Department of Mathematics and Computer Science
Institute of Computer Science

Efficient Data Structures for Automated Theorem Proving in Expressive Higher-Order Logics

submitted by
Alexander Steen
a.steen@fu-berlin.de

in partial fulfillment of the requirements
for the degree of
Master of Science

Submission date: 17.10.2014
Supervisor: PD Dr. Christoph Benzmüller
Second examiner: Prof. Dr. Marcel Kyas

This page is left blank until stated otherwise

Declaration of authorship

I hereby certify, that I have written this thesis entirely on my own and that I have not used any other materials than the ones referred to. This thesis or parts of it have not been submitted for a degree at this or any other university.

Berlin, ____________________

____
Alexander Steen

Abstract

Church's Simple Theory of Types (STT), also referred to as classical higher-order logic, is an elegant and expressive formal system built on top of the simply typed λ-calculus. Its mechanisms of explicit binding and quantification over arbitrary sets and functions allow the representation of complex mathematical concepts and formulae in a concise and unambiguous manner. Higher-order automated theorem proving (ATP) has recently made major progress and several sophisticated ATP systems for higher-order logic have been developed, including Satallax, Isabelle/HOL and LEO-II. Still, higher-order theorem proving is not as mature as its first-order counterpart, and robust implementation techniques for efficient data structures are scarce.

In this thesis, a higher-order term representation based upon the polymorphically typed λ-calculus is presented. This term representation employs spine notation, explicit substitutions and perfect term sharing for efficient term traversal, fast β-normalization and reuse of already constructed terms, respectively. An evaluation of the term representation is performed on the basis of a heterogeneous benchmark set. It shows that while the presented term data structure performs quite well in general, the normalization results indicate that a context dependent choice of reduction strategies is beneficial.

A term indexing data structure for fast term retrieval based on various low-level criteria is presented and discussed. It supports symbol-based term retrieval, indexing of terms via structural properties, and subterm indexing.

Contents

1 Introduction **1**

2 Preliminaries **5**
2.1 The λ-Calculus . . . 5
2.1.1 A nameless representation . . . 8
2.2 Typed Variants of the Classical λ-Calculus . . . 10
2.2.1 The $\lambda\rightarrow$-Calculus . . . 10
2.2.2 Extensions of Typed λ-Calculi . . . 13
2.3 Higher-Order Logic . . . 14
2.4 Automatic Theorem Proving . . . 15

3 The Term Language for Leo-III **19**

4 Internal Term Representation **24**
4.1 Terms in Spine Notation . . . 26
4.2 Explicit Substitutions . . . 29
4.3 Combining Spines and Explicit Substitutions . . . 32
4.4 Shared Representation of Terms . . . 38
4.5 Evaluation of Term Representation . . . 41
4.6 Term Orderings . . . 47

5 Term Indexing **49**
5.1 Indexing in Automated Theorem Proving . . . 50
5.2 Term Indexing of Leo-III . . . 51
5.2.1 Term structure indexing . . . 52
5.2.2 Symbol based indexing . . . 53
5.2.3 Subterm indexing . . . 54
5.2.4 Bound variable indexing . . . 55
5.3 Summary and Preliminary Evaluation . . . 56

6 Further Data Structures **58**
6.1 The Signature . . . 58
6.2 Literals and Clauses . . . 59
6.3 Input preprocessing . . . 60

7 Conclusion **61**
7.1 Related Work . . . 62
7.2 Further Work . . . 63

8 Bibliography **64**

List of Figures

1.1 Layered architecture of data structures . . . 2
2.1 Typing rules for $\lambda\!\rightarrow$ terms . . . 12
2.2 Barendregt's λ-cube . . . 13
2.3 Simplistic view on automated theorem proving . . . 16
3.1 Typing rules for System F terms . . . 21
4.1 Terms of $\lambda\!\rightarrow$ in spine notation . . . 27
4.2 Term graph example in Spine Notation . . . 28
4.3 Explicit substitutions of $\lambda\sigma$ . . . 31
4.4 β-normalization strategy **SpClos** . . . 36
4.5 Normalization benchmark of domain S4 . . . 44
4.6 Normalization benchmark of domain Church I . . . 45
4.7 Leo-III's class architecture for native support of term orderings . . . 47
5.1 $n : 1$ indexing and $n : m$ indexing . . . 50
5.2 Outline of the term insertion process into Leo-III's term index . . . 52
5.3 Simplified example of scope numbers for bound variable indexing . . . 56

List of Tables

1 Reduction count measurement of β-reductions . . . 43
2 Time measurement of β-reductions in μs . . . 43
3 Reduction count measurement of head-symbol query . . . 46
4 Results of term sharing measurements . . . 47

1 Introduction

Church's Simple Theory of Types (STT) [Chu40], also referred to as classical higher-order logic, is an elegant and expressive formal system built on top of the simply typed λ-calculus. Its mechanisms of explicit binding and quantification over arbitrary sets and functions allow the representation of complex mathematical concepts and formulae in a concise and unambigious manner [Far07]. Unfortunately, as a consequence of Gödel's Incompleteness Theorem [Gö31], all proof calculi for full STT are necessarily incomplete, thus eliminating the hope of effective automation. The research of Leon Henkin, however, led to the discovery of a generalized notion of semantics for STT in which completeness can be achieved, also called Henkin semantics [Hen50]. Today, a number of complete proof calculi – with respect to Henkin semantics – exist for higher-order logic, among them several well-suited for automation.

The expressivity of higher-order logic is not only exploited by mathematicians or logicians, but also in the field of formal methods in computer science and engineering (e.g. for software and hardware verification) and in computer linguistics. A more recent and innovative field of study is the application of expressive logics in theoretical philosophy and metaphysics [FZ07]. Classical higher-order logic can in fact be used to simulate a large number of expressive logics using a semantic embedding approach [Ben10]. Hence, efficient automation of higher-order logic also permits automation results for practical relevant logics in philosophy and semantic systems, such as modal logics or conditional logics [BR13, BGGR12], where commonly no special-purpose automated theorem provers are availabe.

Motivation of the Thesis Higher-order automated theorem proving (ATP) has recently made major progress and several sophisticated ATP systems for higher-order logic have been developed, including Satallax [Bro12], Isabelle/HOL [NWP02] and LEO-II [BPTF08]. Still, higher-order theorem proving is not as mature as its first-order counterpart, and robust implementation techniques for efficient data structures are scarce. Of course, the effectivity of an automated theorem prover crucially depends on its proof calculus. Nevertheless, data structure choices are a critical part of a theorem prover and permit reliable increases of overall performance, when implemented properly. Key aspects for efficient theorem proving have been an intensive research topic [Ria03] and have reached a maturity with respect to first-order theorem proving. The employment of reasonable term indexing is, in particular, a key feature of state-of-the-art theorem provers. For the first-order case, several robust term-indexing techniques exist. As for its higher-order counterpart, only a few term indexing techniques are developed.

This thesis' aim is to take a first step towards gathering a complete set of efficient data structures for higher-order automated theorem proving. The data structure base of a theorem proving system can be seen as a layered architecture which is displayed in Figure 1.1.
The most fundamental data structure layer contains those who can generically be ap-

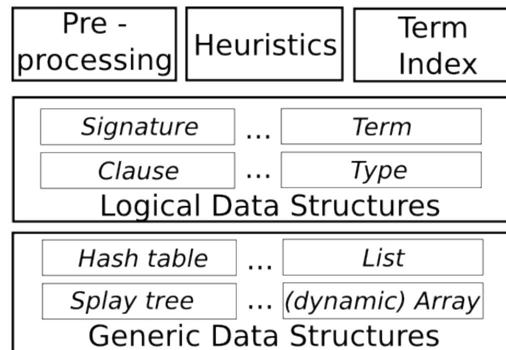


Figure 1.1: Layered architecture of data structures

plied in different contexts, including all common structures such as Hash tables, (self-balancing) search trees, heaps and arrays. The layer of logical structures contains problem-specific data structures with importance for automated reasoning (e.g. terms, clauses, and many more). On top of that, utility structures like the term index or the preprocessing structure offer advanced functionality for logical data structures.

This thesis discusses the following logical and utility data structures for automated theorem proving in a more or less detailed manner:

Term The core data structure that internally represents λ-terms. All common manipulation operations, such as unification or matching tests, are performed on this structure.

Type In a higher-order setting, types have to be managed in a well-structured way. A data structure for types not only support the creation of various types but also allows for efficient instantiation (of polymorphic types).

Term Indexing Term indexing data structures serve as local databases that allow efficient retrieval of terms with certain properties.

Literal, Clause Special term representation that is crucial during resolution-based proof procedures. Literals and clauses often employ various heuristics for choosing the approximately "most productive" ones for a certain resolution step.

Signature A central data structure that manages the logical symbols that are currently used along with their type and possibly their definition. The signature also holds information about the source of certain terms (e.g. introduced during skolemization) and provides mechanism for creating fresh uninterpreted symbols.

Preprocessing The task of preprocessing input problems can also be viewed as a responsibility of an independent data structure. Preprocessing steps could, for instance, include simplification, mini scoping, relevance filtering and normalization.

In this thesis, we heavily focus on the representation of terms as efficient underlying implementation. To that end, a brief overview of requirements is discussed and, consequently, several techniques are introduced that tackle problems of classical λ-term representations with respect to that requirements. A further data structure, the term index, is discussed more thoroughly. The described approach allows simple indexing for fast retrieval of terms with respect to relatively low-level queries. The remaining data structures will be described less intensively but include a sketch of important properties a certain data structure should have to effectively fit in the whole package.
Terms are the most general and common pieces of information that are accessed, manipulated and created by a several routines inside the system. It is therefore not surprising, that the internal representation of terms is a crucial detail and has direct consequences on the efficiency of the whole system. For that reason, we conjecture that substantial performance gains, and thus a more effective theorem proving system, are achieved when intensively focusing on sophisticated term representation techniques together with suitable term indexing mechanism.

This thesis contributes an explicit practical description of the combination of perfect term sharing with spine notation [CP03] and explicit substitutions [ACCL90] lifted to the polymorphically typed case. Additionally, two normalization strategies are described and evaluated in practical applications. Some low-level indexing techniques that were deployed in the LEO-II prover [TB06] are adapted and adjusted for the spine representation case.

The Leo-III Project This thesis is motivated in the context of the recently started Leo-III project [WSB14]. The goal of that project is to turn the LEO-II prover into a prover based on ordered paramodulation/superposition which benefits from massive parallelism in the form of a multi-agent blackboard architecture. From the beginning of the project, careful attention is payed to a robust fundament of efficient data structures and compatibility with other proof systems and the TPTP architecture.
At the current state of development, a number of data structures (including further utility components, such as parser and a shell) and a proof-of-concept of the blackboard and related structures are already implemented, counting roughly 8000 lines of code. The code of the project – including the code described in this thesis – can be found at the Leo-III project repository[1]

Outline of this thesis As an introductory section, §2 first roughly surveys important notions of the λ-*calculus* in its untyped and its simply typed variant. Then, on the basis of the simply typed λ-calculus, classical *higher-order logic* (used synonymously for Church's Simple Theory of Types) is introduced and a short overview of its syntax and semantics is given. Finally, the field of *automated theorem proving* including its history and applications is discussed.

[1] The project repository fork of the author can be found at https://github.com/lex-lex/Leo-III. Access rights might need to be granted explicitly and can be requested via E-Mail to the author.

The next section, §3, agrees on a term language for the Leo-III prover that supports *parametric polymorphism*, commonly referred to as *System F*. Advantages as well as problems of polymorphic types are discussed.

Subsequently, in §4, efficiency considerations concerning the internal representation of polymorphically typed λ-terms are discussed. In particular, the representation techniques of *spine notation*, *explicit substitutions* and *perfect term sharing* are addressed and finally combined. The resulting term data structure is evaluated in terms of β-normalization and head symbol query efficiency and memory consumption.

In the next section, §5, the concept of *term indexing* is discussed and the preliminary term indexing data structure of Leo-III is presented. The term index supports fast retrieval of terms with respect to a number of different query conditions. The concrete query conditions are discussed and motivated by practical application examples.

A brief overview of further important data structures for automated theorem provers is given in §6.

Finally, §7 concludes and underlines the importance of efficient data structures for term representation and indexing.

2 Preliminaries

Although it is assumed that the reader is familiar with most of the basic notions of higher-order logic and its theoretical foundations, we give a short overview of important concepts, mainly to establish a common notation that is used throughout the remainder of this thesis.

In the first section, the classical untyped λ-calculus is introduced along with a brief overview of its history and origin. As a step towards higher-order logic, the next section presents the (simply) typed λ-calculus together with the concept of types. Additionally, some important properties of that calculus are presented. Finally, the basic notions of higher-order logic and its automation are coarsely surveyed.

2.1 The λ-Calculus

In the late 1920's Alonzo Church worked on a formal system to provide a foundation of logic built on top of functions rather than sets, which was published 1932 [Chu32]. His system is an untyped logic that uses a notion of explicit function abstraction, presenting an early version of the system that is now commonly known as the λ-*calculus*.

One of the motivations for introducing explicit abstraction is to formally distinguish functions from values. As an example, consider the conjunction $(a \wedge T)$ (where T denotes truth) that may be used in both of these roles: As a part of a larger formula, say, of a universal quantification $\forall a.\ (a \wedge T)$ the above term embodies a function that assigns the input Boolean to itself (Boolean identity function), whereas in $(a \wedge T) = T$, it represents a value equal to the value of a (which, of course, depends on the given context). Although this distinction is often omitted in usual mathematical notation or even natural language (e.g. "x^2 is continuous", whereby, more precisely, the function $x \mapsto x^2$ is meant), it emphasizes a fundamental difference of both objects. As for the above example, it would apparently not be reasonable to test the truth-value T and the function that maps a to $(a \wedge T)$ for equality – it is not clear how values and functions should be compared.

Church presented a notation that makes this difference explicit by introducing a *function abstraction* denoted by the letter λ, yielding the term $(\lambda a.\ (a \wedge T))$ for the function role of our example. Application of a function to an argument is denoted by juxtaposition, $((\lambda a.\ a \wedge T)T)$ meaning the application of the $(a \wedge T)$-function to the argument T, yielding $T \wedge T$, hence T. Notions of explicit abstraction also appeared in other logical systems, e.g. by Gottlob Frege [Ang84] or Giuseppe Peano [G. 89]. Unlike the others, Church included a formal description and thorough studies of conversion rules for the proposed abstraction mechanism.

The abstraction and application terms, together with so-called *atoms* (i.e. variables or constants of the domain), form the concise syntax of the λ-calculus. As it turned out, the revised version of the λ-calculus [Chu33] played a remarkable role in several research fields, such as logics or computer science. The λ-calculus also presented a formal model of computation, which was shown to capture the intuition of effectively computable functions, hence being as expressive as Turing machines [Tur37] or μ-recursion [Kle36].

Whereas the λ-calculus was not well-established among mathematicians, it was gradually picked up by computer scientists, logicians, philosophers and linguists and is viewed as one of the major developments of the younger history. An exhaustive overview of the impact of the λ-calculus can be found in, e.g., [Bar97], its history in, e.g., [CH06].

We now give a brief introduction of the pure (untyped) λ-calculus, closely following [Bar85].

Definition 2.1 (Terms of the λ calculus)
The terms of the λ-calculus are given by the following abstract syntax:

$$\begin{aligned} s, t ::= \ & x \in Id && \text{(Atom)} \\ \mid \ & (\lambda x.\ s) && \text{(Abstraction)} \\ \mid \ & (s\ t) && \text{(Application)} \end{aligned}$$

where Id denotes the set of identifiers. The set of all terms is denoted Λ. ⌟

Intuitively, an abstraction term $(\lambda x.\ s)$ is an anonymous function with formal parameter x and body s, where all free occurrences of x in s are bound by the λ head. A term of the form $(s\ t)$ represents an application of the function term s to the argument t. As notational agreement, parenthesis may be dropped whenever the term's meaning remains unchanged, that is, application is assumed to associate to the left and abstraction to the right. Hence we may write $s\,t\,(\lambda x.\ u)\,v$ instead of $(((s\,t)\,(\lambda x.\ u))\,v)$. We further agree that a λ-binder consumes as much terms as possible to the right, e.g., $(\lambda x.\ \lambda y.\ s\ t)$ stands for $(\lambda x.\ (\lambda y.\ (s\ t)))$.

In order to describe the semantics of term application, a conversion of abstractions and applications called β-conversion (or β-reduction) is introduced. Interestingly, all of the reduction semantics of the λ-calculus can be described using this single rule. The β-conversion is commonly defined with help of substitutions, presented in a capture-free version by

Definition 2.2 (Substitution of λ-terms)
Let $s \in \Lambda$ be a λ-term. Within s, capture-free substitution of variable x by term $t \in \Lambda$, denoted $s[x/t]$ is defined by

$$\begin{aligned} x[x/t] &:= t \\ y[x/t] &:= y \quad \text{provided } x \neq y \\ (s\ s')[x/t] &:= (s[x/t]\ s'[x/t]) \\ (\lambda x.\ s)[x/t] &:= (\lambda x.\ s) \\ (\lambda y.\ s)[x/t] &:= (\lambda y.\ s[x/t]) \quad \text{provided } x \neq y \text{ and } y \notin fv(t) \end{aligned}$$

where $fv(.)$ denotes the set of free variables of a term. ⌟

The justification of considering capture-free substitution only, is given by another conversion method used for renaming, called α-conversion. α-conversion embodies the

intuition that functions are considered equal if they differ in naming of parameters only. As an example, it should be apparent that the terms $(\lambda x.\ square\ x)$ and $(\lambda y.\ square\ y)$ should be equivalent. This conversion method is formally stated by

Definition 2.3 (α-conversion for the λ-calculus)
For a term $s \in \Lambda$ and two variables $x, y \in Id$, the α-conversion rule, denoted $\longrightarrow_\alpha$ is given by

$$(\lambda x.\ s) \longrightarrow_\alpha (\lambda y.\ s[x/y]), \quad \text{given } y \notin fv(s)$$

We say that two terms $s, t \in \Lambda$ are *α-equivalent* (i.e. equivalent up to renaming), denoted $s =_\alpha t$, iff there exists a series $(u_i)_{0 \le i \le k} \subseteq \Lambda$ with $u_0 = s$, $u_k = t$ and $u_i \longrightarrow_\alpha u_{i+1}$. ⌟

α-conversion rewriting is used to avoid variable capture when applying substitution for evaluating terms. The rewriting rule that completely described the reduction semantics of the λ-calculus, is given by

Definition 2.4 (β-reduction for the λ calculus)
For terms $s, t \in \Lambda$ and variable $x \in Id$, the β-reduction rule, denoted $\longrightarrow_\beta$, is given by

$$(\lambda x.\ s)\ t \longrightarrow_\beta s[x/t]$$

Let further $\longrightarrow^*_{\alpha\beta}$ be the smallest equivalence relation closed under $\longrightarrow_\alpha$ and $\longrightarrow_\beta$. Then, we say that two terms $s, t \in \Lambda$ are *β-equivalent* (i.e. equivalent up to β-reduction), denoted $s =_\beta t$, iff $s \longrightarrow^*_{\alpha\beta} t$. ⌟

A term of the form $(\lambda x.s)t$ is called *redex*, and terms containing at least one redex are called β-reducible, hence a β-conversion rewriting step can be applied.

There is a further conversion rule, called η-conversion, which captures the principle of functional extensionality, i.e., that two functions are considered equal, iff the function values agree for all arguments. This is formalized by

Definition 2.5 (η-conversion)
For a term $s \in \Lambda$ and a variable $x \in Id$, the η-conversion rule, denoted $\longrightarrow_\eta$, is given by

$$(\lambda x.\ s\,x) \longrightarrow_\eta s, \quad \text{given } x \notin fv(s)$$

The application of the η-conversion rule is also commonly called *η-contraction*. When considered in the opposite direction (i.e from right to left), we say that the left term results from the right term by *η-expansion*. $\beta\eta$-equality of terms, denoted $=_{\beta\eta}$ and $\longrightarrow^*_{\beta\eta}$ are defined in the analogously to the β-only case. ⌟

The η-conversion rule can additionally be considered during normalization.
The notion of fully evaluated terms is given by normal forms. Intuitively, when no application of a conversion rule is possible, the term is in normal form with respect to the given rewrite rule. More formally,

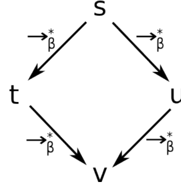
Definition 2.6 (Normal forms)
A term $s \in \Lambda$ is said to be in

(a) *β-normal form*, iff there is no subterm s' of s where s' is a redex.
(b) *η-normal form*, iff there is no subterm s' of s that can be η-contracted

A term is in $\beta\eta$-normal form, if both above cases apply. ⌟

An important result towards a deeper understanding of β-reduction and its normal forms was published by 1936 by Alonzo Church and John Barkley Rosser, known as the Church-Rosser Theorem [CR36].

Theorem 2.1 (Church-Rosser Theorem)
For any terms $s, t \in \Lambda$ it holds that if $s \longrightarrow_\beta^* t$ and $s \longrightarrow_\beta^* u$, then there exists a term $v \in \Lambda$ with $t \longrightarrow_\beta^* v$ and $u \longrightarrow_\beta^* v$. This fact is illustrated by the diagram below.



⌟

This theorem is equivalent to the fact that, as a rewriting system, $\longrightarrow_\beta$ is *confluent*. One of the most important corollaries of the Church-Rosser Theorem is that there is at most one normal form of a term, hence all normal forms are α-equivalent. Due to the uniqueness of a term's normal form, if existent, we can describe the *normal form of a term s* in a well-defined way: The normal form of a term s, here denoted $s{\downarrow}_\beta$ is the term $t \in \Lambda$ such that $s =_\beta t$ and t is in normal form, if existent, or undefined otherwise. The theorem also holds for $\beta\eta$-normalization.

As is turned out, there are indeed terms for which no β-normal form exists. In that case, we say that Λ is not *strongly normalizing* (in fact, not even *weakly normalizing*) [Bar85]. Consider the term $(\lambda x.\ x\,x\,x)\ (\lambda x.\ x\,x\,x)$ which yields an infinite sequence of β-reductions

$$\begin{aligned}(\lambda x.\ x\,x\,x)\ (\lambda x.\ x\,x\,x) &\longrightarrow_\beta (\lambda x.\ x\,x\,x)\ (\lambda x.\ x\,x\,x)\ (\lambda x.\ x\,x\,x)\\ &\longrightarrow_\beta (\lambda x.\ x\,x\,x)\ (\lambda x.\ x\,x\,x)\ (\lambda x.\ x\,x\,x)\ (\lambda x.\ x\,x\,x)\\ &\longrightarrow_\beta \ldots\end{aligned}$$

2.1.1 A nameless representation

In practical systems, such as automated theorem provers or logic programming systems, the explicit handling of variable capture in term processing can be error-prone and cumbersome. Additionally, respecting α-convertibility can lead the necessity of inefficient comparisons. As a solution, Nicolas Govert de Bruijn proposed in 1972 a nameless representation of λ terms, where the named variables are replaced by natural numbers, the so-called *de Bruijn indices* [Bru72]. The idea is that variables don't have names associated with them anymore, but are instead represented by the number of λ-binders in scope up to the one that binds that variable.
The syntax is given by ($i \in \mathbb{N}$):

$$s, t := i \mid \lambda.\ t \mid s\, t$$

describing variables, abstractions and applications, respectively. Here, a variable i is called *bound* if there are (at least) i λ-binders in scope. We omit a description of the reduction semantics and postpone it to the introduction of Leo-III's term representation. As a example, consider the term

$$\lambda x.\lambda y.y(\lambda z.xyz)x$$

that is represented by

$$\lambda.\lambda.1(\lambda.321)2$$

using de-Bruijn indices.
As one can see, every α-convertible term reads syntactically equal thus eliminating the need of considering α-conversion while comparing terms for equality. Hence, the use of de-Bruijn indices offers a unique representation for α-equivalent terms.

Note that, in this notation, the same λ-binder has, in general, different indices associated with it: In the above example the first (second) λ – that binds x (y) in the named variant – binds indices 3 and 2 (1 and 2). Also, same indices may be bound to different λs. This is one of the reasons why terms with de-Bruijn indices are not as human-readable as their named counterpart and term processing systems may use nameless term representations internally and assign names to variables on the user interface level.

The rigid use of de-Bruijn indices is, however, not only complicated for humans but also costly for automation when it comes to *shifting*: A common operation is to increment all indices of a term that are greater than a specific bound. This includes, of course, all free variables of a term since they are represented by a index value that is greater than the maximal number of λ-binders in scope.

As a reasonable compromise, Barendregt mentioned a combination of natural numbers for bound variables and names for free variables [Bru72]. This technique is now widely known as *locally nameless* representation of terms (see, e.g., [Cha11]). The locally nameless approach combines that (externally) defined terms can, again, be managed easily and need not be renamed during β-reduction while bound variables can still be

treated elegantly. Throughout this thesis, we will use a locally nameless representation of terms.

For each λ-calculus variant mentioned in the remainder, we can define a locally nameless version. In order to avoid boilerplate definitions, an explicit description of (locally) nameless representations will only be provided if it is of particular interest.

There are only few other techniques that try to cope with α-conversion: A relatively popular exception is usage of nominal representations as proposed by Pitts and others (e.g. [GP02, Pit03]).

2.2 Typed Variants of the Classical λ-Calculus

There exist many variants of the classical λ-calculus that have been studied since the original publication by Alonzo Church in the 1930s. Since the logic addressed in this thesis is of *higher-order* (a brief introduction can be found in §2.3), we only focus on *typed* variants of λ-calculi. This is due to the fact that untyped calculi suffer from inconsistencies as, for example, first shown by Stephen Kleene and John Barkley Rosser in 1935 for the untyped λ-calculus [KR35]. Later, Haskell Curry simplified the essence of the inconsistency and published a simpler variant, known as *Curry's paradox* [Cur41]. Also, an encoding of the well-known Russel paradox is possible in the untyped λ-calculus.

Classical higher-order logic is traditionally based on the simply-typed λ-calculus [Chu40], often denoted $\lambda \rightarrow$, which is subsequently introduced. It has led to development of Church's Simple Theory of Types (STT), which is used here synonymous to higher-order logic. The use of types imposes a rather strict restriction on the expressivity of the system: As an example, general recursion cannot be formulated directly in $\lambda\rightarrow$, hence sacrificing Turing-completeness. On the other hand, types are a useful tool for giving meaningful semantics and avoiding inconsistencies. Only those terms which can be assigned a type (called *well-typed terms*) need to be considered.

For numerous application, the expressibility of $\lambda\rightarrow$ is too low, and various *extensions* have been investigated. Some of them pursue a rather practical aspect of expressibility and include certain common type constructions (such as sum types, product types, ...) or other pragmatic mechanisms suited for the application to the theory of programming languages. Extensions with deep theoretic influence generally consider more general notions of computability, e.g., the addition of different variants of recursion. A good overview of type systems and their practical applications can be found in Benjamin Pierce's monograph [Pie02].

In the following, the simply typed λ-calculus is introduced along with its most important properties. A brief overview of extensions to $\lambda\rightarrow$ is given subsequently. Here, we focus on a particular hierarchy of extensions representing the axes of refinement in the *calculus of constructions* [CH88].

2.2.1 The $\lambda\!\rightarrow$-Calculus

As mentioned in the previous section, the simply-typed λ-calculus was invented as a formal foundation of logics as part of Church's *Simple Theory of Types* [Chu40]. The discovery of the Kleene-Rosser paradox and, later, Curry's Paradox, stressed the necessity of distinguishing different "sorts" of objects in mathematics and meta-mathematics. To overcome the inconsistencies, Russel proposed a formal type theory, which was first depicted in the appendix *doctrine of types* of [Rus03]. The strict distinction between "sorts" of objects was already known to Frege, who distinguished predicates from objects [Mos68]. Church's Simple Theory of Types (or simply: Simple type theory, STT), is considered a simplification of the type theory of Russel and Whitehead [Rus08, WR26]. Subsequently, the simply typed λ-calculus, denoted $\lambda\!\rightarrow$, is briefly introduced. A detailed discussion of typed λ-calculi including $\lambda\!\rightarrow$ can be found in [Hin14].

Types In $\lambda\!\rightarrow$ the notion of *types* is introduced. A type τ is given by the following abstraction syntax;

$$\begin{aligned} \tau, \nu ::= &\ t \in T && \text{(Base type)} \\ \mid &\ \tau \rightarrow \nu && \text{(Abstraction type)} \end{aligned}$$

where T is a (finite) set of base types. The set of all terms generated by the abstract syntax above is denoted $\mathcal{T}$. We agree that function type constructor $\rightarrow$ associates to the right. As an example, the term $\tau_1 \rightarrow \tau_2 \rightarrow \tau_3$ is equivalent to $\tau_1 \rightarrow (\tau_2 \rightarrow \tau_3)$, describing the type of a curried function that takes two parameters.
$\lambda\!\rightarrow$ allows to express *higher-order* terms, i.e. terms that take terms of function type as parameter. The order of a type is defined by

Definition 2.7 (Order of types)
For each type $\tau \in \mathcal{T}$ let $order(\tau)$ denote the *order of* τ, where *order* is given by

$$order(t) = 0, \text{ for each } t \in T \quad order(\tau \rightarrow \nu) = \max\{order(\tau) + 1, order(\nu)\}$$

We say that a type τ is a *higher-order type*, if $order(\tau) > 1$. ⌟

Terms The term syntax is essentially the same as for untyped λ-terms, except that each term is associated a type, written as subscript. In particular, term abstractions now denote the type of the abstraction parameter variable at binder position. Another commonly used notation for typed terms is to write a term followed by colon and a type (curry-style, see below), e.g. s_τ is equivalent to $s : \tau$. We will use the first notation throughout this thesis.

Let Σ be a set of typed constant symbols with each $c_\tau \in \Sigma$ having type τ, and $\mathcal{V}$ a set of variables with infinite many of variable symbols $X_\tau^1, X_\tau^2, \ldots$ for each type τ. The

terms of $\lambda\!\rightarrow$ are then given by

$$
\begin{aligned}
s, t ::= \ & X_\tau \in \mathcal{V} && \text{(Variable)} \\
\mid \ & c_\tau \in \Sigma && \text{(Constant)} \\
\mid \ & (\lambda x_\tau.\, s_\nu)_{\tau \to \nu} && \text{(Term abstraction)} \\
\mid \ & (s_{\tau \to \nu}\ t_\tau)_\nu && \text{(Term application)}
\end{aligned}
$$

We use the symbol Λ_τ for the set of terms each with type τ and $\Lambda := \bigcup_{\tau \in \mathcal{T}} \Lambda_\tau$ for the set of all terms. The type of a term may be omitted if clear from the context.

The notion of conversions and normal forms are defined analogously to § 2.1. The most important difference is that during β-reduction on an application, the left term must have function type whereas the right term must match the parameter type of the left one, formalized by

$$(\lambda x_\tau.\, s_\nu)_{\tau \to \nu}\ t_\tau \longrightarrow_\beta s_\nu[x_\tau / t_\tau]$$

There are two different type semantics in use [Rey98]: The so-called *church-style* (or intrinsic interpretation) that is commonly used in the context of higher-order logic (and in this thesis) and the *curry-style* (or extrinsic interpretation) that is rather used in the context of programming languages. In the first, the type of a term is considered a part of its name and is thus fixed and cannot change between different contexts. This also means that there are only well-typed terms since we cannot (syntactically) construct terms that are not well-types (i.e. a type is *intrinsic* to a term). The latter variant views types as an additional information assigned to terms from some external context (*extrinsic* types). Consequently, terms itself do not carry any type information: The term $\lambda x.\, x$ can syntactically constructed and the typing $\lambda x.\, x : \tau \to \tau$ and $\lambda x.\, x : \nu \to \nu$ (where $\tau \neq \nu$) can be inferred depending on the context the term occurs in. Here, typing is rather considered a verifying process for certain properties. Both styles are equally expressive since they can simulate each other. Curry-style can be mimicked by Church-style by type-erasure, and the other way around by type reconstruction.

The type of a term can algorithmically be verified and inferred, known as *type checking* and *type reconstruction*, respectively. This fact is formally by

Theorem 2.2 (Type checking is decidable in $\lambda\!\rightarrow$)
The decision problem, whether a given term s has type τ is decidable. In particular, there exists a polynomial-time algorithm for inferring the type of a term. ⌟

A prominent example algorithm for type reconstruction is *Algorithm W* in Hindley-Milner type inference [Hin69, Mil78], which can, in fact, perform type inference for a much greater class of terms. The set of typing rules for $\lambda\!\rightarrow$ [Hin14] is given in Figure 2.1. Here, the judgement s *has type* τ *under context* Γ is denoted by $\Gamma \vdash s : \tau$.

$$\frac{c_\tau \in \Sigma}{\Gamma \vdash c : \tau}\ \text{(Ty-Const)} \qquad \frac{X : \tau \in \Gamma}{\Gamma \vdash X : \tau}\ \text{(Ty-Var)}$$

$$\frac{\Gamma, X : \tau \vdash s : \nu}{\Gamma \vdash (\lambda X.\ s) : \tau \to \nu}\ \text{(Ty-Abstr)} \qquad \frac{\Gamma \vdash s : \tau \to \nu \qquad \Gamma \vdash t : \tau}{\Gamma \vdash (s\ t)_\nu}\ \text{(Ty-App)}$$

Figure 2.1: Typing rules for $\lambda\!\to$ terms

The confluence result from the untyped λ-calculus can be modified, yielding

Theorem 2.3 (Church-Rosser Theorem)
For any terms $s, t \in \Lambda_\tau$ it holds that if $s \longrightarrow_\beta^* t$ and $s \longrightarrow_\beta^* u$, then there exists a term $v \in \Lambda_\tau$ with $t \longrightarrow_\beta^* v$ and $u \longrightarrow_\beta^* v$. ⌟

Unlike the untyped λ-calculus, the set of terms of $\lambda\!\to$ has the strong normalization property:

Theorem 2.4 (Strong normalization)
For each type τ, the set of terms Λ_α is *strongly normalizing*, that is, for typed λ-terms every sequence of β-conversions finally terminates. In particular, the β-normal form of a term can be calculated by a finite number of β-reduction steps. ⌟

The original proof and a simplified version is due to Tait [Tai67, Tai75]. Strong normalization implies that every calculation in $\lambda\!\to$ terminates in finite time. Thus, $\lambda\!\to$ cannot be Turing-complete, we cannot, for example, express general recursion. Theorems 2.3 and 2.4 hold analogously when considering $\beta\eta$-normalization.

2.2.2 Extensions of Typed λ-Calculi

There exist many extensions to $\lambda\!\to$ that address a broad variety of aspects of expressibility augmentation. Popular system enhancements add (bounded) polymorphism [CW85], type operators, dependent types, coercion subtyping or many other language features.

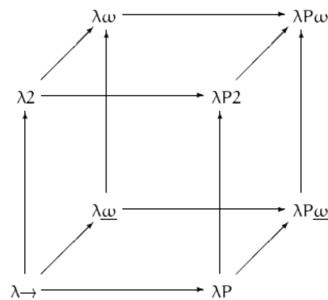


Figure 2.2: Barendregt's λ-cube

A classification of a signification class of extensions is represented by Barendregt's λ-cube [Bar91], see Fig. 2.2. The cube illustrates the axes of refinement of the calculus of constructions (CoC) [CH88] by dissecting the orthogonal language components. Hence, self-contained individual subsystems of CoC, which are all strict extensions to $\lambda\!\rightarrow$, are identified in a hierarchical ordering.

The axes represent the additional ability to express functions from terms to types, types to types and types to terms within the system located at the end of that particular edge. Many systems described by the λ-cube have been studied intensively in the past: The programming language Haskell [HPJWe92] was formerly based on $\lambda\omega$, before it was remodeled to use a even more expressive calculus, known as System Fc. The system λP is used by one of the AUTOMATH systems and the basis of the LF logical framework and various implementations [PS99].

2.3 Higher-Order Logic

The simply-typed λ-calculus as meta-theory basis for higher-order logic was already introduced in § 2.2.1.

In the setting of higher-order logic, we choose the set of base types T to be $\{o, \iota\}$, where o is denotes the type of truth values and ι the type of individuals. The set Σ of constant symbols, contains at least the logical connectives $\neg_{o\rightarrow o}$ and $\vee_{o\rightarrow o\rightarrow o}$ for negation and disjunction, respectively. Additionally, we consider for each type $\tau \in \mathcal{T}$ two connectives $\Pi^{\tau}_{(\tau\rightarrow o)\rightarrow o}$ and $=^{\tau}_{\tau\rightarrow\tau\rightarrow o}$ for universal quantification over objects of type τ and primitive quality on type τ, respectively. We may also include the choice operator $\varepsilon_{(\tau\rightarrow o)\rightarrow\tau}$ for each type $\tau \in \mathcal{T}$. The remaining logical connectives can be defined in the usual way, e.g. by $\wedge_{o\rightarrow o\rightarrow o} := \lambda s_o.t_o.\ \neg_{o\rightarrow o}(\vee_{o\rightarrow o\rightarrow o}\ (\neg s)(\neg t))$. We use binder notation $\forall X_\tau.\ s_o$ as shorthand for universal quantification given by $\Pi^{\tau}_{(\tau\rightarrow o)\rightarrow o}\lambda X_\tau.\ s_o$. For additional convenience, we write the binary logical connectives in infix position, e.g. write $s_o \vee_{o\rightarrow o\rightarrow o} t_o$ instead of $\vee_{o\rightarrow o\rightarrow o}\ s_o\ t_o$.
A *formula* is a λ-term $s \in \Lambda_o$, hence of type o. As usual, a sentence is a closed formula.

The semantics of higher-order logic is well-understood and thorough introductions can be found in the literature (e.g. [And14, BBK04]).
A *frame* is a collection $\{\mathcal{D}_\tau\}_{\tau\in\mathcal{T}}$ with $\mathcal{D}_o = \{T, F\}$ (for truth and falsehood) and $\mathcal{D}_{\tau\rightarrow\nu}$ a set of functions from $\mathcal{D}_\tau$ to $\mathcal{D}_\nu$. An interpretation $(\{\mathcal{D}_\tau\}_{\tau\in\mathcal{T}}, I)$ is a pair, where I is a function that maps each constant c_τ in Σ to an element of $\mathcal{D}_\tau$ (the *denotation* of c_τ). The function I is chosen such that the logical connectives $\neg_{o\rightarrow o}$, $\vee_{o\rightarrow o\rightarrow o}$, $=_{\alpha\rightarrow\alpha\rightarrow o}$ and $\Pi_{(\alpha\rightarrow o)\rightarrow o}$ have their usual meaning. A variable assignment ϕ is a function that maps variables X_τ to an element in $\mathcal{D}_\tau$. An interpretation is called *standard model* iff the sets $\mathcal{D}_{\tau\rightarrow\nu}$ are chosen to be the *complete set* $\mathcal{D}_\nu^{\mathcal{D}_\tau}$ of function from domain τ to codomain ν. The notion of general models (or *Henkin models*) is, in contrast, defined by choosing $\mathcal{D}_{\tau\rightarrow\nu} \subset \mathcal{D}_\nu^{\mathcal{D}_\tau}$ such that it contains "sufficiently many", but not necessarily all, functions. More formally, M is a general model iff there exists a function V that assigns for each variable assignment ϕ and each term s_τ a denotation $V_\phi\ s_\tau \in \mathcal{D}_\tau$, such that

(i) $V_\phi\ X_\tau = \phi\ X_\tau$ and $V_\phi\ c_\tau = I\ c_\tau$
(ii) $V_\phi\ (s_{\tau\to\nu}\ t_\tau) = (V_\phi\ s_{\tau\to\nu})\ (V_\phi\ t_\tau)$
(iii) $V_\phi\ (\lambda X_\tau.\ s_\nu)$ is a function $f \in \mathcal{D}_{\tau\to\nu}$ s.t. for all $z \in \mathcal{D}_\tau$ it holds that $f(z) = V_{\phi[X_\tau/z]}\ s_\nu$, where $\phi[X_\tau/z]\ X_\tau = z$ and $\phi[X_\tau/z]\ Y_\tau = \phi\ Y_\tau$ given $X_\tau \neq Y_\tau$

The function V is called *valuation* and is uniquely determined for a general model. Of course, every standard model is also a general model.

For a model $M = (\{D_\tau\}_{\tau\in\mathcal{T}}, I)$ and a formula $s_o \in \Lambda_o$, s_o is called *valid in M* if $V_\phi\ s_o = T$ for all variable assignments ϕ. A formula $s_o \in \Lambda_o$ is called *Henkin-valid*, written $\models^{HOL} s_o$, if s_o is valid in every Henkin model.

As a consequence of Gödel's incompleteness theorem [Gö31], the standard semantics of higher-order logic is incomplete. However, it shows that in most practical applications the weaker form of general semantics, developed by Leon Henkin, is sufficiently expressive. Henkin investigated on weaker, but still meaningful, models with which he described a notion of completeness for higher-order logic. This notion of completeness (with respect to general models) is given by the following theorem [Hen50]:

Theorem 2.5 (Henkin's Completeness and Soundness Theorem)
A formula is a theorem if and only if it is Henkin-valid. ⌟

Note that, throughout this thesis, the term *higher-order logic* always refers to general semantics.

Higher-order logic with respect to general semantics is as expressive as (many-sorted) first order logic and inherits convenient model theoretic properties. Nevertheless, preferring higher-order logic over of its first-order counterpart comes with a rich expressiveness in a *practical sense* [Far07]. Not only that mathematical concepts can be encoded more directly and concise, the explicit use of types supports well-structured formalization, and algebraic objects such as sets and tuples can easily be encoded on top of HOL. Also, rich extensions such as polymorphism, dependent types or subtypes can be considered.

The above introduction of higher-order semantics follows a compact introduction by P. Andrews [And14]. However, there exist a number of further generalizations of Henkin models that capture notions of weaker semantic structures. A discussion of these structures is omitted here, but can be found, e.g., in [BBK04].

2.4 Automatic Theorem Proving

Automated theorem proving (ATP) denotes the automation of proof procedures that, given a set of axioms and a mathematical conjecture as input, use a computer program to prove or refute that the input conjecture is a logical consequence of the (possible empty) set of assumptions without human assistance. ATP systems are applied in various fields, such as in mathematics, software and hardware verification, or knowledge-based systems with query answering. In the last 50 years, major progress has been made in the field

of automated reasoning, practical relevant systems have been developed and achieved acclaim in industrial and in academic applications. The goal that computer systems would solve hard open mathematical problems is still a rather idealistic conception.
Figure 2.3 displays the schematic top-level structure of the automated reasoning process.

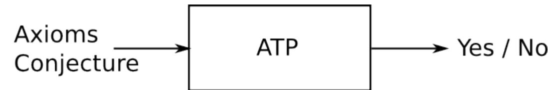


Figure 2.3: Simplistic view on automated theorem proving

For applying an ATP system to a problem, the input axioms and conjectures have to be stated formally and self-contained (i.e. all assumptions need to be explicit). This is rather cumbersome since most of common mathematical formalizations in practical use are too "colloquial" in the context of automated theorem proving, thus substantial work has to be done for using larger problems as input.

The output of an automated theorem prover is in the simplest case a single yes or no answer, depending on whether the problem could be proven of refuted (respectively). More sophisticated prover might also output a *proof object* that describes the used inferences or similar. In the negative result case, a counter-example (or counter model) might be returned in order to provide additional information about the cause of the conjecture's falsehood. It is of course preferable that some of these supplemental output information are returned. In practice, however, the generated proof object is widely unreadable and often does not contribute to the understanding of the proof itself.

The efficiency of automated theorem proving is severely restricted by the computational complexity of the underlying logic, or more formally, the underlying decision procedure for that particular logic. These procedures often have, even for relatively weak logics, a high computational complexity – or, even worse, are undecidable for logics that are at least as expressive as first-order logic [Chu36, Tur36]. The identification of decidable fragments of first-order logic was intensively researched. As an example, there exist decision procedures for propositional logic (NP-complete) and propositional modal logic (PSPACE-complete).

Origin and History This historical overview loosely follows [Mac95], in which a more detailed review of the history of automated theorem proving can be found.

Formalizations of validity of deductions originate from the ancient Greeks, notably Aristotle, who used so-called *syllogisms* to represent reasoning with inferences. In more recent history, the idea of formal deduction based on a set of inference rules goes back to Leibniz, who stated the vision of creating a universal logical language (*characteristica universalis*) and a calculus for reasoning within this language (*calculus ratiocinator*), yielding a analytical method for determining the truth or falsehood of propositions [Lei51]. A notable step towards Leibniz' vision, and thereby modern logics, is due to Gottlob Frege, who described a formal proof calculus for a system similar to second-order logic [Fre79].

The field of automated theorem proving, hence the automation of logical calculi as described above, can be dated back to the late 1950s and early 1960s, where the *Logic Theory Machine* was built as an automated decision program for propositional logic [McC04]. This program found proofs for 38 of 52 selected theorems from the *Principia*, in particular a simpler proof to a specific theorem, compared to the one given by Whitehead and Russel [Dru08].
Some time later, one of the most popular machine-oriented proof procedures, the resolution calculus in its modern form, was presented in 1965 [Rob65], and achieved substantial success in automation. In resolution-based theorem proving, the initial conjecture is negated, transformed into sets of clauses, and successively saturated using a set of inference rules. If, after a number of inferences, the empty clause is inferred, a contradiction is found hence the initial conjecture is confirmed. Today, there exist a number of different calculi well-suited for automation of propositional and first-order logic, including Tableau and Connection calculi (and several variations thereof).

For the higher-order case, however, machine-oriented proof calculi were scarce for a long time. Andrews presented a resolution-based calculus in 1971 [And83], but with strong limitations for effective automation. Further calculi better suited for automation of higher-order logic, were developed subsequently, including Huet's constrained resolution [Hue72] and extensional higher-order resolution [BK98].

Early sophisticated theorem provers include the Boyer-Moore theorem prover (1971) [BKM95], Coq (1989) [Pau11] and Isabelle (1989) [NWP02]. A thorough survey of automated theorem proving in the higher-order setting can be found in, e.g., [BM14].

The TPTP Infrastructure The *Thousands of Problems for Theorem Provers* (TPTP) problem library [Sut09] provides a coherent environment for testing automated theorem provers for their correctness and performance. To that end, it postulates a standardized and stable formula representation syntax for every supported logic language (e.g. FOF for first-order formulas or THF for typed higher-order formulas [SB10]) Currently, the TPTP problem set contains over 19000 problems for ATP, including 7971 first-order and 3036 higher-order problems, grouped in problem domains. Furthermore, the TPTP web site offers several services that become handy when testing ATPs in a more specialized way: Problems can be found by their domain, type, occurrence of equality, size, and many more attributes. Sample problems, and even user contributed problems, can be tested interactively on multiple ATPs through a web interface.

A relatively young language proposal, known as *TFF1*, offers built-in language support for rank-1 polymorphism for typed first-order formulas.

Some provers Today, a large variety of automated theorem provers for first-order logic, and a still manageable amount of higher-order logic theorem provers exist. The following list contains some prominent examples but claims by no means to be complete.

First-order provers include, for example,

- Vampire [KV13]
- E [Sch02]
- OTTER [McC90]
- Prover9 [McC10]

Prominent higher-order provers are, for example,

- TPS [AB06]
- Isabelle/HOL [NWP02]
- LEO-II [BPTF08]
- Satallax [Bro12]
- agsyHOL

Recent Development In more recent history, automated theorem provers could successfully be applied to relevant mathematical problems. The first serious computer assisted proof was conducted 1976 where the four-color problem (colorings of maps with four colors) was proven by an exhaustive case distinction analysis [AH76]. Another example is the Robbins conjecture about certain identities in a special algebra which was proven with the help of the theorem prover EQP in the late 1990s [Mcc97] or Kepler's conjecture about optimal sphere packing which was proved with extensive use of computers [Lag11].

In the higher-order setting, comparably young efforts were made to semantically embed various expressive non-classical logics into classical higher-order logic [BR13, BGGR12]. Those embeddings could then be employed to encode and automate proofs from theoretical philosophy, such as Gödel's ontological proof of God's existence [BP14].

3 The Term Language for Leo-III

As for the term language of the Leo-III prover, we do not base it on the simply typed λ-calculus, but on a more expressive extension that admits built-in support for polymorphism. This section's introduction addresses the presentation of the formal system as is it used on a theoretical level. The next section, §4, then focuses on alternative representations that are due to efficiency considerations for automation.

The motivation for native support of polymorphism is, among others, the restricted expressiveness of simple types. The reuse of similar terms (e.g. extensional equality on different types of functions) is one the major benefits. As an example, consider the universal quantification symbol Π. If we plan to use multiple types $\tau_1, \tau_2, \dots$ in reasoning and to quantify over them, we would have to include appropriate symbols Π^τ of type $(\tau \to o) \to o$ for *each* type τ_i, even if their definition all might be alike. Similar considerations apply, e.g., for primitive equality $=^\tau$. With the use of polymorphic types, we could provide *a single* instance of, say, equality that applies to objects of *all types*.

The need for polymorphism is underlined by the increasing support of such mechanisms, for instance, by the TFF1-language extension [BP13] for typed first-order formulas. In the context of TFF1, so-called *rank-1 polymorphism* and type constructors are employed on top of typed first-order formulae, embodying a, in a sense, even more expressive extension[2]. Another example is Isabelle/HOL [NWP02] where polymorphic types can be stated in a ML-like syntax.

We restrict ourselves to parametric polymorphism and do not consider the addition of, say, type operators (i.e. function from types to types). The reason for that is a simple trade-off between expressibility and complexity. More sophisticated type systems require a more careful analysis of the complexity of the system to be implemented. It is planned to include further extensions to Leo-III's term language in the future, but focus on parametric polymorphism within this thesis.

The language. The term calculus that is used as the basis of the Leo-III prover is one that extends the simply typed λ-calculus with parametric polymorphism, yielding the *second-order polymorphically typed λ-calculus* (corresponding to $\lambda 2$ in Barendregt's λ-cube [Bar91]). In particular, the system under consideration was independently developed by John C. Reynolds [Rey74] and Jean-Yves Girard [Gir72] and is commonly called *System F* today.

The main extension compared to the simply typed λ-calculus is that System F offers variables ranging over types, additionally to variables ranging over terms of a specific type. Together with a new binding mechanism via the Λ-binder, terms can now be abstracted to take a type as formal parameter. The syntax is described in the following:

[2]The term language proposed by TFF1 is not directly comparable to the one considered here, since it adds type operators (thus, being more expressive) but only allows rank-1 polymorphism (and is thereby less expressive in the context of parametric polymorphism).

Types. As usual, let T be a non-empty set of distinct base types; The set of types $\mathcal{T}$ is then constructed by the following abstract syntax definition:

$$\begin{aligned}\tau, \nu ::= \;& t \in T && \text{(Base type)}\\ |\;& \alpha && \text{(Type variable)}\\ |\;& \tau \to \nu && \text{(Abstraction type)}\\ |\;& \forall \alpha.\, \tau && \text{(Polymorphic type)}\end{aligned}$$

Two cases, the second case and the fourth case, are added in this setting. Whereas the first offers a mechanism to use *type variables* insides types, the latter is an *all-quantified type*, which is the type of a polymorphic function with type variable α, ranging over all types from T (including the above type itself). Similar to the binding of variables on term level, the type variable α is bound by the $\forall$-quantifier, which abstracts all free occurrences of α inside τ.

Example 3.1 (Polymorphic types)
All the types

(i) $\iota \to \iota \to o$ (Type of a binary relation over individuals)

(ii) $o \to o$ (Type of an unary logical connective)

(iii) $(\iota \to o) \to o$ (Type of some higher-order function, e.g. quantification over individuals)

are simple (monomorphic) types that can also be expressed in $\lambda\!\to$. In contrast, the types

(iv) $\forall \alpha. \alpha \to \alpha \to o$ (Type of a binary relation over every type, e.g. equality)

(v) $\forall \alpha. \alpha \to \alpha$ (Type of a generic identity function)

(vi) $\forall \alpha.\, (\alpha \to o) \to o$ (Type of a polymorphic higher-order func., e.g. quantif. over objects)

(vii) $(\forall \alpha.\, (\alpha \to \alpha) \to \alpha \to \alpha) \to \forall \alpha.\, (\alpha \to \alpha) \to \alpha \to \alpha$ (Type of a more complex function)

are due to the extension to System F.
Note that one can construct types that are not inhabited by any term, regardless of the choice of base types. As an example, there is no term of type $\forall \alpha.\alpha$ since any term t with that type would need to have *all types*, which contradicts the uniqueness of term typing. At the end of this section, we will give meaningful examples of terms that have complex types such as (vi) or (v). ⌟

Terms. As for the types, two new cases are added on term level: The type abstraction $\Lambda\alpha.\, s$ yields a new function that takes a *type* as formal parameter and abstracts from all (free) occurrences of type variables α inside term s. That includes the occurrences of α in type annotations, e.g., of term abstractions, as well as in *type abstractions*: A term $(s\ \tau)$ is called type application and denotes the application of the (type-abstracted) term

s with argument τ. Again, let Σ be some set of fixed symbols and $\mathcal{V}$ the set of variables (cf. § 2.2.1). The abstract syntax of terms is then given by:

$$
\begin{aligned}
s, t ::= &\ X_\tau \in \mathcal{V} && \text{(Variable)}\\
&\mid c_\tau \in \Sigma \quad \mid d_\tau, \text{ where } d_\tau := s_\tau \in \Sigma_\delta && \text{(Constant / defined atom)}\\
&\mid (\lambda x_\tau .\ s_\nu)_{\tau \to \nu} \quad \mid (s_{\tau \to \nu}\ t_\tau)_\nu && \text{(Term abstraction / application)}\\
&\mid (\Lambda \alpha.\ s_\tau)_{\forall \alpha.\ \tau} \quad \mid (s_{\forall \alpha.\ \tau}\ \nu)_{\tau[\alpha/\nu]} && \text{(Type abstraction / application)}
\end{aligned}
$$

We additionally consider defined atoms, that can be used as a shorthand for some (complex) terms. Those atoms are collected in a set Σ_δ together with its definition, yielding objects of the form $d_\tau := s_\tau$ where s is the definition of the definiendum c. A term consisting of one or more defined atoms can be *δ-expanded*, yielding a term where all occurrences of d_τ are substituted by s_τ, for $d_\tau := s_\tau \in \Sigma_\delta$. This is formally stated by

Definition 3.1 (δ-expansion)
For $d_\tau := s_\tau \in \Sigma_\delta$, the δ-expansion rule, written $\longrightarrow_\delta$, is given by

$$d_\tau \longrightarrow_\delta s_\tau$$

Note that this rule is not considered during β-normalization. ⌟

Typing In 1998, Joe Wells showed that the problem of type reconstruction in System F is undecidable [Wel98]. Since the LEO-III project does not aim to provide a full *programming language*, but rather an expressive language for the automation of logic, the above undecidability result does not impose a major problem in our case. The input problems are expected to be explicitly typed by the user, as it is common practice for typed input languages of the TPTP platform [Sut09]. The problem of *type checking* (here in the sense that given a term s and a type τ, the question of whether s_τ a well-typed term, i.e. is τ a valid type for s?) for System F terms can still be easily solved by a simple top-down algorithm. The typing rules of System F are shown in Figure 3.1.

$$\frac{c_\tau \in \Sigma}{\Gamma \vdash c : \tau}\ \text{(Ty-Const)} \qquad \frac{X_\tau \in \Gamma}{\Gamma \vdash X : \tau}\ \text{(Ty-Atom)}$$

$$\frac{\Gamma, X : \tau \vdash s : \nu}{\Gamma \vdash (\lambda X.\ s) : \tau \to \nu}\ \text{(Ty-Abstr)} \qquad \frac{\Gamma \vdash s : \tau \to \nu \qquad \Gamma \vdash t : \tau}{\Gamma \vdash (s\ t) : \nu}\ \text{(Ty-App)}$$

$$\frac{\Gamma, \alpha : \star \vdash s : \tau}{\Gamma \vdash (\Lambda \alpha.\ s) : \forall \alpha.\ \tau}\ \text{(Ty-TyAbstr)} \qquad \frac{\Gamma \vdash s : \forall \alpha.\ \tau \qquad \nu \in \mathcal{T}}{\Gamma \vdash (s\ \nu) : \tau[\alpha/\nu]}\ \text{(Ty-TyApp)}$$

Figure 3.1: Typing rules for System F terms

The expression $\tau[\alpha/\nu]$ denotes the substitution of all free occurrences of α by ν in τ. The judgment $\alpha : \star$ means that α is a type.

Like the simply typed λ-calculus, $\lambda\!\to$, System F is confluent and strongly normalizing. Proofs can be found, e.g., in [Mat99].
In contrast to the untyped case, where normalization using η-expansion cannot be applied in a meaningful way, we consider terms that are maximally η-expanded as η-normalized, hence we agree on η-long normal form. In order to avoid reductions loops with β-reduction and η-expansion interplay, e.g.,

$$s_{\tau\to\nu}\; t_\tau \quad \longrightarrow_\eta \quad (\lambda X_\tau.\; s_{\tau\to\nu}\; X_\tau)\; t_\tau \quad \longrightarrow_\beta \quad s_{\tau\to\nu}\; t_\tau \quad \longrightarrow_\eta \ldots$$

we consider conditional η-expansion: A (sub-)term $(s_{\tau\to\nu})$ is expanded to $(\lambda X_\tau.\; s_{\tau\to\nu}\; X)$ if and only if (1) it is not applied to an argument and (2) if it is not a abstraction.

Expressibility. The full (unrestricted) System F is a very powerful and expressive term language. Here, expressiveness is intended to capture two quite complementary intuitions: First, the term language has a sophisticated *practical* expressiveness as it allows to structure input theories (or programs) in terms of reusability [Tho97]. This is due to the fact that one can formulate terms (e.g. functions) that are generically applicable to terms of all types, as, for instance, the identity function $\Lambda\alpha.\lambda x_\alpha.\; x$. In contrast, in the simply typed λ-calculus, for each type a identity function has to be provided. Another practical benefit is the ability to define various data types (algebraic data types), including recursive types (such as lists).

Example 3.2 (Encoding of Booleans in System F)

In System F, we now can define polymorphic versions of the Church encoding of Booleans [Chu40]. Using polymorphic variants comes with several benefits, such as uniqueness of data type instances: In the simply typed λ-calculus, for instance, the Boolean value **true** can be represented by the term $\lambda x_\tau.\lambda y_\tau.\; x$ for each type τ. This implies that for each type τ there is an instance of **true** (of that respective type) and it's not intuitively clear how different instances of that value can be compared or are interchangeable.
The Boolean values **true** and **false** can be defined as follows in a polymorphic λ-calculus:

$$\mathbf{true} := \Lambda\alpha.\lambda x_\alpha.\lambda y_\alpha.\; x \qquad \mathbf{false} := \Lambda\alpha.\lambda x_\alpha.\lambda y_\alpha.\; y$$

each of type $\forall a.a \to a \to a$. Thus we can use the abbreviation $\mathbf{Bool} := \forall a.a \to a \to a$ for the type of Boolean terms and define the following Boolean connectives of type $\mathbf{Bool} \to \mathbf{Bool}$ and $\mathbf{Bool} \to \mathbf{Bool} \to \mathbf{Bool}$ respectively:

$$\mathbf{not} := \lambda x_{\mathbf{Bool}}.\; x\; \mathbf{Bool}\; \mathbf{false}\; \mathbf{true}$$

$$\mathbf{and} := \lambda x_{\mathbf{Bool}}.\lambda y_{\mathbf{Bool}}.\; x\; \mathbf{Bool}\; y\; \mathbf{false}$$

The remaining connectives can be defined in the usual way. ⌟

Secondly, the *theoretical* expressiveness of System F terms is amazingly rich: Every computable function that can be proved total using second-order Peano arithmetic is representable (typable) in System F [GTL89]. This is, for instance, a strict superset of typable terms compared to Hindley-Milner types. One of the strengths of System F (in terms of typability) is that it offer *impredicative* polymorphism, i.e. impredicative quantification over type variables.

From a programming language implementer's point of view, type reconstruction is a crucial part for the convenience and the usability of the language. That is why often only a predicative fragment of System F is used, e.g. by Haskell [HPJWe92] or ML [MTM97]. Here, some further restrictions are imposed to achieve decidability of type reconstruction.

4 Internal Term Representation

In the previous section we have agreed on a term language for Leo-III on a theoretical level. In this section, practical realization aspects of such a language's underlying implementation are examined and evaluated.

Terms, considered as internal objects of data, are the most general and common pieces of information that are accessed, manipulated and created by a several routines inside the system. It is therefore not surprising, that the internal representation of terms is a crucial detail and has direct consequences on the efficiency of the whole system.

As an example, consider the common notion of term application of the λ-calculus: The true structural nature of the term $(f_{\tau\to\tau\to\tau\to\nu}\ s_\tau\ t_\tau\ u_\tau)$, which is the application of some function term f to three arguments s, t and u, is hidden by simplified notation. Whereas by common intuition this term could simply be traversed from left to right (yielding the head symbol f first and, subsequently, the three arguments), this is not possible in the classical λ-term representation. This is due to the involved currying of function application what, of course, is elegant from theoretical point of view, but hampers the term traversal in a meaningful order.

With term traversal being one of the most performed operations during the proof procedure (i.e. unification, matching, head symbol query, cf. §5) and with hundreds of such operations per second, it is apparent that the term data structure is thus a sensitive bottleneck of the whole proving system. In particular, due to the high number of term manipulation operations, even minor improvements of the term data structure may significantly increase the provers performance.

The problem mainly addressed in this section is, consequently, how terms can effectively be represented as data structures such that common operations on them can be performed efficiently. In order to provide such a term data structure, is it indispensable to identify which (possibly low-level) operations dominate during the proof procedure.

From a low-level point of view, one of the most dominant operation is term traversal as it is used for routines such as (syntactic) equivalence checks, (pre-) unification and matching between terms. Of course, all routines that compare terms in some sense also involve β-normalizing, and hence the normalization process itself has be to efficient. Yet, there is a fundamental difference between term traversal and normalization operations: Whereas terms are commonly kept in their normal form anyway, β-normalization needs only to be done once for each term (and, of course, for each manipulated term). In contrast, one and the same term may need to be traversed multiple times in the context of different routines. It follows that the term representation should support classical left-to-right traversal for operations such as unification and matching tests if an efficient manner.

Hence, the key aspects for efficient term representations focused in this thesis are fast β-reduction, quick α-equivalence checks, fast access to head symbols and, in general, efficient left-to-right traversal of terms for unification and matching.

In particular, the following three aspects of common λ-terms representations are dealt with in this section:

- A representation with explicit names complicates the identification of α-equivalent terms: $(\lambda X_o.\, p_{o\to o}\, X)$ is semantically equivalent to $(\lambda Y_o.\, p_{o\to o}\, Y)$, but a rather involved routine including term traversal and variable renaming is needed to compute this.
- The term application and type application are left-associative, yielding deeply buried arguments within the term's graph structure. As a consequence, term traversal does not reflect intuitive left-to-right reading of the term.
- The notion of bound variable substitution in β-reduction is a meta-level operation which, in naive implementations, does not allow fine-grained control over the carried out substitution walks.

In this section, the above issues are tackled by the following techniques: Constant time term comparison of syntactic equality can be achieved by using de-Bruijn indices (see below) together with sharing of syntactically equal subterms. We further employ an alternative term graph construction known as *spine notation* [CP03]. Additionally, the spine representation of polymorphically typed λ-terms is enriched with *explicit substitutions* [ACCL90] yielding more efficient normalization routines. The latter two techniques are thoroughly described in the following sections.

Nameless representation The use of de-Bruijn indices as introduced in §2.1.1 eliminates the use of bound variable names and thus simplifies the comparison of terms for equality. In a (locally) nameless term representation, two terms are α-equivalent if and only if they are structurally identical (i.e. α-equivalent terms have a unique term representation). Nevertheless, without any employment of further techniques, syntactical equivalence checks require linear time with respect to the term's size. In the following, the use of de-Bruijn indices is enhanced to (bound) type variables [KRTU99] for eliminating type variable names in polymorphic types. The definition of de-Bruijn indices for type variables is analogous to the one for term variables.

As an example, consider the polymorphically typed term

$$\Lambda\alpha.\lambda P_{\alpha\to o}.\ \Pi_{\forall\beta.\ (\beta\to o)\to o}\ \alpha\ (\lambda Y_\alpha.\ P\ Y)$$

that can be represented with de-Bruijn indices for term variables and type variables as

$$\Lambda.\lambda_{\underline{1}\to o}.\ \Pi_{\forall.(\underline{1}\to o)\to o}\ \underline{1}\ (\lambda_{\underline{1}}.\ 2\ 1)$$

where $\underline{i}$ denotes the de-Bruijn index i for type variables. Since types can be used as terms (i.e. as argument to type abstractions), the syntactical distinction is necessary to retain unambiguous syntax.

Plans of this section Firstly, the core ingredients of Leo-III's internal term representation of $\lambda 2$ are introduced in §4.1 (spine notation) and §4.2 (explicit substitutions). We omit detailed descriptions of typing and reduction rules of each intermediate system, but instead focus on the motivation and benefits of each particular representation. Consequently, the individual ingredients are combined in §4.3. Here, we discuss normalization properties, typing aspects and further details. Also, an explicit formulation of a normalization strategy is given. The technical aspect of perfect term sharing is introduced in §4.4.
Finally, in §4.5 a preliminary evaluation of the term representation is given; §4.6 discusses the native support for term orderings.

4.1 Terms in Spine Notation

While currying is – from a theoretical point of view – an elegant technique for uniform treatment of functions of all arities, it comes with a major drawback for automation. Consider the term $(f\ a\ b\ c\ d)$, where types are omitted for the moment. Here, f is some term denoting a function and a, b, c and d are some argument terms which the function f is applied to. Due to its applicative position, f is also referred to as the *head symbol* of the term. There are many applications where the head symbol of a term needs to be accessed, and, more generally, the individual argument terms of an application need to be examined in the left-to-right reading order. This is the case, for instance, during unification or matching tests. Hence, the access to the head symbol of terms, and, more generally, left-to-right traversal, needs to be performed efficiently. If we recall the above example under this aspect more formally, it can be observed that the head symbol is buried under several application layers as it properly reads $((((f\ a)\ b)\ c)\ d)$. The term's actual structure is unwarily hidden when omitting exhaustive bracketing. As a consequence, during term analysis, four traversal steps are required to access its head symbol. In general, the number of traversal steps is linear in the number of applied terms. Even worse, in operations which first analyze a term's head symbol and subsequently all its arguments, the whole terms is possibly traversed again and again, or a stack of unprocessed terms of size $O(n)$ needs to be maintained.

Cervesato and Pfenning proposed a λ-term representation called spine calculus, or, more commonly, *spine notation*, that copes with the above problems [CP03]. Terms in spine notation adopt the usual mathematical function notation as in $f(x, y)$. Here, the head symbol is located at top level and followed by a linear list of function arguments. In spine notation, the above example reads

$$f \cdot (a \cdot \textsc{Nil}; b \cdot \textsc{Nil}; c \cdot \textsc{Nil}; d \cdot \textsc{Nil}; \textsc{Nil})$$

where Nil denotes the empty argument list (called *spine*) and cons of arguments (prepending) to spines is denoted with ; (semicolon). The argument cons operation (;) is hereby right-associative.

When considering β-normal terms in spine notation, access to the head symbol requires always exactly one traversal step, no matter how many arguments are applied.

Figure 4.1 gives the syntax of terms in spine notation. As usual, term abstraction is denoted by λ. The application term of the usual λ-calculus is represented by a *root* $H \cdot S$, where H is a *head* and S is a *spine*. Intuitively, heads correspond the bound variables and constants[3] and whole terms in the unnormalized case, whereas spines compare to lists of arguments and collect destructors that are applied to the head. The atom Nil denotes the empty spine.

Terms The terms s in spine notation are defined by ($\tau \in \mathcal{T}$)

$$\begin{aligned} s &::= (\lambda X_\tau.\ s_\nu)_{\tau\to\nu} \mid (H \cdot S)_\tau \\ S &::= \textsc{Nil} \mid s_\tau ; S \\ H &::= c_\tau \mid X_\tau \mid s_\tau \end{aligned}$$

where S denote *spines* and H denote *heads*. As usual, c_τ is a constant of the context.

Figure 4.1: Terms of $\lambda\!\to$ in spine notation

Since every β-normal λ-term is of the form $(\lambda X^1_{\tau_1}. \cdots \lambda X^n_{\tau_n}.\ h_\nu\ s^1_{\nu_1}\ \cdots\ s^m_{\nu_m})$, where h is either a bound variable or a constant (and type ν is chosen properly) and the s^i are terms, the head of a β-normal term in spine notation indeed corresponds to either a bound variable or a constant symbol.

The graph structure of the term $(\lambda X.\ a\ b\ c\ ((\lambda X.\ X)\ d)\ e)$ in both classical representation and spine notation is given in Figure 4.2. Note that spine concatenation (denoted by ";") is right-associative, yielding an apparent different graph as opposed to classical left-associative term application (denoted "@" in the example). All Nil atom occurrences are explicitly shown. In this representation, left-to-right traversal of terms can be performed more efficiently as the subterms occur in intuitive reading order inside the term graph.

Examples In order to get familiar to the somehow unusual notation for λ-terms, some examples are shown in both spine notation and classical representation. For reasons of readability, we omit the nameless representation and use variable names instead of de-Bruijn indices.

$$\begin{aligned} \lambda X_\tau.\lambda Y_\nu.f_{\tau\to\nu\to\nu} \cdot (X \cdot \textsc{Nil}; Y \cdot \textsc{Nil}; \textsc{Nil}) &\leftrightsquigarrow \lambda X_\tau.\lambda Y_\nu.(f_{\tau\to\nu\to\nu} X)Y \\ (\lambda X_\tau.\ X \cdot \textsc{Nil}) \cdot (\lambda X_\tau.c_{\tau\to\tau} \cdot (X \cdot \textsc{Nil})\ ;\ \textsc{Nil}) &\leftrightsquigarrow (\lambda X_\tau.X)(\lambda X_\tau.c_{\tau\to\tau} X) \\ h_\tau \cdot \textsc{Nil} &\leftrightsquigarrow h_\tau \end{aligned}$$

where the left side of the examples are terms in spine notation and the right side is its equivalent in classical representation. Due to the numerous occurrences of the Nil atom, the terms tend to be slightly longer and possibly not as clearly intuitive to understand. For reasons of readability, in the following Nil atoms are omitted whenever possible.

[3] The name *head* possibly derives from the fact that β-normalized λ-terms are of the form $(\lambda X_1. \cdots \lambda X_n.\ h\ s_1\ \cdots\ s_m)$ where h is either a bound variable or a constant, and the s_i are terms.

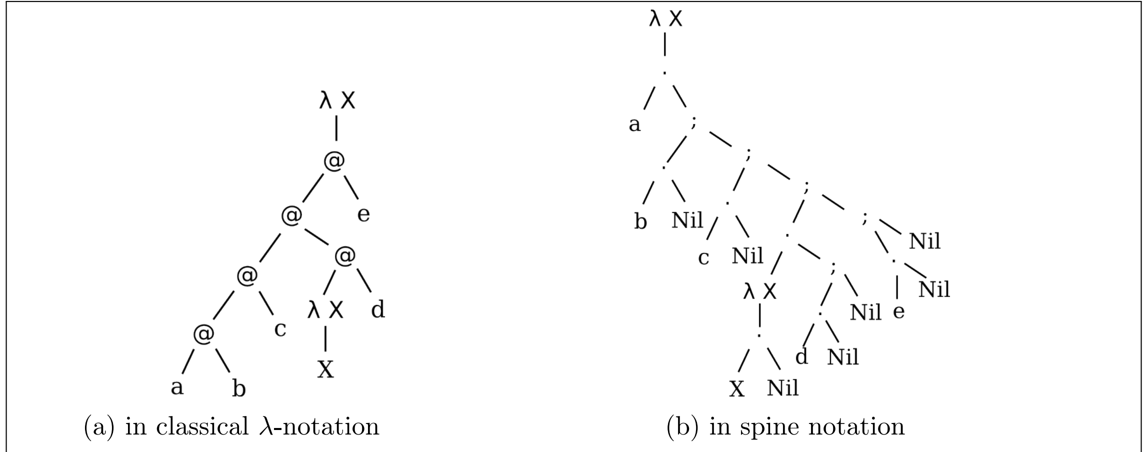


Figure 4.2: Graph structure of the term $(\lambda X.\ a\ b\ c\ ((\lambda X.\ X)\ d)\ e)$

Reductions As a consequence of the explicit linear enumeration of application arguments, a reduction rule corresponding to β-normalization needs to apply the possibly remaining spine (with one argument less) to the whole term again. This reflects the implicit nesting of function arguments inside reducible terms. Hence, the β-reduction rule is given by

$$(\lambda X_\tau . s_\nu) \cdot (t_\tau ; S) \longrightarrow_\beta (s_\nu[X/t] \cdot S)$$

As Cervesato and Pfenning point out, the above β-reduction rule does not suffice to model the common notion of β-normal forms. Since spines are modeled with an explicit termination atom, Nil, possibly empty spines that remain after exhaustive β-reduction need to be eliminated too. Consider the example

$$\begin{array}{ccc} (\lambda X_{\tau\to\tau}.\ X) \cdot ((\lambda Y_\tau.\ Y \cdot \textsc{Nil}); \textsc{Nil}) & \longrightarrow_\beta & (\lambda Y_\tau.\ Y \cdot \textsc{Nil}) \cdot \textsc{Nil} \\ \updownarrow & & \updownarrow ? \\ (\lambda X_{\tau\to\tau}.\ X)\ (\lambda Y_\tau.\ Y) & \longrightarrow_\beta & \lambda Y_\tau.\ Y \end{array}$$

Here, after normalization, a redundant Nil atom remains that cannot be eliminated with any of the currently available rules. Hence, we need to introduce a new kind of reduction, the so-called Nil-reduction, given by

Definition 4.1 (Nil Reduction)
The Nil reduction rule, written $\longrightarrow_{\textsc{Nil}}$ is given by

$$(s_\tau \cdot \textsc{Nil}) \cdot \textsc{Nil} \longrightarrow_{\textsc{Nil}} s_\tau \cdot \textsc{Nil}$$

for any term $s_\tau \in \Lambda_\tau$. ⌟

Considering $\longrightarrow_\beta$ and $\longrightarrow_{\text{Nil}}$ during β-reduction preserves the usual confluence and normalization properties of the simply typed λ-calculus in spine representation [CP03]. However, both of the above reduction rules are only sufficient when all terms are in η-long normal form. To illustrate the problem, consider the following term and normalization:

$$
\begin{array}{c}
(\lambda X_\tau.\ f_{\tau\to\tau\to\nu}\ X)\ c_\tau\ d_\tau \\
\updownarrow \\
(\lambda X_\tau.\ f_{\tau\to\tau\to\nu} \cdot (X; \text{NIL})) \cdot (c_\tau; d_\tau; \text{NIL}) \\
\longrightarrow_\beta (f_{\tau\to\tau\to\nu} \cdot (c_\tau; \text{NIL})) \cdot (d_\tau; \text{NIL}) \\
\longrightarrow_? f_{\tau\to\tau\to\nu} \cdot (c_\tau; d_\tau; \text{NIL}) \\
\updownarrow \\
f_{\tau\to\tau\to\nu}\ c_\tau\ d_\tau
\end{array}
$$

Note that, after one step of β-reducing, no rules for further normalizing apply (here marked with a "?"). Whenever β-reducing a term and the head of its redex (i.e. the head of the term itself) is not a abstraction, its spine needs to be Nil – then, because of Nil-reductions, normal form will eventually be achieved. This restriction is only true for η-long normalized terms though, since due to η-expansion, there are as many abstractions as expected (and not yet applied) parameters to each symbol.

The spine calculus was originally developed as a framework for η-long terms. However, also terms that are not η-long can be normalized if we include the rule

$$(H \cdot S) \cdot S' \longrightarrow_{\text{Mrg}} H \cdot (S +\!\!+ S')$$

Here, $S +\!\!+ S'$ denotes the meta-level operation of *spine merging*, given by $(t_1; t_2; \cdots; t_n; \text{NIL}) +\!\!+ S' = (t_1; t_2; \cdots; t_n; S')$. The spine merging operations is also denoted @ in the original publication.

The ingredient of Leo-III's internal term data structure surveyed in this section, is the representation of λ-terms as (essentially) pairs of heads and spines that allow efficient left-to-right term traversal.

4.2 Explicit Substitutions

The notion of β-reduction is one of the most central operational aspects of the λ-calculus. It is most commonly defined by a form of substitution of free variables for arbitrary terms. This substitution operation can, however, not be expressed as a native construct of the term language itself, but rather denotes a *meta level operation*. Thereby, certain implementation details of substitution routines remain unspecified and might be, depending on the concrete implementation, impractical and a source of subtle errors. When considering de-Bruijn indices in a nameless calculus, the substitution procedure is even more involved and careful attention needs to be payed to its implementation. Also, from a theoretical point of view, the meta level character of substitution does not allow any

reasoning about the reduction process itself, e.g. time and space consumption of λ-terms when considered as programs.

From the perspective of efficient β-reduction, it is generally not a reasonable idea to implement a single monolithic substitution operation that is then gradually applied during normalization. A more efficient β-normalization routine would rather collect variables that are to be substituted and postpone the actual substitutions until they can be applied on the term during a single traversal.
As an example, consider the term $((\lambda X.\lambda Y.X\ Y\ c)\ a\ b)$ (types omitted). Here, naive reduction routines would first traverse the whole term replacing all occurrences of variable X by term a, and then, on a second run, replace all variables Y by b.

In the context of functional programming and logic programming, weaker notions of normal forms, so called (weak) head normal forms, are considered. Here, β-reductions are not exhaustively applied, but only until a certain structural term property is achieved. In the context of theorem proving, however, those weaker normalization notions are often not adequate since the structure of terms under abstractions needs to be examined as well.

One approach to eliminate chaotic handling of substitutions in concrete term languages is to promote the meta-operation of substitution to be a first-class citizen of the term language itself. The idea to make reduction an explicit part of the language was picked up with the development of combinatory logic [Cur58] in which reduction is modeled as a syntactic primitive. As a downside, the number of atomic steps in this model can grow very large as opposed to regular β-reduction steps in the classical λ-calculus. A first serious competitive model was the notion of an calculus of *explicit substitutions*, denoted $\lambda\sigma$, which allows fine-grained syntactic control of reduction semantics [ACCL90].

Today, currently available systems of explicit substitutions can roughly be categorized into those systems, who allows substitution fusion (i.e. the combination of individual substitutions s.t. multiple substitutions can be carried out during one term traversal) and those, which does not. Famous example of the first kind are the $\lambda\sigma$ calculus [ACCL90], ΛCCL [Fie90] and the suspension calculus [Nad96]. Systems of the latter kind are, e.g., the λv calculus [BBLRD96], the λs calculus [KR95] or the λs_e calculus [KR97].

Merging of substitutions have shown to be a substantial efficiency advantage to practical applications [LNQ04]. That is why we chose a explicit substitution calculus with merging capabilities as the basis of the internal representation of $\lambda 2$. More precisely, the $\lambda\sigma$-calculus is used in the following, since it is also well-established in different sophisticated systems (e.g. TWELF [PS99]) and has a fair complexity.

Explicit Substitutions of $\lambda\sigma$ Figure 4.3 displays the terms and reduction rules of $\lambda\sigma$. We here examine explicit substitutions of $\lambda\sigma$ for the untyped case. Although Adabi et al. discuss second-order theories (i.e. typed terms with parametric polymorphism) as well, they mainly focus on constructions well-suited for type-checking rather than on evaluation semantics. This is why, in the remainder, we investigate the principles of explicit substitution for untyped terms and get familiar with its most basic notions.
The terms are built on top of a nameless representation using de-Bruijn indices. A term

Terms The terms of $\lambda\sigma$ are defined by

$$
\begin{aligned}
s, t &::= 1 \mid \lambda s \mid s\ t \mid s[\sigma] \\
\sigma, \rho &::= id \mid\ \uparrow\ \mid s \cdot \sigma \mid \sigma \circ \rho
\end{aligned}
$$

Reductions The rule $\underset{Beta}{\longrightarrow}$ is given by

$$(\lambda s)t \longrightarrow s[t \cdot id]$$

The ruleset $\underset{\sigma}{\longrightarrow}$ is given by

$$
\begin{aligned}
1[id] &\longrightarrow 1 \\
1[s \cdot \sigma] &\longrightarrow s \\
(s\ t)[\sigma] &\longrightarrow s[\sigma]\ t[\sigma] \\
(\lambda s)[\sigma] &\longrightarrow \lambda(s[1 \cdot (\sigma \circ \uparrow)]) \\
(s[\sigma])[\rho] &\longrightarrow s[\sigma \circ \rho] \\
id \circ \sigma &\longrightarrow \sigma \\
\uparrow \circ id &\longrightarrow \uparrow \\
\uparrow \circ (s \cdot \sigma) &\longrightarrow \sigma \\
(s \cdot \sigma) \circ \rho &\longrightarrow s[\rho] \cdot (\sigma \circ \rho) \\
(\sigma_1 \circ \sigma_2) \circ \sigma_3 &\longrightarrow \sigma_1 \circ (\sigma_2 \circ \sigma_3)
\end{aligned}
$$

Figure 4.3: Explicit substitutions of $\lambda\sigma$

$s[\sigma]$ is called *closure* and denotes the (postponed) application of substitution σ on term s. A substitution σ is basically an infinite sequence of de-Bruijn index replacements $\sigma = \{1/t_1, 2/t_2, 3/t_3, \ldots\}$ where, during application, de-Bruijn index i is replaced by t_i. We may also write $\sigma(i) = t_i$ to underline this mapping.

id denotes the identity substitution, i.e. the substitution σ that assigns $\sigma(i) = i$ to each de-Bruijn index i. The shifting operator $\uparrow$ denotes the substitution σ with $\sigma(i) = i + 1$. The original syntax only includes the de-Bruijn index 1, since the index $n + 1$ can be represented as $1[\uparrow^n]$, where $\uparrow^n$ denotes the n-fold composition $\uparrow \circ \cdots \circ \uparrow$ (see below). The *cons* of a term s to a substitution σ is given by $s \cdot \sigma$ and corresponds to the substitution σ' with $\sigma'(1) = a$ and $\sigma'(i) = \sigma(i)$ for all $i > 1$. Finally, the composition of two substitutions σ and ρ, written $\sigma \circ \rho$, produces a substitution that assigns $(\sigma \circ \rho)(i) = \sigma(i)[\rho]$ for each index i.

The first reduction rule of $\lambda\sigma$, denoted Beta, introduces new substitutions (by means of closures) to the term. The remaining ten reduction rules push closures further inwards and deal with reading the value of a de-Bruijn index with respect to a given substitution.

Consider the following example, in which the mechanism of producing and merging

closures is depicted. We omit types to address the original syntax of $\lambda\sigma$ as given above and combine multiple steps per shown inference.

$$
\begin{array}{ll}
 & (\lambda.\lambda.\ 2\ 1\ (\lambda.\ 2))\ s\ t \\
\longrightarrow & (\lambda.\ 2\ 1\ (\lambda.\ 2))[s \cdot id]\ t \\
\longrightarrow & (\lambda.\ (2\ 1\ (\lambda.\ 2))[1 \cdot ((s \cdot id) \circ \uparrow)])\ t \\
\longrightarrow & ((2\ 1\ (\lambda.\ 2))[1 \cdot ((s \cdot id) \circ \uparrow)])[t \cdot id] \\
\longrightarrow & (2\ 1\ (\lambda.\ 2))[(1 \cdot ((s \cdot id) \circ \uparrow)) \circ (t \cdot id)] \\
\longrightarrow & (2\ 1\ (\lambda.\ 2))[t \cdot ((s \cdot id) \circ \uparrow) \circ (t \cdot id)] \\
\longrightarrow & (2\ 1\ (\lambda.\ 2))[t \cdot (s[\uparrow] \cdot \uparrow) \circ (t \cdot id)] \\
\longrightarrow & (2\ 1\ (\lambda.\ 2))[t \cdot s[\uparrow \circ t] \cdot \uparrow \circ (t \cdot id)] \\
\longrightarrow & (2\ 1\ (\lambda.\ 2))[t \cdot s[\uparrow \circ t] \cdot id] \\
\longrightarrow & s[\uparrow \circ t]\ \ t\ \ (\lambda.\ 2[1 \cdot (t \cdot s[\uparrow \circ t] \cdot id) \circ \uparrow]) \\
\longrightarrow & s[\uparrow \circ t]\ \ t\ \ (\lambda.\ t[\uparrow])
\end{array}
$$

which eventually reduces to $(s\ t\ (\lambda.\ t))$ given that s and t are closed terms (i.e. contain no free variables). Note that only a single substitution application run is employed to replace all de-Bruijn indices. Besides that traversal, a calculation involving substitution compositions needs to be done. For efficiency considerations, it shows that the latter calculations can be implemented in such a way that their costs are not bigger than those of the saved traversal runs [LNQ04].

The $\lambda\sigma$-calculus is confluent on ground terms (i.e. terms without meta variables), but not confluent in general. While this is for our context sufficient, it does imply that special treatment needs to be employed when dealing with existential variables. Confluent calculi of explicit substitutions exist [CHL96], but they are far more complex and do not greatly contribute to our problem.

4.3 Combining Spines and Explicit Substitutions

In this section, we combine the previously introduced concepts of spine notation and explicit substitutions (of the $\lambda\sigma$-calculus) and adjust them for a polymorphically typed λ-calculus.

The first step, starting from the term syntax of the spine calculus of §4.1, is to re-introduce the Λ-binder for type abstractions (cf. §3) to the term language. As a consequence, spines not only contain terms as elements (as in $(s_\tau; S)$, for some spine S) but also contain *types* that are arguments for type-abstracted terms. Thus, $(\tau; s_\tau; \text{NIL})$ denotes a valid spine which could, for example, be applied to a polymorphic identity function $(\Lambda.\lambda_{\underline{1}}.\ 1_{\underline{1}})$. Since we are considering explicit substitutions, we also allow types to be prepended to substitutions σ, as in $\sigma' = (\tau \cdot \sigma)$. The reduction semantics of this augmented term language is further below.
In summary, the syntax of the resulting term language is given by

Definition 4.2 (Internal Term syntax)
The terms, denoted s, are given by the following abstract syntax declaration

$$\begin{aligned} s &::= (h_\tau \cdot S_{\tau>\nu})_\nu \mid (s_\tau \cdot S_{\tau>\nu})_\nu \mid (\lambda_\tau.\ s_\nu)_{\tau\to\nu} \mid (\Lambda.\ s_\tau)_{\forall.\tau} \mid s[\sigma] \\ h &::= i_\tau \mid c_\tau \mid h[\sigma] \\ S &::= \textsc{Nil} \mid s_\tau; S \mid \tau; S \mid S[\sigma] \\ \sigma, \rho &::= \uparrow^i \mid s_\tau.\sigma \mid \tau.\sigma \mid \sigma \circ \rho \end{aligned}$$

where $i \in \mathbb{N}$ and $c \in \Sigma$ or $(c := d) \in \Sigma_\delta$ and $\tau, \nu \in \mathcal{T}$. ⌟

Again, the language includes three further primitives: Heads, Spines and substitutions denoted h, S and σ respectively.

The original notion of heads h, where h could be a symbol or again a complex term, is modified such that there are two distinct top-level cases $h \cdot S$ and $s \cdot S$ which we call *root* and *redex* respectively. Hence, in this context, a head h is an atom which is either a de-Bruijn index i or a constant symbol c from the signature (or, of course, a closure of those). The notion of reducible terms, e.g. $(\lambda_\tau.\ s_\nu) \cdot S$, is now explicitly covered by the *redex* term case. This distinction is motivated by practical reasons of implementation details: Internally, the data type of heads is distinct from the data type of terms and some operations are meaningful for heads only. A internal conversion between head data types and term data types seems to be out of place. Additionally, by distinguishing roots and redexes, we have a rather simple criterion for checking if a term is already in weak head normal form.

Due to the introduction of spines to the term construction, the typing rules of those terms look a bit different. The typing of spines S is denoted by $S : \tau > \nu$, which intuitively means that S is a spine from heads of type τ to type ν, i.e. $h \cdot S$ has type ν if h has type τ. A discussion of the typing rules can be found in [CP03], we now focus on term reduction.

Implementation considerations. In a polymorphically typed language, substitutions also need to be applied to types rather than only to terms themselves. In the original publication of the $\lambda\sigma$-calculus, a substitution σ is simply augmented to contain terms as well as types [ACCL90]. While this may be practical for type checking applications, it is not suited for efficient reduction. This is because a substitution σ is constructed such that the replacement for a de-Bruijn index i is stored at the i-th position in σ. If types and terms are both contained in the same substitution, this is not true anymore. In order to retrieve the substitute for a term-de-Bruijn index i, σ would have to be traversed from left to right until the i-th term is encountered, skipping type entries. For efficiency considerations this is not convenient, since we could otherwise employ a random-access data structure that allows constant time access to the i-th component.

To this end, we split a substitution σ in two substitution components, the first being a substitution in which all terms substitutes are listed, the second for the type substitutes. Hence, in the following, a substitution σ denotes a pair $\sigma = (\sigma_{term}, \sigma_{type})$. Composition is component-wise, i.e. $\sigma \circ \rho = (\sigma_{term} \circ \rho_{term}, \sigma_{type} \circ \rho_{type})$

Data structure 1 (Term) The term data structure $\mathfrak{T}$ describes the terms of Definition 4.2 and is given by the above description. ⌟

Computing the β-normal form The initial idea for using explicit substitutions is to avoid redundant term traversal during β-normalization. The here presented approach tries to achieve this by consuming abstractions and their associated arguments on top-level as far as possible (i.e. until the top-level term is not an abstraction anymore) and storing the latter in a substitution σ. The so obtained term structure has form $(h \cdot S) \cdot S'$ or simply $(h \cdot S) \cdot \textsc{Nil}$ if the original term was η-expanded. In the first case, the spines S and S' are merged into a single spine $S +\!\!+ S'$. This term's structure can be seen as a *weak head pre-normal form* of the original term, since it has the same structure as a weak head normal form but with no substitutions applied so far.

At this point, it is not clear whether the application of σ to the term's head h results in introducing new λ-binders, as in $\sigma(h) = (\lambda_\tau.\ s)$, or if h will be replaced by a constant symbol or a de-Bruijn index. In the first case, further arguments could possibly be consumed from the spine (denoted above by S'), while in the latter case, the term normalization traversal can be continued inside the subterm, since constants or de-Bruijn indices cannot be further β-reduced. As consequence of that, the substitution σ is only applied to the term's head h and a case distinction is done for the above described cases.

After this step, the term reached its actual *weak head normal form*, i.e. the top-level symbol is fully normalized with respect to σ and the arguments are possibly reducible since σ was not yet applied to the spine. This is done in the last step, where the argument spine is normalized component-wise using the whole described routine recursively.

In summary, the above routine for normalizing $(s \cdot S)$ is divided into the following steps:

(i) Consume λ-binders, gather arguments in σ (*weak head pre-normal form*)
  (a) Reduce s until it is of form $h \cdot S'$
  (b) Fuse spines S' and S, if necessary
(ii) Apply substitution to h
  (a) Proceed with step (i) if new λ-binders are introduced
  (b) Otherwise proceed with step (iii)
(iii) Recursively normalize spine with respect to current substitution

This β-normalization routine is formally described by Figure 4.4. Here, for a term s, the β-normal form of s, denoted $s{\downarrow_\beta}$, is given by $s{\downarrow_\beta^{id}}$. For redex terms $(s \cdot S)$, the application of this normalization routine with respect to a substitution σ is defined by $(s \cdot S){\downarrow_\beta^\sigma} := (s \cdot S){\downarrow_\beta^{\sigma,\sigma}}$. The benefit of using two substitutions as a parameter for redex-terms, is that we can elegantly manage substitutions that only apply for the redex' head and its spine, respectively.

We write $\tau[\sigma_{type}]$ for the application of type-substitution σ_{type} on type τ, given by a straight-forward meta-level operation, that instantiates each type-de-Bruijn index by a type or type variable according to σ_{type}. This operation is similar to the original

notion of substitution but for nameless types. Type closures (i.e. closures appearing in types) could of course also be allowed, enabling explicit treatment of substitutions as we did for terms. But since we have fairly simple types (without any means of abstractions *within* types), calculating type substitutions does not involve extensive type traversal and thus does not contribute to efficient normalization in this case. This could change if we employ more expressive type systems, as for instance types of System Fω [Gir72]. Here, λ-abstractions can also appear in types as well – an explicit treatment of substitutions for type normalization might be beneficial in this case.

For two spines S, S' and two substitutions σ, σ', the meta operation of *spine closure merging* $S'[\sigma] +\!\!\!+ S[\sigma']$, i.e. merging of spines with respect to postponed substitutions, is given by

$$S'[\sigma] +\!\!\!+ S[\sigma'] = \begin{cases} s[\sigma]; (S_{tail}[\sigma] +\!\!\!+ S[\sigma']) & \text{, if } S' = s; S_{tail} \\ \tau[\sigma_{type}]; (S_{tail}[\sigma] +\!\!\!+ S[\sigma']) & \text{, if } S' = \tau; S_{tail} \\ S''[\rho \circ \sigma] +\!\!\!+ S[\sigma'] & \text{, if } S' = S''[\rho] \\ S[\sigma'] & \text{, if } S' = \textsc{Nil} \end{cases}$$

The normalization of closures is done by rules (Clos), (RxClos) and (SpClos) for top-level closures and closures occurring inside of redexes and spines, respectively. Since only one substitution parameter is used for the normalization routine of roots $h \cdot S$, head closures are explicitly introduced during normalization. This is because head substitutions (introduced by (RxRMrg)) must not get in scope of its spine S. However, all remaining kinds of substitutions for roots affect both its head and its spine. Head closures $h[\sigma]$ are handled explicitly by rules (RAtomClos), (RBndClos), (RTermClos) and (RClosClos).

Note that the only rules that produce roots from redexes are the rules (RxSpNil) and (RxRMrg) which correspond the Nil-reduction and spine merging ($+\!\!\!+$) (cf. §4.1), respectively. A root can be transformed into a redex by rules (RTermSub) and (RTermClos). Rules (Abs) and (TyAbs) correspond to the inwards-shifting of substitutions as described by $\lambda\sigma$ (cf. §4.2).

The β-normalization routine of Fig. 4.4 is currently implemented as part of the term data structure.

β-normalization variants Based on the previously described β-normalization routine, a number of variants can be examined. Simple enhancements include:

(A) *Strict composition of substitutions*
Instead of using closures during substitution composition, i.e. $(s{\cdot}\sigma') \circ \rho \longrightarrow s[\rho]{\cdot}\sigma' \circ \rho$, the composition could be calculated more strictly. More precisely, if we calculate $(s \cdot \sigma') \circ \rho = s[\rho] \downarrow_\beta \cdot \sigma' \circ \rho$ the closure $s[\rho]$ is evaluated strictly.

Pro: A lot of term closure introductions can be avoided
Con: The normalization of the closure causes more term traversals, even if the term itself is never used.

$$\frac{(c \cdot S) \downarrow_\beta^\sigma \qquad c \in \Sigma \text{ or } c := d \in \Sigma_\delta}{c \cdot S \downarrow_\beta^\sigma} \text{(RAtom)}$$

$$\frac{(i_\tau \cdot S) \downarrow_\beta^\sigma \qquad \sigma_{term}(i) = j}{j_{\tau[\sigma_{type}]} \cdot S \downarrow_\beta^\sigma} \text{(RBndSub)} \qquad \frac{(i \cdot S) \downarrow_\beta^\sigma \qquad \sigma_{term}(i) = s}{(s \cdot S) \downarrow_\beta^{(id,\sigma_{type}),\sigma}} \text{(RTermSub)}$$

$$\frac{(c[\rho] \cdot S) \downarrow_\beta^\sigma \qquad c \in \Sigma \text{ or } c := d \in \Sigma_\delta}{c \cdot S \downarrow_\beta^\sigma} \text{(RAtomClos)} \qquad \frac{(i_\tau[\rho] \cdot S) \downarrow_\beta^\sigma \qquad (\rho_{term} \circ \sigma_{term})(i) = j}{j_{\tau[\rho_{type} \circ \sigma_{type}]} \cdot S \downarrow_\beta^\sigma} \text{(RBndClos)}$$

$$\frac{(i[\rho] \cdot S) \downarrow_\beta^\sigma \qquad (\rho_{term} \circ \sigma_{term})(i) = s}{(s \cdot S) \downarrow_\beta^{(id,\rho_{type} \circ \sigma_{type}),\sigma}} \text{(RTermClos)} \qquad \frac{(h[\rho'][\rho] \cdot S) \downarrow_\beta^\sigma}{(h[\rho' \circ \rho] \cdot S) \downarrow_\beta^\sigma} \text{(RClosClos)}$$

Root rules

$$\frac{(\lambda_\tau.\ s) \downarrow_\beta^\sigma}{\lambda_{\tau[\sigma_{type}]}.\ s \downarrow_\beta^{(1.\sigma_{term} \circ \uparrow, \sigma_{type})}} \text{(Abs)} \qquad \frac{(\Lambda.\ s) \downarrow_\beta^\sigma}{\Lambda.\ s \downarrow_\beta^{(\sigma_{term}, 1.\sigma_{type} \circ \uparrow)}} \text{(TyAbs)} \qquad \frac{(s[\sigma']) \downarrow_\beta^\sigma}{s \downarrow_\beta^{\sigma' \circ \sigma}} \text{(Clos)}$$

Abstraction/Closure rule

$$\frac{(s \cdot \text{NIL}) \downarrow_\beta^{\sigma,\sigma'}}{s \downarrow_\beta^\sigma} \text{(RxSpNil)} \qquad \frac{(s \cdot S[\rho]) \downarrow_\beta^{\sigma,\sigma'}}{(s \cdot S) \downarrow_\beta^{\sigma,\rho \circ \sigma'}} \text{(RxSpClos)}$$

$$\frac{(s \cdot t; S_{tail}) \downarrow_\beta^{\sigma,\sigma'} \qquad s = \lambda_\tau.\ s'}{(s' \cdot S_{tail}) \downarrow_\beta^{(t[\sigma].\sigma_{term},\sigma_{type}),\sigma'}} \text{(RxApp)} \qquad \frac{(s \cdot \tau; S_{tail}) \downarrow_\beta^{\sigma,\sigma'} \qquad s = \Lambda.\ t}{(t \cdot S_{tail}) \downarrow_\beta^{(\sigma_{term},\tau[\sigma'_{type}].\sigma_{type}),\sigma'}} \text{(RxTyApp)}$$

$$\frac{(s \cdot S) \downarrow_\beta^{\sigma,\sigma'} \qquad s = h \cdot S'}{(h[\sigma] \cdot S'[\sigma] +\!\!+ S[\sigma']) \downarrow_\beta^{(id,id)}} \text{(RxRMrg)} \qquad \frac{(s \cdot S) \downarrow_\beta^{\sigma,\sigma'} \qquad s = t \cdot S'}{(t \cdot S'[\sigma] +\!\!+ S[\sigma']) \downarrow_\beta^{\sigma,(id,id)}} \text{(RxRxMrg)}$$

$$\frac{(s \cdot S) \downarrow_\beta^{\sigma,\sigma'} \qquad s = t[\rho]}{(t \cdot S) \downarrow_\beta^{\rho \circ \sigma,\sigma'}} \text{(RxClos)}$$

Redex rules

$$\frac{\text{NIL} \downarrow_\beta^\sigma}{\text{NIL}} \text{(SpNil)} \qquad \frac{(S[\rho]) \downarrow_\beta^\sigma}{S \downarrow_\beta^{\rho \circ \sigma}} \text{(SpClos)} \qquad \frac{(s_0; S_{tail}) \downarrow_\beta^\sigma}{(s_0 \downarrow_\beta^\sigma); S_{tail} \downarrow_\beta^\sigma} \text{(SpApp)} \qquad \frac{(\tau; S_{tail}) \downarrow_\beta^\sigma}{(\tau[\sigma_{type}]); S_{tail} \downarrow_\beta^{\rho \circ \sigma}} \text{(SpTyApp)}$$

Spine rules

Figure 4.4: β-normalization strategy **SpClos**

(B) *Strict (RxApp) rule*
Instead of using a closure in (RxApp), calculate $t[\sigma']$ directly by $t[\sigma'] = t{\downarrow}_{\beta}^{\sigma'}$.

Pro: The strict calculation avoids the introduction of term closure constructions which might cause overhead
Con: As for (A), the number of term traversals increase. Also, the normalized term might not be used.

(C) *Partial η-contraction inside of (RxApp)*
Replace the cons of $t[\sigma']$ to σ_{term} in (RxApp) by an η-expanded cons, i.e. by $\eta(t[\sigma']) \cdot \sigma_{term}$, where $\eta(.)$ is a meta-level function that applies η-reduction on the term's top level structure.

Pro: Reduce λ-abstractions that would cause more applications of (RxApp) after instantiation by (RTermSub).
Con: An extra term traversal of t is needed in order to identify if certain bound variables qualify for η-reduction.

(D) *Normalize substitutions before application in Root rules*
Before application of (RTermSub) or (RTermClos), normalize the current substitution σ such that all terms contained in σ are β-normalized.

Pro: When β-normalizing inside of σ, possibly multiple normalization runs for different instances of the same term are avoided.
Con: Exhaustive normalizing of all terms in σ might not be needed since some of the terms are possibly never substituted.

(E) *Normalize substitutions on-demand*
Same as (D), but do not normalize all terms of σ, but just those which are being instantiated by rule (RTermSub), and update σ accordingly.

Pro: Same as (D)
Con: The update of the substitution might be costly

A thorough evaluation of these normalization variants is needed to estimate their impact on the efficiency of the β-normalization routine. As a first step, the normalization strategy of Figure 4.4, denoted **SpClos**, is compared to the variant (A) (denoted **SpStrict** in the following) in §4.5.

Weak head normal form (WHNF) For some applications it might be sufficient to only calculate the WHNF of a term, instead of its full β-normal form. This might be the case for the application of inference rules to literals during the proof procedure, where it is often only necessary to inspect a literal's head symbol. When applying a

substitution to a literal, one could calculate the WHNF of that new literal and postpone the application of the substitution to the remaining subterms.

This notion of normalization can easily be supported by the presented term data structure: A term is normalized until it is of the form $h \cdot S$ and a substitution σ still needs to be applied to S. Then, σ can elegantly be stored as closure, yielding $h \cdot S[\sigma]$.

η-normalization For a β-normalized term s, given by

$$\lambda_{\tau_1}.\cdots.\lambda_{\tau_k}.\, h \cdot (E^1_{\nu_1};\ldots;E^m_{\nu_m})$$

with head h of type $\nu_1 \to \ldots \to \nu_m \to \ldots \to \nu_n \to a$ (where $a \in T$ is a base type), the η-long normal form of s is given by

$$\lambda_{\tau_1}.\cdots.\lambda_{\tau_k}.\lambda_{\nu_{m+1}}.\cdots.\lambda_{\nu_n}.\, h' \cdot (E'^1_{\nu_1};\ldots;E'^m_{\nu_m};(m+1_{\nu_{m+1}})\cdot \textsc{Nil};\ldots;1_{\nu_n}\cdot \textsc{Nil})$$

where $h' = h[\uparrow^{n-m}]$, and all $E'^i = E^i[\uparrow^{n-m}]$ are η-expanded. Hence, a certain number of abstractions needs to be introduced, and the spine is appropriately lifted and augmented with the de-Bruijn indices.

4.4 Shared Representation of Terms

Another key aspect of term managing appears when handling a big number of formulae at once. Regardless of whether a complex input problem file is read or several intermediate terms are generated throughout the proving process – an automated theorem prover often has to store and manage 10^5 - 10^6 terms during a single run [NHRV01]. This high number of terms might be critical concerning the performance of several heavy-weighted operations on terms, as, for instance, when sophisticated term indexing techniques are employed. Also, memory limitations may quickly be reached, depending on the term's internal representation.

In order to tackle the above problems, we employ a *perfectly shared* representation of terms which offers not only a reduced memory consumption, but more importantly the advantage of sharing knowledge about term properties (thus eliminating redundant recomputations). A perfectly shared term representation denotes a technique in which syntactical equal terms are represented by the (physically) same term structure instance in memory. This technique is used in several ATPs, such as E [Sch02] or LEO-II [TB06, BTPF08].

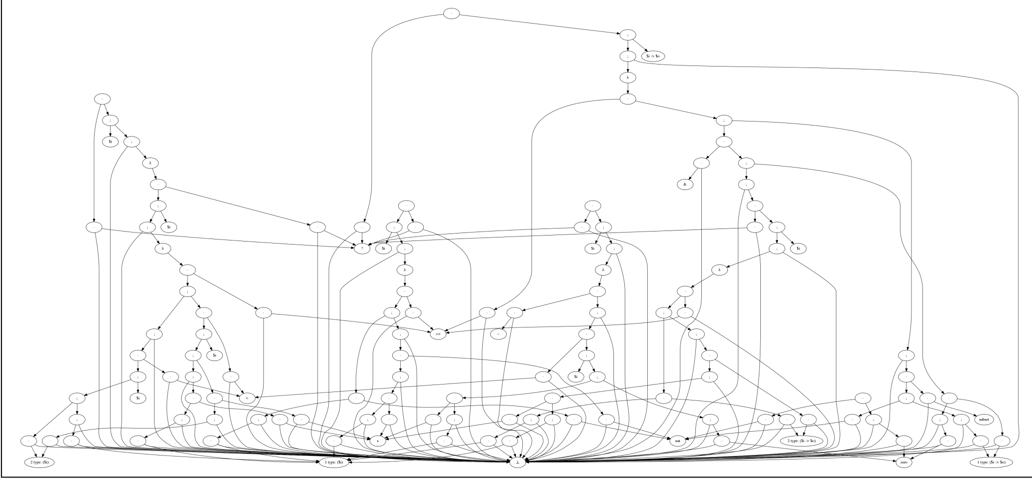
A data structure that manages terms in the above manner is commonly called *Term Bank*. The Term Bank collects all terms that have been created so far and offers operations to create new ones. Here, terms are only created and stored once and then reused as subterm occurrences among possibly many different terms. Terms can be created by composition operations, such as `mkTermApp(s,t)` or `mkTypeAbs(t)` (for creating (st) from s and t, and $\Lambda.\, t$ from t, respectively), or converted directly from a local representation (see below) by deep traversal (essentially a folding of the appropriate constructors).
Since terms are represented as tree-like structures, a set of terms can be represented by a *forest*. When sharing of subterms is considered, the forest is then condensed into

a directed acyclic graph (DAG) whose vertices are terms nodes and whose edges are subterm pointers. As an example, consider the shared representation of the five Peano axioms:

Example 4.1 (Shared Representation of Peano's axioms)
Let the terms $p_1, \ldots, p_5$ be given by

$$\begin{aligned} p_1 &= nat\, zero_\iota \\ p_2 &= \forall n_\iota.\; nat\, n \supset nat\,(s\, n) \\ p_3 &= \forall n_\iota.\; \neg(s\, n = zero) \\ p_4 &= \forall n_\iota \forall m_\iota.\; s\, n = s\, m \supset n = m \\ p_5 &= \forall p_{\iota\to o}.(p\, zero \wedge \forall n_\iota.\; p\, n \supset p\,(s\, n)) \supset subset\; nat\; p \end{aligned}$$

where we identify sets by its characteristic function, in particular $nat_{\iota\to o}$ representing the natural numbers and $s_{\iota\to\iota}$ represents the successor function, and $subset_{(\iota\to o)\to(\iota\to o)\to o}$ is chosen appropriately. The graph representation of the five terms $p1, \ldots, p5$ can be seen in the following picture:



As one can see, all constant symbols (such as $zero$ and nat), logical connectives ($=$ and $\supset$), bound indices and other term parts are shared between all terms. ⌟

We allow *local terms*, that is, terms which are not in a shared representation. This is due to performance considerations: Consider term manipulation tasks (i.e. tasks that create new (different) terms), which might turn out to be needless in some sense (e.g. represent intermediate results). Here, the insertion of the term to the Term Bank increases the search space for other operations and, additionally, takes at least linear time on the size of the term. Hence, Leo-III 's agents can create local terms whenever reasonable and later publish them into the Term Bank to make them globally visible and shared.

The characterization of the Term Bank data structure is given by

Data structure 2 (Term Bank) A Term Bank $\mathfrak{T} = \{Term\}$ is a set of terms together with the following operations

- `insert`$_{\mathfrak{T}}$: $Term \to Term$
  Converts the input term into a shared representation and stores it, yielding the reference of the (syntactically) equal term in shared representation.
- `get`$_{\mathfrak{T}}$: $\{Term\}$
  Returns a set of all stored terms.
- `mkAtom`$_{\mathfrak{T}}$: $Atom \to Term$
  Returns a term $t = c \cdot \textsc{Nil}$ that represents the input atom c as a term.
- `mkBound`$_{\mathfrak{T}}$: $Type \to \mathbb{N} \to Term$
  Returns a term $t = i_\tau \cdot \textsc{Nil}$ with input type τ and de-Bruijn index i.
- `mkTermApp`$_{\mathfrak{T}}$: $Term \to Term^* \to Term$
  Returns a term $t = f \cdot (t_1; t_2; \ldots; t_n; \textsc{Nil})$ with function term f and argument terms $(t_i)_{1 \leq i \leq n}$.
- `mkTypeApp`$_{\mathfrak{T}}$: $Term \to Type^* \to Term$
  Returns a term $t = f \cdot (\tau_1; \tau_2; \ldots; \tau_n; \textsc{Nil})$ with (polymorphic) term f and argument types $(\tau_i)_{1 \leq i \leq n}$.
- `mkApp`$_{\mathfrak{T}}$: $Term \to (Term + Type)^* \to Term$
  Same as `mkTermApp` and `mkTypeApp` but with mixed arguments (terms and types).
- `mkTermAbs`$_{\mathfrak{T}}$: $Type \to Term \to Term$
  Returns a term $t = \lambda_\tau. s$ with body s and abstracted variable of type τ.
- `mkTypeAbs`$_{\mathfrak{T}}$: $Term \to Term$
  Returns a term $t = \Lambda. s$ with body s.

All operations except `insert` expect that its term arguments are already in shared representation. ⌟

Implementation The term sharing graph is implemented using hash tables: For each sort of primitive symbols, i.e., bound indices and atoms, and for each composite node structure, i.e., abstractions, roots, spines, ..., a (possibly nested) hash table is employed. This allows for a fine-grained control of node sharing and, together with a set of pointers to already created terms, constant time check of (syntactic) term equivalence by pointer comparison. The following hash tables are employed:

- `boundAtoms` : $\mathcal{T} \to \mathbb{N} \to Head$ — for input parameters τ and n, store bound variable n_τ
- `symblAtoms` : $\mathbb{N} \to Head$ — for input parameter n, store appropriate constant symbol c (cf. §6.1)
- `roots` : $Head \to Spine \to \Lambda$ — for input parameters h and S, store root $(h \cdot S)$
- `redexes` : $\Lambda \to Spine \to \Lambda$ — for input parameters rx and S store redex $(rx \cdot S)$
- `termAbstractions` : $\Lambda \to \mathcal{T} \to \Lambda$ — for input parameters s and τ, store $(\lambda_\tau. s)$
- `typeAbstractions` : $\Lambda \to \Lambda$ — for input paramter s, store $(\Lambda. s)$

- `spines` : $(\Lambda + \mathcal{T}) \rightarrow \mathit{Spine} \rightarrow \mathit{Spine}$ — for input parameter s or τ and S, store $(s; S)$ or $(\tau; S)$

where $(\Lambda + \mathcal{T})$ denotes the sum type of terms and types.
The hash maps used for implementation are taken from Scala's collection library and guarantee effectively constant time access for term lookup and insertion of new terms. This is important since every call to a constructor (factory) method for shared terms invokes a constant number of queries to some hash tables; hence all `mkX` operations run effectively in constant time and `insert` takes time linear in the number of nodes in the argument term (since it can be implemented as a folding of the `mkX` operations).

Further remarks Additionally to perfect sharing of terms using a term bank, types could be treated in a similar way, yielding *perfect shared types* with the use of a *type bank*. In the higher-order setting, where types are excessively used in problem modelling, employment of shared terms could further reduce memory consumption. In particular, every term is associated a type and the representation of abstractions and bound variables also includes explicit type annotations. Implementation of perfectly shared types is a straight-forward extension and can, as for terms, represented by directed acyclic graphs.

The term bank data structure can also be used for *global unfolding of definitions*, where in every term a given defined constant symbol is replaced by its definiendum. This can be implemented by traversing the term bank and replacing each occurrence of that symbol. Due to the term sharing, that symbol is consequently replaced in all terms.

In literature, usage of term banks is often used synonymously with the concept of *term indexing* (cf. §5). More accurately, term banks address reduction of memory consumption and simple term comparison, whereas term indexing addresses the efficient *retrieval* of terms with respect to a certain condition. Nevertheless, both terms are closely related since the implementation of term banks offers several data structures that can be used within the term index. This observation is discussed in §5.2.1.

4.5 Evaluation of Term Representation

A first (preliminary) evaluation of the currently implemented term representation of $\lambda 2$ is presented and discussed. The benchmark measurements of this evaluation experiment focus primarily on giving evidence to the key benefits that are expected due to the combination of spine notation, explicit substitutions and perfect sharing (as described in previous parts of this section). A substantial overall improvement of the efficiency of common operations on terms is expected, compared to a naive term representation (standard curried λ-calculus representation without sharing) with simple β-reduction techniques (normalization by gradual substitution of bound variables). Here, the meaning of efficiency of course depends on the operation we are examining.

Since the Leo-III project is still at its very beginning and Leo-III's proof calculus is not set up yet, the evaluation examines the data structures and its operations isolated from the rest of the prover. Although these type of benchmarks might be somewhat biased given that operations are benchmarked out of context, the measurements are nevertheless meaningful and give rise to high potential for more realistic applications. This is due to the fact that common bottlenecks of theorem prover performance appear on term data structure level [Ria03]. We omit theoretical performance analysis as it tends to not reflect the observed efficiency results when applied to practical problems and, more importantly, the different behaviours of different problem domains [NHRV01].

The benchmarks considered here are (1) reduction count and time measurement of β-normalization, (2) reduction count measurement of head symbol queries and (3) space consumption of shared term representation in contrast to non-shared terms. For each time measurement, five independent benchmark runs have been carried out and the median of those results is presented. The choice for presenting the median value, referred to as *mean* value, instead of other parameters (i.e. arithmetic average) is that it is more robust against statistical outliers that might occur due to different CPU load during measurement. The remaining benchmarks have been run once, since reductions counts and space consumption are measurable in a deterministic fashion without any measurement error.

The input problems that are used to study the performance of the operations are chosen from a relatively broad field of diversity: Two sets of input problems, denoted *Church I* and *Church II*, contain arithmetic terms in polymorphic Church numerals encoding; the set *S4* contains all problems (that are, roughly, 3500 files) of the QMLTP library [RO12] as semantically embedded HOL formulas [BR13] (with respect to S4 semantics)[4]; the remaining sets are chosen from the TPTP problem library [Sut09]. Using these input problems, a variety of term characteristics is covered. In particular, rank-2 polymorphic terms (arithmetic operations on polymorphic Church Numerals), higher-order formulae with numerous occurrences of λ-term applications (S4), and "ordinary" small and large first-order and higher-order formulas are considered. The precise sets of terms are

Church I	The set of numeral terms of the form $mult \cdot (i; i)$, for $i = 5, \ldots, 100$
Church II	The set of numeral terms of the form $power \cdot (i; 3)$, for $i = 10, \ldots, 30$
S4	The complete set of S4-embedded QMLTP formulae in THF format
SET, GRA, QUA	The FOF, TFF and THF subset of the corresponding domain

The time measurements were taken on a 64-bit machine with Core i3-2120M, 4x2.5 Ghz.

In the remainder of the section the experiments and the results are described more thoroughly.

[4]The S4-encoding of the QMLTP problem library can be found at `http://page.mi.fu-berlin.de/cbenzmueller/papers/THF-S4-ALL.zip`

Normalization The normalization benchmark examines the reduction counts and the time that is required to calculate the β-normal form of each δ-expanded formula of each input problem of the corresponding domain. For that purpose, each problem file is parsed, transformed into the internal term representation and, finally, δ-expanded. Subsequently, the reduction count and time measurement for normalization is taken. Each domain is benchmarked with three different normalization strategies, one of them being the naive strategy on standard λ-terms, denoted **Naive**. The other two strategies are **SpClos** and **SpStrict** as described in Sect. 4.3. The reduction count results and the time measurement results can be found in Table 1 and Table 2 respectively.

	Naive			**SpClos**			**SpStrict**		
	Mean	*Min/Max*	*Sum*	*Mean*	*Min/Max*	*Sum*	*Mean*	*Min/Max*	*Sum*
Church I	83456	313/546848	14368968	13736	153/50408	1712528	9514	255/32650	1150800
Church II	4884	824/15244	126819	2110	560/4660	48160	1650	550/3350	36960
S4	10886	334/1672702	357470394	3997	157/315048	81319105	–[5]	–[5]	–[5]
GRA	96413	595/922889	22703578	6419	511/21379	864294	23351	511/350941	7964587
QUA	262	42/884	7300	139	37/424	3620	438	54/2349	13237
SET	571	1/222750	1317336	358	1/65084	788386	749	1/5345479	17782389

Table 1: Reduction count measurement of β-reductions

	Naive			**SpClos**			**SpStrict**		
	Mean	*Min/Max*	*Sum*	*Mean*	*Min/Max*	*Sum*	*Mean*	*Min/Max*	*Sum*
Church I	36854	1084/247132	6167083	11383	2433/46099	1321107	2536	942/11887	330718
Church II	2712	1439/7614	71696	11144	2589/17493	219424	2560	1690/3824	55012
S4	14738	307/4724178	732305877	2359	106/403043	44276842	–[5]	–[5]	–[5]
GRA	336145	600/7081224	119590722	2893	263/82841	598646	7739	211/197813	2544876
QUA	5281	347/29068	142064	1748	325/5642	40199	3901	290/20185	112355
SET	693	1/465028	2125999	355	1/93775	849116	390	1/2142305	6504471

Table 2: Time measurement of β-reductions in μs

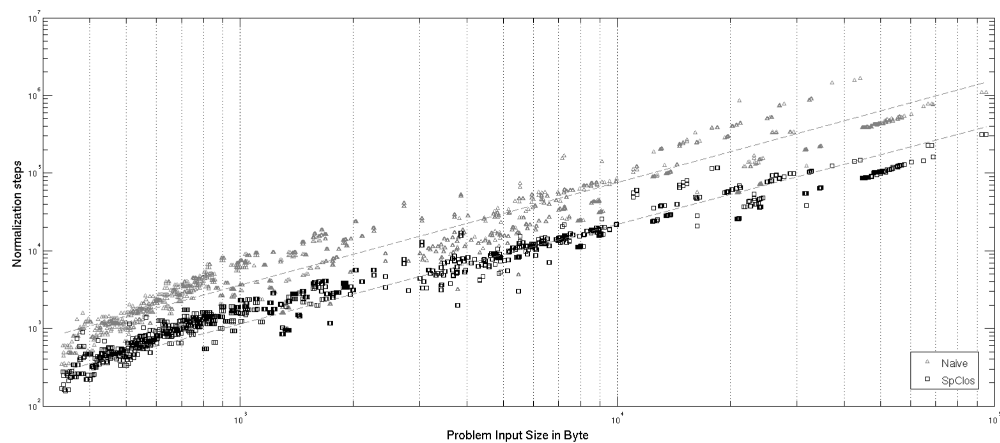
It shows that the normalization routine using explicit substitutions is dramatically superior to naive beta normalizing in all benchmark classes. When inspecting reduction counts, it can be seen that the mean "performance gain" (i.e. fraction of reductions saved) is up to 15.01 at peak (GRA) whereas the overall performance gain (the sum of all reduction steps over the whole problem domain) is with 26.27 even higher (also GRA). Interestingly, the reduction step performance in the benchmark ranges from relatively low (SET: 1.59 mean speed-up, 1.67 overall) up to outstanding results (GRA: 15.01 mean speed-up, 26.27 overall). Another observation is that, even though the term representation presented in this thesis performs better than the naive approach in all reduction benchmarks, the normalization strategy **SpClos** (derived from the normalization strategy stated in [ACCL90]) gives not best result in all benchmark domains: In Church I and Church II, the strategy **SpStrict** performed significantly better the **SpClos**. While comparing results of both strategies in the Church problem domains, we can observe a performance difference roughly of factor 1.3 - 1.5 (mean and overall) in favor of **SpStrict**.

The time measurements confirm the theoretical reduction count analysis: We can observe a speed-up ranging from 2.5 (SET) to 116.2 (GRA) on mean time and 1.3 (Church II) to 199.77 (GRA) at overall time measurement (cf. Table 2). The results

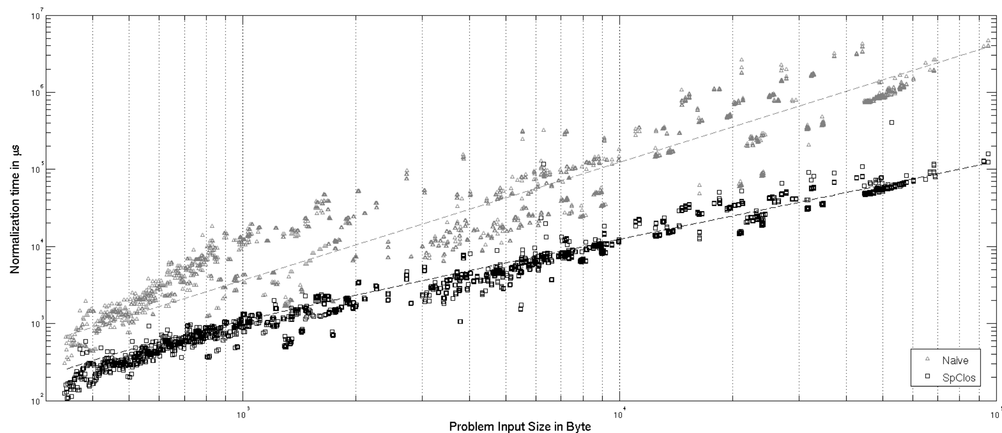
[5] The normalization strategy **SpStrict** performs quite poorly in the context of problem set S4, being widely out of range to be efficient.

of problem domains Church I and Church II clearly favor normalization strategy **Sp-Clos**, although there seem to be formulas in domain Church II that require even less normalization time in naive representation. Hence, the time measurements confirm that the most efficient normalization strategy depends on the problem structure. A careful analysis needs to be performed in order to identify which parameters affect the effectiveness of a particular normalization strategy or representation.

The tables of measurement data present somehow "flattened" results, in a sense that per-term differences in both term representation and normalization strategy cannot be overlooked in a fine-grained manner. Figure 4.6 and Figure 4.5 show the measurement data for each input problem of the problem domains Church I and S4 respectively.

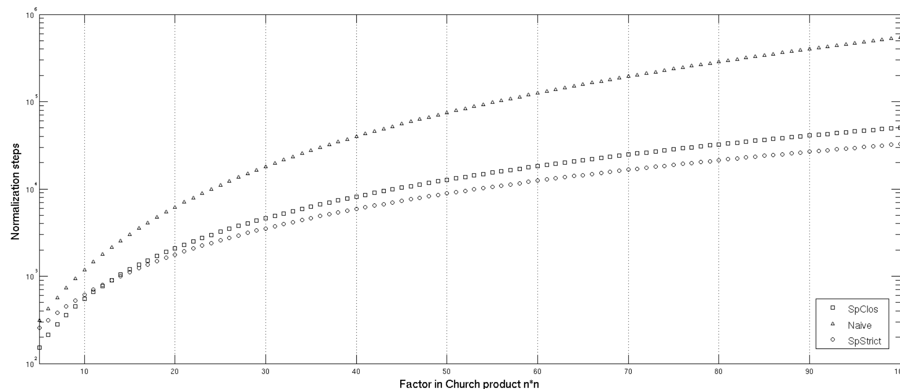


(a) Reduction step measurement

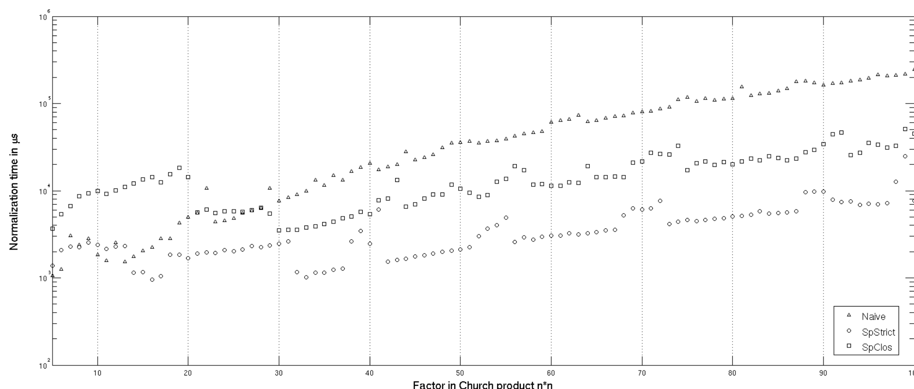


(b) Time measurement

Figure 4.5: Normalization benchmark of domain S4



(a) Reduction step measurement



(b) Time measurement

Figure 4.6: Normalization benchmark of domain Church I

Head symbol queries With this benchmark, the proposed improvement of head symbol access due to spine representation is tested. From a theoretical point of view, this operation should only require one step per formula – in order to estimate the impact in realistic application scenarios, the measurements are once more employed on the problem domains stated above. To that end, the reduction counts (i.e. traversal steps) of head symbol extraction are compared, see Table 3. As before, "Naive" denotes a simple, curried λ-term representation, whereas "Spine" denotes the term representation depicted in § 4.3.

Indeed head symbol queries require only one step. It can be seen that the overall number of reductions is significantly lower when using spine notation (reduction factor between 2 and 3). Note that the columns of the table refer to the mean (min/max) reduction count over all *problem files*, not formulas. Consequently, the mean reduction count reflects the mean number of formulae per problem file (excluding imported files, such as axioms).

	Naive			**Spine**		
	Mean	*Min/Max*	*Sum*	*Mean*	*Min/Max*	*Sum*
Church I	3	3/3	228	1	1/1	96
Church II	3	3/3	63	1	1/1	21
S4	21	3/422	232948	8	1/191	110618
GRA	3	3/21	878	1	1/7	302
QUA	3	3/3	60	1	1/1	20
SET	3	1/246	12821	1	1/110	4434

Table 3: Reduction count measurement of head-symbol query

Term sharing The final benchmark inspects the space consumption of terms in unshared and shared representation. Both space consumptions are approximated by (the sum of) the number of nodes within the graph structure of each problem term and the directed acyclic graph containing all terms of the benchmark problem (respectively). We claim that the number of term nodes is indeed a meaningful measure: The actual physical space consumption is of course dependent on the runtime environment the program runs on, but should behave linear to the number of term nodes. The notion of *term sizes*, as given by Definition 4.3, mimics the node count of the term's corresponding graph representation are is used throughout this benchmark to measure term sizes.

Definition 4.3 (Term size)
Let t be an arbitrary term. Then, the *size of* t, denoted $\#t$, is given by:

$$\begin{aligned} \#(h \cdot S) &= 2 + \#S \\ \#(r \cdot S) &= 1 + \#r + \#S \\ \#(\lambda_\tau .\ t) &= 1 + \#t \\ \#(\Lambda .\ t) &= 1 + \#t \end{aligned}$$

Extended to spines by

$$\begin{aligned} \#(\textsc{Nil}) &= 1 \\ \#(t; S') &= 1 + \#t + \#tail \\ \#(\tau; S') &= 1 + \#tail \end{aligned}$$

where $s, t, r\Lambda_\tau$, $\tau \in \mathcal{T}$, h and S, S' are terms, types, heads and spines respectively. ⌟

Subsequently, both node counts are compared for all problem files of problem sets S4, GRA, QUA, SET. Additionally, we inspect the term DAG's *density*. Density of a graph is in general defined to be the ratio of edges in that graph compared to the maximal number of edges. In the context of directed acyclic graphs $G = (V, E)$, the density of G, denoted d_G, is given by $d_G = \frac{|E|}{\binom{|V|}{2}} = \frac{2|E|}{|V| \cdot (|V|-1)}$. For unshared term graphs, each node is commonly connected to $1-2$ other term nodes with outgoing edges (e.g. abstraction node with one outgoing edge vs. head nodes with two outgoing edges), yielding density values less or equal than $\frac{4}{|V|-1}$. We compare the mean density of the term DAG to the maximal density of an unshared term graph forest of size n, where n is the mean number of nodes per problem. The quotient of the former and the latter value is denoted *Packing ratio* and describes the factor of density increasing.

	S4	**GRA**	**QUA**	**SET**
Mean/Min/Max number of terms	52/44/236	10/10/28	24/24/25	21/10/155
Mean Avg/Max size of terms	34.89/95.5	279.6/2597	28.68/69.5	39/100
Mean/Min/Max problem size	1900/1245/39142	2796/524/6140	689/645/819	886/226/10404
Mean/Min/Max number of DAG nodes	241/147/6572	458/59/1190	96.5/86/116	123/5/1502
Mean/Best/Worst space ratio	.127/.0803/.219	.174/.113/.194	.139/.133/.142	.136/.031/.166
Mean density of term DAG	0.035	0.017	0.087	0.059
Mean density of term forest [in 10^{-3}]	2.20595	1.431127	5.81971	4.88997
Packing ratio	15.866	11.879	14.949	12.066

Table 4: Results of term sharing measurements

Table 4 displays the results of the above described measurements. It can be observed that in all problem sets the memory consumption, even in worst case, is reduced by at least (roughly) 80%. In problem set S4 and SET the mean memory consumption is reduced by approximately 87% and 86%. There are also cases in which over 90% of the original space can be saved (S4: 92%, SET: 97%). From the perspective of graph density, the mean term DAG density exceeds the corresponding unshared term graph density roughly by a factor of 11 to 16. A crucial consequence of higher density is that more subterm properties can be shared between terms, such as normal form of indexing information. Also, due to the reduced memory consumption, a number of additional pre-computed values (e.g. heuristic weights) can be saved along with the terms.

4.6 Term Orderings

Term orderings (also called simplification orderings) provide an important technique for proving termination properties of rewriting systems and are of particular interest in certain proof calculi, as for instance in ordered paramodulation [NR99], for gaining completeness. Another aspect of term orderings is that they are used to restrict the search space and thus the number of inferences performed during refutation. In general, this is done by performing inferences in which terms of a certain weight are rewritten to "smaller' terms (i.e. terms of lower weight) with respect to a term-ordering $\prec$.

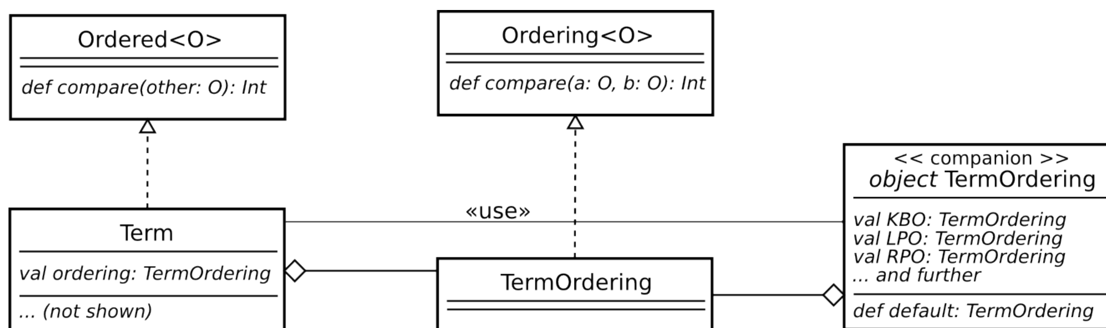


Figure 4.7: Leo-III's class architecture for native support of term orderings

The current implementation of Leo-III offers native support for employment of multiple term orderings. It is important that, inside the system, access to and use of various different orderings is offered in a well-structured way, since it might be necessary to choose the currently used term orderings depending on the input problem. The most popular orderings used in practice are the *Knuth-Bendix Ordering* (KBO), the *Recursive Path Ordering* (RPO) and the *Lexicographic Path Ordering* (LPO).

Figure 4.7 presents a class diagram of the generic term ordering architecture.

Here, we use a modular approach and encapsulate orderings in individual objects, here as realizations of the TermOrdering interface. Terms are marked as ordered, delegating ordering calls to a TermOrdering object that is defined to be "default" at a central position, e.g. in the TermOrdering companion object.
At the current state of development, specific term orderings are not yet implemented.

5 Term Indexing

In a large number of *information processing systems* pieces of information are (syntactically) manipulated, compared and created anew on the basis of a set of processing rules. In the context of automated theorem provers, those information chunks are usually represented by some collections of first-order or higher-order terms that are created, stored and looked up during the proof procedure. Similar information representations are used in term rewriting, functional programming and symbolic computation systems. Whereas the creation and plain storing of terms is, in general, not a challenging problem, the retrieval of already stored terms that are helpful in some sense poses a crucial efficiency requirement on the implementation of such systems [SRV01]. The judgment whether a stored term is helpful towards a given processing goal can be formalized by a retrieval relation called the *query condition*. This condition is formally defined by a relation R between each potential query result term in the term storage set I and some input element s. It can then be used to construct the result set of terms we are interested in, i.e. the set $\{t \in I \mid R(s,t)\}$. As the number of (intermediate) terms that are considered during proof search can easily reach magnitudes of $10^5 - 10^6$ terms, sophisticated techniques are needed to compute the above set in reasonable time [NHRV01]. It is apparent that a simple linear search through all terms does not scale for such large numbers. This is underlined by the observation of a phenomenon called *degradation*, as described by Larry Wos [Wos92]. Here the number of inferences in a proof procedure falls quickly as the time progresses. This is due to the fact that in certain automated theorem proving systems, the set of stored terms grows very quickly and thus the search for useful terms for the given goal taken more and more time. In numbers, L. Wos observed that after a few minutes of proving, the number of inferences drops to merely 1% of the inference rate of the beginning of the procedure.

To overcome this degradation and generally speeding up term look-ups, a common approach is employing an indexing data structure that stores terms in a way that certain queries can efficiently be served. This technique is known as *term indexing* and closely related to indexing from data base systems. Here, data base entries are indexed using B-trees or hash tables in order to support fast result retrieval for queries such as ”retrieve all data, where attribute `attr` equals v”.

In general, indexing a set of data (here terms) is used to allow quick access to sets for which a certain criterion, the query condition, holds. Let I denote the indexed set of terms, and Q the query set. The query condition can the be stated as a relation $R \subseteq I \times Q$. The general goal is then to find all data $I' \subseteq I$ s.t. $R(i,q)$ holds for all $i \in I'$ and $q \in Q$. This is also referred to *perfect indexing* since the result set is exact in the sense that it contains only those terms which match the query condition, but not more. Another common technique is considering *imperfect indexing*, in which a candidate set is returned which is a under-approximation, i.e. contains terms for which the query condition does not hold. The fact that the imperfect variant also returns false positives is evened out by simpler retrieval and storing algorithms.

The query set Q is often considered a singleton set containing the only query term,

for instance a term s for which all terms t are retrieved that are unifiable with s. This can easily be generalized to a query set containing more than one element, e.g. retrieval of all terms t in which all symbols $s \in Q$ occur. The former indexing is called $n : 1$, the latter $n : m$ (cf. Figure 5.1).

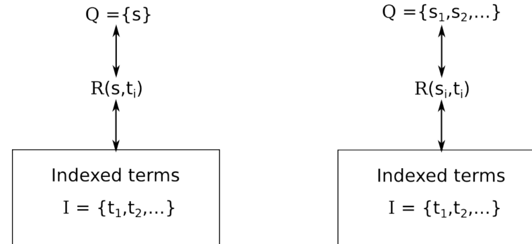


Figure 5.1: $n : 1$ indexing (left) and $n : m$ indexing (right)

A thorough discussion of different indexing techniques can be found in [Gra96] and [SRV01].

5.1 Indexing in Automated Theorem Proving

Although the efficiency of an automated theorem prover primarily depends on the effectiveness of the implemented proof calculus together with its search heuristics, employment of term indexing produces a reliable speed-up [Gra96].

As mentioned before, the term base that is stored during the proof procedure commonly contains a huge number of terms. This is not necessarily because the problem input is that complex, but rather due to the nature of the proof process. In the context of *saturation-based* theorem proving, the initial set of formulas constantly grows as the proof procedure essentially calculates the closure of those formulas with respect to a given set of inference rules. Also there exist simplification and redundancy elimination rules that replace or even delete terms from the term set.

In order to identify candidate terms for which certain inference and simplification rules are applicable, appropriate retrieval operations can be performed on the term index. Common retrieval queries are based on unifiablity, instance and generalization check, and subsumption. Those query conditions can be formalized by relations `unif`, `inst`, `gen` and `sub`, respectively, given by

$$\texttt{unif}(s,t) :\Leftrightarrow \exists\sigma : s\sigma = t\sigma$$

$$\texttt{inst}(s,t) :\Leftrightarrow \exists\sigma : s = t\sigma$$

$$\texttt{gen}(s,t) :\Leftrightarrow \exists\sigma : s\sigma = t$$

$$\texttt{sub}(c,d) :\Leftrightarrow \exists\sigma : c\sigma \subseteq d, \|c\| \leq \|d\|$$

where σ is a substitution, s, t are terms and c, d are clauses.

Additionally to the above given relations, there exists a broad variety of advanced query conditions for more specialized purposes. As an example, priorities can be included when

terms that are *simpler* are to be preferred during retrieval, e.g. retrieval of all terms s with

$$R(s,t) \wedge \forall s'.priority(s') > priority(s) \Rightarrow \neg R(s',t)$$

where *priority* is defined appropriately.

Term indexing is used in all major theorem provers, such as Vampire [KV13], E [Sch02], OTTER [McC90], LEO-II [BPTF08], and Satallax [Bro12]. As an example, Larry Wos observed that with its use of term indexing, program degradation in OTTER could be improved in such a way that after 19 hours of reasoning, it still runs inferences at a rate of 460 per second (while it started with 550 inferences per second) [Wos92].

First-order ATP In first-order theorem proving, robust indexing techniques exist and are thoroughly researched. Popular techniques include path indexing, substitution tree indexing, context tree indexing and many others. A survey can be found in, e.g., [SRV01].

Higher-order ATP In higher-order theorem proving, there exist only a few indexing approaches. This is due to the fact that most operations for building a term index are undecidable, e.g., computing the most specific generalization, or higher-order unification. For the latter case, however, there exists a decidable unification fragment, so called higher-order pattern unification [Mil91a], but algorithms for those fragment are highly complex and seem not to be efficient in practice [PP03].

A popular exception for indexing in the higher-order case is substitution tree indexing based on *linear* higher-order patterns [Pie09]. Another approach is taken by the LEO-II prover, where term indexing is based on rather low-level retrieval operations [TB06].

5.2 Term Indexing of LEO-III

The term index of LEO-III supports fast retrieval of terms with respect to a number of different query conditions. All the term retrieval techniques surveyed in this section rely on rather low-level indexing data structures, primarily on (cascaded) hash tables and sets.

In contrast to the functional programming language setting of LEO-II, we can exploit the object-oriented paradigm of the SCALA language for assigning certain *local* indexing structures to each individual term itself. Hence, data structures for indexing can be employed as an attribute (field) of a given term object, as long as the corresponding queries can be answered based on the sole knowledge of that particular term. Retrieval from such an indexing structure can then be done by simple field or method access instead of making a detour using hash tables, including the calculation of hash values. More complex (non-local) retrieval operations are, however, implemented by dedicated data structures (e.g. hash tables) gathered within the term index structure.

An invariant of the term index is that all stored terms are kept in $\beta\eta$-normal form (here: β-normal and η-long). The reason for preferring η-long form over the corresponding short variant is that it simplifies internal term handling: While in higher-order logic

it is allowed (and often perfectly reasonable) that a function symbol is only applied to *some* arguments, exhaustive η-expansion guarantees that all function symbol occurrences are applied to their maximal number of arguments. Consequently, we don't have to cope with partial argument application in pattern matches and other case distinctions. Another benefit of considering η-expanded terms only is that, during β-renormalization, the normalization rules (RxRMrg) and (RxRxMrg) (cf. Figure 4.4) can be omitted [CP03]. As a further example, Huet's (pre-)unification algorithm can be greatly simplified in the presence of η-long normal forms [Hue75].

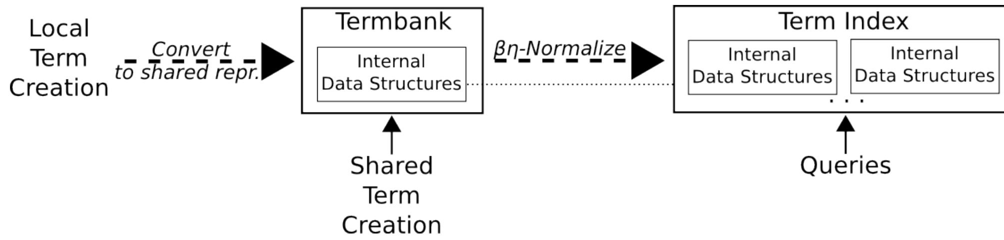


Figure 5.2: Outline of the term insertion process into Leo-III's term index

Figure 5.2 outlines the insertion of new (potentially non-normalized) terms to the index. If the term to be inserted is in local representation (i.e. not included in the shared term bank), it is first converted to shared representation. Subsequently, the input term is $\beta\eta$-normalized and, finally stored in the index data structures.

As an example, insertion of the term $t := ((\lambda R_{\tau\to\tau\to\nu}.\ R\ x_\tau)\ r_{\tau\to\tau\to\nu})$ into the term index involves calculating its $\beta\eta$-normal form, given by $t{\downarrow_{\beta\eta}} = (\lambda Y_\tau.\ r\ x\ Y)$[6]; it is assumed that t is already in shared representation.

As for the current state of development, indexing based on term structure and function symbols, subterm indexing to a certain extent, and a first draft of bound variable indexing is employed. In the remainder of this section, the term indexing techniques of Leo-III are discussed more thoroughly.

5.2.1 Term structure indexing

Term retrieval based on structural properties of terms are strongly linked to query conditions used in a relational database setting. We can, for instance, select all term abstractions which introduce a function that takes a parameter of given type τ, or retrieve all terms that are applications of a particular function f_τ on some arguments.

This kind of indexing requires no introduction of further data structures, since the implementation of the Term Bank already offers that kind of functionality. While the term index abstracts the access to those query operations, it uses, under the hood, the hash tables of the Term Bank.

[6]For reasons of legibility, the term is presented in common curried form

Recall the following hash tables of the Term Bank (cf. §4.4)

- `roots` : $Head \to Spine \to \Lambda$
- `termAbstractions` : $\mathcal{T} \to \Lambda \to \Lambda$
- `spines` : $(\Lambda + \mathcal{T}) \to Spine \to Spine$

Retrieving of terms by their structure is then done by simple table look-ups and intersection/union of results tables, depending on the query condition. One possible application may be arise while (pre-)unification constraint solving, when searching for solutions of a *flex-rigid-pair* [Ben99]: Here one may search, in addition to the use of *general bindings*, for all known appropriate terms in the term set with a head symbol given by the top-level symbol of the rigid side. Those terms can be found by queries to the hash table `roots`.

5.2.2 Symbol based indexing

A rather traditional approach to indexing is *symbol based indexing* [Gra96] in which the function symbols occurring in a given term are stored in such a way that all symbols of a given term and all terms containing a given symbol can be retrieved efficiently.

In the context of Leo-III, the fast retrieval of all function symbols of a given term is implemented as a attribute of the term itself. This set is build inductively during term construction and uses hash tables (for recording the number of occurrences per function symbol). Additionally, the head symbol of a given term is stored, again, as an attribute of the term itself; yielding

- $\texttt{funcsymbols}_t$: $\mathbb{N} \to \mathbb{N}$ — Hash table that stores to each function symbol (first parameter) the number of occurrences of that symbol
- $\texttt{headsymbol}_t$: $Head$ — Attribute value containing the head symbol of t

Note that symbols contained in the signature are encoded as natural numbers (cf. §6.1).

Both function symbol and head symbol indexing are actually independent of the term index structure. For terms not contained in the index, this attributes are computed lazily, i.e. on first access. During indexing, the calculation of both values is enforced to ensure constant time access.

The index further supports retrieval of all terms in which a given symbol occurs. To that end, a further hash table is introduced (where we write A^* for *collections of elements* of type A):

> `symbol_of` : $\mathbb{N} \to \Lambda^*$ — Hash table that stores for each constant symbol all terms in which that symbols occurs (at any position)

Hence we can effectively find all term with a given symbol using a table look-up. If we are interested in the set of terms in which all symbols $\{c_1, c_2, \ldots\}$ occur, we can easily retrieve the candidate sets using `symbol_of` and then intersect all results.

Symbol based indexing can, e.g., be used to implement an efficient term matching pretest: A term s can only be an instance of t, if the number of function symbols of s is greater or equal to the number of function symbols of t.

5.2.3 Subterm indexing

Since Leo-III is outlined as a theorem prover based on (ordered) paramodulation/superposition [WSB14], we also consider explicit indexing of subterms. This is due to the fact that in paramodulation based theorem proving, we are interested in *all subterms* of a given term with a certain property (e.g. being unifiable with, an instance of, or equal to a specific term) [NR99]. Subterm positions are given in Dewey decimal notation (see, e.g., [DJ90]), formally given by:

Definition 5.1 (Subterm Positions)
For a term $s \in \Lambda$ the *subterm of s at position p*, denoted $s|_p$, is the term t that is achieved by traversing the term tree of s, starting from the root, as given by the *position string p*. A position string p is a finite concatenation of natural numbers, indicating which subtree is traversed at each step, where ε denotes the empty sequence. Formally,

$$
\begin{aligned}
t|_\varepsilon &= t \\
(s \cdot S)|_{i.p} &= \begin{cases} s|_p & \text{, if } i = 1 \\ S|_p & \text{, if } i = 2 \end{cases} \\
(\Lambda.\ s)|_{1,p} &= s|_p \\
(\lambda_\tau.\ s)|_{1,p} &= s|_p \\
(S_1; S_2; \ldots; S_n)|_{i,p} &= S_i|_p
\end{aligned}
$$

⌟

We employ subterm indexing by means similar to *coordinate indexing* [Sti89] and store for each term t, the subterm $t|_p$ occurring at position p and, for each subterm t', in which superterms (and positions) it occurs. Due to the perfectly shared term representation (cf. §4.4), the indexing of subterms can be realized in a straight-forward manner. The key observation here is, that indexing of subterms of a particular term (and its subterms) need only to be done once, because further occurrences of a particular term are already indexed (due to its shared representation).

The implementation is similar to that one of LEO-II's subterm index [TB06]. One difference is that the storage of subterm occurrences of a given term t is implemented as property (here: as hash table) of the term itself, hence as the field

$\texttt{occurrences}_t : \Lambda \to Pos^*$

Therefore, only a single hash table per term and only one look-up per subterm query is required. In contrast, LEO-II uses cascaded hash tables.

Additionally, the term index contains the following hash tables for storing all terms in which a given subterm occurs:

- `occurs_in` : $\Lambda \to (\Lambda, Pos)^*$ — Hash table that stores for each argument term in which terms (and at which position) the former term occurs as a subterm

- `occurs_at` : $\Lambda \to Pos \to \Lambda^*$ — Hash table that stores for each argument term and path, in which terms the former term occurs at the given position

Although the information stored by the former hash table can be reconstructed using the latter, that information is additionally stored. This is due to performance considerations: In order to retrieve all positions and terms in which a given term t occurs as subterm, the result of `occurs_at` applied to t, a hash table itself, needs to be traversed for computing the union of each individual result set. This takes linear time in the number of terms in which t occurs as a subterm, whereas direct retrieval from `occurs_in` can be done in effectively constant time.

5.2.4 Bound variable indexing

In order to further speed-up β-reduction, a simple indexing technique that keeps track of bound variables within a term is employed. Each term node in the index is assigned a *scope number* that represents the maximal de-Bruijn index occurring in the term, which bind to a λ-binder outside of it (so-called *loose* bound variables). Hence, the scope number of a bound variable index i (i.e. the term $i \cdot \textsc{Nil}$) equals to $-i$, since its λ-binder is located in a superterm i abstractions above it. The scope number of a constant symbol ($c \cdot \textsc{Nil}$) is 0 as there exists no corresponding λ-binder to that term. The scope numbers of the remaining term nodes are determined inductively: If terms s and t_i have scope number n and m_i (respectively), then the

- term abstraction $(\lambda_\tau.\ s)$ has scope number $n - 1$,
- term application $s \cdot (t_1; t_2; \ldots; t_n; \textsc{Nil})$ has scope number $\min\{n, m_1, \ldots, m_n\}$,
- type abstraction $(\Lambda.\ s)$ has scope number n, and
- type application $s \cdot (\tau; \textsc{Nil})$ has scope number n.

Intuitively, if the scope number of a term is zero or positive then the term is closed, i.e. contains no loose bound variables referring to a binder outside, thus does not need to be considered during re-normalization. If the scope number is negative, then this term needs to be traversed during re-normalization since it *may* contain De-Bruijn indices.

As an example, Figure 5.3 displays selected scope numbers of the term $(\lambda.\lambda.\ f \cdot ((1 \cdot \textsc{Nil}); (\lambda.\ 1 \cdot 2); (2 \cdot \textsc{Nil}); \textsc{Nil}))$ (where types and some $\textsc{Nil}$s are omitted). The idea is that certain subtrees of the term's syntax tree can be skipped during normalization traversing, thereby reducing traversing and, therefore, normalization time.
The concept of scope numbers is extended to types as well. In a polymorphically typed λ-calculus, bound variables also occur in types and might need to be substituted during normalization. This *type scope number* is defined analogously to the scope number above, yielding the term property

$$\texttt{scopeNumber}_t : (\mathbb{N}, \mathbb{N})$$

The scope number of a term t is determined inductively during term construction.

Figure 5.3: Simplified example of scope numbers for bound variable indexing

Considering scope numbers as described above during re-normalization is an under-approximation, as subterms may be traversed although no meaningful substitution of bound variables occurs within it. This is due to the fact that potentially many different bound variables can be substituted during the same term traversal, and, without inspection of the current substitution, it cannot be decided if a particular subterm traversal can safely be skipped. This situation is also shown in Figure 5.3, where the first subtree of the spine needs to be traversed although no meaningful substitution is carried out.

In contrast to LEO-II's bound variable indexing, where for each λ-binder a partial syntax tree of the original term is stored [TB06], this method seems unpractical in this situation. This is because possibly a lot of different bound variables are substituted during one term traversal and it is not clear how to navigate through a set of partial syntax trees at the same time. This could, however, be addressed by investigating how efficient partial syntax trees can be merged.

A first draft of the above surveyed bound variable indexing is currently implemented. In order to achieve a substantial speed-up, this method has to be improved. One possible improvement involves replacing scope numbers by a collection of all de-Bruijn indices occurring within it. This can then be used during β-normalization for checking against the current substitution whether traversing that term tree actually makes sense.

5.3 Summary and Preliminary Evaluation

A provisional implementation of the term index (as described in the previous section) is currently implemented in the Leo-III project. A summary of the retrieval methods of Leo-III's term index is given by

Data structure 3 (Term Index) The term index data structure $\mathfrak{I}$ has the following operations:

- `byHeadsymbol`$_{\mathfrak{I}}$: $Head \to \Lambda^*$ — return all terms that have a given symbol (first argument) as head symbol
- `abstractionsOfType`$_{\mathfrak{I}}$: $\mathcal{T} \to \Lambda^*$ — return all terms that are abstractions with a given abstracted type (first argument)

- $\texttt{bySymbol}_{\mathfrak{I}}$: $Head \rightarrow \Lambda^*$ — return all terms that contain a given symbol (first argument)
- $\texttt{bySubterm}_{\mathfrak{I}}$: $\Lambda \rightarrow (\Lambda, Pos)^*$ — return all terms (and positions) in which the first argument occurs as subterm
- $\texttt{bySubtermAtPos}_{\mathfrak{I}}$: $\Lambda \rightarrow Pos \rightarrow \Lambda^*$ — return all terms in which the first argument occurs as subterm at a given position (second argument)

The remaining retrieval operations

- $\texttt{headsymbol}_{\mathfrak{T}}$: $Head$ — the head symbol of the term
- $\texttt{occurences}_{\mathfrak{T}}$: $\Lambda \rightarrow Pos^*$ — all subterm occurences inside the term
- $\texttt{symbols}_{\mathfrak{T}}$: $\Lambda \rightarrow Head^*$ — all symbols occuring inside the term
- $\texttt{scopeNumber}_{\mathfrak{T}}$: $\mathbb{N}$ — the scope number of the term

are performed on the term data structure itself. ⌟

At the current state of development, the term indexing of LEO-III cannot be evaluated in a meaningful way. Firstly, the implementation of the term index needs to be thoroughly tested for robustness. Secondly, no currently available proof procedure makes use of the term index at this time. There are, however, certain operations for which we predict a performance gain:

The paramodulation-based proof procedure can strongly benefit from the subterm indexing, since occurs checks and replacements in given terms are more efficient: An occurs check can easily be performed on a given term t, simply by a look-up operation of the corresponding $\texttt{occurrences}_t$ table, which takes constant time. The result information contains all positions of subterm occurrences and can thus be used for replacement operations. Here, branches in which the given term does not occur are skipped during traversal.
Also, general occurs checks can efficiently be performed on non-primitive terms.

A particular interesting question is to what extent the here presented bound variable indexing contributes for the speed-up of β-reductions. Here, intensive tests and benchmarks need to be performed.

Another question is whether the collection data structures taken from the SCALA library are well-suited for our purposes. Depending on the kind of queries, different advanced data structures might, in practice, perform better. This could even be the case, if that structure's theoretical performance is worse (i.e. worst case time complexity for certain operations). A thorough empirical analysis is required to justify certain data structure choices.

6 Further Data Structures

The data structures mainly focused in this thesis have been discussed in the previous sections. Nevertheless, further data structures are needed to operate an automated theorem prover such as LEO-III. To this end, an overview of these data structures is presented in this section, closing the survey of necessary structures.

6.1 The Signature

The management of a signature is an important component of a theorem prover. The signature data structure stores all logical symbols (such as connectives, constant symbols and definitions) given by the underlying logic as well as those used by the current problem. Typically, the signature contains the fixed and defined symbols of the logic by default and stores all processed problem-specific symbols on-demand.

Formally, a signature for a many-sorted logic is given by

Definition 6.1 (Signature)
Let L be a language, $S \subseteq \Sigma^*$ the set of symbols of L, and $\tau : S \rightarrow \mathcal{T}$ a function assigning every language symbol to its corresponding type.
Then, the structure $\boldsymbol{\mathcal{S}}_L = (S, \mathcal{T}, \tau)$ is called *Signature of L*. $\boldsymbol{\mathcal{S}}_L$ may be abbreviated by $\boldsymbol{\mathcal{S}}$ if L is clear from the context. ⌟

Note that a signature is also often called Σ in literature.
For a practical implementation of the signature, several additional information needs to be stores along with the symbols. Those information data contains not only the name (in terms of a string representation) and the type of the symbol, but also the following:

- The "nature" of the symbol: Is it a primitive (logical) symbol that is fixed by the logic, such as, say, $\sim$ (negation) or $|$ (disjunction), or is it a defined symbol that can be expanded to its appropriate definition, like & (conjunction) with definition $\lambda_o\lambda_o. \sim \cdot(| \cdot (\sim \cdot 2; \sim \cdot 1))$. Further sorts include uninterpreted symbols and base type symbols.

- The kind of the symbol, if applicable (can be used for general type constructors in a later phase of the LEO-III project).

- Symbol properties such as commutativity or similar

- Symbol source: On which occasion was the symbol added to the signature? During problem processing, for skolemization and by user interaction?

There are many more types of information that can be thought of as helpful for employment at different tasks apart from the actual reasoning process. That is why we assign each symbol a generic information annotation, called *Meta*.

Additionally, each symbol will be replaced by a certain key that is used to identify that specific symbol. This is to encode it as a shorter and efficiently comparable representation. Firstly, operations on strings are costly, so simple (syntactic) equality-checks may slow down access to the signature. Secondly, user-defined names of constants and functions may be very long. In order to circumvent this aspects, the signature data structure essentially consists of two dictionaries: One for mapping the original name to a unique, internally used key; and one for mapping those keys to their assigned *Meta* – yielding the signature data structure $\mathfrak{S}$:

Data structure 4 (Signature) The signature data structure $\mathfrak{S}$ is given by the pair $\mathfrak{S} = (Id \to Key, Key \to \mathfrak{M})$ with some key set Key, a $Meta$ structure $\mathfrak{M}$, and the following operations:

- $\texttt{addFixed}_{\mathfrak{S}} : Id \to \mathcal{T} \to Key$ — adds a fixed logical symbol with given name and type (first and second argument, respectively)
- $\texttt{add}_{\mathfrak{S}} : Id \to \mathcal{T} \to \mathfrak{T} \cup \{\bot\} \to Key$ — adds a symbol with a given name (first argument) and a given type (second argument) and possibly a given definition
- $\texttt{exists}_{\mathfrak{S}} : Id \to Bool$ — returns whether a constant symbol with the given name (first argument) exists
- $\texttt{meta}_{\mathfrak{S}} : Key \cup Id \to Meta$ — return according meta data structure

The structure $\mathfrak{M}$ has at least the following operations:

- $\texttt{nameOf}_{\mathfrak{M}} : Id$ — name of the symbol
- $\texttt{symbolSort}_{\mathfrak{M}} : \{Fixed, Uninterpreted, Defined, Type\}$ — "nature" of the symbol
- $\texttt{typeOf}_{\mathfrak{M}} : \mathcal{T} \cup \{\bot\}$ — the term of the symbol, if it is a constant symbol
- $\texttt{kindOf}_{\mathfrak{M}} : \mathcal{T} \cup \{\bot\}$ — the kind of the symbol, if it is a type symbol
- $\texttt{defOf}_{\mathfrak{M}} : Term \cup \{\bot\}$ — return the definition of the symbol, if it exists

where $\bot$ denotes an undefined value in this context. There may be more operations on $\mathfrak{M}$ depending on which information are saved along with a symbol. ⌟

The signature data structure is implemented using a hash table and a integer map for storing the name-to-key-dictionary and the key-to-meta-dictionary, respectively.

6.2 Literals and Clauses

Further important data structures include the effective representation of literals and clauses which are intensively used and manipulated during the proof procedure. An important aspect for both data structures is the retrieval of the "best" literal with respect to a given heuristic. These heuristics are in general given by a function that assigns each clause (and literal) a *weight* based on certain characteristics.

Data structure 5 (Literal) The literal data structure $\mathfrak{L}$ offers at least the following operations:

- $\texttt{weight}_{\mathfrak{L}} : \mathbb{N}$ — heuristic priority measure

- $\texttt{polarity}_{\mathfrak{L}} : \{+, -\}$ — the literal's polarity
- $\texttt{term}_{\mathfrak{L}} : \mathit{Term}$ — the underlying term
- $\texttt{isUnifConstraint}_{\mathfrak{L}} : \mathit{Bool}$ — returns whether the literal is a unification constraint
- $\texttt{isFlexFlex}_{\mathfrak{L}} : \mathit{Bool}$ — returns whether the literal is a *flex-flex* unif. constaint
- ... *and further* ⌟

As usual, literals are assigned a polarity and an underlying term that represents the literal itself. Structural query operations may also be included to the literal data structure, such as operations determining whether the literal is a (*flex-flex*) unification constraint.

Clauses are disjunctive concatenation of literals and can be represented by sets of literals and, consequently, also offers a weight heuristic function.

Data structure 6 (Clause) The clause data structure $\mathfrak{C}$ offers at least the following operations:

- $\texttt{weight}_{\mathfrak{C}} : \mathbb{N}$ — heuristic priority measure
- $\texttt{id}_{\mathfrak{C}} : \mathbb{N}$ — monotonously increasing identifier for clauses
- $\texttt{source}_{\mathfrak{C}} : \mathit{Source}$ — additional information about the clause's source
- $\texttt{literals}_{\mathfrak{C}} : \mathfrak{L}^*$ — the set of underlying literals ⌟

The choice of weight functions for clauses is one of the most crucial points in heuristic search for efficient proof procedures [Sch02] and many different approaches are known. A simple but rather effective approach counts all symbols depending on their arity, or prefers older clauses over newer ones. The first weight function can directly be implemented using the function symbol index of each literal's underlying term structure, the latter is supported by the unique clause counter $\texttt{id}_{\mathfrak{C}}$.

During the proof procedure, not only clauses but rather *sets of clauses* are maintained to store intermediate results. In order to provide quick access to "best clauses" (with respect to the weight function) within that set, an appropriate data structure needs to by employed for representing those sets. Depending on the performed queries, ordered sets can be represented using heaps or (balanced) search trees. A rather popular example for the latter approach are *splay trees* [ST85].

6.3 Input preprocessing

Another important component in automated theorem proving is careful *preprocesssing* of input problems. Certain transformations reduce the complexity of subsequent steps and make the problem more probable to be solved, although the original semantics is maintained. Common transformations range from rather simple techniques such as simplification and normalization (e.g. to negation normal form) to more involved ones include miniscoping, relevance checks, redundancy detection, and equality substitution. Extensive use of the preprocessing techniques need to be employed, possibly with dedicated control mechanism that allow to enable or disable certain transformation steps.

7 Conclusion

In this thesis, an internal term representation for polymorphically typed λ-terms based on spine notation and explicit substitutions is given. In contrast to first-order term representations, a naive adoption of λ-term structures for higher-order terms is problematic and lacks support for efficient term traversal and fast β-reduction. The development of sophisticated data structures for term representations are, together with intensive use of indexing techniques and search heuristics, key aspects of an efficient state-of-the-art theorem prover.

Costs for term normalization are one important aspect that are tackled in this thesis with the employment of explicit substitutions. The fine-grained control over normalization routines and the combination of substitution runs are a major benefit of introducing substitutions as part of the term language, rather than on a meta-level.

On the other hand, term traversal is the most crucial bottleneck of nearly all terms operations that are performed during the proof procedure (such as unification steps, equality checks, head symbol queries) and must therefore be carried out efficiently. An adaption of the spine notation to a polymorphically typed language is used to allow efficient left-to-right traversal of term graph structures. In particular, this representation guarantees constant time access to a term's head symbol.

Both of the above techniques are merged with perfect sharing, yielding the preliminary term data structure for the Leo-III prover. The evaluation measurements are promising, as a substantial number of reduction steps are saved during common operations such as β-normalization, and head symbol queries. Simultaneously, the memory consumption of terms is reduced by roughly 90%. Additionally, properties of subterms are shared between all occurrences and, hence, only need to be computed once. The use of de-Bruijn indices further simplifies internal term operations and allows, in combination with perfect sharing, constant time equality checks of terms by a simple pointer comparison.

Subsequently, another important data structure, the term index, was discussed and presented. The term index of Leo-III supports fast retrieval of terms with respect to a number of different query conditions. All the term retrieval techniques rely on rather low-level indexing data structures, primarily on (cascaded) hash tables and sets. Important features are indexing of function symbols and subterm occurrences. The indexing of bound variables within a term can be used to speed up β-normalization, but intensive improvements need to be done at this point in order to achieve substantial advantages.

Further important data structures for automated theorem proving are briefly discussed in the last part of this thesis. It is pointed out that effective search heuristics, e.g. for clause selection, are crucial.

At the current state of development, preliminary implementations of the described data structures (except for clauses and literals) are given in the Scala language.

7.1 Related Work

There are only a few systems with built-in support for polymorphism as presented in this thesis. Popular exceptions are Coq [Pau11] and the TWELF system [PS99] (based on Conquand's calculus of constructions) and Isabelle [NWP02] which offers a rich type system including polymorphism and type classes.

Term representations are intensively studied for the first-order case: *Flatterms* [Chr93] are linear term representations similar to singly linked lists that allow efficient left-to-right (preorder) traversal. They include pointer to next subterms for fast subterm skipping. However, this representation does not allow sharing of identical subterms.

Prolog terms [DEDC96] follow a similar approach as in spine notation. They are an optimized version of conventional term graphs, where function nodes are of variable size and store all pointers to child terms (arguments). A particular interesting choice is the notion of variable nodes, which are essentially a pointer to themselves. This construction allows constant time variable instantiation by resetting the pointer to the intended term node. This is, however, a destructive term manipulation.

Early systems that employed a higher-order term representation were Elf [Pfe94] and λProlog [Mil91b]. In contrast to the data structures discussed in this thesis, they chose a term representation that reflects the theoretical term definition, including the discussed disadvantages.

Spine representation is used in major systems, such as the reimplementation of Elf, the TWELF system, or LLF [CP02]. Also, the teyjus implementation of λProlog uses a spine calculus [Nad01]. TWELF also employs explicit substitutions based on $\lambda\sigma$, but for the case of dependent types.

Since the original publication of explicit substitutions of $\lambda\sigma$, a large number of alternative systems have emerged, with rather different theoretical properties (e.g. confluence and strong normalization). A notable system for explicit substitutions is the Suspension Calculus [Nad96], which also supports fusion of substitutions. It is used as a basis for an implementation of λProlog and evaluated in the context of different normalization strategies in [LNQ04].

Perfect shared terms are a commonly used representation technique. They are employed in, e.g., LEO-II [TB06, BPTF08] or E [Sch02]. The latter system also allows sharing of rewritings.

Higher-order term indexing techniques are scarce, only a few approaches exist. A popular exception the indexing of terms using substitution trees with respect to linear higher-order patterns [Pie09]. In this thesis, we follow the indexing approach of LEO-II [TB06], where rather low-level techniques are employed. First-order term indexing techniques are well-researched and can be found in various survey papers [SRV01, Gra96].

7.2 Further Work

Efficient term indexing data structures for higher-order terms need to be researched intensively. The bound variable indexing is of particular interest for speeding up β-normalization routines. The current implementation of bound variable indexing only indicates whether *some* de-Bruijn index exists in a term that binds to a λ outside. As a consequence, term traversal during normalization will consider subterms that are not relevant for the current substitution run. As a optimization, bound variable indexing might exploit the explicit substitution's structure to decide whether a subterm needs to be traversed. One possible step is storing a list of all loose de-Bruijn indices with the term node and check if any meaningful substitutions will be carried out for each de-Bruijn index (i.e. a replacement different from $\sigma(i) = i$).

The memory consumption and maintenance costs of the subterm index need to be evaluated compared to its actual advantages for the proof procedure. A possible parameter might be the size of the subterms to be indexed.

The evaluation of normalization strategies showed that there exists no most efficient strategy for all problems. The efficiency of a strategy rather depends on some characteristics of the input problem. This observation has to be thoroughly investigated, since there might be heuristics that can approximate which normalization strategy is the most efficient one in the current context.

As a next step, further data structures, e.g. for clauses and literals, have to be implemented. Together with a (maybe simplistic) proof calculus, first significant benchmark results could be achieved. Additionally, the currently implemented data structures need to be analyzed: Most data structures are implemented using high-level algebraic data types (*case classes* of SCALA). However, some underlying data structures could be exchanged by more involved implementations that give better worst-time complexity for certain operations, e.g. random access on substitutions rather than linear traversal.

8 Bibliography

[AB06] P. B. Andrews and C. E. Brown. Tps: A hybrid automatic-interactive system for developing proofs. *Journal of Applied Logic*, 4(4):367 – 395, 2006. Towards Computer Aided Mathematics.

[ACCL90] M. Abadi, L. Cardelli, P.-L. Curien, and J.-J. Levy. Explicit substitutions. In *Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '90, pages 31–46, New York, NY, USA, 1990. ACM.

[AH76] K. Appel and W. Haken. Every planar map is four colorable. *Bulletin of the American Mathematical Society*, 82(5):711–712, 09 1976.

[And83] P. B. Andrews. Resolution in type theory. In JörgH. Siekmann and Graham Wrightson, editors, *Automation of Reasoning*, Symbolic Computation, pages 487–507. Springer Berlin Heidelberg, 1983.

[And14] P. B. Andrews. Church's type theory. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Spring 2014 edition, 2014.

[Ang84] I. Angelelli. Frege and abstractions. *Philosophia Naturalis*, 21(2-4):453–471, 1984.

[Bar85] H. P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics.* Studies in Logic and the Foundations of Mathematics. Elsevier Science, 1985.

[Bar91] H. P. Barendregt. Introduction to generalized type systems. *J. Funct. Program.*, 1(2):125–154, 1991.

[Bar97] H. P. Barendregt. The impact of the lambda calculus in logic and computer science. *The Bulletin of Symbolic Logic*, 3(2):pp. 181–215, 1997.

[BBK04] C. Benzmüller, C. E. Brown, and M. Kohlhase. Higher-order semantics and extensionality. *Journal of Symbolic Logic*, 69(4):1027–1088, 2004.

[BBLRD96] Z. Benaissa, D. Briaud, P. Lescanne, and J. Rouyer-Degli. λv, a calculus of explicit substitutions which preserves strong normalisation. *Journal of Functional Programming*, 6:699–722, 9 1996.

[Ben99] C. Benzmüller. Extensional higher-order paramodulation and RUE-resolution. In Harald Ganzinger, editor, *Automated Deduction - CADE-16, 16th International Conference on Automated Deduction, Trento, Italy, July*

7-10, 1999, Proceedings, number 1632 in LNCS, pages 399–413. Springer, 1999.

[Ben10] C. Benzmüller. Simple type theory as framework for combining logics. In *Contest paper at the World Congress and School on Universal Logic III (UNILOG)*, Lisbon, Portugal, 2010. The conference had no published proceedings; the paper is available as arXiv:1004.5500v1.

[BGGR12] C. Benzmüller, D. Gabbay, V. Genovese, and D. Rispoli. Embedding and automating conditional logics in classical higher-order logic. *Annals of Mathematics and Artificial Intelligence*, 66(1-4):257–271, 2012.

[BK98] C. Benzmüller and M. Kohlhase. Extensional higher-order resolution. In Claude Kirchner and Hélène Kirchner, editors, *Automated Deduction — CADE-15*, volume 1421 of *Lecture Notes in Computer Science*, pages 56–71. Springer Berlin Heidelberg, 1998.

[BKM95] R.S. Boyer, M. Kaufmann, and J.S. Moore. The boyer-moore theorem prover and its interactive enhancement. *Computers & Mathematics with Applications*, 29(2):27 – 62, 1995.

[BM14] C. Benzmüller and D. Miller. Automation of higher-order logic. In Jörg Siekmann, Dov Gabbay, and John Woods, editors, *Handbook of the History of Logic, Volume 9 — Logic and Computation*. Elsevier, 2014. In print.

[BP13] J. C. Blanchette and A. Paskevich. TFF1: The TPTP Typed First-Order Form with Rank-1 Polymorphism. In Maria Paola Bonacina, editor, *Automated Deduction – CADE-24*, volume 7898 of *Lecture Notes in Computer Science*, pages 414–420. Springer Berlin Heidelberg, 2013.

[BP14] C. Benzmüller and B. Woltzenlogel Paleo. Automating Gödel's ontological proof of god's existence with higher-order automated theorem provers. In Torsten Schaub, Gerhard Friedrich, and Barry O'Sullivan, editors, *ECAI 2014*, volume 263 of *Frontiers in Artificial Intelligence and Applications*, pages 163 – 168. IOS Press, 2014.

[BPTF08] C. Benzmüller, L. C. Paulson, F. Theiss, and A. Fietzke. Leo-ii - a cooperative automatic theorem prover for classical higher-order logic (system description). In Alessandro Armando, Peter Baumgartner, and Gilles Dowek, editors, *Automated Reasoning*, volume 5195 of *Lecture Notes in Computer Science*, pages 162–170. Springer Berlin Heidelberg, 2008.

[BR13] C. Benzmüller and T. Raths. HOL based first-order modal logic provers. In Kenneth L. McMillan, Aart Middeldorp, and Andrei Voronkov, editors, *Proceedings of the 19th International Conference on Logic for Programming, Artificial Intelligence and Reasoning (LPAR)*, volume 8312 of *LNCS*, pages 127–136, Stellenbosch, South Africa, 2013. Springer.

[Bro12] C. E. Brown. Satallax: An automatic higher-order prover. In *Proceedings of the 6th International Joint Conference on Automated Reasoning*, IJCAR'12, pages 111–117, Berlin, Heidelberg, 2012. Springer-Verlag.

[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.

[BTPF08] C. Benzmüller, F. Theiss, L. Paulson, and A. Fietzke. LEO-II - a cooperative automatic theorem prover for higher-order logic. In Alessandro Armando, Peter Baumgartner, and Gilles Dowek, editors, *Automated Reasoning, 4th International Joint Conference, IJCAR 2008, Sydney, Australia, August 12-15, 2008, Proceedings*, volume 5195 of *LNCS*, pages 162–170. Springer, 2008.

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

[CH06] F. Cardone and J. R. Hindley. History of lambda-calculus and combinatory logic. *Handbook of the History of Logic*, 5, 2006.

[Cha11] A. Charguéraud. The locally nameless representation. *Journal of Automated Reasoning*, pages 1–46, 2011. 10.1007/s10817-011-9225-2.

[CHL96] Pierre-Louis Curien, Therese Hardin, and Jean-Jacques Levy. Confluence properties of weak and strong calculi of explicit substitutions. *JOURNAL OF THE ACM*, 43:43–2, 1996.

[Chr93] J. Christian. Flatterms, discrimination nets, and fast term rewriting. *Journal of Automated Reasoning*, 10(1):95–113, 1993.

[Chu32] A. Church. A Set of Postulates for the Foundation of Logic. *Annals of Mathematics*, 33(2):346–366, 1932.

[Chu33] A. Church. A Set of Postulates for the Foundation of Logic, Second Paper. *The Annals of Mathematics*, 34(4):839–864, October 1933.

[Chu36] Alonzo Church. An unsolvable problem of elementary number theory. *American Journal of Mathematics*, 58(2):345–363, April 1936.

[Chu40] A. Church. A formulation of the simple theory of types. *J. Symb. Log.*, 5(2):56–68, 1940.

[CP02] I. Cervesato and F. Pfenning. A linear logical framework. *Information and Computation*, 179(1):19 – 75, 2002.

[CP03] I. Cervesato and F. Pfenning. A linear spine calculus. Technical report, Journal of Logic and Computation, 2003.

[CR36] A. Church and J. B. Rosser. Some properties of conversion. *Transactions of the American Mathematical Society*, 39:472–482, 1936. http://www.jstor.org/stable/2268573Electronic Edition.

[Cur41] H. B. Curry. The Paradox of Kleene and Rosser. *Transactions of the American Mathematical Society*, 50(3):454–516, November 1941.

[Cur58] Haskell B. Curry. *Combinatory logic / [by] Haskell B. Curry [and] Robert Feys. With two sections by William Craig.* North-Holland Pub. Co Amsterdam, 1958.

[CW85] L. Cardelli and P. Wegner. On understanding types, data abstraction, and polymorphism. *ACM Comput. Surv.*, 17(4):471–523, December 1985.

[DEDC96] Pierre Deransart, AbdelAli Ed-Dbali, and Laurent Cervoni. Prolog data structures. In *Prolog: The Standard*, pages 5–10. Springer Berlin Heidelberg, 1996.

[DJ90] N. Dershowitz and J.-P. Jouannaud. Handbook of theoretical computer science (vol. b). chapter Rewrite Systems, pages 243–320. MIT Press, Cambridge, MA, USA, 1990.

[Dru08] T. Drucker. *Perspectives on the History of Mathematical Logic.* Modern Birkhäuser Classics. Birkhäuser Boston, 2008.

[Far07] W. M. Farmer. The seven virtues of simple type theory, 2007.

[Fie90] J. Field. On laziness and optimality in lambda interpreters: Tools for specification and analysis. In *Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '90, pages 1–15, New York, NY, USA, 1990. ACM.

[Fre79] G. Frege. *Begriffsschrift, eine der arithmetischen nachgebildete Formelsprache des reinen Denkens.* Verlag von Louis Nebert, Halle, 1879.

[FZ07] Branden Fitelson and EdwardN. Zalta. Steps toward a computational metaphysics. *Journal of Philosophical Logic*, 36(2):227–247, 2007.

[G. 89] G. Peano. The principles of arithmetic, presented by a new method. 1889.

[Gir72] J.Y. Girard. *Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur.* 1972.

[GP02] M. J. Gabbay and A. M. Pitts. A new approach to abstract syntax with variable binding. *Formal Aspects of Computing*, 13(3-5):341–363, 2002.

[Gra96] P. Graf. *Term Indexing*, volume 1053 of *Lecture Notes in Computer Science.* Springer, 1996.

[GTL89] J.-Y. Girard, P. Taylor, and Y. Lafont. *Proofs and Types.* Cambridge University Press, New York, NY, USA, 1989.

[Gö31] K. Gödel. Über formal unentscheidbare sätze der principia mathematica und verwandter systeme. *Monatshefte für Mathematik und Physik*, 38(1):173–198, 1931.

[Hen50] L. Henkin. Completeness in the theory of types. *Journal of Symbolic Logic*, 15(2):81–91, 06 1950.

[Hin69] R. Hindley. The principal type-scheme of an object in combinatory logic. *Transactions of the American Mathematical Society*, 146:pp. 29–60, 1969.

[Hin14] J. R. Hindley. Lambda calculus with types (perspectives in logic) by henk barendregt, wil dekkers and richard statman. *Bulletin of the London Mathematical Society*, 2014.

[HPJWe92] P. Hudak, S. Peyton Jones, and P. Wadler (editors). Report on the Programming Language Haskell, A Non-strict Purely Functional Language (Version 1.2). *ACM SIGPLAN Notices*, 27(5), May 1992.

[Hue72] G. P. Huet. *Constrained Resolution: A Complete Method for Higher Order Logic.* Reports. Case Western Reserve university, Cleveland, OH, 1972.

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

[Kle36] S. C. Kleene. λ -definability and recursiveness. *Duke Mathematical Journal*, 2(2):340–353, 06 1936.

[KR35] S. C. Kleene and J. B. Rosser. The inconsistency of certain formal logics. *Annals of Mathematics*, 36(3):pp. 630–636, 1935.

[KR95] F. Kamareddine and A. Ríos. A lambda-calculus ‘a la de bruijn with explicit substitutions. In *Proceedings of the 7th International Symposium on Programming Languages: Implementations, Logics and Programs*, PLILPS ’95, pages 45–62, London, UK, UK, 1995. Springer-Verlag.

[KR97] F. Kamareddine and A. Ríos. Extending a λ-calculus with explicit substitution which preserves strong normalisation into a confluent calculus on open terms. *J. Funct. Program.*, 7(4):395–420, July 1997.

[KRTU99] A. J. Kfoury, S. Ronchi Della Rocca, J. Tiuryn, and P. Urzyczyn. Alpha-conversion and typability. pages 1–21, 1999.

[KV13] L. Kovács and A. Voronkov. First-order theorem proving and vampire. In Natasha Sharygina and Helmut Veith, editors, *Computer Aided Verification*, volume 8044 of *Lecture Notes in Computer Science*, pages 1–35. Springer Berlin Heidelberg, 2013.

[Lag11] J.C. Lagarias. *The Kepler Conjecture: The Hales-Ferguson Proof*. Springer New York, 2011.

[Lei51] G. W. Leibniz. The Art of Discovery (1685). In Philip P. Wiener, editor, *Leibniz: Selections*, page 51. Charles Scribner's Sons, New York, 1951.

[LNQ04] C. Liang, G. Nadathur, and X. Qi. Choices in representation and reduction strategies for lambda terms in intensional contexts. *J. Autom. Reasoning*, 33:89–132, 2004.

[Mac95] D. MacKenzie. The automation of proof: A historical and sociological exploration. *IEEE Ann. Hist. Comput.*, 17(3):7–29, September 1995.

[Mat99] R. Matthes. *Extensions of System F by Iteration and Primitive Recursion on Monotone Inductive Types*. Informatik [Essen : Westarp]. Utz, 1999.

[McC90] W. McCune. OTTER 2.0. In *10th International Conference on Automated Deduction, Kaiserslautern, FRG, July 24-27, 1990, Proceedings*, pages 663–664, 1990.

[Mcc97] William Mccune. Solution of the robbins problem. *Journal of Automated Reasoning*, 19(3):263–276, 1997.

[McC04] P. McCorduck. *Machines Who Think: A Personal Inquiry into the History and Prospects of Artificial Intelligence*. AK Peters Ltd, 2004.

[McC10] W. McCune. Prover9 and mace4. `http://www.cs.unm.edu/~mccune/prover9/`, 2005–2010.

[Mil78] R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

[Mil91a] D. Miller. Unification of simply typed lambda-terms as logic programming. In *In Eighth International Logic Programming Conference*, pages 255–269. MIT Press, 1991.

[Mil91b] Dale Miller. A logic programming language with lambda-abstraction, function variables, and simple unification. In Peter Schroeder-Heister, editor, *Extensions of Logic Programming*, volume 475 of *Lecture Notes in Computer Science*, pages 253–281. Springer Berlin Heidelberg, 1991.

[Mos68] A. Mostowski. *Synthese*, 18(2/3):pp. 302–305, 1968.

[MTM97] R. Milner, M. Tofte, and D. Macqueen. *The Definition of Standard ML*. MIT Press, Cambridge, MA, USA, 1997.

[Nad96] G. Nadathur. A fine-grained notation for lambda terms and its use in intensional operations. *Journal of Functional and Logic Programming*, 1999, 1996.

[Nad01] G. Nadathur. The metalanguage lambda-prolog and its implementation. In *Functional and Logic Programming, 5th International Symposium, FLOPS 2001, Tokyo, Japan, March 7-9, 2001, Proceedings*, pages 1–20, 2001.

[NHRV01] R. Nieuwenhuis, T. Hillenbrand, A. Riazanov, and A. Voronkov. On the evaluation of indexing techniques for theorem proving. In Rajeev Goré, Alexander Leitsch, and Tobias Nipkow, editors, *Automated Reasoning*, volume 2083 of *Lecture Notes in Computer Science*, pages 257–271. Springer Berlin Heidelberg, 2001.

[NR99] R. Nieuwenhuis and A. Rubio. Paramodulation-based theorem proving, 1999.

[NWP02] T. Nipkow, M. Wenzel, and L. C. Paulson. *Isabelle/HOL: A Proof Assistant for Higher-order Logic*. Springer-Verlag, Berlin, Heidelberg, 2002.

[Pau11] C. Paulin-Mohring. Introduction to the coq proof-assistant for practical software verification. In *Tools for Practical Software Verification, LASER, International Summer School 2011, Elba Island, Italy, Revised Tutorial Lectures*, pages 45–95, 2011.

[Pfe94] Frank Pfenning. Elf: A meta-language for deductive systems (system description). In *In 12th International Conference on Automated Deduction*, pages 811–815. Springer-Verlag, 1994.

[Pie02] B.C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[Pie09] B. Pientka. Higher-order term indexing using substitution trees. *ACM Trans. Comput. Logic*, 11(1):6:1–6:40, November 2009.

[Pit03] A. M. Pitts. Nominal logic, a first order theory of names and binding. *Information and Computation*, 186(2):165 – 193, 2003. Theoretical Aspects of Computer Software (TACS 2001).

[PP03] B. Pientka and F. Pfenning. Optimizing higher-order pattern unification. In *19th International Conference on Automated Deduction*, pages 473–487. Springer-Verlag, 2003.

[PS99] F. Pfenning and C. Schürmann. System description: Twelf - a meta-logical framework for deductive systems. In *CADE*, pages 202–206, 1999.

[Rey74] J. C. Reynolds. Towards a theory of type structure. In *Symposium on Programming*, pages 408–423, 1974.

[Rey98] J. C. Reynolds. *Theories of Programming Languages*. Cambridge University Press, 1998. Cambridge Books Online.

[Ria03] A. Riazanov. *Implementing an efficient theorem prover*. PhD thesis, 2003.

[RO12] T. Raths and J. Otten. The qmltp problem library for first-order modal logics. In B. Gramlich, D. Miller, and U. Sattler, editors, *Automated Reasoning*, volume 7364 of *Lecture Notes in Computer Science*, pages 454–461. Springer Berlin Heidelberg, 2012.

[Rob65] J. A. Robinson. A machine-oriented logic based on the resolution principle. *J. ACM*, 12(1):23–41, January 1965.

[Rus03] B. Russell. *The Principles of Mathematics*. Number v. 1 in The Principles of Mathematics. University Press, 1903.

[Rus08] B. Russell. Mathematical logic as based on the theory of types. *American Journal of Mathematics*, 30(3):pp. 222–262, 1908.

[SB10] G. Sutcliffe and C. Benzmüller. Automated reasoning in higher-order logic using the TPTP THF infrastructure. *Journal of Formalized Reasoning*, 3(1):1–27, 2010.

[Sch02] S. Schulz. E - a brainiac theorem prover. *AI Commun.*, 15(2,3):111–126, August 2002.

[SRV01] R. Sekar, I. V. Ramakrishnan, and A. Voronkov. Handbook of automated reasoning. chapter Term Indexing, pages 1853–1964. Elsevier Science Publishers B. V., Amsterdam, The Netherlands, The Netherlands, 2001.

[ST85] Daniel Dominic Sleator and Robert Endre Tarjan. Self-adjusting binary search trees. *J. ACM*, 32(3):652–686, July 1985.

[Sti89] M. E. Stickel. The Path-Indexing Method For Indexing Terms. Technical report, Artificial Intelligence Center, SRI International, 333 Ravenswood Ave., Menlo Park, CA 94025, 1989.

[Sut09] G. Sutcliffe. The TPTP Problem Library and Associated Infrastructure: The FOF and CNF Parts, v3.5.0. *Journal of Automated Reasoning*, 43(4):337–362, 2009.

[Tai67] W. W. Tait. Intensional interpretations of functionals of finite type i. *Journal of Symbolic Logic*, 32(2):198–212, 1967.

[Tai75] W. W. Tait. A realizability interpretation of the theory of species. In Rohit Parikh, editor, *Logic Colloquium*, volume 453 of *Lecture Notes in Mathematics*, pages 240–251. Springer Berlin Heidelberg, 1975.

[TB06] F. Theiss and C. Benzmüller. Term indexing for the LEO-II prover. In *IWIL-6 workshop at LPAR 2006: The 6th International Workshop on the Implementation of Logics*, Pnom Penh, Cambodia, 2006.

[Tho97] S. Thompson. Higher-order + Polymorphic = Reusable. May 1997.

[Tur36] A. M. Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, 2(42):230–265, 1936.

[Tur37] A. M. Turing. Computability and λ-Definability. *The Journal of Symbolic Logic*, 2:153–163, 1937.

[Wel98] J. B. Wells. Typability and Type Checking in System F Are Equivalent and Undecidable. *Annals of Pure and Applied Logic*, 98:111–156, 1998.

[Wos92] L. Wos. Note on mccune's article on discrimination trees. *J. Autom. Reasoning*, 9(2):145–146, 1992.

[WR26] A. N. Whitehead and B. Russell. Principia mathematica. *Mind*, 35(137):130, 1926.

[WSB14] M. Wisnieski, A. Steen, and C. Benzmüller. The Leo-III project. In Alexander Bolotov and Manfred Kerber, editors, *Joint Automated Reasoning Workshop and Deduktionstreffen*, page 38, 2014.

Index

λ-calculus, 5
- α-conversion, 7
- β-conversion, 7
- η-conversion, 7
- normal forms, 8
- simply typed, 10
- untyped, 6

$\lambda\sigma$-calculus, 30
NIL-reduction, 28

Automated Theorem Proving, 15

Church-Rosser, 8, 13
Church-style, 12
Clause, 60
- Data structure, 60
- sets of, 60

Closure, 30
Curry-style, 12

De-Bruijn index , *see* Nameless representation

Head, 27
Higher-order logic, 14

Literal, 59
- Data structure, 59

Nameless representation, 8

Preprocessing, 60

Root, 27

Shifting, 31
Signature, 58
- Data structure, 59

Spine, 27
- merging, 29

Strong normalization, 13
Substitution
- as meta operation, 6
- Composition of, 31
- Explicit, 29

System F, 19

Term Bank, 38
- Data Structure, 39
- Implementation, 40

Term indexing, 49
- in automated theorem proving, 50
- of bound variables, 55
- of function symbols, 53
- of subterms, 54
- of term structure, 52
- of LEO-III, 51

Term representation
- evaluation of, 41
- normalization of, 34
- of LEO-III, 32
- perfectly shared, 38

TPTP, 17
Type, 11
- Order of, 11
- polymorphic, 20