization for lambda calculi have long been used as case studies for the formalization of programming language meta-theory. An early example is the strong normalization proof for System F by Altenkirch [Alt93], other examples include [Abe04, BBLS06, DX07, SS08]. Normalization proofs provide interesting case studies for formalization, because they combine syntactic as well as more semantic arguments about terms and reduction: one must deal with variable binding, renaming, and substitution, but one also employs semantically interesting techniques like logical relations.

Logical relations for monadic types The $\top\top$-lifting technique is a method for proofs via logical relations. It allows us to handle type constructors with elimination constructs that are not inductive on the type structure. One example for such types is the monadic type constructor T of Moggi's computational metalanguage [Mog91], which provides a type-theoretic framework for the description of effectful computations.

In the case of function types $\sigma \to \tau$, the elimination construct, application, yields terms of the smaller type τ. This is used in the definition of the logical

relation at function type. In contrast, the elimination construct for terms of type $T\,\sigma$, the to-binding[1], has the following typing rule:

$$\frac{\Gamma \vdash s : T\,\sigma \qquad \Gamma; x : \sigma \vdash t : T\,\tau}{\Gamma \vdash s \text{ to } x \text{ in } t : T\,\tau}\ ,$$

which results in terms of arbitrarily complex type $T\,\tau$. The $\top\top$-lifting approach uses an auxiliary structure of *stacks* to obtain a logical relation, defined by induction on the type structure, even for the type constructor T. Moreover, the approach does not only apply to Moggi's computational metalanguage λ_{ml} but can also be applied to the λ-calculus with sum-types and to the Call-By-Push-Value calculus [Lev99, Doc07], which has several different computation types.

The Barendregt Variable Convention In the programming language literature, terms are often identified up to α-equivalence. Issues arising from this identification are usually glossed over in informal proofs and [LS05] is no exception. In particular in any proof by induction on the structure of λ-terms, the cases involving binders are only shown for some variable which is chosen to be suitably fresh. This is usually referred to as *Barendregt Variable Convention*: "If $M_1 \dots M_n$ occur in a certain mathematical context (e.g. definition, proof), then in these terms all bound variables are chosen to be different from the free variables"[Bar85, Page 26]. Use of this convention allows fairly slick informal proofs. Unfortunately, formalizing this convention in proof assistants is far from trivial.

Choice of the proof assistant Available proof assistants vary greatly, both in their logical foundations as well as usability and the quality of documentation. One goal of this thesis was to get acquainted with a tool that is applicable to a broad range of applications. This ruled out proof assistants like Twelf[2], which is based on the Edinburgh Logical Framework [HHP93]. Compared with other proof assistants, Twelf has a relatively weak metalogic and is specifically targeted at developing metatheory of programming languages and logics.

Among the general purpose interactive theorem provers, Coq[3] and Isabelle[4] are widely used. The current version of Coq is based on an extension of the Calculus of Constructions [CH88] with inductive types [CP90]. This gives rise to a very powerful logic, the Calculus of inductive Constructions. The main focus of the Coq system is the development of certified programs.

[1] We differ in our notation from [LS05], see Section 2.1 for details and motivation.
[2] http://www.twelf.org
[3] http://coq.inria.fr
[4] http://isabelle.in.tum.de

Isabelle and its main object logic HOL, implementing classical higher-order logic, offer good support for proof automation and are rather well documented. With the Isar structured proof language [Wen02], the Isabelle system is geared towards creating formal proof documents – documents which are machine checkable while remaining human readable.

Another aspect influencing the choice of the proof assistant is the handling of bound variables. In Coq, there is limited support for reasoning using a locally nameless representation (see Section 6.3.1). The Isabelle approach to handling binders builds upon nominal logic [Pit03, Pit06], a logic specifically designed to support names and binding. The Isabelle logic HOL-Nominal is an extension of classical higher-order logic with support for reasoning about names and binding. In particular, it allows the definition of data types with built-in α-equivalence. It also provides induction principles for α-equated terms and inductively defined relations that allow reasoning close to the Barendregt Variable Convention. This, together with my background in higher-order logic and simple type theory, made Isabelle/HOL-Nominal a sensible choice.

Formalizing $\top\top$-lifting The main goal of this thesis is not to increase the trust in Lindley and Stark's strong normalization result on the λ_{ml}-calculus. This result is well known and had already been established by Benton et al. [BBDP98] via translation into the simply-typed λ-calculus with sum types. Instead, our formalization of $\top\top$-lifting is a non-trivial case study with a particular emphasis placed on the handling of bound variables. The reduction relation for λ_{ml} includes the commuting conversion:

$$(s \;\mathsf{to}\; x \;\mathsf{in}\; t) \;\mathsf{to}\; y \;\mathsf{in}\; u \mapsto s \;\mathsf{to}\; x \;\mathsf{in}\; (t \;\mathsf{to}\; y \;\mathsf{in}\; u) \text{ where } x \notin fv(u) \ ,$$

which involves multiple binders and a change in the binding structure. As we will see in Chapter 5, handling of this rule shows some of the major limitations of the Nominal package in its current state.

Furthermore, we want to obtain a faithful formalization of Lindley and Stark's proof. The Isar structured proof language and the document preparation system allow the creation of formal proof documents. This thesis, in particular Chapter 5, is such a formal proof document. Although the detail required to allow machine checking greatly increases the length of the proofs, the central arguments are still those of [LS05]. Furthermore, the reasoning principles employed in the key lemmas are established separately. This makes it possible to follow the reasoning of Lindley and Stark in those parts that are spelled out in [LS05] rather closely. Since the informal proof has a rather moderate size, the formalization, though being substantially longer, can still be presented in its entirety.

1.1 Outline

In Chapter 2 we briefly outline the central ideas of ⊤⊤-lifting. Chapter 3 contains an introduction to nominal logic. This forms the theoretical base for the implementation of the Isabelle logic HOL-Nominal, which is described together with a general overview of the Isabelle system in Chapter 4. Hence, no familiarity with nominal logic or Isabelle is assumed. Chapter 5 contains the full formalization of the strong normalization proof for λ_{ml} using ⊤⊤-lifting. This chapter is generated from the theory file itself. In Chapter 6 we evaluate how the Nominal package aids in keeping the formalization close to informal reasoning. We conclude with a discussion of related work and possible research directions.

CHAPTER 2

Strong Normalization via ⊤⊤-lifting

In this chapter we will briefly describe the central aspects of Lindley and Stark's paper *Reducibility and* ⊤⊤*-lifting for Computation Types* [LS05], which presents a type directed proof of strong normalization for Moggi's computational metalanguage [Mog91]. The computational metalanguage is an extension of the simply-typed λ-calculus with monadic types.

Normalization proofs using the logical relations proof technique are very modular in nature, since every syntactic construct and its associated typing rule can be handled mostly independent of the others. In fact, [LS05] only mentions abstractions and products when introducing the calculus and refers to Girard et al.'s book [GTL89] for the corresponding cases of their proofs. We want to base our formalization on [Nom], a formalization of [GTL89, Chapter 6], which is distributed with the Isabelle proof assistant. This formalization does not include products. Therefore, we will only deal with a reduced fragment of the metalanguage disregarding product types.

In the next section we introduce the computational metalanguage. In Section 2.2 we state the normalization theorem we want to formalize. We also introduce the inductive characterization of strong normalization employed Chapter 5 and relate it to the definition in terms of infinite sequences usually found in the literature. Section 2.3 briefly explains the structure of Lindley and Stark's normalization proof.

2.1 The λ_{ml}-calculus

We use a presentation of the λ_{ml}-calculus (Figure 2.1), which differs slightly from the version presented by Lindley and Stark. Instead of the Church-style typed variables employed in [LS05], we use a Curry-style/domain-free

[BS00] presentation of the calculus. One reason for this is the way binding is handled in the nominal package and is explained in Section 4.5.1.

Furthermore, we make slight changes to the syntax of the calculus, because we want to use the syntax presented here also as concrete syntax for the calculus in the formalization in Chapter 5. Therefore, we use a different syntactic form for the monadic binding (sequencing) construct. Whereas Lindley and Stark use a monadic let binding written $let\ x \Leftarrow s\ in\ t$, we use a monadic to-binding, written $s\ \mathsf{to}\ x\ \mathsf{in}\ t$. This notation, which is also used in Call-By-Push-Value [Lev99], has two benefits. In our notation, the order of the subterms represents the binding structure which, as will be seen later, interacts nicely with the nominal datatype declaration of the Nominal package. Furthermore, the commuting conversion ($T.assoc$ in Figure 2.1) really looks like an associativity rule rule. Also, we write abstractions as $\Lambda x.t$ since using λ would clash with abstraction in Isabelle/HOL.

The last change is in the reduction relation. For readability, reduction in λ-calculi is usually presented by distinguishing some top-level reductions and the contexts in which these reductions may occur. If one wants to prove things formally, this requires formalizing some relation of top-level reductions, contexts with holes, and the action of plugging a term into a hole all in the presence of alpha equivalence and binding issues. Although this approach might scale better to calculi with lots of reduction rules it would complicate the proof we are aiming for. Instead, we opt for a formalization of a reduction relation which directly includes context rules.

2.2 Strong Normalization

The theorem we ultimately want to formalize is the following:

Theorem 2.2.1. *If $\Gamma \vdash t : \tau$ then t is strongly normalizing.*

Strictly speaking, we have only defined the premise of the theorem so far. So in order to formalize the theorem in a proof assistant we also need to make explicit what it means for a term to be strongly normalizing. The standard definition of strong normalization is that a term t is strongly normalizing if there is no infinite sequence of reductions beginning with t. We call a term with such a sequence (possibly) *diverging* written as formula:

Definition 2.2.2 (Diverging)**.**

$$DIV\ t \equiv \exists S \in \mathbb{N} \to trm.\ S\,0 = t \wedge \forall n.\ S\,n \mapsto S(n+1)$$

The drawback of this definition is that, as shown below, the absence of such infinite sequences alone does not provide an upper bound on the length of the longest reduction sequence.

Syntax:

$$\begin{array}{ll} \text{Types:} & \sigma, \tau \in ty ::= \iota \mid \sigma \to \tau \mid T\,\sigma \\ \text{Terms:} & s, t \in trm ::= x \mid \Lambda x.t \mid s\,t \mid s \text{ to } x \text{ in } t \mid [s] \end{array}$$

Typing:

$$\frac{(x:\tau) \in \Gamma}{\Gamma \vdash x : \tau}$$

$$\frac{\Gamma; x:\sigma \vdash t : \tau}{\Gamma \vdash \Lambda x.t : \sigma \to \tau} \qquad \frac{\Gamma \vdash t : \sigma \to \tau \qquad \Gamma \vdash s : \sigma}{\Gamma \vdash t\,s : \tau}$$

$$\frac{\Gamma \vdash t : \tau}{\Gamma \vdash [t] : T\,\tau} \qquad \frac{\Gamma \vdash s : T\,\sigma \qquad \Gamma; x:\sigma \vdash t : T\,\tau}{\Gamma \vdash s \text{ to } x \text{ in } t : T\,\tau}$$

Reductions:

$$\frac{s \mapsto s'}{s\,t \mapsto s'\,t} \quad \frac{t \mapsto t'}{s\,t \mapsto s\,t'} \quad \frac{}{(\Lambda x.t)s \mapsto t[x ::= s]} \to.\beta$$

$$\frac{t \mapsto t'}{\Lambda x.t \mapsto \Lambda x.t'} \quad \frac{x \notin fv(t)}{\Lambda x.t\,x \mapsto t} \to.\eta$$

$$\frac{s \mapsto s'}{s \text{ to } x \text{ in } t \mapsto s' \text{ to } x \text{ in } t} \quad \frac{t \mapsto t'}{s \text{ to } x \text{ in } t \mapsto s \text{ to } x \text{ in } t'}$$

$$\frac{}{[s] \text{ to } x \text{ in } t \mapsto t[x ::= s]} T.\beta \quad \frac{}{s \text{ to } x \text{ in } [x] \mapsto s} T.\eta$$

$$\frac{x \notin fv(u)}{(s \text{ to } x \text{ in } t) \text{ to } y \text{ in } u \mapsto s \text{ to } x \text{ in } (t \text{ to } y \text{ in } u)} T.assoc$$

$$\frac{t \mapsto t'}{[t] \mapsto [t']}$$

Figure 2.1: *The λ_{ml}-calculus*

Example 2.2.3. The relation $\{(o, n) \mid n \in \mathbb{N}\} \cup \{(n+1, n) \mid n \in \mathbb{N}\}$ is terminating but there is no upper bound on the length of the longest sequence beginning at o.

To obtain such and upper bound, which Lindley and Stark denote with $max(t)$ and frequently do induction on, one has to know that the reduction relation is finitely branching as well as the fact that for any finitely branching relation without upper bound on the longest sequence in the relation there also exists an infinite sequence (Königs Lemma, a special case of [Kőn26]).

One can, however, obtain the same result using an inductive formalization of strong normalization. The two requirements on the notion of strong normalization are that (i) there should be some induction principle allowing us to apply the induction hypothesis whenever a strongly normalizing term makes a reduction step, and (ii) it should exclude infinite sequences. This leads to the following single rule inductive definition of strong normalization which is originally due to Altenkirch [Alt93].

Definition 2.2.4 (Strong normalization)**.**

$$\frac{\forall t'.\ t \mapsto t' \Longrightarrow SN\,t'}{SN\,t}$$

This definition also appears in [Nom]. Requirement (i) is satisfied, because we get the following induction principle, which is automatically derived by Isabelle from Definition 2.2.4.

$$\frac{SN\,t \qquad \forall t.\ (\forall t'.\ t \mapsto t' \Longrightarrow SN\,t' \wedge \forall t'.\ t \mapsto t' \Longrightarrow P\,t') \Longrightarrow P\,t}{P\,t}$$

The definition also fulfills (ii) since we can prove $SN\,t \iff \neg DIV\,t$. We defer formally establishing this fact to Chapter 5.

Thus, we have established the meaning of Theorem 2.2.1. In the next section we will shortly outline the central ideas of the original proof in [LS05]. We assume some familiarity with the logical relations proof technique for the simply-typed λ-calculus. See [GTL89] for details.

2.3 ⊤⊤-lifting for Computation Types

The logical relations proof technique for proving strong normalization of the simply-typed λ-calculus proceeds in three steps. First, one defines a logical (reducibility) relation by induction on the type structure of the calculus. Then, one proves, by induction on the type structure, simultaneously that all reducible terms are strongly normalizing, reducibility is preserved under

reduction, and satisfies the expansion property – if a term t is neutral (does not interact with its context) and all immediate successors are reducible, so is t. Finally, one shows the fundamental theorem of logical relations, i.e. that all typeable terms are reducible and hence strongly normalizing.

For the base type ι, reducibility is just strong normalization, and for function types, the reducibility relation is easily defined using the elimination construct for the $\to$ type constructor.

Definition 2.3.1 (Reducibility, I)**.**

$$\begin{aligned} t \in \mathit{RED}_\iota &\iff \mathit{SN}\, t \\ t \in \mathit{RED}_{\sigma\to\tau} &\iff \forall s \in \mathit{RED}_\sigma.\ t\, s \in \mathit{RED}_\tau \end{aligned}$$

However, the elimination construct for monadic types is *not* inductive on the type structure. This needs to be the case since the type constructor T is meant to encapsulate arbitrary computational effects. So going from $T\,\sigma$ to σ would mean "forgetting" the computational effects. To salvage this, Lindley and Stark introduce stacks: nested sequences of to-bindings.

Definition 2.3.2 (Stacks and dismantling)**.** Stacks are given through the grammar:

$$K \in \mathit{stack} ::= [x]s \gg K \mid \mathit{Id}\ ,$$

and $t \star k$, the associated action of dismantling a term t into k is defined as:

$$\begin{aligned} t \star \mathit{Id} &= t \\ t \star ([x]s \gg K) &= (t\ \mathsf{to}\ x\ \mathsf{in}\ s) \star K \end{aligned}$$

Note that the argument order of the dismantling operation is switched in comparison to [LS05] and that we use $\star$ instead of @, since the latter clashes with list concatenation in Isabelle. It would produce two valid types for $t@[s]$. Using these definitions, one can define reducibility for for stacks and computation types.

Definition 2.3.3 (Reducibility, II)**.**

$$\begin{aligned} t \in \mathit{RED}_{T\,\sigma} &\iff \forall K \in \mathit{SRED}_\sigma.\ \mathit{SN}(t \star K) \\ K \in \mathit{SRED}_\sigma &\iff \forall t \in \mathit{RED}_\sigma.\ \mathit{SN}([s] \star K) \end{aligned}$$

Although the reducibility relation is defined by induction on the type structure of the calculus, the elements of the various relations are untyped terms. For example, any normal term is an element of RED_ι. Similarly, variables are reducible at any type. The reducibility predicate satisfies the following conditions, which were already mentioned above (where *neutral* terms are variables and applications).

Definition 2.3.4 (Properties of reducibility)**.**

$$\begin{array}{lllll}
\text{(CR1)} & & t \in RED_\tau & \Longrightarrow & SN(t) \\
\text{(CR2)} & t \in RED_\tau \quad \wedge \quad t \mapsto t' & & \Longrightarrow & t' \in RED_\tau \\
\text{(CR3)} & neutral(t) \quad \wedge \quad (\forall t'.t \mapsto t' \Longrightarrow t' \in RED_\tau) & & \Longrightarrow & t \in RED_\tau
\end{array}$$

In [LS05] the authors only show the cases for the type constructor T. The subcases for CR1 and CR2 are rather simple but CR3 requires to extend the notion of reduction and strong normalization to stacks.

Definition 2.3.5 (Stack reduction)**.** $K \mapsto K' \equiv \forall t.\ t \star K \mapsto t \star K'$

Strong normalization for stacks then looks exactly as the corresponding property of terms given in Section 2.2. The proof idea for CR3 is then to take some neutral term t such that $t \mapsto t' \Longrightarrow t' \in RED_{T\,\sigma}$ and some $K \in SRED_\sigma$, use the induction hypothesis to show that K is strongly normalizing, and then show $SN(t \star K)$ by induction on the length of the longest reduction sequence beginning at K.

Once the properties of reducibility are established, the next step is to prove the fundamental theorem of logical relations: All (well-typed) terms are reducible. This amounts to proving that all typing rules preserve reducibility. Since abstraction and sequencing bind variables, one has to generalize the claim to Γ-closing substitutions. A substitution is Γ-closing if it substitutes terms which are reducible at the corresponding type for all variables mentioned by the context Γ.

Theorem 2.3.6 (Fundamental Theorem of Logical Relations)**.**
If θ is Γ-closing and $\Gamma \vdash t : \tau$ then $\theta{<}t{>} \in RED_\tau$.

For the case of the to-binding one has show:

Lemma 2.3.7. *if $s \in RED_{T\,\sigma}$ and $\forall p \in RED_\sigma.\ t[x ::= p] \in RED_{T\,\tau}$ then s to x in $t \in RED_{T\,\tau}$.*

Unfolding all the definitions, this is a straightforward consequence of:

Lemma 2.3.8. *Let p and n be terms and K a stack such that $SN(p)$ and $SN(n[x ::= p] \star K)$. Then $SN(([p]$ to x in $n) \star K)$*

This lemma turned out to be difficult to formalize. Lindley and Stark [LS05] prove this by induction on $|K| + max(n \star K) + max(p)$, which cannot be formalized directly due to our inductive characterization of strong normalization. The reasoning then continues by case distinction on the successors of $([p]$ to x in $n) \star K$, which also was surprisingly tedious to formalize. We defer the details to Chapter 5.

Given Theorem 2.3.6, strong normalization of all well-typed terms is a trivial consequence using the identity substitution and CR1.

CHAPTER 3

Nominal Logic

When reasoning about structures with binders, such as λ-calculi, these structures and most of their properties are usually defined by induction. Therefore it seems natural to do most proofs concerning these calculi by structural induction. Unfortunately, it happens quite frequently that in proofs by structural induction the case of a binding construct can be shown easily for names that are "suitably fresh", but not for all names, as the structural induction principle requires.

To simplify proofs and concentrate on the crucial aspects, terms are often identified up to α-equivalence. The cases of variable-binding constructs within inductive proofs are usually only shown for some suitably chosen representative of the α-equivalence class. This is often referred to as the *Barendregt Variable Convention* [Bar85], which states that one can choose all bound variables mutually distinct and distinct from any variables occurring in the current context. There are two underlying assumptions to this approach, which are usually not verified in pen-and-paper proofs.

1. There are always variables that are fresh for the current context.
2. All the predicates and constructions one ever deals with are *equivariant*, meaning that they are invariant under consistent renaming of variables.

Both of these assumptions are nontrivial and need to be considered when formalizing proofs with an interactive theorem prover like Isabelle/HOL. The first assumption above might be invalidated if the context mentions the set of all variables. The second assumption is also nontrivial because not everything that looks like the definition of a function is actually consistent.

More precisely, it need not be well defined on α-equivalence classes, as shown in the example below

Example 3.0.9. The following is a "function" that returns the set of immediate subterms of some lambda term.

$$\begin{aligned} ist\ x &= \emptyset \\ ist(t_1\ t_2) &= \{t_1, t_2\} \\ ist(\lambda x.t) &= \{t\} \end{aligned}$$

This is a perfectly valid function when defined on raw (unequated) λ-terms. However, if we identity terms up to alpha equivalence, we have $\lambda x.x = \lambda y.y$ but $ist(\lambda x.x) = \{x\} \neq \{y\} = ist(\lambda y.y)$ which would introduce an inconsistency.

In the remainder of this chapter, we will give an overview of the development of Nominal Logic (cf. [Pit03, Pit06]) which addresses the issues above. The logic is built around the notions of atoms, permutations, and support, as described in the next section.

3.1 Atoms, Permutations, and Support

To formalize any sort of binding, one first has to formalize the entities which can be bound. These entities are commonly referred to as *atoms*. The terminology dates back to the roots of Nominal Logic in the Fraenkel-Mostowski permutation model of set theory [Jec71]. Atom sorts are abstract sets with infinitely many atoms of that sort. To accommodate calculi with multiple bindable entities, as for example the second-order λ-calculus [GTL89, Rey74] with its term and type variables, there is an infinite number of atom sorts.

Definition 3.1.1 (Atoms)**.** We fix a countably infinite family of *atom sorts* $(\mathbb{A}_i)_{i\in\mathbb{N}}$ such that the sets $\mathbb{A}_i$ are countably infinite and mutually disjoint and define $\mathbb{A} = \bigcup_i \mathbb{A}_i$

Definition 3.1.2 (Permutations)**.** Let *Perm* be the set of all finite, sort-respecting atom permutations.

$$Perm \equiv \left\{ \pi \in \mathbb{A} \to \mathbb{A} \;\middle|\; \begin{array}{c} \pi \text{ is bijective } \wedge \text{ finite } \{a \mid \pi(a) \neq a\} \\ \wedge\ \forall i \forall a \in \mathbb{A}_i.\ \pi(a) \in \mathbb{A}_i \end{array} \right\}$$

We denote the identity on $\mathbb{A}$ by id and the special case of a transposition of a and b is written $(a\,b)$.

Note that the finite sort-respecting atom permutations form a group under composition which is generated from the set of atom transpositions. Next we define what it means for an element of *Perm* to act on a set.

Definition 3.1.3 (Action)**.** An *action of Perm on a set* X is a function $\cdot : \mathit{Perm} \times X \to X$ such that $\mathit{id} \cdot x = x$ and $\pi \cdot (\pi' \cdot x) = (\pi \circ \pi') \cdot x$. For the set $\mathbb{A}$ we define $\pi \cdot a = \pi\, a$, which gives an action of *Perm* on $\mathbb{A}$.

The permutation action on some set X also determines the notion of support for its elements $x \in X$. The notion of support is the central definition of nominal logic. It is meant to capture the intuition of "free" occurrences of names in finite structures, but is general enough to also apply to infinite structures.

Definition 3.1.4 (Support)**.** The support of x is defined as:

$$\mathrm{supp}(x) \equiv \bigcup_i \{a \in \mathbb{A}_i \mid \mathrm{infinite}\{b \in \mathbb{A}_i \mid (a\, b) \cdot x \neq x\}\}$$

Example 3.1.5 (λ-calculus)**.** Consider the λ-calculus with the set of variables chosen to be some atom sort $\mathbb{A}_i$. If we use raw λ-terms and the permutation action specified below we obtain the following notion of support:

$$\begin{array}{rr} \pi \cdot x = \pi\, x & \mathrm{supp}(x) = \{x\} \\ \pi \cdot (s\, t) = (\pi \cdot s)\, (\pi \cdot t) & \mathrm{supp}(s\, t) = \mathrm{supp}(s) \cup \mathrm{supp}(t) \\ \pi \cdot (\lambda x.t) = \lambda(\pi \cdot x).\pi \cdot t & \mathrm{supp}(\lambda x.t) = \mathrm{supp}(t) \cup \{x\} \end{array}$$

Note that the action of perm above is also well defined, if we identify terms up to α-equivalence. In this case, we obtain the support below, which exactly captures the usual notion of free variables.

$$\begin{aligned} \mathrm{supp}(x) &= \{x\} \\ \mathrm{supp}(s\, t) &= \mathrm{supp}(s) \cup \mathrm{supp}(t) \\ \mathrm{supp}(\lambda x.t) &= \mathrm{supp}(t) - \{x\} \end{aligned}$$

Hence, it is natural do define freshness of some atom a for some object x, written $a \sharp x$, simply as a not being in the support of x.

Definition 3.1.6 (Freshness)**.** $a \sharp x \equiv a \notin \mathrm{supp}(x)$

Although the support of x precisely captures the usual intuition of free variables, this definition is cumbersome to work with. Most of the time, one is only interested in showing that some object has finite support. The reason for this is the following proposition which allows us to obtain fresh variables for any finitely supported context.

Proposition 3.1.7. $\mathrm{finite}(\mathrm{supp}(x)) \Longrightarrow \forall i\ \exists a \in \mathbb{A}_i.\ a \sharp x$

Hence, one defines the auxiliary notion of a set supporting some object, which, as we will see, is one way to quickly show that something has finite support. The definition is also fairly natural. It only requires that swapping two atoms which are not in the supporting set should leave the object unchanged.

Definition 3.1.8 (Supports)**.** S supports $x \equiv \forall a\, a' \notin S.\ (a\, a') \cdot x = x$

This auxiliary definition has a number of nice properties.

Proposition 3.1.9. *For any finite sets A and B we have:*
A supports $x \wedge B$ supports $x \Longrightarrow A \cap B$ supports x.

Proposition 3.1.10. $\mathrm{supp}(x)$ supports x

Proposition 3.1.11. $\mathrm{finite}\, S \wedge S$ supports $x \Longrightarrow \mathrm{supp}(x) \subseteq S$

For a proof of the last proposition see [Urb08]. From these properties it immediately followings that for any finitely supported object, $\mathrm{supp}(x)$ is indeed the intersection over all finite supports. The example below however shows that the restriction to finite supporting sets is necessary in each of the propositions.

Example 3.1.12. Consider some $\mathbb{A}_i$ with the action of *Perm* on $\mathcal{P}(\mathbb{A}_i)$ defined by $\pi \cdot X = \{\pi \cdot x \mid x \in X\}$. We split $\mathbb{A}_i$ into $even \uplus odd$ using some bijection from $\mathbb{A}_i$ to $\mathbb{N}$. Both sets support themselves and each other but $\mathrm{supp}(even) = \mathrm{supp}(odd) = \mathbb{A}_i$. Hence, neither set contains $\mathrm{supp}(even)$ and their intersection, the empty set, supports neither *odd* nor *even*.

The example above shows that structures with infinite support can behave rather unintuitively. Furthermore, we certainly want all structures incorporating binders to be finitely supported to guarantee the existence of fresh variables. This leads to the definition of a nominal set below.

Definition 3.1.13 (Nominal set)**.** A *nominal set* is a set X together with an action of *Perm* such that

$$\forall x \in X : \mathrm{finite}(\mathrm{supp}\, x))$$

There are quite a lot of examples of nominal sets and in particular, as we will see later, it is even possible to build a logic in which everything is finitely supported.

Example 3.1.14.

1. Both the raw lambda terms and the α-equated λ-terms from above are nominal sets with respect to the given permutation action.
2. Each $\mathbb{A}_i$ is a nominal set with permutation action $\pi \cdot a = \pi(a)$
3. The set $\mathbb{N}$ of natural numbers becomes a nominal set with $\pi \cdot n = n$ as action of *Perm*.

3.2 Products and Functions

In order to use nominal reasoning in any expressive logic, it is necessary to lift nominal sets along the type structure of the logic. For product types this is straightforward as show in the definition below.

Definition 3.2.1. Given two nominal sets X and Y, the set $X \times Y$ is a nominal set, defining $\pi \cdot (x, y) = (\pi \cdot x, \pi \cdot y)$ to be the action of perm associated with $X \times Y$. One can easily show that $\text{supp}((x, y)) = \text{supp}(x) \cup \text{supp}(y)$

For function types this is however a little more complicated. One can obtain a well behaved action of perm on function types.

Definition 3.2.2. Given two nominal sets X and Y we obtain an action of perm on $X \to Y$ by setting $\pi \cdot f = \lambda x.\ \pi \cdot (f(\pi^{-1} \cdot x))$. Here $X \to Y$ denotes the set of (set theoretic) functions from X to Y.

While maybe looking unusual at first sight, the action on $X \to Y$ is equivalent to $\pi \cdot (f\, x) = (\pi \cdot f)\, (\pi \cdot x)$, which ensures that function application is respected by atom permutations. The definition is forced from the requirement that $X \to Y$ together with the usual application function be the exponential in the Cartesian closed category whose objects are sets equipped with an action of perm and whose morphisms preserve this action. Unfortunately, not every element of $X \to Y$ is finitely supported with respect to the atom permutation above. One such example is shown below.

Example 3.2.3. For any function $f : \mathbb{N} \to \mathbb{A}_i$, $\text{Ran}(f) \subseteq \text{supp}(f)$. For this we show that for any $a \in \text{Ran}(f)$ the set $\{b \mid (a\, b) \cdot f \neq f\}$ is infinite. We have for any $a \in \text{Ran}(f)$, $b \neq a$ and $n \in f^{-1}(a)$ that:

$$((a\, b) \cdot f)\, n = (\lambda n.(a\, b) \cdot f((a\, b) \cdot n))\, n = (a\, b) \cdot a = b \neq a = f\, n\,.$$

Since for any $a \in \text{Ran}$ there exists some $n \in f^{-1}(a)$ and infinitely many b different from a, we have $Ran(f) \subseteq \text{supp}(f)$. On the other hand there certainly are surjective functions in $\mathbb{N} \to \mathbb{A}_i$, which cannot be finitely supported.

One can restrict to finitely supported functions defining.

$$X \to_{fs} Y = \{f : X \to Y \mid \text{finite}(\text{supp}(f))\}$$

This makes $X \to_{fs} Y$ a nominal set as well. Using the standard isomorphism between $\mathcal{P}(X)$ and $X \to \mathbb{B}$ one can also define the nominal set of finitely supported subsets $\mathcal{P}_{fs}(X)$ for any nominal set X. One can easily calculate that the action of perm derived this way is the one used in Example 3.1.12.

3.3 Finite Support vs. Choice

Most of the functions, sets, and predicates one encounters when reasoning about programming languages are finitely supported. There is however one very important exception. There exists no function $C : (\mathbb{A} \rightarrow_{fs} \mathbb{B}) \rightarrow_{fs} \mathbb{B}$ satisfying

$$\exists x.f\, x \Longrightarrow f(C\, f)\ ,$$

the *axiom of choice*. See [Pit06, Example 3.4]. This is be no means unexpected since Nominal Logic builds on the Fraenkel-Mostowski permutation model of set theory [Jec71], which is also incompatible with the axiom of choice. On the other hand, one can show that the function application operation, the currying operation as well as the constantly true function and the equality predicate all define functions which have empty support. Hence one has a *Finite Support Principle* stating that any function defined from finitely supported functions using classical higher-order logic without choice is itself finitely supported [Pit06].

Thus, there are in principle two ways to deal with the issue of finite support. First, can build a logic, without a choice operator, in which everything is finitely supported. This has the nice side effect that Proposition 3.1.7 simplifies to $\forall i\ \exists a \in \mathbb{A}_i.\ a \sharp x$, hence one can always obtain a fresh name in any circumstance. Such a logic was developed by Gabbay [Gab02].

The drawback of this approach is that some very basic parts of the libraries found in proof assistants like Isabelle/HOL are built using the axiom of choice. Hence, all these libraries would have to be adapted. Many applications of the axiom of choice could probably be replaced by explicit constructions. However, Section 5.3 shows that using the axiom of choice can simplify formal reasoning significantly.

The second approach to dealing with finite support is to not restrict to finitely supported functions and relations and work in standard higher-order logic including choice. This has the benefit that all the libraries remain usable, but at the price that one has to show finite support whenever one wants to apply a theorem relying on it. This includes in particular α-structural recursion and induction principles as developed in [Pit06]. We will make use of this approach in the context of Isabelle/HOL-Nominal, an extension to Isabelle/HOL that is built on this second approach to nominal logic [Urb08].

CHAPTER 4

Introduction to Isabelle/HOL-Nominal

The formalization in Chapter 5 uses the Isabelle system. In particular, the the chapter is generated through the Isabelle document preparation system, which is centered around the development of formal proof documents. In this chapter we introduce the basics of the Isabelle system, its logic and proof methods as well as a brief overview of the Isar structured proof language. Any such overview is, by definition, incomplete. The goal of this chapter is mainly to provide the information necessary to understand the development in Chapter 5.

Section 4.1 describes the meta-logic of the Isabelle system. Sections 4.2 and 4.3 describe two approaches to developing proofs in Isabelle. Having described the structure of Isabelle proofs, Section 4.4 briefly explains the most important proof methods used in Chapter 5. The introduction to Isabelle is supplemented by Section 4.5, a description of how the Nominal package implements nominal logic in Isabelle.

4.1 Short Description of the Isabelle System

Isabelle is designed to be a generic framework for developing formal proofs in a variety of different logics, including first-order logic (FOL), higher-order logic (HOL), higher-order logic of computable functions (HOLCF), ZF set theory, etc. All these logics are specified in the *meta-logic* Pure, which is an instance of intuitionistic higher-order logic. The meta-logic uses the simply-typed λ-calculus with $\alpha\beta\eta$-conversion as term language. It has the logical connectives $\Longrightarrow$, $\bigwedge$, and $\equiv$, representing logical entailment (meta-implication), generality (meta-universal-quantifier) and equality (cf. [PN94, Pau89]).

This simple logic is then used to specify the axioms for the logical constants $(\forall, \exists, \longrightarrow, \neg, \wedge, \ldots)$ of the various object logics. Using the notation $[\![P_1\ ;\ \ldots\ ;\ P_n]\!] \Longrightarrow Q$ for $P_1 \Longrightarrow \ldots \Longrightarrow P_n \Longrightarrow Q$, the introduction and elimination rules for implication and universal quantifier for example are:

$$\begin{array}{rl} (P \Longrightarrow Q) \Longrightarrow P \longrightarrow Q & \text{(impI)} \\ [\![P \longrightarrow Q\ ;\ P\ ;\ Q \Longrightarrow R]\!] \Longrightarrow R & \text{(impE)} \end{array}$$

$$\begin{array}{rl} (\bigwedge x.\ P\ x) \Longrightarrow \forall x.\ P\ x & \text{(allI)} \\ [\![\forall x.\ P\ x\ ;\ P\ x \Longrightarrow R]\!] \Longrightarrow R & \text{(allE)} \end{array}$$

The rules above are presented in the same way they are displayed when using the search function of the Isabelle system within the HOL logic. This hides the fact that there is a difference between meta-level terms and object-level terms. Whenever appropriate, object-level formulae are automatically coerced into meta-level propositions. It is useful to think of meta-level terms of the form $[\![P;\ Q]\!] \Longrightarrow R$ as rules of the form

$$\frac{P \qquad Q}{R}$$

However, the premises of these rules may be rules themselves. The meta-logic distinguishes two different types of variables which logically serve the same purpose but are handled differently by the system. Schematic variables, written *?x*, are variables which may be instantiated using (higher-order) unification when applying the rule. Ordinary variables, as used with the $\bigwedge$ binder, need to be instantiated separately. Hence, $\bigwedge x.\ P\ x \Longrightarrow Q\ x$ and $P\ ?x \Longrightarrow Q\ ?x$ are logically the same, but are handled differently by the Isabelle system. As done in the rules for the logical connectives, we will drop the *?* most of the time, especially if all variables occurring free in the rule are schematic.

Within the Isabelle system, the terms of the meta-logic actually serve several purposes. In addition to representing inference rules as described above, derived rules of the form $[\![G_0\ ;\ \ldots;\ G_n]\!] \Longrightarrow C$ are also used to represent proof states, where the G_i are the current goals and C is the claim to be proven. This proof state is then refined using resolution of the proof state with some rule to form a new proof state [Pau89, Chapter 1]. This process uses higher-order unification. While higher-order unification lacks most general unifiers and is undecidable, algorithms performing well in practice were already developed by Huet in 1975 [Hue75].

When one tries to prove some proposition C, the initial proof state is the trivial rule $C \Longrightarrow C$. which is then refined until all subgoals are "discharged" and one obtains C as derived rule. This style of proving theorems is commonly referred to as backwards proof. The remainder of this chapter deals with the practical aspects of proving. A very good starting point for this is the extensive Isabelle tutorial [NPW09].

4.2 Writing Proof Scripts in Isabelle

There are two ways of proving facts in the Isabelle system. Both start by specifying the fact in the theory file using the **lemma** or **theorem** directive. This puts the Isabelle interpreter into the *prove* mode which presents the lemma to be proven as a goal to the user. This means that of the internal proof state $[\![G_0 \; ; \; \ldots ; \; G_n]\!] \Longrightarrow C$ only the G_i are shown. From here, there are two different ways to procede.

In *prove* mode one can apply a proof method via **apply**(*method*). This may modify the current proof state in an arbitrary manner and keeps the Isabelle interpreter in *prove* mode. This allows several **apply** statements to be issued one after the other until no proof goals remain and the proof can be finished via **done**. This style of proof has the advantage that backward proofs are usually relatively easy to come up with, and it is also quite close to the internals of the Isabelle system.

But this kind of proof also has its disadvantages. The first disadvantage is that the proof state remains implicit, and therefore the proof document (the LaTeX version of the theory file) does not contain this information. This makes the theory files impossible to read for humans. The second disadvantage is one of stability. Long sequences of **apply** statements may break between one version of Isabelle and the next, because improvements to the various proof methods may cause these to return another (maybe simpler) proof state in which the subsequent method is no longer applicable, causing the proof to fail at that point.

Furthermore, the backward structure of apply-style proofs is incompatible with the forward reasoning usually employed in informal documents. Hence, if one wants to preserve the structure of some existing informal proof, some forward reasoning infrastructure is required.

4.3 The Isar Structured Proof Language

The currently preferred way of writing Isabelle proof scripts is to use the Isar structured proof language. It mimics to some extent the language usually used in pen-and-paper proofs and is meant to produce formal proof documents that remain human readable. Furthermore, it can be used to close the gap between the backwards proof centered Isabelle system and the forward reasoning style that mathematics is usually presented in.

A simplified grammar for the Isar language developed by Wenzel [Wen02] and presented for the practical user in [Nip, Wen08] is displayed in Figure 4.1. A proof is either atomic using **by** or compound using **proof** ... **qed**. A simple compound proof may begin with some initial proof step and

$$
\begin{aligned}
\mathit{proof} &::= \textbf{proof}\ \mathit{method}^{?}\ \mathit{statement}^{*}\ \textbf{qed} \\
&\mid \textbf{by}\ \mathit{method} \\
\mathit{statement} &::= \textbf{fix}\ \mathit{variables} \\
&\mid \textbf{assume}\ \mathit{proposition}^{+} \\
&\mid (\textbf{from}\ \mathit{fact}^{*})^{?}\ (\textbf{show} \mid \textbf{have})\ \mathit{proposition}^{+}\ \mathit{proof} \\
\mathit{proposition} &::= (\mathit{label}\ \textbf{:})^{?}\ \mathit{formula} \\
\mathit{fact} &::= \mathit{label}
\end{aligned}
$$

Figure 4.1: *Simplified Isar Grammar*

then consists of a series of statements ending with a **show** statement, which should establish the conclusion of the theorem.

The most important statement is of the form **from** *facts* **have** *l: proposition proof* which takes some set of named facts and establishes a new fact named l to be used later in the proof. Since the proof machinery of Isabelle/HOL is built on natural deduction and backwards proofs, the stating of intermediate facts with **from** ... **have** can be used to facilitate forward reasoning in a backwards proof oriented system.

Note that both the statement of a claim as well as the initial proof method may introduce several independent subgoals which need to be solved individually using different **show** statements. When a claim stated using **show** is proven, Isabelle creates a rule. All variables introduced using **fix** become schematic variables, all assumptions made using **assume** become premises, and the final claim becomes the conclusion. This rule is then used to solve one of the current goals.

Hence, **fix** is used to introduce meta-level universal quantifiers and **assume** is used to introduce meta-level implications. This connects the Isar language very tightly to the meta-logic Pure.

4.3.1 Calculatorial reasoning

In addition to the simple grammar above there are a number of extensions that are used heavily in the formal development of Chapter 5. There are some special names for facts that are currently in scope in a proof. The name *this* always refers to the most recently established fact. Since **from** *this* is needed very often, it can be abbreviated by **then**. Furthermore, **then show** is abbreviated by **thus** and **then have** is abbreviated by **hence**. Therefore, one of the most important ways to simplify proof scrips is to use proper chaining in order to avoid explicit naming of facts.

Another special name is *calculation*. It can be used to successively establish a list of facts needed for some key step in the proof. The use is

have A ...
moreover have B ...
moreover have C ...
ultimately show *theorem*

A variant of the chaining scheme above is the combination of **also** and **finally** which does not collect the different facts that are established in a list, but applies transitivity rules to them. This allows equational rewriting in the style $m_0 = m_1 = \ldots = m_n$, where each equality $m_i = m_{i+1}$ is justified by a separate (usually atomic) proof. In this context "..." refers to the right hand side of the last claim.

4.3.2 Generalized elimination

The Isar language also provides a nice generalized elimination construct. The statement

obtain x **where** φ **by** *method*

first establishes

$$\bigwedge \mathit{thesis} \,.\, (\bigwedge x \,.\, \varphi \Longrightarrow \mathit{thesis}) \Longrightarrow \mathit{thesis}$$

as a soundness check to justify the existence of some x with property φ. Afterwards, x is fixed as a new name, and φ is introduced as a new fact. The most common use of this pattern is existential elimination where $\exists x.\varphi$ is a fact from the current context. In Chapter 5, this scheme is used frequently to obtain fresh variables.

4.3.3 Induction and case analysis

The Isar language also has support for inductive proofs and case analysis. In an inductive proof there usually tend to be a large number of universally quantified variables and hypothesis which all originate from the applied induction rule. To prove any of these cases one would need to use **fix** and **assume** to get all these hypotheses before beginning the actual proof. To do all of this in one step there is the **case** keyword, which is used **case**(*CaseId vars*), and binds all the hypotheses to *CaseId* with the newly introduced variables named according to *vars*. For instance, given the following list datatype:

datatype *a list* = *Nil* | *Cons a list* ,

the skeleton of a proof by structural induction on a list *ls* would look as follows:

proof(*induct ls*)
  case *Nil* ... **show** *?case*
next
  case (*Cons x xs*) ... **show** *?case*
qed

Within the cases of an an inductive proof, like the one on the variable *ls* above, this variable is *replaced* by the variables introduced in the various cases. Therefore, any facts referring to *ls* need to be chained into the proof as current facts to obtain variants where *ls* is replaced by *Nil* or *Cons x xs* respectively.

Note the use of *?case* in the code fragment above. When opening some case of an inductive proof or case analysis, *?case* is always bound to the conclusion of the associated subgoal. Furthermore, in any proof, *?theorem* is bound to the conclusion of the theorem.

4.3.4 Raw proof blocks

Another useful tool for more complicated proofs are raw proof blocks. A raw proof block behaves like a normal proof, but it does not establish some explicitly stated fact. Instead, the result of the proof block is the last claim that is established. Known facts pass **{** unchanged, but facts involving locally fixed variables and assumptions are generalized by **}**. For example, the proof block

$$\textbf{\{fix } t' \textbf{ assume } t \mapsto t' \textbf{ have } SN\ t'\ \textit{proof}\textbf{\}}$$

results in $t \mapsto ?t' \Longrightarrow SN\ ?t'$, which can be used to prove $SN\ t$, following the informal style of not mentioning the intermediate universal quantifier.

4.4 The Most Frequently used Proof Methods

This section explains the most frequently used proof methods in the formalization in Chapter 5. A very fundamental proof method is *rule*. Given a rule r, *rule* r tries to use the current facts to eliminate premises of r and then unifies the conclusion of r with the current goal. Without argument, *rule* tries to choose the rule automatically from predefined rule sets (see Section 4.4.2). As simple proofs "**by** *rule*" are fairly common, the Isar language introduces ".." as an abbreviation. In this way, one can extend the example above to

$$\textbf{\{fix } t' \textbf{ assume } t \mapsto t' \textbf{ have } SN\ t'\ \textit{proof}\textbf{\} thus } SN\ t\ ..$$

Further methods include simplification and (automated) classical reasoning as explained next.

4.4.1 Simplification

The starting point for all simplification is the *default simpset* which contains a number of theorems of the form

$$[\![Q_1 \; ; \ldots ; \; Q_n]\!] \Longrightarrow P_l \; = P_r$$

The simplification method *simp* tries to simplify the current subgoal via rewriting, matching some part of the current goal or the premises with P_l and replacing this with P_r. This process uses higher order unification and may instantiate meta variables both in the current goal and the simplification rule being applied. If the list of premises is nonempty, n new subgoals are created where each of them requires some Q_i to be proven from the premises of the original goal. Each of these is again subject to simplification.

After applying a rewrite rule, a solver tool is called which tries to solve the simplified subgoal using some restricted set of rules. Usually, this only includes solving trivial goals like $t = t$ or $True$. This process repeats until no further simplification rules apply. Since the power of the simplifier depends heavily on the simpset, this can be modified when calling the simplifier. Thus, a standard proof by simplification might look as follows:

by(*simp add*: *abs-fresh sapp-fresh*)

In rare occasions, a rewriting might render other rules inapplicable, causing the proof to fail. In these situations it is necessary to also delete some rules from the simpset, which can be done using *del* instead of *add*.

4.4.2 The classical reasoner

An important method for automated proving is the *blast* method which invokes a classical tableau prover. It performs proof search in an untyped manner, reconstructing an Isabelle natural deduction proof once a tableau proof has been found. In contrast to the simplifier which simplifies the subgoal as far as possible and then returns this as the new proof state, *blast* either proves the goal fully automatically or fails outright.

The classical reasoner uses the default classical ruleset to prove the subgoal. This includes, in particular, rules for the logical connectives and all the facts that are declared as introduction rules using the [intro] or the [intro!] attribute. This is particularly useful for inductively defined relations such as $\mapsto$ in Section 5.2. The difference between the two attributes is that the "!" marks the rule as safe, which means that it is applied eagerly without backtracking. This also applies to [elim] and [dest] which add elimination and destruction rules respectively. As with the simpset for simplification, the set of classical rules can also be modified directly when calling the classical reasoner.

4.4.3 The auto method

The *auto* method is the most commonly used proof method in this formalization. It combines simplification with classical reasoning, making it a very powerful tool. All options modifying either the simpset or the classical rule set are also applicable to *auto*. Since auto is targeted at solving all the trivial subgoals, by default, it applies to all the current goals at the same time. This makes it especially suited to prove all the trivial subgoals that remain in an inductive proof, once the interesting cases have been dealt with.

4.4.4 Sledgehammer and metis

Apart from the proof search mechanisms that are provided by the Isabelle system, one can also use external provers like E[1] or SPASS[2]. When issuing the **sledgehammer** command, the current goal and *all* currently known facts are converted to clausal form and transfered to the external tool. These industrial strength provers can sometimes deal with this enormous amount of information and find a proof for the current subgoal.

However, since these tools are not aware of the Isabelle system, they do not return a proof object that can be used to verify the proof found. Instead, the external tool returns the subset of the known facts that are part of the proof which was found. This set of facts can then be given to *metis*[3], which in most cases is capable of finding a proof with this reduced set of facts and outputs a detailed proof Isabelle can parse.

4.5 From HOL to HOL-Nominal

With the basics of the Isabelle system in place, we now explain briefly how α-equated terms are implemented in Isabelle. The Isabelle logic image HOL-Nominal is an implementation of the second approach noted in Section 3.3. It provides infrastructure for reasoning about atoms and support without restricting everything to be finitely supported. We follow the description given by Urban [Urb08]. HOL-Nominal is only a definitional extension of the HOL logic. As long as one trusts in the soundness of HOL, there is no soundness argument required.

[1] http://www4.informatik.tu-muenchen.de/~schulz/WORK/eprover.html
[2] http://www.spass-prover.org/
[3] http://www.gilith.com/software/metis/

4.5.1 Atoms and support in HOL-Nominal

Each theory in HOL-Nominal begins with some **atom-decl** *id-list* which specifies the names of the atom sorts one wants to use in the formalization. Hence, for practical purposes, one has to restrict to finitely many atom sorts. It is for this reason that we cannot use Church-style typed variables. This would require an infinite supply of atom sorts, indexed by the types of the λ_{ml}-calculus, which cannot be represented using the nominal package.

The Isabelle implementation uses lists of swappings to represent elements of *Perm*, with the permutation action on atoms defined by recursion on the list:

$$[] \cdot a = a$$

$$((a_1\, a_2) :: \pi) \cdot a = \begin{cases} a_2 & \text{if } \pi \cdot a = a_1 \\ a_1 & \text{if } \pi \cdot a = a_2 \\ a & \text{otherwise} \end{cases}$$

Due to the typing constraints on lists, this representation of permutations does not generate the whole group *Perm* but only those permutations affecting a single atom sort. For the formalization of λ_{ml}, we only need one atom sort, hence, we ignore this issue and restrict to a single atom sort called *name*. Any element of (*name*×*name*) *list* certainly represents some element of *Perm* in the sense of Section 3.1, hence all the definitions from Chapter 3 apply. However, the list representation of permutations is not unique, so one also needs a notion of equivalence of permutations.

Definition 4.5.1 (Permutation Equality)**.** Two permutations are equal, written $\pi_1 \triangleq \pi_2$, if for all atoms a, we have $\pi_1 \cdot a = \pi_2 \cdot a$.

Definition 4.5.2 (Permutation types and finitely supported types)**.** A type α is a permutation type, written $\mathsf{pt}\ \alpha$, if there is an action of perm defined on it which satisfies the conditions from Definition 3.1.3 and also satisfies $\pi_1 \triangleq \pi_2 \Longrightarrow \pi_1 \cdot x = \pi_2 \cdot x$. A permutation type α whose elements are all finitely supported is called a finitely supported type, written $\mathsf{fs}\ \alpha$.

Note that $\mathsf{fs}\ \alpha$ coresponds to the type α being a nominal set. In HOL-Nominal, the type predicates pt and fs are used as specifications of type classes over HOL types. Following the constructions in Section 3.2, one obtains the following propositions:

Proposition 4.5.3. *If* $\mathsf{pt}\ \alpha$ *and* $\mathsf{pt}\ \beta$ *we have* pt *name,* pt *bool,* pt *unit,* $\mathsf{pt}\ (\alpha \times \beta)$, $\mathsf{pt}\ (\alpha \to \beta)$, $\mathsf{pt}\ (\alpha\ set)$, *and* $\mathsf{pt}\ (\alpha\ list)$ *using the permutation actions from Figure 4.2.*

Proposition 4.5.4. *If* $\mathsf{fs}\ \alpha$ *and* $\mathsf{fs}\ \beta$ *we have* fs *name,* fs *bool,* fs *unit,* $\mathsf{fs}\ (\alpha \times \beta)$, *and* $\mathsf{fs}\ (\alpha\ list)$ *using the permutation actions from Figure 4.2*

$$
\begin{array}{rc}
\mathbb{A}: & \pi \cdot a = \pi\, a \\
bool: & \pi \cdot b = b \\
unit: & \pi \cdot () = () \\
\alpha \times \beta: & \pi \cdot (x_1, x_2) = (\pi \cdot x_1, \pi \cdot x_2) \\
\alpha \Rightarrow \beta: & \pi \cdot f = \lambda x.\pi \cdot (f(\pi^{-1} \cdot x)) \\
\alpha\ set: & \pi \cdot X = \{\pi \cdot x \mid x \in X\} \\
\alpha\ list: & \pi \cdot [\,] = [\,] \\
 & \pi \cdot (x :: xs) = (\pi \cdot x) :: (\pi \cdot xs)
\end{array}
$$

Figure 4.2: *Permutation actions on HOL types*

4.5.2 Identifying terms up to α-equivalence

The cornerstone of the HOL-Nominal logic is an implementation of datatypes with a built-in notion of α-equivalence. This is achieved by first defining pre-terms, using weak higher-order abstract syntax, and then inductively defining a subset of these pre-terms, whose elements correspond to α-equivalence classes. For example, one defines a nominal datatype of λ-terms as follows:

nominal-datatype *trm* = *Var name* | *App trm trm* | *Lam* ≪*name*≫ *trm*

Here, *Lam* ≪*name*≫ *trm* means that a is bound in *Lam a t*. This datatype declaration is first translated into a regular datatype, which uses weak higher-order abstract syntax for the case of abstractions.

datatype *trm′* = *Var′ name*
| *App′ trm′ trm′*
| *Lam′* (*name* ⇒ *trm′ option*)

As equality for functions of type *name* ⇒ *trm′ option* is undecidable, one wants to restrict terms of the type *trm′* to only use some restricted set of functions in the *Lam* case which represent α-equivalence classes and for which equality is decidable. For this reason one introduces *abstraction functions*, which represent α-equivalence classes.

Definition 4.5.5 (Abstraction functions)**.**

$$[a].t \equiv \lambda b.\ \mathsf{if}\ b = a\ \mathsf{then}\ t\ \mathsf{else}\ \mathsf{if}\ b \sharp t\ \mathsf{then}\ Some((a\,b) \cdot t)\ \mathsf{else}\ None$$

Finally, one can (inductively) define a subset of the type trm', restricting the constructor Lam to abstraction functions. The **nominal-datatype** declaration derives this subset and exports it as type trm to the user, translating $Lam\ a\ t$ to $Lam'\ ([a].t)$, Var to Var', and App to App'.

Nominal datatypes derived in this way are equipped with an action of perm that simply pushes the permutation to the subterms (see Example 3.1.5). A **nominal-datatype** declaration may only refer to finitely supported types, making the datatype itself finitely supported. Hence, together with Proposition 4.5.4, this allows the *type checker* to immediately establish finite support almost all cases, in particular for tuples of λ-terms.

4.5.3 Induction and recursion over α-terms

Having identified terms up to α-equivalence, one can build up infrastructure to allow formal proofs close to the informal reasoning using the Barendregt Variable Convention. For these proofs, the Nominal package provides, in addition to the standard structural induction principle, also a strong induction principle. Using this strong induction principle, one needs to prove the cases for binders only for names which are suitably fresh. In this context, suitably fresh means fresh for any finitely supported context that is provided when starting the induction. A direct comparison of the standard and the strong induction rules for the λ_{ml}-calculus, as used in the formalization, can be found in Figure 5.1.

Furthermore, the **nominal-datatype** command provides a primitive recursion combinator which allows freshness conditions on the binders occurring in the defining equations. This greatly simplifies defining functions by primitive structural recursion over nominal datatypes.

Unfortunately, not every function one wants to use in formal developments can be defined using primitive recursion. However, we will see that it is more difficult, but still feasible, to define functions over nominal datatypes that use more general recursion schemes.

The rather technical theoretical development of these recursion and induction principles appears in [Pit06] and the adaptation to the Isabelle framework can be found in [Urb08]. One of the technical aspects of the primitive recursion combinator is the *freshness condition for binders* (FCB). Informally stated the FCB requires that any binder that occurs on the left hand side of a defining equation must be fresh for the right hand side of that equation. Consider the variable x in the to-case of substitution:

$$x \sharp (s,y,v) \Longrightarrow (s\ to\ x\ in\ t)[y::=v] = s[y::=v]\ to\ x\ in\ t[y::=v]$$

Here, one has to choose x to be fresh not only for y and v to avoid capture and allow the substitution to be moved to the subterm. To satisfy the FCB,

x also has to be fresh for the term s, which is outside of the scope of x. As this example shoes, these additional freshness conditions may lead an unusual presentation of affected concepts.

4.5.4 Rule inductions and nominal inversion

While using the variable convention in inductive proofs over terms is always possible, using the variable convention in rule inductions can lead to faulty reasoning. First, not every inductively defined relation is equivariant. But even for equivariant relations, it is not always sound to assume that the binders occurring in a rule are fresh for the context. Consider the Unbind relation for the simply-typed λ-calculus from [UBN07],

$$\frac{}{x \hookrightarrow [], x} \qquad \frac{}{s\,t \hookrightarrow [], s\,t} \qquad \frac{t \hookrightarrow xs, t'}{\lambda x.t \hookrightarrow x :: xs, t'}$$

which is well defined and equivariant. However, assuming that x is fresh for xs may lead to faulty reasoning as x also occurs free in the conclusion of the rule. In [UBN07], Urban et al. introduce a sufficient condition, that allows reasoning using the variable condition in rule inductions. An inductively defined relation is *vc-compatible* if:

1. all the predicates and functions occurring in the side conditions are equivariant, and
2. the side conditions imply that all variables occurring in binding position are mutually distinct and fresh for the conclusion of the rule.

Note that this second condition fails for the Unbind relation above.

Berghofer and Urban [BU08] show that the same condition also allows the strengthening of the inversion principles in such a way that the variables occurring in the various cases can be chosen upon instantiation of the rule. Suppose we know s to x in $t \mapsto r$ and want to use inversion of the $\mapsto$ relation. Using the standard inversion principle the case of a β-reduction gives the following equations: s to x in $t = [u]$ to y in v and $r = v[y ::= u]$. Using the injectivity principle for α-terms, this would require reasoning about the equality of $[x].t$ and $[y].v$. With the strong inversion principle one can chose y to be x, provided x is fresh for s and r, giving the equations s to x in $t = [u]$ to x in v and $r = v[x ::= u]$, from which $t = v$ follows immediately. This simplifies inversion significantly.

Now we have everything in place to start the formalization.

CHAPTER 5
Formalization

This chapter contains the full formalization of the strong normalization theorem for the λ_{ml}-calculus. The first section deals with the formalization of syntax, typing, and substitution. Section 5.2 contains the formalization of the reduction relation. For technical reasons, the reduction relation needs to be stated with additional freshness requirements on the variables occurring in binding position. So we show that these freshness conditions do not change the relation being defined. Section 5.3 contains a formal treatment of the inductive characterization of strong normalization. The rest of the Chapter deals with stacks, reducibility, and the normalization theorem.

5.1 The Calculus

As explained in Section 4.5.1, we begin the theory file by declaring *name* to be our only atom sort. Furthermore, we use a nominal datatype to represent the terms of the λ_{ml}-calculus, using the syntax introduced in Chapter 2

atom-decl *name*

nominal-datatype *trm* =
  Var name
| *App trm trm*
| *Lam* ≪*name*≫*trm* (Λ - . - [*0*,*120*] *120*)
| *To trm* ≪*name*≫*trm* (- *to* - *in* - [*141*,*0*,*140*] *140*)
| *Ret trm* ([-])

The weak and strong induction principles provided by the **nominal-datatype** declaration above are listed in Figure 5.1. The first rule is weak,

trm.induct:

$$\frac{\begin{array}{l}\bigwedge name.\ ?P\ (Var\ name)\\ \bigwedge trm1\ trm2.\ [\![?P\ trm1;\ ?P\ trm2]\!] \Longrightarrow ?P\ (App\ trm1\ trm2)\\ \bigwedge name\ trm.\ ?P\ trm \Longrightarrow ?P\ (\Lambda\ name\ .\ trm)\\ \bigwedge trm1\ name\ trm2.\ [\![?P\ trm1;\ ?P\ trm2]\!] \Longrightarrow ?P\ (trm1\ to\ name\ in\ trm2)\\ \bigwedge trm.\ ?P\ trm \Longrightarrow ?P\ [trm]\end{array}}{?P\ ?trm}$$

trm.strong-induct:

$$\frac{\begin{array}{l}\bigwedge name\ z.\ ?P\ z\ (Var\ name)\\ \bigwedge trm1\ trm2\ z.\ [\![\bigwedge z.\ ?P\ z\ trm1;\ \bigwedge z.\ ?P\ z\ trm2]\!] \Longrightarrow ?P\ z\ (App\ trm1\ trm2)\\ \bigwedge name\ trm\ z.\ [\![name \sharp z;\ \bigwedge z.\ ?P\ z\ trm]\!] \Longrightarrow ?P\ z\ (\Lambda\ name\ .\ trm)\\ \bigwedge trm1\ name\ trm2\ z.\\ \quad [\![name \sharp z;\ name \sharp trm1;\ \bigwedge z.\ ?P\ z\ trm1;\ \bigwedge z.\ ?P\ z\ trm2]\!]\\ \quad \Longrightarrow ?P\ z\ (trm1\ to\ name\ in\ trm2)\\ \bigwedge trm\ z.\ (\bigwedge z.\ ?P\ z\ trm) \Longrightarrow ?P\ z\ [trm]\end{array}}{?P\ ?z\ ?trm}$$

Figure 5.1: *Induction principles for the term datatype*

because one has to prove the binder cases (third and fourth premise) for *all* names, whereas the second rule only requires one to prove the binder cases for names which are suitably fresh.

Note that the premises of these rules neither refer to the context *?z*, nor to the variable *?trm*, but to universally quantified variables of the same type. This means that any current facts referring to *?z* or *?trm* need to be replaced by new facts as explained in Section 4.3.3.

We add the injectivity principle for *trm* to the default simpset and instantiate some of the lemmas from the parent theory Nominal to the current setting.

declare *trm.inject*[*simp*]
lemmas *name-swap-bij* = *pt-swap-bij''*[*OF pt-name-inst at-name-inst*]
lemmas *ex-fresh* = *exists-fresh'*[*OF fin-supp*]

The second **lemmas** statement produces the lemma

$$ex\text{-}fresh{:}\exists\, c{::}name.\ c \sharp (z{::}'a{::}fs\text{-}name)$$

where z can be instantiated to any finitely supported term. Some other frequently used, automatically derived facts can be found in Figure 5.2.

$$
\begin{array}{rl}
\text{alpha:} & ([a].x = [b].y) = \\
& (a = b \wedge x = y \vee a \neq b \wedge x = [(a,\, b)] \cdot y \wedge a \sharp y) \\
\text{alpha':} & ([a].x = [b].y) = \\
& (a = b \wedge x = y \vee a \neq b \wedge [(b,\, a)] \cdot x = y \wedge b \sharp x) \\
\text{abs-fresh:} & b \sharp [a].x = (b = a \vee b \sharp x) \\
\text{fresh-atm:} & a \sharp b = (a \neq b) \\
\text{fresh-prod:} & a \sharp (x,\, y) = (a \sharp x \wedge a \sharp y) \\
\text{exists-fresh':} & \mathit{finite}\ (\mathit{supp}\ x) \Longrightarrow \exists\, c.\ c \sharp x
\end{array}
$$

Figure 5.2: *Some automatically derived facts*

Furthermore, we establish a useful variant of the other alpha renaming lemmas.

```
lemma alpha'' :
  fixes x y :: name and t::trm
  assumes a: x ♯ t
  shows [y].t = [x].([(y,x)] · t)
proof −
  from a have aux: y ♯ [(y, x)] · t
    by (subst fresh-bij[THEN sym, of - - [(x,y)]])
       (auto simp add: perm-swap calc-atm)
  thus ?thesis
    by(auto simp add: alpha perm-swap name-swap-bij fresh-bij)
qed
```

5.1.1 Typing

Even though our types do not involve binders, we still need to formalize them as nominal datatypes to obtain a permutation action. This is required to establish equivariance of the typing relation.

```
nominal-datatype ty =
    TBase
  | TFun ty ty (infix → 200)
  | T ty
```

Since, as explained in Section 4.5.1, we cannot use typed variables; we have to formalize typing contexts. Isabelle does not provide a type for finite functions, hence typing contexts are formalized as lists. A context is *valid* if no name occurs twice.

```
inductive
  valid :: (name×ty) list ⇒ bool
where
  v1[intro]: valid []
| v2[intro]: ⟦valid Γ;x♯Γ⟧⟹ valid ((x,σ)#Γ)
```

equivariance *valid*

lemma *fresh-ty*:
  fixes x :: *name* **and** τ::*ty*
  shows $x \sharp \tau$
by (*induct* τ *rule*: *ty.induct*) (*auto*)

lemma *fresh-context*:
  fixes Γ :: (*name*×*ty*)*list*
  assumes *a*: $x \sharp \Gamma$
  shows $\neg(\exists\ \tau\ .\ (x,\tau)\in set\ \Gamma)$
using *a*
by (*induct* Γ) (*auto simp add*: *fresh-prod fresh-list-cons fresh-atm*)

inductive
  typing :: (*name*×*ty*) *list*⇒*trm*⇒*ty*⇒*bool* (- ⊢ - : - [*60*,*60*,*60*] *60*)
where
  t1[*intro*]: ⟦*valid* Γ; $(x,\tau)\in set\ \Gamma$⟧ ⟹ $\Gamma \vdash Var\ x : \tau$
| *t2*[*intro*]: ⟦$\Gamma \vdash s : \tau\to\sigma$; $\Gamma \vdash t : \tau$⟧ ⟹ $\Gamma \vdash App\ s\ t : \sigma$
| *t3*[*intro*]: ⟦$x \sharp \Gamma$; $((x,\tau)\#\Gamma) \vdash t : \sigma$⟧ ⟹ $\Gamma \vdash \Lambda\ x\ .\ t : \tau\to\sigma$
| *t4*[*intro*]: ⟦ $\Gamma \vdash s : \sigma$ ⟧ ⟹ $\Gamma \vdash [s] : T\ \sigma$
| *t5*[*intro*]: ⟦$x \sharp (\Gamma,s)$; $\Gamma \vdash s : T\ \sigma$; $((x,\sigma)\#\Gamma) \vdash t : T\ \tau$ ⟧
    ⟹ $\Gamma \vdash s\ to\ x\ in\ t : T\ \tau$
equivariance *typing*
nominal-inductive *typing*
  by(*simp-all add*: *abs-fresh fresh-ty*)

Except for the explicit requirement that contexts be valid in the variable case and the freshness requirement on s in *t5*, this typing relation is a direct translation of the original typing relation in [LS05] to the setting using contexts. The **nominal-inductive** command derives the strong induction and case rules described in Section 4.5.4. The strong induction principle will be used in the proof of the fundamental theorem of logical relations.

5.1.2 Substitution

Here we introduce substitution on the terms defined above. Since we need parallel substitution for the fundamental theorem, we define it first and introduce ordinary substitution as an abbreviation. Unfortunately, the function type *name* ⇒ *trm* is not finitely supported. Thus, as was the case with contexts, the easiest approach is to formalize substitutions as lists.

fun
  lookup :: (*name*×*trm*) *list* ⇒ *name* ⇒ *trm*
where
  lookup [] x = *Var* x
| *lookup* $((y,e)\#\theta)$ x = (*if* x=y *then* e *else lookup* θ x)

lemma *lookup-eqvt*[*eqvt*]:
  fixes *pi*::*name prm*
  and *θ*::(*name*×*trm*) *list*
  and *x*::*name*
  shows *pi* ∙ (*lookup θ x*) = *lookup* (*pi* ∙ *θ*) (*pi* ∙ *x*)
by (*induct θ*) (*auto simp add*: *eqvts*)

nominal-primrec
  psubst :: (*name*×*trm*) *list* ⇒ *trm* ⇒ *trm* (-<-> [*95*,*95*] *205*)
where
    θ<*Var x*> = *lookup θ x*
  | *θ*<*App s t*> = *App* (*θ*<*s*>) (*θ*<*t*>)
  | *x* ♯ *θ* ⟹ *θ*<Λ *x* .*s*> = Λ *x* . (*θ*<*s*>)
  | *θ*<[*t*]> = [*θ*<*t*>]
  | ⟦ *x* ♯ *θ* ; *x* ♯ *t* ⟧ ⟹ *θ*<*t to x in s*> = (*θ*<*t*>) *to x in* (*θ*<*s*>)
by(*finite-guess*+ , (*simp add*: *abs-fresh*)+ , *fresh-guess*+)

lemma *psubst-eqvt*[*eqvt*]:
  fixes *pi*::*name prm*
  shows *pi* ∙ (*θ*<*t*>) = (*pi* ∙ *θ*)<(*pi* ∙ *t*)>
by(*nominal-induct t avoiding*: *θ rule*:*trm.strong-induct*)
  (*auto simp add*: *eqvts fresh-bij*)

The lemma *psubst-eqvt* states that substitution, like all our constructions, is equivariant. Having defined parallel substitution, we define substitution for a single variable as an abbreviation of the parallel case. Furthermore, we show the usual defining equations as a lemma and add it to the default simpset. The effect of this is that single variable substitution behaves just as if defined directly, and also interacts smoothly with the parallel case.

abbreviation
  subst :: *trm* ⇒ *name* ⇒ *trm* ⇒ *trm* (-[-::=-] [*200*,*100*,*100*] *200*)
where
  t[*x*::=*t'*] ≡ ([(*x*,*t'*)])<*t*>

lemma *subst*[*simp*]:
shows (*Var x*)[*y*::=*v*] = (*if x* = *y then v else Var x*)
  and (*App s t*)[*y*::=*v*] = *App* (*s*[*y*::=*v*]) (*t*[*y*::=*v*])
  and *x* ♯ (*y*,*v*) ⟹ (Λ *x* . *t*)[*y*::=*v*] = Λ *x* .*t*[*y*::=*v*]
  and *x* ♯ (*s*,*y*,*v*) ⟹ (*s to x in t*)[*y*::=*v*] = *s*[*y*::=*v*] *to x in t*[*y*::=*v*]
  and ([*s*])[*y*::=*v*] = [*s*[*y*::=*v*]]
by(*simp-all add*: *fresh-list-cons fresh-list-nil*)

5.1.3 Facts about substitution

To be able to work comfortably with substitution, we need a couple of lemmas about substitution that concern the interaction of substitution and

freshness.

lemma *subst-rename*:
  assumes *a*: $y \sharp t$
  shows $([(y,x)]\cdot t)[y::=v] = t[x::=v]$
using *a*
by(*nominal-induct t avoiding*: *x y v rule*: *trm.strong-induct*)
   (*auto simp add*: *calc-atm fresh-atm abs-fresh fresh-prod fresh-aux*)
lemmas *subst-rename'* = *subst-rename*[*THEN sym*]

lemma *forget*: $x \sharp t \Longrightarrow t[x::=v] = t$
by(*nominal-induct t avoiding*: *x v rule*: *trm.strong-induct*)
   (*auto simp add*: *abs-fresh fresh-atm*)

lemma *fresh-fact*:
  fixes *x*::*name*
  assumes *x*: $x \sharp v \quad x \sharp t$
  shows $x \sharp t[y::=v]$
using *x*
by(*nominal-induct t avoiding*: *x y v rule*: *trm.strong-induct*)
   (*auto simp add*: *abs-fresh fresh-atm*)

lemma *fresh-fact'*:
  fixes *x*::*name*
  assumes *x*: $x \sharp v$
  shows $x \sharp t[x::=v]$
using *x*
by(*nominal-induct t avoiding*: *x v rule*: *trm.strong-induct*)
   (*auto simp add*: *abs-fresh fresh-atm*)

lemma *subst-lemma*:
  assumes *a*: $x \neq y$
  and *b*: $x \sharp u$
  shows $s[x::=v][y::=u] = s[y::=u][x::=v[y::=u]]$
using *a b*
by(*nominal-induct s avoiding*: *x y u v rule*: *trm.strong-induct*)
   (*auto simp add*: *fresh-fact forget*)

lemma *id-subs*:
  shows $t[x::=Var\ x] = t$
by(*nominal-induct t avoiding*: *x rule*:*trm.strong-induct*) *auto*

In addition to the facts on simple substitution we also need some facts on parallel substitution. In particular we want to be able to extend a parallel substitution with an ordinary one.

lemma *lookup-fresh*:
  fixes *z*::*name*
  assumes $z\sharp\theta \quad z\sharp x$
  shows $z\sharp\ lookup\ \theta\ x$

```
using assms
by(induct rule: lookup.induct)
  (auto simp add: fresh-list-cons)

lemma lookup-fresh':
  assumes a: z♯θ
  shows lookup θ z = Var z
using a
by (induct rule: lookup.induct)
   (auto simp add: fresh-list-cons fresh-prod fresh-atm)

lemma psubst-fresh-fact:
  fixes x :: name
  assumes a: x ♯ t and b: x ♯ θ
  shows x ♯ θ<t>
using a b
by(nominal-induct t avoiding: θ x rule:trm.strong-induct)
    (auto simp add: lookup-fresh abs-fresh)

lemma psubst-subst:
  assumes a: x ♯ θ
  shows θ<t>[x::=s] = ((x,s)#θ)<t>
  using a
by(nominal-induct t avoiding: θ x s rule: trm.strong-induct)
    (auto simp add: fresh-list-cons fresh-atm forget
      lookup-fresh lookup-fresh' fresh-prod psubst-fresh-fact)
```

5.2 The Reduction Relation

With substitution in place, we can now define the reduction relation on λ_{ml}-terms. To derive strong induction and case rules, all the rules must be vc-compatible. In the case of the reduction relation stated in Chapter 2, this requires some additional freshness conditions. Note that in this particular case the additional freshness conditions only serve the technical purpose of automatically deriving strong reasoning principles. To show that the version with freshness conditions defines the same relation as the one in Figure 2.1, we also state this version and prove equality of the two relations.

```
inductive std-reduction :: trm ⇒ trm ⇒ bool (- ⇝ - [80,80] 80)
where
  std-r1[intro!]:s ⇝ s' ⟹ App s t ⇝ App s' t
| std-r2[intro!]:t ⇝ t' ⟹ App s t ⇝ App s t'
| std-r3[intro!]:App (Λ x . t) s ⇝ t[x::=s]

| std-r4[intro!]:t ⇝ t' ⟹ Λ x . t ⇝ Λ x . t'
| std-r5[intro!]:x ♯ t ⟹ Λ x . App t (Var x) ⇝ t
```

```
| std-r6[intro!]:⟦ s ⇝ s' ⟧ ⟹ s to x in t ⇝ s' to x in t
| std-r7[intro!]:⟦ t ⇝ t' ⟧ ⟹ s to x in t ⇝ s  to x in t'
| std-r8[intro!]:[s] to x in t ⇝ t[x::=s]
| std-r9[intro!]:x ♯ s ⟹ s to x in [Var x] ⇝ s
| std-r10[intro!]: ⟦ x ♯ y; x ♯ u⟧
                    ⟹ (s to x in t) to y in u ⇝ s to x in (t to y in u)
| std-r11[intro!]: s ⇝ s' ⟹ [s] ⇝ [s']

inductive
  reduction :: trm ⇒ trm ⇒ bool (- ↦ - [80,80] 80)
where
  r1[intro!]:s ↦ s' ⟹ App s t ↦ App s' t
| r2[intro!]:t ↦ t' ⟹ App s t ↦ App s t'
| r3[intro!]:x ♯ s ⟹ App (Λ x . t) s ↦ t[x::=s]

| r4[intro!]:t ↦ t' ⟹ Λ x . t ↦ Λ x . t'
| r5[intro!]:x ♯ t ⟹ Λ x . App t (Var x) ↦ t

| r6[intro!]:⟦ x ♯ (s,s') ; s ↦ s' ⟧ ⟹ s to x in t ↦ s' to x in t
| r7[intro!]:⟦ x ♯ s ; t ↦ t' ⟧ ⟹ s to x in t ↦ s  to x in t'
| r8[intro!]:x ♯ s ⟹ [s] to x in t ↦ t[x::=s]
| r9[intro!]:x ♯ s ⟹ s to x in [Var x] ↦ s
| r10[intro!]: ⟦ x ♯ (y,s,u) ; y ♯ (s,t) ⟧
                ⟹ (s to x in t) to y in u ↦ s to x in (t to y in u)
| r11[intro!]: s ↦ s' ⟹ [s] ↦ [s']
equivariance reduction
nominal-inductive reduction
  by(auto simp add: abs-fresh fresh-fact' fresh-prod fresh-atm)
```

In order to show adequacy, the extra freshness conditions in the rules *r3*, *r6*, *r7*, *r8*, *r9*, and *r10* need to be discharged. Furthermore, we make the classical reasoners, used in *auto* and *blast*, aware of the new rules by adding them as introduction rules to the classical rule set. To avoid conflicts, the old rules are then deleted.

```
lemma r3'[intro!]: App (Λ x . t) s ↦ t[x::=s]
proof −
  obtain x'::name where s: x' ♯ s and t: x' ♯ t
    using ex-fresh[of (s,t)] by (auto simp add: fresh-prod)
  from t have App (Λ x . t) s = App (Λ x' . ([(x,x')] ∙ t)) s
    by (simp add: alpha'')
  also from s have ... ↦ ([(x, x')] ∙ t)[x'::=s] ..
  also have ... = t[x::=s] using t
    by (auto simp add: subst-rename') (metis perm-swap)
  finally show ?thesis .
qed
declare r3[rule del]
```

```
lemma r6'[intro]:
  fixes s :: trm
  assumes r: s ↦ s'
  shows s to x in t ↦ s' to x in t
using assms
proof −
  obtain x'::name where s: x' ♯ (s, s') and t: x' ♯ t
    using ex-fresh[of (s,s',t)] by (auto simp add: fresh-prod)
  from t have s to x in t = s to x' in ([(x,x')] · t)
    by (simp add: alpha'')
  also from s r have ... ↦ s' to x' in ([(x, x')] · t) ..
  also from t have ... = s' to x in t
    by (simp add: alpha'')
  finally show ?thesis .
qed
declare r6[rule del]

lemma r7'[intro]:
  fixes t :: trm
  assumes t ↦ t'
  shows s to x in t ↦ s to x in t'
using assms
proof −
  obtain x'::name where f: x' ♯ t   x' ♯ t'   x' ♯ s   x' ♯ x
    using ex-fresh[of (t,t',s,x)] by(auto simp add:fresh-prod)
  hence a: s to x in t = s to x' in ([(x,x')] · t)
    by (auto simp add: alpha'')
  from assms have ([(x,x')] · t) ↦ [(x,x')] · t'
    by (simp add: eqvts)
  hence r: s to x' in ([(x,x')] · t) ↦ s to x' in ([(x,x')] · t')
    using f by auto
  from f have s to x in t' = s to x' in ([(x,x')] · t')
    by (auto simp add: alpha'')
  with a r show ?thesis by (simp del: trm.inject)
qed
declare r7[rule del]

lemma r8'[intro!]: [s] to x in t ↦ t[x::=s]
proof −
  obtain x'::name where s: x' ♯ s and t: x' ♯ t
    using ex-fresh[of (s,t)] by (auto simp add: fresh-prod)
  from t have [s] to x in t = [s] to x' in ([(x,x')] · t)
    by (simp add: alpha'')
  also from s have ... ↦ ([(x, x')] · t)[x'::=s] ..
  also have ... = t[x::=s] using t
    by (auto simp add: subst-rename') (metis perm-swap)
  finally show ?thesis .
qed
declare r8[rule del]
```

```
lemma r9'[intro!]: s to x in [Var x] ↦ s
proof −
  obtain x'::name where f: x' ♯ s   x' ♯ x
    using ex-fresh[of (s,x)] by(auto simp add:fresh-prod)
  hence s to x' in [Var x'] ↦ s by auto
  moreover have s to x' in ([Var x']) = s to x in ([Var x])
    by (auto simp add: alpha fresh-atm swap-simps)
  ultimately show ?thesis by simp
qed
declare r9[rule del]
```

While discharging these freshness conditions is easy for rules involving only one binder it unfortunately becomes quite tedious for the assoc rule *r10*. This is due to the complex binding structure of this rule which includes *four* binding occurrences of two different names. Furthermore, the binding structure changes from the left to the right: On the left hand side, x is only bound in t, whereas on the right hand side the scope of x extends over the whole term *t to y in u*.

```
lemma r10'[intro!]:
  assumes xf: x ♯ y   x ♯ u
  shows (s to x in t) to y in u ↦ s to x in (t to y in u)
proof −
  obtain y'::name — suitably fresh
    where y: y' ♯ s   y' ♯ x   y' ♯ t   y' ♯ u
    using ex-fresh[of (s,x,t,u,[(x, x')] • t)]
    by (auto simp add: fresh-prod)
  obtain x'::name
    where x: x' ♯ s   x' ♯ y'   x' ♯ y  x' ♯ t   x' ♯ u
          x' ♯ ([(y,y')] • u)
    using ex-fresh[of (s,y',y,t,u,([(y,y')] • u))]
    by (auto simp add: fresh-prod)
  from x y have yaux: y' ♯ [(x, x')] • t
    by(simp add: fresh-left perm-fresh-fresh fresh-atm)
   have (s to x in t) to y in u = (s to x in t) to y' in ([(y,y')] • u)
    using ‹y' ♯ u› by (simp add: alpha'')
  also have ... = (s to x' in ([(x,x')] • t)) to y' in ([(y,y')] • u)
    using ‹x' ♯ t› by (simp add: alpha'')
  also have ... ↦ s to x' in (([(x,x')] • t) to y' in ([(y,y')] • u))
    using x y yaux by (auto simp add: fresh-prod)
  also have  ... = s to x' in (([(x,x')] • t) to y in u)
    using ‹y' ♯ u› by (simp add: abs-fun-eq1 alpha'')
  also have ... = s to x in (t to y in u)
  proof (subst trm.inject)
    from xf x have swap: [(x,x')] • y = y   [(x,x')] • u = u
      by(auto simp add: fresh-atm perm-fresh-fresh )
    with x show s = s ∧ [x'].([(x, x')] • t) to y in u = [x].t to y in u
      by (auto simp add: alpha''[of x' - x] abs-fresh abs-fun-eq1 swap)
```

```
  qed
  finally show ?thesis .
qed
declare r10[rule del]
```

Since now all the introduction rules of the vc-compatible reduction relation exactly match their standard counterparts, both directions of the adequacy proof are trivial inductions.

```
theorem adequacy: s ↦ t = s ⇝ t
by (auto elim:reduction.induct std-reduction.induct)
```

Next we show that the reduction relation preserves freshness and is in turn preserved under substitution.

```
lemma reduction-fresh:
  fixes x::name
  assumes r: t ↦ t'
  shows x ♯ t ⟹ x ♯ t'
using r
by(nominal-induct t t' avoiding: x rule: reduction.strong-induct)
    (auto simp add: abs-fresh fresh-fact fresh-atm)

lemma reduction-subst:
  assumes a:  t ↦ t'
  shows t[x::=v] ↦ t'[x::=v]
using a
by(nominal-induct t t' avoiding: x v rule: reduction.strong-induct)
  (auto simp add: fresh-atm fresh-fact subst-lemma fresh-prod abs-fresh)
```

5.3 Strong Normalization

Next we need to formalize what it means for a term to be strongly normalizing. As already motivated in Section 2.2, we use an inductive variant of strong normalization, It allows for inductive proofs on terms being strongly normalizing, without establishing that the reduction relation is finitely branching.

```
inductive
  SN :: trm ⇒ bool
where
  SN-intro: (⋀ t' . t ↦ t' ⟹ SN t') ⟹ SN t
```

It remains to be shown that this definition actually excludes infinite sequences of reductions. We define a term t to be diverging, written $DIV\ t$, if there is some infinite sequence S of reductions beginning at t.

```
constdefs
  DIV :: trm ⇒ bool
  DIV t ≡ ∃ (S::nat ⇒ trm) .  t = S 0 ∧ (∀ n . S n ↦ S (n + 1))

theorem SN t ⟹ ¬ DIV t
proof (induct rule:SN.induct)
  case (SN-intro t)
  have ih: ⋀t'. t ↦ t' ⟹ ¬ DIV t' by fact
  moreover have DIV t ⟹ ∃ t' . t ↦ t' ∧ DIV t'
  proof −
    assume DIV t from this obtain S::nat⇒trm
      where S: t = S 0 ∧ (∀ n . S n ↦ S (n + 1))
      unfolding DIV-def ..
    let ?t = S 1 let ?S = λ n . S (n + 1)
    from S have  t ↦ ?t by auto
    moreover {
      from S have ?t = ?S 0 ∧ (∀ n . ?S n ↦ ?S (n + 1)) by auto
      hence DIV ?t unfolding DIV-def by auto}
    ultimately show ?thesis by blast
  qed
  ultimately show ¬ DIV t using ih by blast
qed
```

Incidentally, the converse direction holds as well. Our proof requires the axiom of choice and hence would not have been possible in a logic where everything has finite support. The claim could probably also be proven without the use of choice, but this would, for example, require the construction of an order on the set of terms to be able to select the least successor that is not *SN*.

```
theorem ¬ SN t ⟹ DIV t
proof −
  fix t assume t: ¬ SN t
  let ?NSN = { t . ¬ SN t }
  have ∀ t ∈ ?NSN .  ∃ t' . t ↦ t' ∧ ¬ SN t'
    by (auto intro: SN-intro)
  hence  ∃ f . ∀ t ∈ ?NSN . t ↦ f t ∧ ¬ SN (f t)
    by (rule bchoice)
  from this obtain f where f: ∀ t ∈ ?NSN . t ↦ f t ∧ ¬ SN (f t) ..
  let ?S = λ n . (f^n) t
  { fix n from t f have  ?S n ↦ ?S (n + 1) ∧ ¬ SN (?S (n + 1))
      by (induct n) auto }
  hence t = ?S 0 ∧ (∀ n . ?S n ↦ ?S (n + 1)) by auto
  thus DIV t unfolding DIV-def  by(rule exI[where x=?S])
qed
```

For the formalization, we merely need that strong normalization is preserved under reduction and some lemmas on normal terms.

lemma *SN-preserved*[*intro*]:
  assumes *a*: *SN t* $t \mapsto t'$
  shows *SN t'*
using *a* **by** (*cases*) (*auto*)

constdefs
  NORMAL :: *trm* ⇒ *bool*
  NORMAL t ≡ ¬(∃ *t'*. *t* ↦ *t'*)

lemma *normal-var*: *NORMAL* (*Var x*)
unfolding *NORMAL-def* **by** (*auto elim*: *reduction.cases*)

lemma *normal-implies-sn* : *NORMAL s* ⟹ *SN s*
unfolding *NORMAL-def* **by**(*auto intro*: *SN-intro*)

5.4 Stacks

As explained in Chapter 2.3, the monadic type structure of the λ_{ml}-calculus does not lend itself to an easy definition of a logical relation along the type structure of the calculus. Therefore, we need to introduce stacks as an auxiliary notion to handle the monadic type constructor T. Stacks can be thought of as lists of term abstractions $[x].t$. Our notation for stacks is chosen with this resemblance in mind.

nominal-datatype *stack* = *Id* | *St* ≪*name*≫*trm stack* ([-]-≫-)

lemma *stack-exhaust* :
  fixes *c* :: *'a*::*fs-name*
  shows *k* = *Id* ∨ (∃ *y n l* . *y* ♯ *l* ∧ *y* ♯ *c* ∧ *k* = [*y*]*n*≫*l*)
by(*nominal-induct k avoiding*: *c rule*: *stack.strong-induct*) (*auto*)

nominal-primrec
  length :: *stack* ⇒ *nat* (|-|)
where
  |*Id*| = *0*
| *y* ♯ *L* ⟹ *length* ([*y*]*n*≫*L*) = *1* + |*L*|
by(*finite-guess*+,*auto simp add*: *fresh-nat*,*fresh-guess*)

5.4.1 Stack dismantling

Together with the stack datatype, we introduce the notion of dismantling a term onto a stack. Unfortunately, the dismantling operation has no easy primitive recursive formulation. The Nominal package, however, only provides a recursion combinator for primitive recursion. This means that for dismantling one has to prove pattern completeness, right uniqueness, and termination explicitly.

This takes a little more effort than using **nominal-primrec** and the mostly canonical proof to discharge the finite support and freshness requirements. However, once this has been done, the defining equations can be used as simplification rules just as if defined with **nominal-primrec**.

```
function
  dismantle :: trm ⇒ stack ⇒ trm (- ⋆ - [160,160] 160)
where
  t ⋆ Id = t |
  x ♯ (K,t) ⟹ t ⋆ ([x]s≫K) = (t to x in s) ⋆ K
proof −     — pattern completeness
  fix P :: bool and arg::trm × stack
  assume id: ⋀t. arg = (t, stack.Id) ⟹ P
    and st: ⋀x K t s. ⟦x ♯ (K, t); arg = (t, [x]s≫K)⟧ ⟹ P
  { assume snd arg = Id
    hence P by (metis id[where t=fst arg] surjective-pairing) }
  moreover
  { fix y n L assume snd arg = [y]n≫L    y ♯ (L, fst arg)
    hence P by (metis st[where t=fst arg] surjective-pairing) }
  ultimately show P using stack-exhaust[of    snd arg    fst arg] by auto
next
    — right uniqueness
    — only the case of the second equation matching both args needs to be shown.
  fix t t′ :: trm and x x′ :: name and s s′ :: trm and K K′ :: stack
  let ?g = dismantle-sumC — graph of dismantle
  assume x ♯ (K, t)    x′ ♯ (K′, t′)
    and  (t, [x]s≫K) = (t′, [x′]s′≫K′)
  thus ?g (t to x in s, K) = ?g (t′ to x′ in s′, K′)
    by (auto intro!: arg-cong[where f=?g] simp add: stack.inject)
qed (simp-all add: stack.inject) — all other cases are trivial
```

Note the use of *metis* for the relatively simple goals above. The reason for this is that *simp* and *auto* diverge if one adds surjective pairing ($t = (fst\ t, snd\ t)$ to the simpset whereas *metis* finds a proof within a fraction of a second. Afterwards, we just have to establish termination which is simple as the length of K decreases with every recursive call.

```
termination dismantle
by(relation measure (λ(t,K). |K| ))(auto)
```

Like all our constructions, dismantling is equivariant. Also, freshness can be pushed over dismantling, and the freshness requirement in the second defining equation is not needed

```
lemma dismantle-eqvt[eqvt]:
  fixes pi :: (name × name) list
  shows pi ∙ (t ⋆ K) =  (pi ∙ t) ⋆ (pi ∙ K)
by(nominal-induct K avoiding: pi t rule:stack.strong-induct)
  (auto simp add: eqvts fresh-bij)
```

lemma *dismantle-fresh*[*iff*]:
  fixes $x :: name$
  shows $(x \sharp (t \star k)) = (x \sharp t \wedge x \sharp k)$
by(*nominal-induct k avoiding*: *t x rule*: *stack.strong-induct*)
  (*simp-all*)

lemma *dismantle-simp*[*simp*]: $s \star [y]n \gg L = (s\ to\ y\ in\ n) \star L$
proof −
  obtain $x::name$ **where** f: $x \sharp s \quad x \sharp L \quad x \sharp n$
    using *ex-fresh*[*of* (*s*,*L*,*n*)] **by**(*auto simp add*:*fresh-prod*)
  hence *t*: $s\ to\ y\ in\ n = s\ to\ x\ in\ ([(y,x)] \cdot n)$
    by(*auto simp add*: *alpha''*)
  from f **have** $[y]n \gg L = [x]([(y,x)] \cdot n) \gg L$
    by (*auto simp add*: *stack.inject alpha''*)
  hence $s \star [y]n \gg L = s \star [x]([(y,x)] \cdot n) \gg L$ **by** *simp*
  also have $\ldots = (s\ to\ y\ in\ n) \star L$ **using** $f\ t$ **by**(*simp del*:*trm.inject*)
  finally show *?thesis* **.**
qed

5.4.2 Reduction and substitution for stacks

We also need a notion of reduction on stacks. This reduction relation allows us to define strong normalization not only for terms but also for stacks and is needed to prove the properties of the logical relation later on.

constdefs
  stack-reduction :: *stack* ⇒ *stack* ⇒ *bool* (- ↦ -)
  $k \mapsto k' \equiv \forall\ (t::trm)\ .\ (t \star k) \ \mapsto (t \star k')$

While one could certainly obtain the same reduction relation by explicitly stating reduction rules, the given definition provides a rather canonical way for lifting properties of the term reduction relation to the reduction relation on stacks. One example, shown below, is that freshness is preserved under stack reduction.

lemma *stack-reduction-fresh*:
  fixes $k :: stack$ **and** $x :: name$
  assumes $r : k \mapsto k'$ **and** $f : x \sharp k$
  shows $x \sharp k'$
proof −
  from *ex-fresh*[*of x*] **obtain** $z::name$ **where** f': $z \sharp x$ **..**
  from r **have** $Var\ z \star k \mapsto Var\ z \star k'$ **unfolding** *stack-reduction-def* **..**
  moreover from $f\ f'$ **have** $x \sharp Var\ z \star k$ **by**(*auto simp add*: *fresh-atm*)
  ultimately have $x \sharp Var\ z \star k'$ **by**(*rule reduction-fresh*)
  thus $x \sharp k'$ **by** *simp*
qed

lemma *dismantle-red*[*intro*]:
fixes $m :: trm$
assumes r: $m \mapsto m'$
shows $m \star k \mapsto m' \star k$
using r
by (*nominal-induct k avoiding*: m m' *rule*:*stack.strong-induct*) *auto*

Next we define a substitution operation for stacks. The main purpose of this is to distribute substitution over dismantling.

nominal-primrec
ssubst :: *name* ⇒ *trm* ⇒ *stack* ⇒ *stack*
where
ssubst x v Id = *Id*
| $y \sharp (k,x,v) \implies$ *ssubst x v* ($[y]n \gg k$) = $[y](n[x::=v]) \gg$(*ssubst x v k*)
by(*finite-guess*+ , (*simp add*: *abs-fresh*)+ , *fresh-guess*+)

lemma *ssubst-fresh*:
fixes y :: *name*
assumes $y \sharp (x,v,k)$
shows $y \sharp$ *ssubst x v k*
using *assms*
by(*nominal-induct k avoiding*: *y x v rule*: *stack.strong-induct*)
(*auto simp add*: *fresh-prod fresh-atm abs-fresh fresh-fact*)

lemma *ssubst-forget*:
fixes x :: *name*
assumes $x \sharp k$
shows *ssubst x v k* = *k*
using *assms*
by(*nominal-induct k avoiding*: *x v rule*: *stack.strong-induct*)
(*auto simp add*: *abs-fresh fresh-atm forget*)

lemma *subst-dismantle*[*simp*]: $(t \star k)[x ::= v] = (t[x::=v]) \star$ *ssubst x v k*
by(*nominal-induct k avoiding*: *t x v rule*: *stack.strong-induct*)
(*auto simp add*: *ssubst-fresh fresh-prod fresh-fact*)

5.5 Reducibility for Terms and Stacks

Following [Nom], we formalize the logical relation as a function *RED* of type *ty* ⇒ *trm set* for the term part and accordingly *SRED* of type *ty* ⇒ *stack set* for the stack part of the logical relation. Showing that these mutually recursive functions terminate is therefore equivalent to showing that the logical relation is correctly defined on the type structure.

lemma *ty-exhaust*: $ty = TBase \vee (\exists\ \sigma\ \tau\ .\ ty = \sigma \to \tau) \vee (\exists\ \sigma\ .\ ty = T\ \sigma)$
by(*induct ty rule*:*ty.induct*) (*auto*)

```
function RED :: ty ⇒ trm set
and      SRED :: ty ⇒ stack set
where
  RED (TBase) = {t. SN(t)}
| RED (τ→σ) = {t. ∀ u ∈ RED τ . (App t u) ∈ RED σ }
| RED (T σ) = {t. ∀ k ∈ SRED σ . SN(t ⋆ k) }
| SRED τ = {k. ∀ t ∈ RED τ . SN ([t] ⋆ k) }
by(auto simp add: ty.inject, case-tac x rule: sum.exhaust,insert ty-exhaust)
  (blast)+
```

This is the second non-primitive function in the formalization. Since types do not involve binders, pattern completeness and right uniqueness are mostly trivial. The termination argument is not as simple as for the dismantling function, because the definiton of *SRED* τ involves a recursive call to *RED* τ of the same size.

```
nominal-primrec
  tsize :: ty ⇒ nat
where
    tsize TBase = 1
  | tsize (σ→τ) = 1 + tsize σ + tsize τ
  | tsize (T τ) = 1 + tsize τ
by (rule TrueI)+
```

In the termination argument below, *Inl* τ corresponds to the call *RED* τ, whereas *Inr* τ corresponds to *SRED* τ

```
termination RED
by(relation measure
    (λ x . case x of Inl τ ⇒ 2 ∗ tsize τ
                   | Inr τ ⇒ 2 ∗ tsize τ + 1)) (auto)
```

5.6 Properties of the Reducibility Relation

After defining the logical relations we need to prove that the relation implies strong normalization, is preserved under reduction, and satisfies the head expansion property.

```
constdefs
  NEUT :: trm ⇒ bool
  NEUT t ≡ (∃ a. t = Var a) ∨ (∃ t1 t2. t = App t1 t2)
```

```
constdefs
  CR1 :: ty ⇒ bool
  CR1 τ ≡ ∀ t. (t∈RED τ ⟶ SN t)

  CR2 :: ty ⇒ bool
  CR2 τ ≡ ∀ t t'. (t∈RED τ ∧ t ↦ t') ⟶ t'∈RED τ
```

CR3-RED :: *trm* ⇒ *ty* ⇒ *bool*
CR3-RED t τ ≡ ∀ *t′*. *t* ↦ *t′* ⟶ *t′*∈*RED τ*

CR3 :: *ty* ⇒ *bool*
CR3 τ ≡ ∀ *t*. (*NEUT t* ∧ *CR3-RED t τ*) ⟶ *t*∈*RED τ*

CR4 :: *ty* ⇒ *bool*
CR4 τ ≡ ∀ *t*. (*NEUT t* ∧ *NORMAL t*) ⟶*t*∈*RED τ*

lemma *CR3-implies-CR4*[*intro*]: *CR3 τ* ⟹ *CR4 τ*
by (*auto simp add*: *CR3-def CR3-RED-def CR4-def NORMAL-def*)

5.6.1 Strong normalization for subterms and stacks

To prove *CR1-3* for the type constructor →, we need a way to obtain *SN s* from *SN* (*App s t*). This can not be defined using a simple function, since HOL is a logic of total functions and we only want to project to the first element of an application. One could use a function from *trm* to *trm option* but this would not generalize to the case of dismantling below. Thus, we define a one case inductive relation between terms establishing the desired connection of *App s t* and *s*.

inductive
FST :: *trm*⇒*trm*⇒*bool* (- ≫ - [*80*,*80*] *80*)
where
fst[*intro*!]: (*App t s*) ≫ *t*

lemma *SN-of-FST-of-App*:
assumes *a*: *SN* (*App t s*)
shows *SN t*
proof −
from *a* **have** ∀ *z*. (*App t s* ≫ *z*) ⟶ *SN z*
by (*induct rule*: *SN.induct*)
(*blast elim*: *FST.cases intro*: *SN-intro*)
then show *SN t* **by** *blast*
qed

This lemma is a simplified version of the one used in [Nom]. Since we have generalized our notion of reduction from terms to stacks, we can also generalize the notion of strong normalization. The new induction principle will be used to prove the *T* case of the properties of the reducibility relation.

inductive
SSN :: *stack* ⇒ *bool*
where
SSN-intro: (⋀ *k′* . *k* ↦ *k′* ⟹ *SSN k′*) ⟹ *SSN k*

Furthermore, the approach for deriving strong normalization of subterms from above can be generalized to terms of the form $t \star k$. In contrast to the case of applications, $t \star k$ does *not* uniquely determine t and k. Thus, the extraction is a proper relation in this case.

```
inductive
  SND-DIS :: trm ⇒ stack ⇒ bool (- ▷ -)
where
  snd-dis[intro!]: t ⋆ k ▷ k
```

Lemmas like *SN-of-FST-of-App* are usually not proven at all or proven by contradiction – using the fact that any infinite sequence in a subterm implies an infinite sequence in the whole term. For this reason, the inductive proof below is shown in length, although it could have been proven using automated reasoning tools similar to the case above.

```
lemma SN-SSN:
  assumes a: SN (t ⋆ k)
  shows  SSN k
proof −
  from a have ∀ z. (t ⋆ k ▷ z) ⟶ SSN z
   proof (induct rule: SN.induct)
    case (SN-intro u)
    have ih: ⋀u'. u ↦ u' ⟹ ∀ z. u' ▷ z ⟶ SSN z by fact
    show ∀ z. u ▷ z ⟶ SSN z
      proof (intro allI impI)
        fix z  assume u ▷ z
        thus SSN z proof (cases rule:SND-DIS.cases)
          case (snd-dis v -)  hence u: u = v ⋆ z by simp
          { fix z' assume z ↦ z'
            with u have u ↦ v ⋆ z' by (simp add: stack-reduction-def)
            hence ∀ z. (v ⋆ z') ▷ z ⟶ SSN z by (rule ih)
            with u have SSN z' by blast }
          thus SSN z ..
        qed
      qed
    qed
  thus SSN k by blast
qed
```

5.6.2 A new case construct on the reducts of $t \star k$

To prove the properties of the logical relation, the authors of [LS05] use a case distinction on the reducts of $t \star k$, where t is a neutral term and therefore no interaction occurs between t and k.

$$\frac{t \star k \mapsto r \qquad \bigwedge t'.\ [\![t \mapsto t';\ r = t' \star k]\!] \Longrightarrow P \qquad NEUT\ t \qquad \bigwedge k'.\ [\![\ k \mapsto k'\ ;\ r = t \star k']\!] \Longrightarrow P}{P}$$

We strive for a proof of this rule by structural induction on k. The general idea of the case where $k = [y]n \gg l$ is to move the first stack frame into the term t and then apply the induction hypothesis as a case rule. Unfortunately, this term is no longer neutral, so, for the induction to go through, we need to generalize the claim to also include the possible interactions of non-neutral terms and stacks.

lemma *dismantle-cases*:
  fixes t :: *trm*
  assumes r: $t \star k \mapsto r$
  and T: $\bigwedge t'\ .\ [\![\ t \mapsto t'\ ;\ r = t' \star k\ ]\!] \Longrightarrow P$
  and K: $\bigwedge k'\ .\ [\![\ k \mapsto k'\ ;\ r = t \star k'\ ]\!] \Longrightarrow P$
  and B: $\bigwedge s\ y\ n\ l\ .[\![\ t = [s]\ ;\ k = [y]n \gg l\ ;\ r = (n[y::=s]) \star l\ ]\!] \Longrightarrow P$
  and A: $\bigwedge u\ x\ v\ y\ n\ l.[\![\ x \sharp y;\ x \sharp n\ ;\ t = u\ to\ x\ in\ v\ ;$
        $k = [y]n \gg l\ ;\ r = (u\ to\ x\ in\ (v\ to\ y\ in\ n)) \star l\ ]\!] \Longrightarrow P$
  shows P
using *assms*
proof (*nominal-induct k avoiding*: t r *rule*:*stack.strong-induct*)
  case (*St y n L*) **note** *yfresh* = ‹$y \sharp t$› ‹$y \sharp r$› ‹$y \sharp L$›
  note *IH* = *St*(*4*)
    and T = *St*(*6*) **and** K = *St*(*7*) **and** B = *St*(*8*) **and** A = *St*(*9*)

The hypothesis we get from the induction on the stack k can be found in Figure 5.3. We immediately strive for the induction hypothesis by using it as case rule moving the first stack frame to the left of the dismantling operator. Hence we get five cases corresponding to the premises of the induction hypothesis.

  thus P **proof** (*cases rule*:*IH*[**where** *b*=*t to y in n* **and** *ba*=*r*])
    case (*2 r'*) **have** *red*: t *to* y *in* $n \mapsto r'$ **and** r: $r = r' \star L$ **by** *fact*+

If m *to* y *in* n makes a step we reason by case distinction on the successors of m *to* y *in* n. We want to use the strong inversion principle for the reduction relation. For this we need that y is fresh for t *to* y *in* n and r'.

    from *yfresh r* **have** y: $y \sharp$ t *to* y *in* n $y \sharp r'$
      by (*auto simp add*: *abs-fresh*)
    obtain z **where** z: $z \neq y$ $z \sharp r'$ $z \sharp t$ *to* y *in* n
      using *ex-fresh*[*of* (y,r',t *to* y *in* n)]
      by(*auto simp add*:*fresh-prod fresh-atm*)
    from *red* r **show** P
    proof (*cases rule*:*reduction.strong-cases*
        [**where** *x*=*y***and** *xa*=*y* **and** *xb*=*y* **and** *xc*=*y* **and** *xd*=*y*
          and *xe*=*y* **and** *xf*=*y* **and** *xg*=*z* **and** *y*=*y*])
      case (*r6 s t' u*) — if t makes a step we use assumption T

We can assume the following hypotheses:

$$\frac{t \mapsto ?t' \qquad r = ?t' \star [y]n\gg L}{P}\ \mathrm{T}$$

$$\frac{[y]n\gg L \mapsto ?k' \qquad r = t \star ?k'}{P}\ \mathrm{K}$$

$$\frac{t = [?s] \qquad [y]n\gg L = [?y]?n\gg ?l \qquad r = ?n[?y::=?s] \star ?l}{P}\ \mathrm{B}$$

$$\frac{\begin{array}{c}?x \sharp ?y \qquad ?x \sharp ?n \qquad t = ?u\ to\ ?x\ in\ ?v \\ [y]n\gg L = [?y]?n\gg ?l \qquad r = (?u\ to\ ?x\ in\ ?v\ to\ ?y\ in\ ?n) \star ?l\end{array}}{P}\ \mathrm{A}$$

And we get the following (large) induction hypothesis:

$$\begin{array}{l}?b \star L \mapsto ?ba \\ \bigwedge t'.\ [\![?b \mapsto t';\ ?ba = t' \star L]\!] \Longrightarrow P \\ \bigwedge k'.\ [\![L \mapsto k' ;\ ?ba = ?b \star k']\!] \Longrightarrow P \\ \bigwedge s\ y\ n\ l.\ [\![?b = [s];\ L = [y]n\gg l;\ ?ba = n[y::=s] \star l]\!] \Longrightarrow P \\ \bigwedge x\ y\ n\ u\ v\ l. \\ \quad [\![x \sharp y;\ x \sharp n;\ ?b = u\ to\ x\ in\ v;\ L = [y]n\gg l; \\ \quad\ ?ba = (u\ to\ x\ in\ v\ to\ y\ in\ n) \star l]\!] \\ \quad \Longrightarrow P \end{array}$$

$$\overline{\qquad\qquad P}$$

Figure 5.3: *Hypotheses for dismantle-cases*

```
    with y have m: t ↦ t'   r' = t' to y in n by auto
    thus P using T[of t'] r by auto
  next
    case (r7 - - n') with y have n: n ↦ n' and r': r' = t to y in n'
      by (auto simp add: alpha)
```

Since $k = [y]n \gg L$, the reduction $n \mapsto n'$ occurs within the stack k. Hence, we need to establish this stack reduction.

```
    have [y]n≫L ↦ [y]n'≫L unfolding stack-reduction-def
    proof
      fix u have u to y in n ↦ u to y in n' using n ..
      hence (u to y in n) ⋆ L ↦ (u to y in n') ⋆ L ..
      thus  u ⋆ [y]n≫L ↦ u ⋆ [y]n'≫L
        by simp
    qed
    moreover have r = t ⋆ [y]n'≫L using r r' by simp
    ultimately show P by (rule K)
  next
    case (r8 s -) — the case of a β-reduction is exactly B
    with y have t = [s]   r' = n[y::=s] by(auto simp add: alpha)
    thus P using B[of s y n L] r by auto
  next
    case (r9 -) — The case of an η-reduction is a stack reduction as well.
    with y have n: n = [Var y] and r': r' = t
      by(auto simp add: alpha)
    { fix u have u to y in n ↦ u unfolding n ..
      hence (u to y in n) ⋆ L ↦ u ⋆ L  ..
      hence u ⋆ [y]n≫L ↦ u ⋆ L by simp
    } hence [y]n≫L ↦ L unfolding stack-reduction-def ..
    moreover have r = t ⋆ L using r r' by simp
    ultimately show P by (rule K)
  next
    case (r10 u - v) — The assoc case holds by A.
    with y z have
      t = (u to z in v)
      r' = u to z in (v to y in n)
      z ♯ (y,n) by (auto simp add: fresh-prod alpha)
    thus P using A[of z y n] r by auto
  qed (insert y, auto)  — No other reductions are possible.
next
```

Next we have to solve the case where a reduction occurs deep within L. We get a reduction of the stack k by moving the first stack frame "[y]n" back to the right hand side of the dismantling operator.

```
  case (3 L')
  hence L: L ↦ L' and  r: r = (t to y in n) ⋆ L' by auto
  { fix s from L have  (s to y in n) ⋆ L ↦ (s to y in n) ⋆ L'
      unfolding stack-reduction-def ..
    hence s ⋆ [y]n≫L ↦ s ⋆ [y]n≫L' by simp
  } hence [y]n≫L ↦ [y]n≫L' unfolding stack-reduction-def by auto
```

```
    moreover from r have r = t ⋆ [y]n≫L' by simp
    ultimately show P by (rule K)
  next
    case (5 x z n' s v K) — The "assoc" case is again a stack reduction
    have xf: x ♯ z   x ♯ n'
      — We get the following equalities
    and red: t to y in n = s to x in v
            L = [z]n'≫K
            r = (s to x in v to z in n') ⋆ K by fact+
    { fix u from red have u ⋆ [y]n≫L = ((u to x in v) to z in n') ⋆ K
        by(auto intro: arg-cong[where f=λ x . x ⋆ K])
      moreover
      { from xf have (u to x in v) to z in n' ↦ u to x in (v to z in n') ..
        hence ((u to x in v) to z in n') ⋆ K ↦ (u to x in (v to z in n')) ⋆ K
          by rule
      } ultimately  have u ⋆ [y]n≫L ↦ (u to x in (v to z in n')) ⋆ K
        by (simp (no-asm-simp) del:dismantle-simp)
      hence u ⋆ [y]n≫L ↦ u ⋆ [x](v to z in n')≫K by simp
    } hence [y]n≫L ↦  [x](v to z in n')≫ K
      unfolding stack-reduction-def by simp
    moreover have r = t ⋆ ([x](v to z in n')≫K) using red
      by (auto)
    ultimately show P by (rule K)
  qed (insert St, auto )
qed auto
```

Now that we have established the general claim, we can restrict t to neutral terms only and drop the cases dealing with possible interactions.

```
lemma dismantle-cases'[consumes 2, case-names T K]:
  fixes m :: trm
  assumes r: t ⋆ k ↦ r
  and NEUT t
  and ⋀ t' . ⟦ t ↦ t' ; r = t' ⋆ k ⟧ ⟹ P
  and ⋀ k' . ⟦ k ↦ k' ; r = t ⋆ k' ⟧ ⟹ P
  shows P
using assms unfolding NEUT-def
by (cases rule: dismantle-cases[of t k r]) (auto)
```

5.6.3 Proof of the properties of reducibility

Now we are only two simple lemmas away from proving the properties of the reducibility relation.

```
lemma red-Ret:
  fixes t :: trm
  assumes [s] ↦ t
  shows  ∃ s' . t = [s'] ∧ s  ↦ s'
using assms by cases (auto)
```

lemma *SN-Ret*: *SN u* $\Longrightarrow$ *SN* [*u*]
by(*induct rule*:*SN.induct*) (*metis SN.intros red-Ret*)

All the properties of reducibility are shown simultaneously by induction on the type. Lindley and Stark [LS05] only spell out the cases dealing with the monadic type constructor T. We do the same by reusing the proofs from [Nom] for the other cases. To shorten the presentation, these proofs are folded as ⟨*Urban*⟩.

lemma *RED-props*:
shows *CR1* τ **and** *CR2* τ **and** *CR3* τ
proof (*nominal-induct* τ *rule*: *ty.strong-induct*)
case *TBase* ⟨*Urban*⟩
next
case (*TFun* $\tau 1$ $\tau 2$) ⟨*Urban*⟩
next
case (*T* σ)
{ case *1* — follows from the fact that *Id* $\in$ *SRED* σ
have *ih-CR1-σ*: *CR1* σ **by** *fact*
{ fix *t* **assume** *t-red*: $t \in$ *RED* (*T* σ)
{ fix *s* **assume** $s \in$ *RED* σ
hence *SN s* **using** *ih-CR1-σ* **by** (*auto simp add*: *CR1-def*)
hence *SN* ([*s*]) **by** (*rule SN-Ret*)
hence *SN* ([*s*] $\star$ *Id*) **by** *simp*
} hence *Id* $\in$ *SRED* σ **by** *simp*
with *t-red* **have** *SN* (*t*) **by** (*auto simp del*: *SRED.simps*)
} thus *CR1* (*T* σ) **unfolding** *CR1-def* **by** *blast*
next
case *2* — follows since *SN* is preserved under redcution
{ fix *t t'*::*trm* **assume** *t-red*: $t \in$ *RED* (*T* σ) **and** *t-t'*: $t \mapsto t'$
{ fix *k* **assume** *k*: $k \in$ *SRED* σ
with *t-red* **have** *SN*$(t \star k)$ **by** *simp*
moreover from *t-t'* **have** $t \star k \mapsto t' \star k$ **..**
ultimately have *SN*$(t' \star k)$ **by** (*rule SN-preserved*)
} hence $t' \in$ *RED* (*T* σ) **by** (*simp del*: *SRED.simps*)
} thus *CR2* (*T* σ)**unfolding** *CR2-def* **by** *blast*
next
case *3* **from** ‹*CR3* σ› **have** *ih-CR4-σ* : *CR4* σ **..**
{ fix *t* **assume** *t'-red*: $\bigwedge t' .\ t \mapsto t' \Longrightarrow t' \in$ *RED* (*T* σ)
and *neut-t*: *NEUT t*
{ fix *k* **assume** *k-red*: $k \in$ *SRED* σ
fix *x* **have** *NEUT* (*Var x*) **unfolding** *NEUT-def* **by** *simp*
hence *Var x* $\in$ *RED* σ **using** *normal-var ih-CR4-σ*
by (*simp add*: *CR4-def*)
hence *SN* ([*Var x*] $\star$ *k*) **using** *k-red* **by** *simp*
hence *SSN k* **by** (*rule SN-SSN*)
then have *SN* (*t* $\star$ *k*) **using** *k-red*
proof (*induct k rule*:*SSN.induct*)

Let t be neutral such that $t' \in RED_{T\,\sigma}$ whenever $t \mapsto t'$. We have to show that $(t \star k)$ is SN for each $k \in SRED_\sigma$. First, we have that $[x] \star k$ is SN, as $x \in RED_\sigma$ by the induction hypothesis. Hence k itself is SN, and we can work by induction on $\max(k)$. Application $t \star k$ may reduce as follows:

- $t' \star k$, where $t \mapsto t'$, which is SN as $k \in SRED_\sigma$ and $t' \in RED_{T\,\sigma}$.
- $t \star k'$, where $k \mapsto k'$. For any $s \in RED_\sigma$, $[s] \star k$ is SN as $k \in SRED_\sigma$; and $[s] \star k \mapsto [s] \star k'$, so $[s] \star k'$ is also SN. From this we have $k' \in SRED_\sigma$ with $\max(k') < \max(k)$, so by induction hypothesis $t \star k'$ is SN.

There are no other possibilities as t is neutral. Hence $t \star k$ is strongly normalizing for every $k \in SRED_\sigma$, and so $t \in RED_{T\,\sigma}$ as required.

Figure 5.4: *Proof of the case $T\,\sigma$ subcase CR3 as in [LS05]*

```
      case (SSN-intro k)
      have ih : ⋀k'. ⟦ k ↦ k' ; k' ∈ SRED σ ⟧ ⟹ SN (t ⋆ k')
        and k-red: k ∈ SRED σ by fact+
      { fix r assume r: t ⋆ k ↦ r
        hence SN r using neut-t
        proof (cases rule:dismantle-cases')
          case (T t') hence t-t': t ↦ t' and r-def: r = t' ⋆ k .
          from t-t' have t' ∈ RED (T σ) by (rule t'-red)
          thus SN r using k-red r-def by simp
        next
          case (K k') hence k-k': k ↦ k' and r-def: r = t ⋆ k' .
          { fix s assume s ∈ RED σ
           hence SN ([s] ⋆ k) using k-red
             by simp
           moreover have [s] ⋆ k ↦ [s] ⋆ k'
             using k-k' unfolding stack-reduction-def ..
           ultimately have SN ([s] ⋆ k') ..
          } hence k' ∈ SRED σ by simp
          with k-k' show SN r unfolding r-def by (rule ih)
        qed } thus SN (t ⋆ k) ..
    qed } hence t ∈ RED (T σ) by simp
  } thus CR3 (T σ) unfolding CR3-def CR3-RED-def by blast
 }
qed
```

The last case above shows that, once all the reasoning principles have been established, some proofs have a formalization which is amazingly close to the informal version. For a direct comparison, the informal proof is presented in Figure 5.4.

Now that we have established the properties of the reducibility relation, we need to show that reducibility is preserved by the various term constructors. The only nontrivial cases are abstraction and sequencing.

5.7 Abstraction Preserves Reducibility

Once again we could reuse the proofs from [Nom]. The proof uses the *double-SN* rule and the lemma *red-Lam* below. Unfortunately, this time the proofs are not fully identical to the proofs in [Nom] because we consider $\beta\eta$-reduction rather than β-reduction. The cases for η-reductions had to be to "patched in", mainly by changing the *red-Lam* lemma accordingly, but some minor adjustments also had to be made to the *abs-RED* lemma.

lemma *double-SN*[*consumes 2*]:
  assumes *a*: *SN a*
  and *b*: *SN b*
  and *c*: $\bigwedge(x::trm)\ (z::trm).$
    $[\![\bigwedge y.\ x \mapsto y \Longrightarrow P\ y\ z;\ \bigwedge u.\ z \mapsto u \Longrightarrow P\ x\ u]\!] \Longrightarrow P\ x\ z$
  shows *P a b*
using *a b c*
⟨*Urban*⟩

lemma *red-Lam*:
  assumes *a*: $\Lambda\ x\ .\ t \mapsto r$
  shows $(\exists t'.\ r = \Lambda\ x\ .\ t' \wedge t \mapsto t') \vee (t = App\ r\ (Var\ x) \wedge x \sharp r\ )$
proof −
  obtain *z*::*name* **where** *z*: $z \sharp x \quad z \sharp t \quad z \sharp r$
    using *ex-fresh*[*of* (*x*,*t*,*r*)] **by** (*auto simp add*: *fresh-prod*)
  have $x \sharp \Lambda\ x\ .\ t$ **by** (*simp add*: *abs-fresh*)
  with *a* **have** $x \sharp r$ **by** (*simp add*: *reduction-fresh*)
  with *a* **show** *?thesis* **using** *z*
    by(*cases rule*: *reduction.strong-cases*
      [**where** *x* =*x* **and** *xa*=*x* **and** *xb*=*x* **and** *xc*=*x* **and**
        xd=*x* **and** *xe*=*x* **and** *xf*=*x* **and** *xg*=*x* **and** *y*=*z*])
      (*auto simp add*: *abs-fresh alpha fresh-atm*)
qed

lemma *abs-RED*:
  assumes *asm*: $\forall\, s{\in}RED\ \tau.\ t[x::=s]{\in}RED\ \sigma$
  shows $\Lambda\ x\ .\ t\ {\in}RED\ (\tau{\rightarrow}\sigma)$
⟨*Urban*⟩

5.8 Sequencing Preserves Reducibility

This section corresponds to the main part of the paper being formalized and as such deserves special attention. In the lambda case one has to formalize doing induction on $\max(s) + max(t)$ for two strongly normalizing terms s and t (cf. [GTL89, Section 6.3]). Above, this was done through a *double-SN* rule. The central Lemma 7 of Lindley and Stark's paper uses an even more complicated induction scheme. They assume terms p and n as well as a stack K such that $SN\ p$ and $SN\ (n[x::=p] \star K)$. The induction is then done on $|K| + max(n \star K) + max(p)$. See Figure 5.5 in for details.

5.8.1 Triple induction principle

Since we have settled for a different characterization of strong normalization, we have to derive an induction principle similar in spirit to the *double-SN* rule. There are, however, some complications. To prove the triple induct principle one needs to keep the variables that the induction is performed on *independent* from one another. Hence the double occurrence of K in the sum above needs to be handled by a suitable abstraction when instantiating the rule.

Furthermore, it turns out that it is not necessary to formalize the fact that stack reductions do not increase the length of the stack.[1] Doing induction on the sum above, this is necessary to handle the case of a reduction occurring in K. We differ from [LS05] and establish an induction principle which to some extent resembles the lexicographic order on

$$(SN, \mapsto) \times (SN, \mapsto) \times (\mathbb{N}, >)\,.$$

A direct translation would correspond to the lemma:

lemma *triple-induct'*[*consumes 2*]:
  assumes *a*: $SN\ p$
  and *b*: $SN\ (q)$
  and *hyp*: $\bigwedge\ (p::trm)\ (q::trm)\ (K::stack)$.
  $[\![\bigwedge\ p'\ q\ K.\ [\![SN\ q; p \mapsto p']\!] \Longrightarrow P\ p'\ q\ K$;
    $\bigwedge\ q'\ K$. $q \mapsto q' \Longrightarrow P\ p\ q'\ K$;
    $\bigwedge\ K'$. $|K'| < |K| \Longrightarrow P\ p\ q\ K']\!] \Longrightarrow P\ p\ q\ K$
  shows $P\ p\ q\ K$
oops

The rule derived this was is, however, more general (and hence more difficult to instantiate) than the one we need. Thus, we use the variation below, in

[1] This possibility was only discovered *after* having formalized $K \mapsto K' \Rightarrow |K| \geq |K'|$. The proof of this seemingly simple fact was about 90 lines of Isar code.

Lemma 5.8.1. *(Lemma 7) Let p, n be terms and K a stack such that $SN(p)$ and $SN(n[x ::= p] \star K)$. Then $SN(([p]\ \mathsf{to}\ x\ \mathsf{in}\ n) \star K)$*

Proof. We show by induction on $|K| + max(n \star K) + max(p)$ that the reducts of $([p]\ \mathsf{to}\ x\ \mathsf{in}\ n) \star K$ are all strongly normalizing. The interesting reductions are as follows:

- $T.\beta$ giving $n[x ::= p] \star K$ which is strongly normalizing by hypothesis.
- $T.\eta$ when $n = [x]$ giving $[p] \star K$. But $[p] \star K = n[x ::= p] \star K$ which is again strongly normalizing by hypothesis
- $T.assoc$ in the case where $K = [y]m \gg K'$ with $x \notin fv(m)$; giving the reduct $([p]\ \mathsf{to}\ x\ \mathsf{in}\ (n\ \mathsf{to}\ y\ \mathsf{in}\ m)) \star K$. We aim to apply the induction hypothesis with K' and $(n\ \mathsf{to}\ y\ \mathsf{in}\ m)$ for K and n respectively. Now

  $$\begin{aligned}(n\ \mathsf{to}\ y\ \mathsf{in}\ m)[x ::= p] \star K' &= (n[x ::= p]\ \mathsf{to}\ y\ \mathsf{in}\ m) \star K' \\ &= n[x ::= p] \star K\end{aligned}$$

  which is strongly normalizing by induction hypothesis. Also

  $$|K'| + max((n\ \mathsf{to}\ y\ \mathsf{in}\ m) \star K') + max(p) < |K| + max(n \star K) + max(p)$$

  as $|K'| < |K|$ and $(n\ \mathsf{to}\ y\ \mathsf{in}\ m) \star K' = n \star K$. This last equation explains the use of $max(n \star K)$; it remains fixed under $T.assoc$ unlike $max(K)$ and $max(n)$. Applying the induction hypothesis gives $SN(([p]\ \mathsf{to}\ x\ \mathsf{in}\ (n\ \mathsf{to}\ y\ \mathsf{in}\ m)) \star K)$ as required.

Other reductions are confined to K, n or p and can be treated by the induction hypothesis, decreasing either $max(n \star K)$ or $max(p)$.

Figure 5.5: *Proof of Lemma 7 as in [LS05]*

which the first hypothesis is only applicable to the original q and k, with the benefit that we can drop the extra premise $SN\ q$.

lemma *triple-induct*[*consumes 2*]:
  assumes *a*: $SN\ (p)$
  and *b*: $SN\ (q)$
  and *hyp*: $\bigwedge\ (p{::}trm)\ (q{::}trm)\ (k{::}stack)$.
  $[\![\bigwedge\ p' .\ p \mapsto p' \Longrightarrow P\ p'\ q\ k\ ;$
    $\bigwedge\ q'\ k\ .\ q \mapsto q' \Longrightarrow P\ p\ q'\ k;$
    $\bigwedge\ k' .\ |k'| < |k| \Longrightarrow P\ p\ q\ k'\ ]\!] \Longrightarrow P\ p\ q\ k$
  shows $P\ p\ q\ k$
proof −
  from *a* **have** $\bigwedge q\ K\ .\ SN\ q \Longrightarrow P\ p\ q\ K$
  proof (*induct p*)
    case (*SN-intro p*)
    have *sn1*: $\bigwedge\ p'\ q\ K\ .\ [\![p \mapsto p'; SN\ q]\!] \Longrightarrow P\ p'\ q\ K$ **by** *fact*
    have *sn-q*: $SN\ q \quad SN\ q$ **by** *fact+*
    thus $P\ p\ q\ K$
    proof (*induct q arbitrary*: *K*)
      case (*SN-intro q K*)
      have *sn2*: $\bigwedge\ q'\ K\ .\ [\![q \mapsto q'; SN\ q']\!] \Longrightarrow P\ p\ q'\ K$ **by** *fact*
      show $P\ p\ q\ K$
      proof (*induct K rule*: *measure-induct-rule*[**where** *f*=*length*])
        case (*less k*)
        have *le*: $\bigwedge\ k'\ .\ |k'| < |k| \Longrightarrow P\ p\ q\ k'$ **by** *fact*
        { fix p' **assume** $p \mapsto p'$
          moreover have $SN\ q$ **by** *fact*
          ultimately have $P\ p'\ q\ k$ **using** *sn1* **by** *auto* **}**
        moreover
        { fix $q'\ K$ **assume** *r*: $q \mapsto q'$
          have $SN\ q$ **by** *fact*
          hence $SN\ q'$ **using** *r* **by** (*rule SN-preserved*)
          with *r* **have** $P\ p\ q'\ K$ **using** *sn2* **by** *auto* **}**
        ultimately show *?case* **using** *le*
          by (*auto intro*:*hyp*)
      qed
    qed
  qed
  with *b* **show** *?thesis* **by** *blast*
qed

5.8.2 Strengthening of the dismantle case rule

Here we strengthen the case rule for terms of the form $t \star k \mapsto r$. This is similar to the nominal inversion rules described in Section 4.5.4. The freshness requirements on x,y, and z correspond to those for the rule *reduction.strong-cases*, the strong inversion principle for the reduction relation.

lemma *dismantle-strong-cases*:
fixes t :: *trm*
assumes r: $t \star k \mapsto r$
and f: $y \sharp (t,k,r)$ $x \sharp (z,t,k,r)$ $z \sharp (t,k,r)$
and T: $\bigwedge t'$. $[\![t \mapsto t' ; r = t' \star k]\!] \Longrightarrow P$
and K: $\bigwedge k'$. $[\![k \mapsto k' ; r = t \star k']\!] \Longrightarrow P$
and B: $\bigwedge s\ n\ l$. $[\![t = [s] ;$
$k = [y]n \gg l ; r = (n[y::=s]) \star l]\!] \Longrightarrow P$
and A: $\bigwedge u\ v\ n\ l$.
$[\![x \sharp (z,n); t = u\ to\ x\ in\ v ; k = [z]n \gg l ;$
$r = (u\ to\ x\ in\ (v\ to\ z\ in\ n)) \star l]\!] \Longrightarrow P$
shows P
proof (*cases rule*:*dismantle-cases*[*of t k r P*])
case (*4 s y' n L*) **have** *ch*:
$t = [s]$
$k = [y']n \gg L$
$r = n[y'::=s] \star L$ **by** *fact+*

The equations we get look almost like those we need to instantiate the hypothesis B. The only difference is that B only applies to y, and since we want y to become an instantiation variable of the strengthened rule, we only know that y satisfies f and nothing else. But the condition f is just strong enough to rename y' to y and apply B.

with f **have** $y = y' \vee y \sharp n$
by (*auto simp add*: *fresh-prod abs-fresh*)
hence $n[y'::=s] = ([(y,y')] \cdot n)[y::=s]$
and $[y']n \gg L = [y]([(y,y')] \cdot n) \gg L$
by(*auto simp add*: *name-swap-bij subst-rename' stack.inject alpha'*)
with *ch* **have** $t = [s]$
$k = [y]([(y,y')] \cdot n) \gg L$
$r = ([(y,y')] \cdot n)[y::=s] \star L$
by (*auto*)
thus P **by** (*rule B*)
next
case (*5 u x' v z' n L*) **have** *ch*:
$x' \sharp z'$ $x' \sharp n$
$t = u\ to\ x'\ in\ v$
$k = [z']n \gg L$
$r = (u\ to\ x'\ in\ v\ to\ z'\ in\ n) \star L$ **by** *fact+*

We want to do the same trick as above but at this point we have to take care of the possibility that x might coincide with x' or z'. Similarly, z might coincide with z'.

with f **have** x: $x = x' \vee x \sharp v\ to\ z'\ in\ n$
and z: $z = z' \vee z \sharp n$
by (*auto simp add*: *fresh-prod abs-fresh*)
from *f ch* **have** x': $x' \sharp n$ $x' \sharp z'$
and xz': $x = z' \vee x \sharp n$
by (*auto simp add*:*name-swap-bij alpha fresh-prod fresh-atm abs-fresh*)
from *f ch* **have** $x \sharp z$ $x \sharp [z'].n$ **by** (*auto simp add*: *fresh-prod*)
with xz' z **have** $x \sharp (z\ ,\ ([(z,\ z')] \cdot n))$

```
    by (auto simp add: fresh-atm fresh-bij name-swap-bij
          fresh-prod abs-fresh calc-atm fresh-aux fresh-left)
  moreover from x ch have t = u to x in ([(x,x')] • v)
    by (auto simp add:name-swap-bij alpha')
  moreover from z ch have k = [z]([(z,z')] • n)≫L
    by (auto simp add:name-swap-bij stack.inject alpha')
```

The first two α-renamings are simple, but here we have to handle the nested binding structure of the assoc rule. Since x scopes over the whole term $v\ to\ z'\ in\ n$, we have to push the swapping over z'

```
  moreover { from x have
    u to x' in (v to z' in n) = u to x in ([(x,x')] • (v to z' in n))
      by (auto simp add:name-swap-bij alpha' simp del: trm.perm)
    also from xz' x' have ... = u to x in (([(x,x')] • v) to z' in n)
      by (auto simp add: abs-fun-eq1 swap-simps alpha'')
         (metis alpha'' fresh-atm perm-fresh-fresh swap-simps(1) x')
    also from z have  ... = u to x in (([(x,x')] • v) to z in ([(z,z')] • n))
      by (auto simp add: abs-fun-eq1 alpha' name-swap-bij )
    finally
      have r =  (u to x in (([(x, x')] • v) to z in ([(z, z')] • n))) ⋆ L
        using ch by (simp del: trm.inject) }
  ultimately show P
    by (rule A[where n=[(z, z')] • n and v=([(x, x')] • v)])
qed (insert r T K, auto)
```

5.8.3 Strong normalization and substitution

The lemma in Figure 5.5 assumes $SN\ (n[x::=p] \star K)$ but the actual induction in done on $SN\ (n \star K)$. The stronger assumption $SN\ (n[x::=p] \star K)$ is needed to handle the β and η cases.

```
lemma sn-forget:
  assumes a: SN(t[x::=v])
  shows SN t
proof −
  def dq: q ≡ t[x::=v]
  from a have SN q unfolding dq .
  thus SN t using dq
  proof (induct q arbitrary: t)
    case (SN-intro q t)
    hence ih: ⋀ t'. ⟦t[x::=v] ↦ t'[x::=v]⟧ ⟹ SN t' by auto
    { fix t' assume t ↦ t'
      hence t[x::=v] ↦ t'[x::=v] by (rule reduction-subst)
      hence SN t' by (rule ih) }
    thus SN t ..
  qed
qed
```

lemma *sn-forget'*:
  assumes *sn*: $SN\ (t[x::=p] \star k)$
  and x: $x \sharp k$
  shows $SN\ (t \star k)$
proof −
  from x **have** $t[x::=p] \star k = (t \star k)[x::=p]$ **by** (*simp add*: *ssubst-forget*)
  with *sn* **have** $SN(\ (t \star k)[x::=p]\ )$ **by** *simp*
  thus *?thesis* **by** (*rule sn-forget*)
qed

abbreviation
$redrtrans :: trm \Rightarrow trm \Rightarrow bool\ (\ - \mapsto^* -\ )$
where $redrtrans \equiv reduction\hat{}**$

To be able to handle the case where p makes a step, we need to establish $p \mapsto p' \Longrightarrow\ m[x::=p] \mapsto^* m[x::=p']$ as well as the fact that strong normalization is preserved for an arbitrary number of reduction steps. The first claim involves a number of simple transitivity lemmas. Here we can benefit from having removed the freshness conditions from the reduction relation as this allows all the cases to be proven automatically. Similarly, in the *red-subst* lemma, only those cases where substitution is pushed to two subterms needs to be proven explicitly.

lemma *red-trans*:
  shows *r1-trans*: $s \mapsto^* s' \Longrightarrow App\ s\ t \mapsto^* App\ s'\ t$
  and *r2-trans*: $t \mapsto^* t' \Longrightarrow App\ s\ t \mapsto^* App\ s\ t'$
  and *r4-trans*: $t \mapsto^* t' \Longrightarrow \Lambda\ x\ .\ t \mapsto^* \Lambda\ x\ .\ t'$
  and *r6-trans*: $s \mapsto^* s' \Longrightarrow s\ to\ x\ in\ t \mapsto^* s'\ to\ x\ in\ t$
  and *r7-trans*: $[\![\ t \mapsto^* t'\ ]\!] \Longrightarrow s\ to\ x\ in\ t \mapsto^* s\ \ to\ x\ in\ t'$
  and *r11-trans*: $s \mapsto^* s' \Longrightarrow [s] \mapsto^* ([s'])$
by − (*induct rule*: *rtranclp-induct* , (*auto intro*: *transitive-closurep-trans'*)[*2*])+

lemma *red-subst*: $p \mapsto p' \Longrightarrow (m[x::=p]) \mapsto^* (m[x::=p'])$
proof(*nominal-induct m avoiding*: *x p p' rule*:*trm.strong-induct*)
  case (*App s t*)
  hence $App\ (s[x::=p])\ (t[x::=p]) \mapsto^* App\ (s[x::=p'])\ (t[x::=p])$
    by (*auto intro*: *r1-trans*)
  also from *App* **have** $\ldots \mapsto^* App\ (s[x::=p'])\ (\ t[x::=p'])$
    by (*auto intro*: *r2-trans*)
  finally show *?case* **by** *auto*
next
  case (*To s y n*) **hence**
    $(s[x::=p])\ to\ y\ in\ (n[x::=p]) \mapsto^* (s[x::=p'])\ to\ y\ in\ (n[x::=p])$
    by (*auto intro*: *r6-trans*)
  also from *To* **have** $\ldots \mapsto^* (s[x::=p'])\ to\ y\ in\ (n[x::=p'])$
    by (*auto intro*: *r7-trans*)
  finally show *?case* **using** *To* **by** *auto*
qed (*auto intro*:*red-trans*)

lemma *SN-trans* : ⟦ $p \mapsto^* p'$; *SN p* ⟧ ⟹ *SN* p'
by (*induct rule*: *rtranclp-induct*) (*auto intro*: *SN-preserved*)

5.8.4 Central lemma

Now we have everything in place we need to tackle the central "Lemma 7" of [LS05]. The proof is quite long, but for the most part, the reasoning is that of [LS05].

lemma *to-RED-aux*:
assumes *p*: *SN p*
and *x*: $x \sharp p \quad x \sharp k$
and *npk*: *SN* $(n[x::=p] \star k)$
shows *SN* $(([p]\ to\ x\ in\ n) \star k)$
proof −

The first problem we need to handle is that the triple induction principle, like any induction rule, allows induction only on distinct variables. Hence, we need to introduce a new variable q. We later want to instantiate q with $n \star k$. Furthermore, we need to generalize the claim to arbitrary terms m, where $q = m \star k$. This is needed to handle reductions occuring in n.

{ fix *q* **assume** *SN q* **with** *p*
have $\bigwedge m\ .\ ⟦\ q = m \star k\ ;\ SN(m[x::=p] \star k)\ ⟧$
$\Longrightarrow$ *SN* $(([p]\ to\ x\ in\ m) \star k)$
using *x*
proof (*induct p q rule*:*triple-induct*[**where** *k*=*k*])
case (*1 p q k*) — We obtain an induction hypothesis for p, q, and k.
have *ih-p*:
$\bigwedge p'\ m\ .\ ⟦p \mapsto p';\ q = m \star k;\ SN\ (m[x::=p'] \star k);\ x \sharp p';\ x \sharp k⟧$
$\Longrightarrow$ *SN* $(([p']\ to\ x\ in\ m) \star k)$ **by** *fact*
have *ih-q*:
$\bigwedge q'\ m\ k\ .\ ⟦q \mapsto q';\ q' = m \star k;\ SN\ (m[x::=p] \star k);\ x \sharp p;\ x \sharp k⟧$
$\Longrightarrow$ *SN* $(([p]\ to\ x\ in\ m) \star k)$ **by** *fact*
have *ih-k*:
$\bigwedge k'\ m\ .\ ⟦\ |k'| < |k|;\ q = m \star k';\ SN\ (m[x::=p] \star k');\ x \sharp p;\ x \sharp k'⟧$
$\Longrightarrow$ *SN* $(([p]\ to\ x\ in\ m) \star k')$ **by** *fact*
have *q*: $q = m \star k$ **and** *sn*: *SN* $(m[x::=p] \star k)$ **by** *fact*+
have *xp*: $x \sharp p$ **and** *xk*: $x \sharp k$ **by** *fact*+

Once again we want to reason via case distinction on the successors of a term including a dismantling operator. Since this time we also need to handle the cases where interactions occur, we want to use the strengthened case rule. We already require x to be suitably fresh. To instantiate the rule, we need another fresh name.

{ fix *r* **assume** *red*: $([p]\ to\ x\ in\ m) \star k \mapsto r$
from *xp xk* **have** *x1* : $x \sharp ([p]\ to\ x\ in\ m) \star k$
by (*simp add*: *abs-fresh*)
with *red* **have** *x2*: $x \sharp r$ **by** (*rule reduction-fresh*)
obtain *z*::*name* **where** *z*: $z \sharp (x,p,m,k,r)$
using *ex-fresh*[*of* (x,p,m,k,r)] **by** (*auto simp add*: *fresh-prod*)

```
have SN r
proof (cases rule:dismantle-strong-cases
    [of   [p] to x in m   k r x x z ])
  case (5 r') have r: r = r' ⋆ k and r': [p] to x in m ↦ r' by fact+
```

To handle the case of a reduction occurring somewhere in $[p]$ *to x in m*, we need to contract the freshness conditions to this subterm. This allows the use of the strong inversion rule for the reduction relation.

```
from x1 x2 r
have xl:(x ♯ [p] to x in m) and xr:x ♯ r' by auto
from z have zl: z ♯ ([p] to x in m)   x ≠ z
  by (auto simp add: abs-fresh fresh-prod fresh-atm)
with r' have zr: z ♯ r'  by (blast intro:reduction-fresh)
— handle all reductions of [p] to x in m
from r' show SN r proof (cases rule:reduction.strong-cases
    [where x=x and xa=x and xb=x and xc=x and xd=x
     and xe=x and xf=xand xg=x and y=z])
```

The case where $p \mapsto p'$ is interesting, because it requires reasioning about the reflexive transitive closure of the reduction relation.

```
  case (r6 s s' t)  hence ch: [p] ↦ s'   r' = s' to x in m
    using xl xr by (auto)
  from this obtain p' where s: s' = [p'] and  p : p ↦ p'
    by (blast dest:red-Ret)
  from p have ((m⋆k)[x::=p]) ↦* ((m⋆k)[x::=p'])
    by (rule red-subst)
  with xk have ((m[x::=p]) ⋆ k) ↦* ((m[x::=p']) ⋆ k)
    by (simp add: ssubst-forget)
  hence sn: SN ((m[x::=p']) ⋆ k) using sn by (rule SN-trans)
  from p xp have xp' : x ♯ p' by (rule reduction-fresh)
  from ch s have rr: r' = [p'] to x in m by simp
  from p q  sn xp' xk
  show SN r unfolding r rr by (rule ih-p)
next

  case(r7 s t m') hence r' = [p] to x in m' and m ↦ m'
    using xl xr by (auto simp add: alpha)
  hence rr: r' = [p] to x in m' by simp
  from q ‹m ↦ m'› have  q ↦ m' ⋆ k by(simp add: dismantle-red)
  moreover have m' ⋆ k = m' ⋆ k .. — a triviality
  moreover { from ‹m ↦ m'› have (m[x::=p]) ⋆ k ↦ (m'[x::=p]) ⋆ k
     by (simp add: dismantle-red reduction-subst)
    with sn have SN(m'[x::=p] ⋆ k) .. }
  ultimately show SN r using xp xk unfolding r rr by (rule ih-q)
next

  case (r8 s t) — the β-case is handled by assumption
  hence r' = m[x::=p] using xl xr by(auto simp add: alpha)
  thus SN r unfolding r using sn by simp
next
```

```
      case (r9 s) — the η-case is handled by assumption as well
      hence m = [Var x] and r' = [p] using xl xr by(auto simp add: alpha)
      hence r' = m[x::=p] by simp
      thus SN r unfolding r using sn by simp
    qed (simp-all only: xr xl zl zr abs-fresh , auto)
    — There are no other possible reductions of [p] to x in m.
  next

    case (6 k')
    have k: k ↦ k' and r: r = ([p] to x in m) ⋆ k' by fact+
    from q k have q ↦ m ⋆ k' unfolding stack-reduction-def by blast
    moreover have m ⋆ k' = m ⋆ k' ..
    moreover { have SN (m[x::=p] ⋆ k) by fact
      moreover have (m[x::=p]) ⋆ k ↦ (m[x::=p]) ⋆ k'
        using k unfolding stack-reduction-def ..
      ultimately have SN (m[x::=p] ⋆ k') .. }
    moreover note xp
    moreover from k xk have x ♯ k'
      by (rule stack-reduction-fresh)
    ultimately show SN r unfolding r by (rule ih-q)
  next
```

The case of an assoc interaction between $[p]$ *to* x *in* m and k is easily handled by the induction hypothesis, since $m[x::=p] \star k$ remains fixed under assoc.

```
    case (8 s t u L)
    hence k: k = [z]u≫L
      and r: r = ([p] to x in (m to z in u)) ⋆ L
      and u: x ♯ u
      by(auto simp add: alpha fresh-prod)
    let ?k = L and ?m = m to z in u
    from k z have |?k| < |k| by (simp add: fresh-prod)
    moreover have q =  ?m ⋆ ?k using k q by simp
    moreover { from k u z xp have (?m[x::=p] ⋆ ?k) = (m[x::=p]) ⋆ k
      by(simp add: fresh-prod forget)
    hence SN (?m[x::=p] ⋆ ?k) using sn by simp }
    moreover from xp xk k have x ♯ p and x ♯ ?k by auto
    ultimately show SN r unfolding r by (rule ih-k)
  qed (insert red z x1 x2 xp xk ,
      auto simp add: fresh-prod fresh-atm abs-fresh)
 } thus SN ((([p] to x in m) ⋆ k) ..
qed }
moreover have SN ((n[x::=p]) ⋆ k) by fact
moreover hence SN (n ⋆ k) using ‹x ♯ k› by (rule sn-forget')
ultimately show ?thesis by blast
qed
```

Having established the claim above, we use it show that to-bindings preserve reducibility.

```
lemma to-RED:
  assumes s: s ∈ RED (T σ)
  and t : ∀ p ∈ RED σ . t[x::=p] ∈ RED (T τ)
  shows s to x in t ∈ RED (T τ)
proof −
  { fix K assume k: K ∈ SRED τ
    { fix p assume p: p ∈ RED σ
      hence snp: SN p using RED-props by(simp add: CR1-def)
      obtain x'::name where x: x' ♯ (t, p, K)
        using ex-fresh[of (t,p,K)] by (auto)
      from p t k have SN((t[x::=p]) ⋆ K) by auto
      with x have SN ((([(x',x)] ∙ t )[x'::=p]) ⋆ K)
        by (simp add: fresh-prod subst-rename)
      with snp x  have snx': SN (([p] to x' in ([(x',x)] ∙ t )) ⋆ K)
        by (auto intro: to-RED-aux)
      from x have [p] to x' in ([(x',x)] ∙ t ) = [p] to x in t
        by simp (metis alpha' fresh-prod name-swap-bij x)
      moreover have ([p] to x in t) ⋆ K  = [p] ⋆ [x]t≫K by simp
      ultimately have snx: SN([p] ⋆ [x]t≫K) using snx'
        by (simp del: trm.inject)
    } hence [x]t≫K ∈ SRED σ by simp
    with s have SN((s to x in t) ⋆ K) by(auto simp del: SRED.simps)
  } thus s to x in t ∈ RED (T τ) by simp
qed
```

5.9 Fundamental Theorem

The remainder of this section follows [Nom] very closely. We first establish that all well typed terms are reducible if we substitute reducible terms for the free variables.

```
abbreviation
 mapsto :: (name×trm) list ⇒ name ⇒ trm ⇒ bool (- maps - to - [55,55,55] 55)
where
 θ maps x to e ≡ (lookup θ x) = e

abbreviation
  closes :: (name×trm) list ⇒ (name×ty) list ⇒ bool (- closes - [55,55] 55)
where
  θ closes Γ ≡ ∀x τ. ((x,τ) ∈ set Γ ⟶ (∃t. θ maps x to t ∧ t ∈ RED τ))

theorem fundamental-theorem:
  assumes a: Γ ⊢ t : τ   and   b: θ closes Γ
  shows θ<t> ∈ RED τ
using a b
proof(nominal-induct  avoiding: θ rule: typing.strong-induct)
  case (t3 a Γ σ t τ θ) — lambda case
  ⟨Urban⟩
```

```
next
  case (t5 x Γ s σ t τ θ) — to case
  have ihs : ⋀ θ . θ closes Γ ⟹ θ<s> ∈ RED (T σ) by fact
  have iht : ⋀ θ . θ closes ((x, σ) # Γ) ⟹ θ<t> ∈ RED (T τ) by fact
  have θ-cond: θ closes Γ by fact
  have fresh: x ♯ θ  x ♯ Γ  x ♯ s by fact+
  from ihs have θ<s> ∈ RED (T σ) using θ-cond by simp
  moreover
  { from iht have ∀ s∈RED σ. ((x,s)#θ)<t> ∈ RED (T τ)
      using fresh θ-cond fresh-context by simp
    hence ∀ s∈RED σ. θ<t>[x::=s] ∈ RED (T τ)
      using fresh by (simp add: psubst-subst) }
  ultimately have (θ<s>) to x in (θ<t>) ∈ RED (T τ) by (simp only: to-RED)
  thus θ<s to x in t> ∈ RED (T τ) using fresh by simp
qed auto — all other cases are trivial
```

The final result then follows using the identity substitution, which is Γ-closing since all variables are reducible at any type. This technique is standard for logical relations proofs.

```
fun
  id :: (name×ty) list ⇒ (name×trm) list
where
  id []    = []
| id ((x,τ)#Γ) = (x,Var x)#(id Γ)

lemma id-maps:
  shows (id Γ) maps a to (Var a)
by (induct Γ) (auto)

lemma id-fresh:
  fixes x::name
  assumes x: x ♯ Γ
  shows x ♯ (id Γ)
using x
by (induct Γ) (auto simp add: fresh-list-nil fresh-list-cons)

lemma id-apply:
  shows (id Γ)<t> = t
by (nominal-induct t avoiding: Γ rule: trm.strong-induct)
   (auto simp add: id-maps id-fresh)

lemma id-closes:
  shows (id Γ) closes Γ
proof −
  { fix x τ assume (x,τ) ∈ set Γ
    have CR4 τ by(simp add: RED-props CR3-implies-CR4)
    hence Var x ∈ RED τ
      by(auto simp add: NEUT-def normal-var CR4-def)
```

```
    hence (id Γ) maps x to Var x ∧ Var x ∈ RED τ
      by (simp add: id-maps)
  } thus ?thesis by blast
qed
```

5.9.1 Strong normalization theorem

```
lemma typing-implies-RED:
  assumes a: Γ ⊢ t : τ
  shows t ∈ RED τ
proof −
  have (id Γ)<t>∈RED τ
  proof −
    have (id Γ) closes Γ by (rule id-closes)
    with a show ?thesis by (rule fundamental-theorem)
  qed
  thust ∈ RED τ by (simp add: id-apply)
qed

theorem strong-normalization:
  assumes a: Γ ⊢ t : τ
  shows SN(t)
proof −
  from a have t ∈ RED τ by (rule typing-implies-RED)
  moreover have CR1 τ by (rule RED-props)
  ultimately show SN(t) by (simp add: CR1-def)
qed
```

This finishes our formalization effort. The last theorem corresponds directly to Theorem 2.2.1, the theorem we set out to prove in Chapter 2. As noted initially, this whole chapter is generated from the Isabelle theory file, which consists of roughly 1500 lines of proof code. The reader is invited to replay some of the more technical proofs using the theory file provided at the authors website.[2] A shorter proof script might have been obtained using simple **apply** scripts instead of Isar structured proofs, but this would have largely obscured the proof.

[2] http://www.ps.uni-sb.de/~doczkal/master/

CHAPTER 6

Evaluation

In the first part of this chapter, we assess how much the nominal package helps in keeping the formalization in Chapter 5 close to the pen-and-paper version. This is followed by a discussion of the trusted base. In Section 6.3, we briefly discuss some alternatives to nominal reasoning in Isabelle. We conclude our evaluation by pointing out two restrictions of the Nominal package that, we believe, warrant further investigation.

6.1 How Faithful is the Formalization

On of the main aspects of the Isabelle/Isar is the creation of formal proof documents, documents that are both, human readable and machine checkable. In this section, we want to evaluate how faithful our formalization is to the (well readable) original proof.

6.1.1 Calculus and basic properties

Using nominal datatypes and some syntax annotations, terms and types can be represented very close to the pen-and-paper version, but as motivated in Section 4.5.1, we had to move to a domain-free version of λ_{ml}. Since we started the formalization of the calculus from scratch, all the simple properties of the calculus, notably those of substitution, need to be proven explicitly. Here one can see one of the strengths of the Nominal package. Substitution is easily defined using the provided primitive recursion combinator, and all the properties of substitution have simple inductive proofs, using only *nominal-induct* and *auto*. This is possible because of the strong induction principles, derived by the nominal package.

In the case of the reduction relation, we need to impose more freshness conditions than the pen-and-paper style presentation. These freshness conditions are needed purely for the technical benefit of automatically deriving strong reasoning principles. This is unfortunate, since the reduction relation is, as we have shown, compatible with reasoning using the variable convention. However, this can not be established automatically, since the relation is not vc-compatible.

6.1.2 Dismantling and the case analysis

The nominal package offers good support for defining primitive recursive functions over α-terms, but support for non-primitive recursion is currently very limited. The Isabelle function package only works on nominal datatypes which do not involve binders and therefore behave like regular datatypes having a permutation action associated with them. Defining non-primitive recursive functions, like the dismantling function, requires explicit proofs of pattern completeness, right-uniqueness and termination. These properties are usually taken for granted in informal proofs. Similarly, we had to invest substantial work to derive the case distinction on reductions of the form $t \star k \mapsto r$. In this particular case, formalizing the central reasoning principle requires about as much work as its applications.

In this context, we note that the preferred nominal reasoning style, avoiding explicit alpha renaming, requires collecting the various freshness conditions during the course of reasoning and satisfying them using some strong induction rule. While this overhead of collecting freshness conditions is negligible for small proofs, it can become a real burden in proofs that are already lengthy and complex. In the main lemmas *dismantle-cases* and *to-RED-aux*, this problem is alleviated to some extent by having shown adequacy of the reduction relation, removing many of the freshness requirements.

6.1.3 Deviations

Apart from working in a domain-free setting, there are two other aspects, where our formalization differs significantly from [LS05]. First, the strong normalization theorem shown in Chapter 5 relies on a formalization of strong normalization differing from the one Lindley and Stark use. As can be seen, for example by comparing Figure 5.4 with the corresponding case of *RED-props*, the differing characterizations have, in most situations, little influence on the course of reasoning.

There is one exception, which is the only point where the argument in Chapter 5 deviates significantly from [LS05]. Lindley and Stark prove their central Lemma 7 (Figure 5.5) by induction on $|k| + max(n \star k) + max(p)$. To be able to apply the induction hypothesis in the case where k makes a step,

this requires establishing the fact that stack reduction does not increase the stack length. Since our notion of strong normalization does not directly provide bounds on the length of reduction sequences, this could not be formalized directly. Hence, in Section 5.8.4, we use the stronger *triple-induct* principle, allowing us to handle the case of a reduction $k \mapsto k'$ by using the induction hypothesis regardless of the length of k'. This simplification naturally also applies to the pen-and-paper version of the proof, making redundant the stack length lemma and its rather informal proof in [LS05].

6.2 Trusted Base

In formal proofs it is reasonable to explicitly state ones trusted base. Here, the trusted base was kept as small as possible. It mainly includes consistency of HOL and correctness of (the proof checking part of) its Isabelle implementation. HOL-Nominal is only a definitional extension of HOL, so it does not increase the trusted base. Beyond that, we have to trust that the definitions and theorems stated in the logic actually have the intended meaning. Due to our use of nominal logic, which allows theorems to be stated exactly as one would do on paper, this *coding gap* is minimal. Using a vc-compatible version of the reduction relation does not extend the trusted base, since we have proven adequacy with respect to the version presented in Section 2.1. Likewise for the nonstandard definition of strong normalization, which is shown equivalent to the prevalent definition in terms of infinite sequences.

6.2.1 Stack reductions and variables

In this context, we mention the one minor inaccuracy we found in [LS05]. Lindley and Stark state the definition of stack reductions as follows:

$$\begin{array}{rcl} k \mapsto k' & \equiv & \forall t.\ t \star k \mapsto t \star k' \\ & \overset{!}{\Longleftrightarrow} & x \star k \mapsto x \star k' \end{array}$$

for *any* variable x, claiming that the equivalence above follows since reduction is preserved under substitution. While the "$\Longrightarrow$" direction holds trivially, the "$\Longleftarrow$" direction only follows from the preservation of reduction under substitution if we have both $x \sharp k$ and $x \sharp k'$. It is possible to drop the second freshness condition via case analysis on the reduction relation. While the proposed proof fails, the claim still appears reasonable, since the reduction relation does not depend on any term being an unbound variable.

In Chapter 5, we only need the trivial direction of the above equivalence, but this hinges on the adequacy proof of the reduction relation. Aydemir et al. [ACP+08] note that one of the drawbacks of the nominal approach, as

opposed to the locally nameless approach (see below), is that the freshness contexts for the variables has to be provided upon *instantiation* of a strong case/induction rule r. This means there is no easy way to ensure freshness of the names stemming from the various cases of r for terms introduced later on. However, fixing arbitrary terms is required for the introduction of stack reductions, if using the definition directly. This provided the initial motivation for the adequacy proof, eliminating freshness restrictions on those rules involved in showing stack reductions. On the other hand, obtaining fresh variables in the middle of a proof is easy. So the above equivalence, even the resticted version, requiring freshness for k and k', could also be used in these cases. This would allow skipping the adequacy proof, adding the trivial, but tedious to obtain, adequacy result to the trusted base.

6.3 Related Work

Nominal reasoning is certainly not the only way to formalize reasoning about inductively defined data structures incorporating binders. Here, we point out two alternatives, both requiring less metatheory than the nominal approach, but at the expense of a significantly larger coding gap.

6.3.1 HOL-Nominal vs. Locally Nameless

Aydemir et al. [ACP+08] develop a lightweight alternative to nominal reasoning. One part of this theme is to use a *locally nameless* representation [MP99, Ler07] of the λ-calculus, which, like the De Bruijn representation [Bru72], has a single representative for every alpha equivalence class. The locally nameless approach uses variables only when they occur free and uses De Bruijn indices for bound variables. Unlike nominal datatypes, where every element of the datatype corresponds to some λ_{ml} term, the locally nameless representation requires an extra predicate to check whether the term is *locally closed*, meaning that all occurring indices are bound by some lambda. If one wants to reason about an abstraction, one can *open* the body of the abstraction with some suitable variable, writing t^x for the body of λt opened with x.

In this context, the authors propose *cofinite quantification* to reason about inductively defined relations involving binders. Instead of specifying rules using the standard (existential) presentation of rules, one employs (cofinitely quantified) rules which are only applicable if the premise holds for all variables not in L, for some finite set L.

$$\frac{x \notin fv(t) \qquad \Gamma, x : \sigma \vdash t^x : \tau}{\Gamma \vdash \lambda t : \sigma \to \tau}\textsc{Exists} \qquad \frac{\forall x \notin L.\ \Gamma, x : \sigma \vdash t^x : \tau}{\Gamma \vdash \lambda t : \sigma \to \tau}\textsc{Cofinite}$$

Proving adequacy of the cofinite version can be difficult, but the cofinite

version provides very strong induction principles. While inverting the existential rule proves the premise only for some specific name, the cofinite rule provides an infinite supply of names for which the premise holds.

Another advantage of the locally nameless approach is that it requires very little infrastructure. Aydemir et al. provide an implementation of the required metatheory in Coq. In the current state of development/documentation, it requires, however, substantial experience with the Coq proof assistant. The technique does not rely on any particular feature of Coq, hence, the approach could be applied just as well to the Isabelle/HOL setting, but this would require reimplementing the metatheory in Isabelle.

6.3.2 Structural Logical Relations in Twelf

We initially noted that Twelf has a relatively weak meta-logic. This precludes a direct definition of reducibility in the style of Section 5.5, but this does not mean that it is entirely impossible to do logical relations proofs in the Twelf framewok. Using *Structural Logical Relations* [SS08] one can also do logical relation proofs in Twelf. The idea is to explicitly represent reasoning about the logical relation in an auxiliary assertion logic. Due to the weak meta-logic of Twelf, this approach requires to assume termination of cut-elimination for this auxiliary logic, because the corresponding algorithm does not termination check in Twelf. Using this approach, Sarnat and Schürmann have even formalized a proof of weak normalization to β-short η-long normal forms for the computational meta-language[1], also following [LS05] in the use of $\top\top$-lifting.

Sarnat and Schürmann handle the problem of alpha equivalence using higher-order abstract syntax, where abstractions are represented as functions from terms to terms. In contrast, the implementation of nominal logic uses weak higher-order abstract syntax, representing abstraction functions as functions of type $name \Rightarrow trm\ option$, as explained in Section 4.5.2.

6.4 Future Research Directions

There are a number of issues that came up while doing the formalization, some of these may warrant further investigation.

6.4.1 Inductively defined relations

In [UBN07] Urban et al. describe conditions which allow the induction principle for inductively defined relations to be strengthened automatically to

[1] www.twelf.org/slr

include a variant of the Barendregt Variable Convention. The most important restriction is that any variable occurring in binding position must be fresh for the conclusion of the rule. Unfortunately, this condition requires the corresponding relations to be stated with additional freshness conditions. As shown in Section 5.2, for some relations these additional freshness conditions (as one would hope) do not change the relation being defined. This implies that for these relations the freshness conditions are semantically not necessary.

One particularly easy case are variables which occur *only* in binding position and all occurrences of the binder in the rule have the same scope. Here one can always establish the required freshness for the conclusion of the rule via alpha renaming. The proofs for *r3*, *r6*, *r7*, *r8*, and *r9* (pages 36-39) all largely follow one of two patterns, depending on whether the reduction occurs underneath the binder or in another subterm. So there is the hope to refine the notion of vc-compatibility or at least automate these proofs. However, the cases where one binder appears in the scope of another – as in the assoc rule *r10* – are rather tedious by hand, and removing these freshness conditions automatically appears to be difficult.

6.4.2 Functions

Another issue worthwile investigating is to add some support for non-primitive recursion involving binders. There seems to be no theoretical work on this subject so far. On the other hand, the right uniqueness proof for the dismantling function only depends on the fact that the function maps α-equivalent arguments to recursive calls on α-equivalent arguments. A first step may be to generalize this to arbitrary tail recursive functions satisfying the invariant above.

BIBLIOGRAPHY

[Abe04] Andreas Abel. Normalization for the Simply Typed λ-calculus in Twelf. In *LFM'04: Fourth International Workshop on Logical Frameworks and Meta-Languages, Informal Proceedings*, 2004.

[ABF+05] Brian E. Aydemir, Aaron Bohannon, Matthew Fairbairn, J. Nathan Foster, Benjamin C. Pierce, Peter Sewell, Dimitrios Vytiniotis, Geoffrey Washburn, Stephanie Weirich, and Steve Zdancewic. Mechanized metatheory for the masses: The poplmark challenge, 2005. http://www.cis.upenn.edu/~plclub/wiki-static/poplmark.pdf.

[ACP+08] Brian Aydemir, Arthur Charguéraud, Benjamin C. Pierce, Randy Pollack, and Stephanie Weirich. Engineering formal metatheory. In *POPL '08: Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 3–15, New York, NY, USA, 2008. ACM.

[Alt93] Thorsten Altenkirch. A formalization of the strong normalization proof for System F in LEGO. In *TLCA*, volume 664 of *Lecture Notes in Computer Science*, pages 13–28. Springer-Verlag, 1993.

[Bar85] Hendrik Pieter Barendregt. *The lambda calculus*, volume 103 of *Studies in logic and the foundations of mathematics*. Elsevier/North-Holland, rev. ed., 2nd printing 1985 edition, 1985.

[BBDP98] P. N. Benton, G. M. Bierman, and V. C. V. De Paiva. Computational types from a logical perspective. *J. Funct. Program.*, 8(2):177–193, 1998.

[BBLS06] Ulrich Berger, Stefan Berghofer, Pierre Letouzey, and Helmut Schwichtenberg. Program extraction from normalization proofs. *Stud. Log.*, 82(1):25–49, 2006.

[Bru72] N. G. De Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the church-rosser theorem. *Indag. Math*, 34:381–392, 1972.

[BS00] Gilles Barthe and Morten Heine Sørensen. Domain-free pure type systems. *J. Funct. Program.*, 10(5):417–452, 2000.

[BU08] Stefan Berghofer and Christian Urban. Nominal Inversion Principles. In *TPHOLs '08: Proceedings of the 21st International Conference on Theorem Proving in Higher Order Logics*, volume 5170 of *Lecture Notes in Computer Science*, pages 71–85, Berlin, Heidelberg, 2008. Springer-Verlag.

[CH88] Thierry Coquand and Gerard Huet. The calculus of constructions. *Inf. Comput.*, 76(2-3):95–120, 1988.

[CP90] Thierry Coquand and Christine Paulin. Inductively defined types. In *COLOG '88: Proceedings of the International Conference on Computer Logic*, pages 50–66, London, UK, 1990. Springer-Verlag.

[Doc07] Christian Doczkal. Strong normalisation of call-by-push-value. B.Sc. thesis, Universität des Saarlandes, 2007.

[DX07] Kevin Donnelly and Hongwei Xi. A formalization of strong normalization for simply-typed lambda-calculus and System F. *Electr. Notes Theor. Comput. Sci.*, 174(5):109–125, 2007.

[Gab02] Murdoch J. Gabbay. FM-HOL, a higher-order theory of names. In F. Kamareddine, editor, *35 Years of Automath*. Heriot-Watt University, Edinburgh, Scotland, April 2002.

[GTL89] Jean-Yves Girard, Paul Taylor, and Yves Lafont. *Proofs and Types*. Cambridge University Press, New York, NY, USA, 1989.

[HHP93] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *J. ACM*, 40(1):143–184, 1993.

[Hue75] Gérard P. Huet. A unification algorithm for typed lambda-calculus. *Theor. Comput. Sci.*, 1(1):27–57, 1975.

[Jec71] Thomas J. Jech. Fraenkel-Mostowski models. In *Lectures in Set Theory with Particular Emphasis on the Method of Forcing*, volume 217 of *Lecture Notes in Mathematics*, pages 122–125. Springer, 1971.

[Kőn26] Dénes Kőnig. Sur les correspondances multivoques des ensembles. *Fund. Math.*, 8:114–134, 1926.

[Ler07] Xavier Leroy. A locally nameless solution to the poplmark challenge. 2007. http://hal.inria.fr/inria-00123945/en/.

[Lev99] Paul Blain Levy. Call-by-push-value: A subsuming paradigm. In *TLCA*, volume 1581 of *Lecture Notes in Computer Science*, pages 228–242. Springer-Verlag, 1999.

[LS05] Samuel Lindley and Ian Stark. Reducibility and $\top\top$-lifting for Computation Types. In *Proceedings of Typed Lambda Calculi and Applications (TLCA '05)*, volume 3461 of *Lecture Notes in Computer Science*, pages 262–277. Springer, Apr 2005.

[Mog91] Eugenio Moggi. Notions of computation and monads. *Inf. Comput.*, 93(1):55–92, 1991.

[MP99] James Mckinna and Robert Pollack. Some lambda calculus and type theory formalized. *J. Autom. Reason.*, 23:373–409, 1999.

[Nip] Tobias Nipkow. *A Tutorial Introduction to Structured Isar Proofs*. http://isabelle.in.tum.de/dist/Isabelle/doc/isar-overview.pdf.

[Nom] Nominal Methods Group. Strong normalisation proof from the proofs and types book. http://isabelle.in.tum.de/dist/library/HOL/HOL-Nominal/Examples/SN.html.

[NPW09] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *A Proof Assistant for Higher-Order Logic*, April 2009. http://isabelle.in.tum.de/dist/Isabelle/doc/tutorial.pdf.

[Pau89] L. C. Paulson. The foundation of a generic theorem prover. *J. Autom. Reason.*, 5(3):363–397, 1989.

[Pit03] Andrew M. Pitts. Nominal logic, a first order theory of names and binding. *Inf. Comput.*, 186(2):165–193, 2003.

[Pit06] Andrew M. Pitts. Alpha-structural recursion and induction. *J. ACM*, 53(3):459–506, 2006.

[PN94] L. C. Paulson and T. Nipkow. *Isabelle: A Generic Theorem Prover*, volume 828/1994 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 1994.

[Rey74] John C. Reynolds. Towards a theory of type structure. In *Programming Symposium, Proceedings Colloque sur la Programmation*, pages 408–423, London, UK, 1974. Springer-Verlag.

[SS08] Carsten Schürmann and Jeffrey Sarnat. Structural Logical Relations. In *LICS '08: Proceedings of the 2008 23rd Annual IEEE Symposium on Logic in Computer Science*, pages 69–80, Washington, DC, USA, 2008. IEEE Computer Society.

[UBN07] Christian Urban, Stefan Berghofer, and Michael Norrish. Barendregts variable convention in rule inductions. In *CADE '07: Proceedings of the 21th International Conference on Automated Deduction*, volume 4603 of *Lecture Notes in Computer Science*, pages 35–50. Springer, 2007.

[Urb08] Christian Urban. Nominal Techniques in Isabelle/HOL. *J. Autom. Reason.*, 40(4):327–356, 2008.

[Wen02] Markus M. Wenzel. *Isabelle/Isar – a versatile environment for human-readable formal proof documents*. PhD thesis, Technische Universiät München, 2002. Revised version to cover Isabelle 2002.

[Wen08] Makarius Wenzel. *The Isabelle/Isar Reference Manual*, June 2008. http://isabelle.in.tum.de/dist/Isabelle/doc/isar-ref.pdf.