University of Warsaw
Faculty of Mathematics, Informatics and Mechanics

Michał Szynwelski

A functional programming language for sets with atoms

PhD dissertation

Supervisor:

prof. dr hab. Bartosz Klin
Department of Computer Science
University of Oxford

March 2022

Supervisor's statement

Hereby I confirm that the presented thesis was prepared under my supervision and that it fulfills the requirements for the degree of PhD of Computer Science.

....

Date prof. dr hab. Bartosz Klin

Author's statement

Hereby I declare that the presented thesis was prepared by me and none of its contents was obtained by means that are against the law.

The thesis has never before been a subject of any procedure of obtaining an academic degree.

Moreover, I declare that the present version of the thesis is identical to the attached electronic version.

....

Date mgr Michał Szynwelski

Abstract

This thesis describes a functional programming language Nλ that operates on infinite structures using sets with atoms. For relational structures satisfying certain conditions, hereditarily orbit-finite sets with atoms admit a finite representation based on first-order logic formulas. Thanks to this property, the presented language enables the implementation of programs that perform calculations on infinite objects, such as graphs or automata. Such infinite objects are accessed in ways similar to their finite counterparts. Therefore, many algorithms operating on finite structures can be naturally transported to the world of infinite objects.

The Nλ language is presented as an extension of the simply typed lambda calculus. The syntax of the language and its denotational and operational semantics are described. Each language term is assigned a type using appropriate typing rules. Additionally, denotational semantics assign meaning to terms in the form of mathematical objects. In this case, they are atoms, truth values, sets, and functions. Operational semantics, through a list of reduction rules, defines the actual computational model of the language.

The language has several expected properties that have been proven. It is shown that the reduction rules preserve the type (subject reduction) and the meaning (denotation) of the term. Additionally, the language has Church-Rosser property and strong normalization.

The above properties have been proven for a core version of the language. To be considered as a full-fledged functional programming language, Nλ must additionally have constructs present in modern, commonly used programming languages. Therefore, possible extensions of the core version are indicated. The possibility of adding recursion, polymorphism, or algebraic types is discussed.

In addition to the theoretical description of the Nλ language, its implementation in Haskell was also provided. As a result, the language user, apart from the implementation of programs that perform calculations on infinite data structures, can use all Haskell functionalities. To solve and simplify first-order logic formulas, the language implementation integrates with an external SMT solver. Some interesting applications of the language for algorithms on infinite graphs and automata are discussed.

ACM Computing Classification System

Mathematics of computing → Mathematical analysis → Calculus → Lambda calculus
Software and its engineering → Software notations and tools → Formal language definitions → Semantics
Software and its engineering → Software notations and tools → General programming languages → Language types → Functional languages

Keywords

sets with atoms, definable sets, lambda calculus, language semantics, functional programming, Haskell, satisfiability modulo theories

Streszczenie

Niniejsza rozprawa przedstawia funkcyjny język programowania Nλ operujący na strukturach nieskończonych opartych na zbiorach z atomami. Dla struktur relacyjnych spełniających określone warunki, zbiory z atomami, które są dziedzicznie skończenie orbitowe, posiadają skończoną reprezentację bazującą na formułach pierwszego rzędu. Przedstawiony język, wykorzystując tę własność, umożliwia implementację programów działających na obiektach nieskończonych takich jak grafy lub automaty. Takie nieskończone obiekty są dostępne za pomocą analogicznych funkcji i interfejsów jak ich skończone odpowiedniki. Dlatego wiele algorytmów działających na konstrukcjach skończonych może być naturalnie wykorzystanych w świecie struktur nieskończonych.

Język Nλ został przedstawiony jako rozszerzenie rachunku lambda z typami prostymi. Opisana została jego składnia, semantyka denotacyjna oraz semantyka operacyjna. Każdy term języka ma przypisany typ za pomocą odpowiednich zasad typowania. Semantyka denotacyjna dodatkowo przypisuje termom ich znaczenie w postaci obiektów matematycznych. W tym wypadku są to atomy, wartości logiczne, zbiory oraz funkcje. Semantyka operacyjna poprzez listę reguł redukcyjnych przedstawia właściwy model obliczeniowy na strukturach nieskończonych.

Tak opisany język ma oczekiwane własności, które zostały udowodnione. Chodzi o zachowywanie przez reguły redukcyjne typów i denotacji (znaczenia) termów, własność Churcha-Rossera oraz silną normalizację.

Powyższe własności zostały wykazane dla podstawowej wersji języka. Jednak aby język Nλ mógł być rozpatrywany jako pełnoprawny język programowania, musi zostać rozszerzony o konstrukcje obecne we współczesnych, powszechnie używanych językach programowania. Dlatego wskazane zostały możliwe rozszerzenia podstawowej wersji o takie konstrukcje jak rekurencja, polimorfizm czy typy algebraiczne.

Poza teoretycznym opisem języka Nλ została również dostarczona jego implementacja w języku Haskell. Dzięki temu użytkownik języka poza implementacją programów wykonujących obliczenia na nieskończonych strukturach danych może korzystać ze wszystkich funkcjonalności Haskella. W celu sprawdzania prawdziwości oraz upraszczania formuł logicznych pierwszego rzędu implementacja języka integruje się z zewnętrznym narzędziem w tym się specjalizującym (SMT solver). Rozprawa opisuje także ciekawsze zastosowania języka do algorytmów na grafach i automatach nieskończonych.

Tytuł pracy w języku polskim

Funkcyjny język programowania dla zbiorów z atomami

Acknowledgements

Most of all, I would like to thank my supervisor, Professor Bartosz Klin. His guidance and extensive knowledge are invaluable to me. I am very grateful for his concern and support. I could always count on his help and advice. Additionally, I would like to thank him for his encouragement, patience, sense of humor, and time spent over the years.

I would like to thank Eryk Kopczyński and Szymon Toruńczyk, who came up with the idea of using formulas to represent orbit-finite sets with atoms, and whose work on the LOIS library has been a source of constant inspiration.

I am also grateful to my co-authors, Alexandra Silva, Joshua Moerman, and Matteo Sammartino, for everything I have learned while working with them. Special thanks to Joshua Moerman, who became the first Nλ programmer, and whose constructive remarks and feature requests were very helpful in making Nλ a practical programming solution.

Many thanks to Henryk Michalewski for the initial supervision of my scientific work.

I am very grateful to the entire Automata Group at the University of Warsaw, especially Mikołaj Bojańczyk, for showing how fascinating this area is in connection with sets with atoms.

I would like to express my gratitude to my parents, Teresa and Piotr, for their unconditional love and great support. Special thanks go to my sisters Małgorzata and Magdalena, my family, Jarosław Paszek, and my friends for always believing in me.

My deepest gratitude goes to my lovely wife Anna and children Zosia, Marysia, Staś, and Jaś. Thank you for your love, understanding, and support. None of the things that I achieved would be possible without you.

Contents

1 Introduction **8**

2 Sets with atoms **13**
- 2.1 Logical structure . . . 13
- 2.2 Cumulative hierarchy . . . 15
- 2.3 Support . . . 15
- 2.4 Sets with atoms . . . 16
- 2.5 Equivariance . . . 17
- 2.6 Orbit-finiteness . . . 17
- 2.7 Set expressions . . . 18

3 Programming language **20**
- 3.1 Types . . . 21
- 3.2 Basic functional language . . . 22
- 3.3 Denotational semantics . . . 24
- 3.4 Logic-based functional language . . . 26
- 3.5 Extended denotational semantics . . . 30
- 3.6 Typing rules . . . 31
- 3.7 Reduction rules . . . 32
- 3.8 Properties of semantics . . . 37

4 Subject reduction **38**
- 4.1 Substitutions in typed terms . . . 38
- 4.2 The Subject Reduction Theorem . . . 40

5 Equivalence of semantics **48**
- 5.1 Substitution in denotation . . . 48
- 5.2 Equivalence theorem . . . 50

6 Church-Rosser Property **61**
- 6.1 Parallel reduction . . . 62
- 6.2 A substitution lemma for parallel reduction . . . 64
- 6.3 Parallel reduction in restricted context . . . 67
- 6.4 Diamond property of parallel reduction . . . 70
- 6.5 Church–Rosser theorem . . . 75

7 Normalization **77**
- 7.1 Reducibility . . . 78
- 7.2 Reducibility to strong normalization . . . 79

7.3 Universality of reducibility . . . 80
7.4 Normal form . . . 84
7.5 Consequences of strong normalization . . . 85

8 Language extensions 86
8.1 Atom constants . . . 86
8.2 Explicit extension . . . 87
8.3 Construction of sets with atoms . . . 91
8.4 Recursion . . . 92
8.5 Polymorphism . . . 93
8.6 Towards an implementation . . . 94
8.7 Relational structures . . . 95

9 Alternative approaches 96
9.1 Hulls, supports and orbits . . . 96
9.2 While programs with atoms . . . 98
9.3 Other alternatives . . . 100

10 Implementation 103
10.1 Features of Haskell . . . 104
10.2 Atom constants and atom variables . . . 106
10.3 Relational structures and logic . . . 108
10.4 Logical formulas . . . 109
10.5 Elimination of quantifiers . . . 111
10.6 SMT Solver . . . 112
10.7 Contexts . . . 115
10.8 Variants . . . 116
10.9 Conditional expressions . . . 117
10.10 Maybe, Either . . . 119
10.11 Nominal types . . . 120
10.12 Sets . . . 122
10.13 Supports and orbits . . . 125
10.14 Show, Read . . . 127
10.15 Installation Guide . . . 128

11 Use cases 130
11.1 Graphs . . . 130
11.2 Graph algorithms . . . 131
11.3 Graph coloring . . . 133
11.4 Automata . . . 135
11.5 Minimization of deterministic automata . . . 138
11.6 Learning automata . . . 140

CHAPTER 1

Introduction

The concept of infinity is an important topic in computer science. On first consideration, it might seem that computer programs, because they use finite memory, can only operate on finite data. However, many definitions, algorithms, and data structures can easily be extended to concepts that have no limitations. For example, the algorithm for adding an item to the beginning of a linked list can work the same whether the list is finite or infinite.

Not all operations have the nice property of expanding into the infinite world. The procedure of searching for a given element in a list does not extend to infinite lists. Moreover, some internal representations of abstract data structures cannot be used for infinite structures. The representation that explicitly stores each list element can only be used for finite lists.

To represent an infinite data structure, it is critical that the representation is finite, and that it allows for efficient operations on it. Ideally, the representation should be abstract enough to hide from the user the property of being infinite. A programmer using such a structure should be allowed the convenient illusion of working directly with an infinite structure in the same way as she would work with a finite one. In particular, operations that extend to infinite concepts should only be presented to the programmer once, without having to handle each case (finite/infinite) separately.

In the world of programming languages, finite representations of infinite structures is often built through a mechanism of lazy evaluation. Instead of storing the individual elements of a structure, a function is stored that can compute such elements. The function call is delayed until the element is needed. Such a representation allows us to efficiently perform operations of mapping or filtering elements of an infinite list. This technique has its limitations, for example in a finite time it does not allow to check whether a given element is a member of a structure, or to display that structure in a human-readable manner.

In mathematics, a rich source of infinite objects that are finitely represented are relational structures that are first-order definable over some fixed structure. It is worth emphasizing that such structures do not suffer from some of the limitations of the lazy approach. For example, a set of pairs of different natural numbers can be written as:

$$\{(a,b) \mid a,b \in \mathbb{N}, a \neq b\}$$

This notation is readable and understandable for a human. Checking whether a given element is a member of this set is checking whether two given natural numbers are not equal.

More complex objects can also be represented with this notation. For example, an infinite clique graph, with vertices as natural numbers and edges as unordered pairs of distinct numbers, is represented by:

$$(\mathbb{N},\ \{\{a,b\} \mid a,b \in \mathbb{N}, a \neq b\})$$

These structures are first-order definable over the set $\mathbb{N}$ of natural numbers with the equality relation. In turn, when considering (the total order of) rational numbers and using the same notation, one can represent, for example, the set of closed intervals as follows:

$$\{\{c \mid c \in \mathbb{Q}, a \leq c \leq b\} \mid a,b \in \mathbb{Q}\} \tag{1.1}$$

This structure is first-order definable over the set $\mathbb{Q}$ with an ordering relation $\leq$.

The elements of the underlying structure of basic values, like natural or rational numbers above, will be called *atoms*. The set of relations that define these structures naturally limits the operations that we will be able to perform on the data. This means that considering natural numbers with the equality relation, we do not have the ambition to check whether a given set contains only even numbers. This property cannot be expressed using the available relation. On the other hand, we would like to be able to check if a given set is empty or if it is contained in another set.

Apart from representing possibly infinite structures, we want to perform computations on them efficiently. This efficiency is heavily based on the first-order properties of the underlying structure of atoms. For example, to make sure that the set (1.1) contains any non-empty interval, one need to know that there are rational numbers a, b, c such that $a \leq c \leq b$. The structure of atoms should therefore allow for efficient checking of such conditions. For this purpose, we shall assume that underlying structures of atoms are uniquely (up to isomorphism) determined as countable models of their first-order theories and that these first-order theories are decidable.

Two structures that meet these conditions and which we will focus on are already mentioned:

- $(N, =)$ – natural numbers with equality,
- $(Q, \leq)$ – rational numbers with ordering.

Our goal is to create a programming language in a functional paradigm that will allow the programmer to operate on such structures. Additionally, when using this language, one should naturally operate on infinite sets in the same way as programming on finite sets, even though the internal representations of infinite sets are quite different from finite sets.

As an example, consider a program implemented in Haskell that computes a transitive closure of a binary relation. We represent such a relation as a set of pairs. First, we define a function that computes the composition of two relations:

```haskell
compose :: (Ord a, Ord b, Ord c) => Set(a,b) -> Set(b,c) -> Set(a,c)
compose r s = sum (map (\(a,b) -> map (\(_,c) -> (a,c))
                                      (filter ((==b) . fst) s))
                       r)
```

This function maps each pair `(a,b)` from a given relation `r` to a set of those pairs `(a,c)` for which there is a pair `(b,c)` in relation `s`. Then the sum (i.e. set-theoretic union) of such sets is returned. With this function, we can compute transitive closures as follows:

```haskell
transitiveClosure :: Ord a => Set(a,a) -> Set(a,a)
transitiveClosure r = let r' = union r (compose r r)
                      in if r==r' then r else (transitiveClosure r')
```

This example uses some auxiliary functions from the standard Haskell `Data.Set` module, which is used to handle finite sets:

```haskell
sum = unions . elems :: Set (Set a) -> Set a
map :: Ord b => (a -> b) -> Set a -> Set b
filter :: (a -> Bool) -> Set a -> Set a
union :: Ord a => Set a -> Set a -> Set a
```

One of our goals is to provide a version of the `Set` type constructor that would allow the programmer to construct both finite and infinite first-order definable sets and then treat them uniformly, so that the above piece of code could be reused to compute the transitive closure of an infinite relation, internally represented by first-order formulas and set-builder expressions such as (1.1).

The main contribution of this thesis is a definition and implementation of a new programming language, an extension of Haskell, called Nλ.[1] The theoretical basis for the language is the simply-typed lambda calculus extended with new types aimed for handling infinite sets represented by first-order formulas. The language has built-in basic functions that create, modify, and check the properties of such sets. One can implement programs such as the transitive closure function above with these ingredients.

For the core Nλ language, syntax and two semantics will be presented. Denotational semantics will provide meaning for language terms, and operational semantics will be the actual computational model of the language. Useful properties of the language like subject reduction, Church-Rosser property, or strong normalization will be proven. The language is implemented using Haskell. The implementation will be described in detail and used for many algorithms presented at the end of the dissertation.

Chapters 3 through 7 describe the core language, which is really a lambda calculus with a certain extended type system rather than a full programming language. In particular, there is no recursion in it. Thanks to this, proofs in Chapters 4 - 7 are variants of classical proofs of the corresponding properties of the simply typed lambda calculus. If recursion were added to the language, these results would be come vastly more complicated (e.g. denotational semantics) or even false (normalization). However, it is worth stating and proving these results even for a simple language without recursion, because it turns out that our type system extension is non-trivial and causes significant technical complications in classical proofs that have to be overcome.

Since the representation of infinite sets is based on first-order formulas, the implementation of such a representation requires the functionality of solving formulas for selected structures of atoms. The well-researched area of *satisfiability modulo*

[1]The letter N comes from the word *nominal*, which is used to describe sets with atoms (see Chapter 2).

theories (SMT) is used here. There are off-the-shelf software tools tuned to that purpose. In our solution, the Z3 theorem prover from Microsoft Research was selected, which offers satisfiability checking for first-order formulas over the theory of equality and the theory of dense total orders without endpoints. The Nλ implementation intensively interacts with this checker to analyse formulas that arise in representations of infinite data structures.

Structure of the thesis

Chapter 2 specifies the conditions that must be met by a relational structure of atoms to admit effective calculations on infinite sets. In addition, the concept of a set with atoms, and conditions that guarantee finite representability of certain sets with atoms, are discussed. Finally, a finite representation is described in detail.

Chapter 3 describes the core Nλ programming language. The semantics of the language is presented in two versions: a basic abstract one, which provides an interface for operations on infinite sets, and the extended, more concrete one, which includes constructs that enable the representation of infinite objects in a finite form. The chapter contains a specification of the language types, typing rules, syntax, and two semantics. Denotational semantics allows us to understand the meaning of individual language terms. Operational semantics, given as a list of reduction rules, describes the computational model and the algorithm for obtaining the result for a given program.

Chapters 4 through 7 prove some desirable properties of the core language. Chapter 4 shows subject reduction, which means that reduction rules preserve types. In Chapter 5 we show that denotational and operational semantics closely correspond to each other. More precisely, the reduction rules of operational semantics do not change the denotational meaning of the program. Chapter 6 proves the Church-Rosser property of the language. It means that the choice of the order of the reduction rules does not affect the final result of the reduction. Strong normalization of the language is demonstrated in Chapter 7. It follows that the reduction process cannot be infinite and eventually terminates in a normal form.

Chapter 8 lists examples of how the core version of a language can be extended to provide more functionality and bring it closer to a modern full-fledged programming language. The extensions apply to the language itself, but also to relational structures of atoms that can be supported.

Alternative approaches to computing with infinite objects are described in Chapter 9. In particular, it presents the immediate predecessor of Nλ, which uses representatives of the orbits to represent infinite sets. In addition, the imperative programming language LOIS is shown that also operates on a first-order formula-based representation of infinite sets.

Chapter 10 explains the nuances of the implementation of Nλ. Arguments are presented why Haskell was chosen to implement the language and how the theoretical description of the language is realized by constructions in Haskell. The data types and type classes implementing particular terms of the language are explained. In addition, issues related to first-order logic and integration with SMT Solver are discussed.

Finally, in Chapter 11, some use cases are shown. The chapter lists examples of solutions implemented in Nλ to problems related to infinite graphs and automata. In particular, graph coloring, minimization of deterministic automata, and automata learning are discussed.

This thesis is not atended as a full documentation of the Nλ implementation; this is available elsewhere [Szynwelski, 2022a, Szynwelski, 2022b, Szynwelski, 2022c].

CHAPTER 2

Sets with atoms

Sets with atoms are a mathematical structure which, under certain conditions (orbit-finiteness), provides a finite representation for infinite objects. They are the basis for a convenient framework for designing and running algorithms operating on infinite data structures.

Sets with atoms were originaly introduced to set theory by Fraenkel [Fraenkel, 1922]. The model was based on the extension of the classical set theory by atoms (also known as ur-elements), which are objects that do not have any elements but are distinct from the empty set. The set of axioms (called ZFA) that formalizes this theory is very similar to the Zermelo-Fraenkel axiomatic system, with some modifications to the empty set axiom and the axiom of extensionality. It also introduces an axiom of atoms, which states that all atoms are empty. One of the applications of this model is to prove the independence of the axiom of choice from other axioms. The theory was then developed by Mostowski [Mostowski, 1938] and therefore these sets are sometimes called Fraenkel-Mostowski sets.

In Computer Science, sets with atoms were rediscovered, under the name of *nominal sets*, by Gabbay and Pitts [Gabbay and Pitts, 2002] to develop a theory of freshness and name abstraction.

Since then, this theory has met with wide interest leading to fruitful results in areas such as semantics (details can be found in the book [Pitts, 2013]), the theory of computation [Bojańczyk et al., 2012, Bojańczyk et al., 2013, Bojańczyk et al., 2014] or distributed computing [Montanari and Pistore, 1999, Montanari and Pistore, 2005]. In addition to the theoretical foundations, sets with atoms have influenced the design of programming languages [Shinwell et al., 2003, Kopczyński and Toruńczyk, 2017] and theorem proving systems [Urban, 2008].

The introductory presentation of the theory in this chapter is based on [Bojańczyk et al., 2014, Klin et al., 2014, Ochremiak, 2016, Bojańczyk, 2019].

2.1. Logical structure

The theory of sets with atoms is parameterized by a countable infinite relational structure over a finite signature. This basic structure is denoted by $\mathcal{A}$. Its elements are called *atoms*. For simplicity, the set of all atoms is also denoted by $\mathcal{A}$. The signature contains symbols of relations interpreted over atoms. Two basic examples of structures used in this thesis are:

- $(\mathbb{N}, =)$ – natural numbers with equality (we call these *equality atoms*)

- $(\mathbb{Q}, \leq)$ – rational numbers with ordering (we call these *ordered atoms*).

An *atom automorphism* is any permutation π of the domain of $\mathcal{A}$ which preserves all relations in the structure. This definition can naturally be extended to n-tuples of atoms:

$$\pi(a_1, \ldots, a_n) = (\pi(a_1), \ldots, \pi(a_n)).$$

The set of all automorphisms of the structure $\mathcal{A}$ is denoted $Aut(\mathcal{A})$.

A *first-order formula over* $\mathcal{A}$ is a first-order logic formula in which atomic statements take the form of applying relations from the structure to atoms, e.g. $a = b$ or $b \leq c$.

In the course of further considerations, we will limit ourselves only to structures that meet the following properties:

- have a decidable first-order theory,
- are oligomorphic,
- are homogeneous.

The latter two conditions are standard properties in Model Theory (see e.g. [Hodges, 1993]). We recall them here for completeness.

Definition 2.1. *A structure $\mathcal{A}$ is called* ***oligomorphic***[1] *if for every $n \geq 0$ there are finitely many n-tuples up to automorphism. More precisely, the equivalence relation defined as follows:*

two n-tuples $(a_1, \ldots, a_n)$ and $(b_1, \ldots, b_n)$ are related if there exists an automorphism π of $\mathcal{A}$, such that $\pi(a_1, \ldots, a_n) = (b_1, \ldots, b_n)$,

has finitely many equivalence classes.

Equality atoms $(\mathbb{N}, =)$ and ordered atoms $(\mathbb{Q}, \leq)$ are oligomorphic. For equality atoms, the number of equivalence classes of n-tuples of atoms is equal to the n-th Bell number.

Definition 2.2. *A structure is called* ***homogeneous*** *if every isomorphism between its finite substructures extends to a full automorphism of the structure.*

Equality atoms and ordered atoms are homogeneous.

A useful property of homogeneous and at the same time oligomorphic structures is that these structures have *quantifier elimination*, i.e., every first-order formula over $\mathcal{A}$ is equivalent to a quantifier-free formula [Bojańczyk, 2019, Theorem 7.6].[2]

Below is an example of a structure that does not satisfy the above properties.

Example 2.3 ([Bojańczyk, 2019, Example 7.4])**.** *The structure $(\mathbb{Z}, \leq)$ of integers with order is neither oligomorphic nor homogeneous.*

Note that an automorphism in this structure is simply a translation, i.e. a function that adds some fixed integer to each argument. So the action of automorphism

[1]The concept of oligomophicity derives from the celebrated Ryll-Nardzewski theorem proved independently by [Ryll-Nardzewski, 1959, Engeler, 1959, Svenonius, 1959]. It follows from the theorem that countable oligomorphic structures are those which are ω-categorical, i.e. are the unique countable models of their first-order theory.

[2]This property will be used in Chapter 10.

on n-tuples of atoms has the form $(a_1, \dots, a_n) \mapsto (a_1 + k, \dots, a_n + k)$ for some $k \in \mathbb{Z}$. For $n = 2$ the equivalence classes are determined by numbers $a_1 - a_2$, which are infinitely many.

The structure is not homogeneous, because the partial function

$$1 \mapsto 1 \qquad 2 \mapsto 3$$

is an isomorphism between finite substructures and cannot be extended to an automorphism.

2.2. Cumulative hierarchy

The *cumulative hierarchy*[3] *over* $\mathcal{A}$ is a family of sets indexed by ordinals α (called ranks) such that

- There is only one set for $\alpha = 0$. Namely, the empty set.
- For $\alpha > 0$, a set of rank at most α, is any set that contains only atoms and sets of rank strictly smaller than α.

We can extend the action of atom automorphism π to the cumulative hierarchy over $\mathcal{A}$ by ordinal induction. For any set X from the cumulative hierarchy:

$$\pi(X) = \{\pi(x) : x \in X\}$$

2.3. Support

For any set of atoms $S \subseteq \mathcal{A}$, an automorphism $\pi \in Aut(\mathcal{A})$ is called an S-*automorphism* when

$$\pi(s) = s \text{ for } s \in S.$$

The set of S-automorphisms is denoted by $Aut(\mathcal{A}, S)$.

We say that the set S *supports* a set X from the cumulative hierarchy over $\mathcal{A}$ (or S is a *support* of X) if $X = \pi(X)$ for every S-automorphism π.

For example, every atom is supported by any set of atoms containing this atom. A tuple $(a_1, a_2, a_3) \in \mathcal{A}^3$ is supported by every set containing a_1, a_2, a_3.

We say that X is *finitely supported* if it is supported by some finite set of atoms. It is *hereditarily finitely supported* if it finitely supported, each of its elements is finitely supported, and so on recursively.

A function is finitely supported if it is a finitely supported set, when seen as a set of pairs (argument, value). We denote such functions by the symbol $\to_{fs}$. The set of all subsets of the set X that are finitely supported is denoted by $\mathcal{P}_{fs}(X)$.

It can be proved for some structures $\mathcal{A}$ that finite supports of a set X from the cumulative hierarchy are closed under intersection. As a result, every finitely supported set X has the least finite support with respect to inclusion. This property is satisfied for equality atoms (see [Gabbay and Pitts, 2002, Proposition 3.4] or [Bojańczyk et al., 2014, Corollary 9.4]) and for ordered atoms (see [Bojańczyk et al., 2014, Corollary 9.5]). On the other hand, an example of a structure that does not have the least finite support property is $(\mathbb{Z}, \leq)$ from Example 2.3.

[3] The above hierarchy is an extension of the standard (not over atoms) cumulative hierarchy called the von Neumann universe introduced in [Zermelo, 1930].

2.4. Sets with atoms

We already have all the necessary concepts to introduce the most important definition of this chapter.

Definition 2.4. *A **set with atoms over** $\mathcal{A}$ is any set X from the cumulative hierarchy over $\mathcal{A}$ which is hereditarily finitely supported.*

In other words, a set with atoms is any set defined with relations from the structure that refers to a finite number of specific atoms as constants. For better understanding, some examples are given below.

Example 2.5. *All of the following sets are sets with atoms (regardless of the choice of the relational structure $\mathcal{A}$):*

- *any classical set without atoms,*
- *the set of atoms $\mathcal{A}$,*
- *the set $\mathcal{A}^n$ of all n-tuples or the set $\mathcal{A}^{(n)}$ of non-repeating n-tuples for $n \in \mathbb{N}$,*
- *the set $\binom{\mathcal{A}}{n}$ of sets of atoms of size n for $n \in \mathbb{N}$,*
- *the set $\mathcal{A}^*$ of finite words over $\mathcal{A}$,*
- *the set $P_{fin}(\mathcal{A})$ of finite subsets of $\mathcal{A}$.*

Tuples and words can be encoded with sets using known techniques from set theory. For example, an ordered pair can be represented using Kuratowski's definition: $(a, b) := \{\{a\}, \{a, b\}\}$ [Kuratowski, 1921] and then naturally expanded to tuples and words.

All the above sets have empty support themselves, and their elements have finite supports. Sets with atoms that have non-empty support are, for example, a set of atoms different from some atom a in equality atoms, or a set of all closed intervals containing a in ordered atoms. In both cases, the singleton set $\{a\}$ is the least support.

It can be shown that for equality atoms a subset of $\mathcal{A}$ is a set with atoms if and only if it is finite or co-finite. For ordered atoms, sets with atoms that are subsets of $\mathcal{A}$ are exactly finite unions of intervals [Bojańczyk, 2019, Exercise 3.7 and 3.8].

Example 2.6. *Examples of sets that are not sets with atoms but are in the cumulative hierarchy over $\mathcal{A}$ are:*

- *the set of odd numbers for equality atoms $\{1, 3, 5, \ldots\}$,*
- *the set $\mathcal{A}^\omega$ of infinite words over $\mathcal{A}$,*
- *the power set $P(\mathcal{A})$.*

All these sets do not meet the condition of being hereditarily finitely supported, as either they have no finite support themselves or some of their elements fail to be finitely supported.

2.5. Equivariance

A set with atoms is *equivariant* if its support is empty. In other words, it can be represented without referring to any specific atoms. All sets from Example 2.5 are equivariant.

We can extend the definition to relations and functions if we treat them as sets of pairs. Unfolding the definition, a relation $R \subseteq X \times Y$ is equivariant if it is closed under automorphisms:

$$(x, y) \in R \quad \Rightarrow \quad (\pi(x), \pi(y)) \in R$$

for any automorphism $\pi \in Aut(\mathcal{A})$.

For functions, $f\colon X \to Y$ is equivariant if and only if

$$f(\pi(x)) = \pi(f(x))$$

for any automorphism $\pi \in Aut(\mathcal{A})$ and element $x \in X$. In other words, equivariant functions commute with automorphisms.

Example 2.7. *For equality atoms, the only equivariant function from $\mathcal{A}$ to $\mathcal{A}$ is the identity. There are exactly two equivariant functions from $\mathcal{A}^2$ to $\mathcal{A}$, namely projections. In the opposite direction, there is only the diagonal function ($a \mapsto (a, a)$). The mapping $(a, b) \mapsto \{a, b\}$ is the only equivariant function from $\mathcal{A}^{(2)}$ to $\binom{\mathcal{A}}{2}$. There is no equivariant function from $\binom{\mathcal{A}}{2}$ to $\mathcal{A}^{(2)}$ [Bojańczyk et al., 2014, Example 2.6]. However, there is an equivariant relation between $\binom{\mathcal{A}}{2}$ and $\mathcal{A}^{(2)}$ that relates $\{a, b\}$ with both (a, b) and (b, a).*

2.6. Orbit-finiteness

For a given finite set S of atoms an *S-orbit* of an element $x \in X$ is the set

$$\{\pi(x) \mid \pi \in Aut(\mathcal{A}, S)\}$$

If the set S is empty, then such an orbit is called an *equivariant orbit*. If S supports the set X, then S-orbits of elements of set X form the partition of the set X.

We call a set with atoms X *orbit-finite* if it is contained in a finite union of S-orbits for some finite set $S \subseteq \mathcal{A}$. It turns out that the orbit-finiteness property is independent of the choice of the set S. That is, if the set X satisfies this property for a certain set S, then it satisfies also for all finite sets of atoms that support X [Bojańczyk, 2019, Theorem 3.16].

A set is *hereditarily orbit-finite* if it is orbit-finite, all their elements are orbit-finite, and so on recursively.

Thus, atoms are oligomorphic if and only if the set $\mathcal{A}^n$ is orbit-finite for all $n > 0$.

Example 2.8. *The set $\mathcal{A}^2$ for equality atoms has two orbits: $\{(a, a) \mid a \in \mathcal{A}\}$ and $\{(a_1, a_2) \mid a_1 \neq a_2$ for $a_1, a_2 \in \mathcal{A}\}$.*

Example 2.9. *For ordered atoms the set $\mathcal{A}^2$ has three orbits: $\{(a, a) \mid a \in \mathcal{A}\}$, $\{(a_1, a_2) \mid a_1 < a_2$ for $a_1, a_2 \in \mathcal{A}\}$ and $\{(a_1, a_2) \mid a_1 > a_2$ for $a_1, a_2 \in \mathcal{A}\}$.*

Assuming that the atoms are oligomorphic, the orbit-finite sets from Example 2.5 are finite sets, $\mathcal{A}^n$, $\mathcal{A}^{(n)}$, $\binom{\mathcal{A}}{n}$ for $n \in \mathbb{N}$.

The sets $\mathcal{A}^*$ and $P_{fin}(\mathcal{A})$ are not orbit-finite because words or sets of a given size form separate orbits.

Orbit-finite sets are a very important class of sets with atoms. As we will see in the next section, these sets can be finely represented and allow us to perform finite computations on them.

2.7. Set expressions

We will represent (potentially infinite) sets with atoms with a finite number of orbits using set expressions.[4]

Definition 2.10. *For a fixed countable set of atom variables and a fixed structure $\mathcal{A}$, a* ***set expression*** *is a finite set, written $\{\xi_1, ..., \xi_n\}$ (or $\{\}$ for the empty set), of expressions of the form*

$$\xi \quad = \quad e \ : \ \phi \ \ \textit{for} \ \ a_1, \ldots, a_k \in \mathcal{A} \tag{2.1}$$

where

- *e is a set expression or an atom variable,*
- *ϕ is a first-order formula over the signature of atoms and over atom variables,*
- *$a_1, \ldots, a_k$ are atom variables.*

If $k = 0$ then we write simply $e \ : \ \phi$ instead of (2.1). We also omit ϕ if it is the always true formula $\top$.

The expression e or the formula ϕ can contain atom variables other than $a_1, \ldots, a_k$. Therefore, the set expression can contain free atom variables. The set of free atom variables[5] is defined inductively by:

$$\begin{aligned} \mathrm{FV_A}(a) &= \{a\} \\ \mathrm{FV_A}(e \colon \phi \text{ for } a_1, \ldots, a_k \in \mathcal{A}) &= (\mathrm{FV_A}(e) \cup \mathrm{FV_A}(\phi)) \setminus \{a_1, \ldots, a_k\} \\ \mathrm{FV_A}(\{\xi_1, \ldots, \xi_n\}) &= \mathrm{FV_A}(\xi_1) \cup \cdots \cup \mathrm{FV_A}(\xi_n) \end{aligned} \tag{2.2}$$

where $\mathrm{FV_A}(\phi)$ is the standard set of free atom variables in a first-order formula.

A *valuation* for a set expression e is a function $\nu \colon \mathrm{FV_A}(e) \to \mathcal{A}$. A set expression e together with a valuation ν denotes a set $[\![e]\!]_\nu$ in the expected way:

$$\begin{aligned} [\![a]\!]_\nu &= \nu(a) \\ [\![e \colon \phi \text{ for } a_1, \ldots, a_k \in \mathcal{A}]\!]_\nu &= \{[\![e]\!]_{\nu[a_i \to v_i]_{i=1}^k} \mid v_1, \ldots, v_k \in \mathcal{A} \text{ s.t. } \mathcal{A}, \nu[a_i \to v_i]_{i=1}^k \vDash \phi\} \\ [\![\{\xi_1, \ldots, \xi_n\}]\!]_\nu &= [\![\xi_1]\!]_\nu \ \cup \ \cdots \ \cup \ [\![\xi_k]\!]_\nu \end{aligned} \tag{2.3}$$

[4] A comparison to other representations of sets with atoms can be found in Chapter 9.

[5] We use the notation with the subscript A to distinguish from the definition of free term variables (see Section 3.4).

where $\mathcal{A}, \nu \vDash \phi$ means that the formula ϕ holds in $\mathcal{A}$ with the valuation ν of the free variables in ϕ. By definition it follows that the set $[\![e]\!]_\nu$ is in the cumulative hierarchy.

For example, over equality atoms, the expression

$$\{(a_1, a_2) : \neg(a_1 = a_2) \text{ for } a_1, a_2 \in \mathcal{A}\},$$

with the empty valuation denotes the set of ordered pairs of distinct atoms. The same definition works for ordered atoms, where we see $a_1 = a_2$ as shorthand for $a_1 \le a_2 \wedge a_2 \le a_1$. Over ordered atoms, the expression

$$\{a_1 : a_1 \le b \text{ for } a_1 \in \mathcal{A},\ a_2 : c \le a_2 \text{ for } a_2 \in \mathcal{A}\}$$

with a valuation $b \mapsto 2$, $c \mapsto 5$, denotes the set of all atoms outside of the open interval $(2; 5)$. The same set is denoted by the expression

$$\{a : a \le b \vee c \le a \text{ for } a \in \mathcal{A}\} \tag{2.4}$$

with the same valuation.

We say that a set of the form $[\![e]\!]_\nu$ is *definable* over $\mathcal{A}$.

Finally, we show a theorem that confirms the correctness of the representation of hereditarily orbit-finite sets by set expressions.

Theorem 2.11. *A set is definable over $\mathcal{A}$ if and only if it is a hereditarily orbit-finite set with atoms over $\mathcal{A}$.*

Proof. Inductive proof: the left-to-right implication on the set expression representing the definable set, the right-to-left implication on the cumulative hierarchy of the hereditarily orbit-finite set. The proof uses the property of oligomorphism (see assumptions in Section 2.1). Details can be found in [Bojańczyk, 2019, Theorem 4.10]. □

CHAPTER 3

Programming language

In chapter 2 we saw how hereditarily orbit-finite sets with atoms can be represented by set expressions. We shall now introduce a calculation model for definable sets. For this purpose, the syntax and semantics of a programming language called Nλ will be presented. Because it is a typed functional language, the simply typed lambda calculus [Church, 1940, Barendregt, 1993] will be a natural starting point for presenting its semantics. The classical calculus will be expanded with constructions on sets with atoms.

The user of the programming language who implements the calculation of the set with atoms will see the result of their program in the form of the set expression. But this does not mean that the user will build sets by defining set expression explicitly. Instead, it will be a natural process of modifying the sets starting with an empty set, or the set of all atoms. More complex sets can be obtained by successive use of operations such as adding or removing elements, computing the sum or intersection of a set, filtering or applying a given function to each element of a set. This is a process analogous to classical programming on finite sets.

The language described here, based on the simply typed lambda calculus, will be the *core* language. Only the necessary structures needed to perform calculations on definable sets will be introduced. The simplicity of the core language will facilitate the presentation of semantics and the proof of expected properties of the language in the following chapters. More complex elements of a fully-fledged modern programming language, like recursion or polymorphism, will be introduced in Chapter 8.

Two semantics of the language will be introduced in this chapter: denotational and operational. The first one will make it easier for the reader to understand the meaning of particular expressions of the language by presenting the interpretation of terms in the form of sets and functions. The second one will present a computational model by listing reduction rules for individual terms, formally describing the steps leading to the calculation of the result.

All considerations below assume that the language parameter is some chosen relational structure $\mathcal{A}$ of atoms (see Section 2.1). For the sake of language presentation, we consider only two relational structures: equality atoms and ordered atoms. The possibility of extending the language with other structures is described in Section 8.7.

3.1. Types

Nλ is a strongly typed language, so we start presenting its semantics with its types. We introduce a division into *element types* and *function types.* This stems from the assumption that sets will only contain values of certain types that exclude function types. This is for the following reasons:

- It can be expected that elements of a set are equipped with a computable equality operation. However, for functions it is difficult (or impossible) to define such an operation. Additionally, for performance reasons, sets often require a defined order on their elements. This is standard in most functional languages (e.g. in ML or Haskell) and will also be seen in the Nλ implementation (see Section 10.11 for the types division in the implementation).
- It simplifies proofs of language properties such as subject reduction, Church-Rosser property or strong normalization.

The types in Nλ are as follows:

$$\begin{aligned} \tau &::= \mathbb{A} \mid \mathbb{B} \mid \mathbb{S}\tau \\ \alpha, \beta &::= \tau \mid \alpha \to \beta \end{aligned}$$

Element types, denoted by τ, can take one of the following forms:

- $\mathbb{A}$ is the type of atoms from the relational structure $\mathcal{A}$,
- $\mathbb{B}$ is a type of boolean values that will then be expanded to first-order logic formulas,
- $\mathbb{S}$ is an unary type constructor that can only be applied to an element type. The intuition is that values of type $\mathbb{S}\tau$ are (definable) sets of values of type τ.

In addition to the above types, there is also a classical type constructor $\to$ that builds function types. Letters α and β range over element or function types.

Let us now define the semantic domains for the above types explaining their meaning:

$$\begin{aligned} [\![\mathbb{A}]\!] &= \mathcal{A} \\ [\![\mathbb{B}]\!] &= \{tt, f\!f\} \\ [\![\mathbb{S}\tau]\!] &= \mathcal{P}_{fs}([\![\tau]\!]) \\ [\![\alpha \to \beta]\!] &= [\![\alpha]\!] \to_{fs} [\![\beta]\!] \end{aligned}$$

The semantic domain for type $\mathbb{A}$ is the set of all atoms, while for type $\mathbb{B}$, the set of two elements containing the values *true* and *false.* For the type $\mathbb{S}\tau$, the domain is the power set for domain of the type τ, but limited to finitely supported subsets (recall the definition in Section 2.3). This type represents sets with atoms which, by definition, are finitely supported.

Finally, we have a domain for a function type, which as a set of functions is also limited to finitely supported functions (see Section 2.3).

3.2. Basic functional language

To provide a functional language that creates and operates on definable sets, we expand standard simply typed lambda calculus terms: variables, abstractions and applications with a collection of constants. Terms of the core language are defined by the grammar:

$$M ::= x \;\Big|\; \lambda x.M \;\Big|\; MM \;\Big|\; C$$

All of these terms form the Nλ language syntax and are available directly to the programmer.

C ranges over the following collection of typed constants:

$\texttt{empty} : \mathbb{S}\tau$	(the empty set)
$\texttt{atoms} : \mathbb{S}\mathbb{A}$	(the set of all atoms)
$\texttt{insert} : \tau \to \mathbb{S}\tau \to \mathbb{S}\tau$	(adds an element to a set)
$\texttt{map} : (\tau_1 \to \tau_2) \to \mathbb{S}\tau_1 \to \mathbb{S}\tau_2$	(applies a function element-wise)
$\texttt{sum} : \mathbb{S}\mathbb{S}\tau \to \mathbb{S}\tau$	(union of a family of sets)
$\texttt{true}, \texttt{false} : \mathbb{B}$	(boolean values)
$\texttt{not} : \mathbb{B} \to \mathbb{B}$	(logical negation)
$\texttt{and}, \texttt{or} : \mathbb{B} \to \mathbb{B} \to \mathbb{B}$	(conjunction and disjunction)
$\texttt{isEmpty} : \mathbb{S}\tau \to \mathbb{B}$	(emptiness test)
$\texttt{if} : \mathbb{B} \to \alpha \to \alpha \to \alpha$	(conditional)

Additionally, we include some constants that depend on the signature of the underlying structure $\mathcal{A}$ of atoms. For equality atoms we take simply:

$$\texttt{eq}_{\mathbb{A}} : \mathbb{A} \to \mathbb{A} \to \mathbb{B} \qquad \text{(equality relation on atoms)}$$

and for ordered atoms, additionally:

$$\texttt{leq}_{\mathbb{A}} : \mathbb{A} \to \mathbb{A} \to \mathbb{B} \qquad \text{(ordering relation on atoms).}$$

If we wanted to introduce other logical structures $\mathcal{A}$, then this part of the language might change.

The formal semantics for all these operations will be introduced in the next section. For now, the meaning of operations should be intuitively clear as specified on the right above.

Note that the *if... then... else...* construct is provided here by the `if` function. In many languages, such a conditional expression is handled by an additional language statement. From the point of view of language semantics, both approaches are acceptable. The function was chosen here because it requires no additional element in the grammar. But the language implementation provides both possible forms: a function and a statement (details are described in Section 10.9).

The grammar of the Nλ language is described using a Curry-style type system [Curry and Feys, 1958]. This means that terms do not have an explicit type assigned and may have many different types depending on the context in which they are used. In particular, the constant `empty` may have type $\mathbb{S}\tau$ for any element type τ. Whereas the function `if` may have type $\mathbb{B} \to \alpha \to \alpha \to \alpha$ for any type α. More on the difference between the Curry's and Church's approaches to the type system and on extending our language with polymorphism can be found in Section 8.5.

From now on, we will use the following notation: letters M, N, P, Q range over terms, letters x, y, z are term variables.

Using the above core functions, we can define additional functions that allow to perform many expected calculations on definable sets. For example, one can define functions such as:

$\texttt{singleton} : \tau \to \mathbb{S}\tau$	(singleton set with a given element)
$\texttt{filter} : (\tau \to \mathbb{B}) \to \mathbb{S}\tau \to \mathbb{S}\tau$	(set with elements that satisfy a predicate)
$\texttt{delete} : \tau \to \mathbb{S}\tau \to \mathbb{S}\tau$	(removes an element from a given set)
$\texttt{exists} : (\tau \to \mathbb{B}) \to \mathbb{S}\tau \to \mathbb{B}$	(is there an element that meets a predicate)
$\texttt{forall} : (\tau \to \mathbb{B}) \to \mathbb{S}\tau \to \mathbb{B}$	(whether all elements satisfy a predicate)
$\texttt{contains} : \mathbb{S}\tau \to \tau \to \mathbb{B}$	(checking if a set contains a given element)
$\texttt{union} : \mathbb{S}\tau \to \mathbb{S}\tau \to \mathbb{S}\tau$	(union of two sets)
$\texttt{intersection} : \mathbb{S}\tau \to \mathbb{S}\tau \to \mathbb{S}\tau$	(intersection of two sets)
$\texttt{isSubsetOf} : \mathbb{S}\tau \to \mathbb{S}\tau \to \mathbb{B}$	(subset relation)
$\texttt{eq}_{\mathbb{S}\tau} : \mathbb{S}\tau \to \mathbb{S}\tau \to \mathbb{B}$	(equality on sets)
$\texttt{implies} : \mathbb{B} \to \mathbb{B} \to \mathbb{B}$	(implication)
$\texttt{eq}_{\mathbb{B}} : \mathbb{B} \to \mathbb{B} \to \mathbb{B}$	(equivalence of booleans)

with the following implementation:

```
     singleton x = insert x empty
      filter f s = sum (map (λx.if (f x) (singleton x) empty) s)
      delete x s = filter (λy.not (eq_τ x y)) s
      exists f s = not (isEmpty (filter f s))
      forall f s = isEmpty (filter (λx.not (f x)) s)
    contains s x = exists (eq_τ x) s
       union s t = sum (insert s (singleton t))
intersection s t = filter (contains t) s
  isSubsetOf s t = forall (contains t) s
       eq_Sτ s t = and (isSubsetOf s t) (isSubsetOf t s)

     implies p q = or (not p) q
        eq_B p q = and (implies p q) (implies q p)
```

The equality function `eq` requires special attention. In the examples above, we see this function in three different versions.

- For atoms we have the function $\mathtt{eq}_{\mathbb{A}}$ which results from the structure $\mathcal{A}$.
- For logical values we have the function $\mathtt{eq}_{\mathbb{B}}$ defined above.
- For sets, the equality function $\mathtt{eq}_{\mathbb{S}\mathcal{T}}$ is defined recursively by indirectly using $\mathtt{eq}_{\mathcal{T}}$ in the definition of the `contains` function.

With the functions $\mathtt{eq}_{\mathbb{A}}$, $\mathtt{eq}_{\mathbb{B}}$ and $\mathtt{eq}_{\mathbb{S}\mathcal{T}}$ one can define a general function:

$$\mathtt{eq} : \tau \to \tau \to \mathbb{B}$$

which can resolve equality for any pair of element type values.[1]

This collection of functions already gives great possibilities for the implementation of various programs. For example, the set represented by the expression (2.4) from the Chapter 2 can be calculated using the program:

$$\mathtt{filter}\ (\lambda \mathtt{x}.\mathtt{or}\ (\mathtt{leq}_{\mathbb{A}}\ \mathtt{x}\ \mathtt{y})\ (\mathtt{leq}_{\mathbb{A}}\ \mathtt{z}\ \mathtt{x}))\ \mathtt{atoms} \tag{3.1}$$

For the valuation of free variables $y \mapsto 2,\ z \mapsto 5$ we get the set:

$$\{a : a \leq 2 \vee 5 \leq a \text{ for } a \in \mathcal{A}\}$$

The more complex example below:

```
sum
  (map
    (λx.map
        (λy.filter
            (λz.(and
                 (and (leq_A x z) (leq_A z y))
                 (and (leq_A x u) (leq_A u y)))
            atoms)
        atoms)
    atoms)
```

computes the set of all closed intervals that contain a certain value u. For the valuation of $u \mapsto 7$ the result will be as follows:

$$\{\{c : a \leq c\ \wedge\ a \leq 7\ \wedge\ c \leq b\ \wedge\ 7 \leq b \text{ for } c \in \mathcal{A}\} : \text{for } a, b \in \mathcal{A}\}$$

3.3. Denotational semantics

The constants that are part of the Nλ language have so far been described informally. To formalize the meaning of individual terms, we will now introduce denotational

[1] The implementation of the `eq` function for more types is described in Section 10.11.

semantics. For each term M from the grammar of the language, a denotation will be assigned in the form of mathematical objects, such as a function or set. It will be denoted by $[\![M]\!]^\rho$, where ρ is a valutation of free term variables of the term M.

The denotation of a term will be an element of the semantic domain for the type of this term. It means that for every term M of type α:

$$[\![M]\!]^\rho \in [\![\alpha]\!]$$

The semantics for basic terms of lambda calculus are standard:

$$[\![x]\!]^\rho = \rho(x) \tag{D1}$$
$$[\![\lambda x.M]\!]^\rho(v) = [\![M]\!]^{\rho[x\mapsto v]} \tag{D2}$$
$$[\![M_1 M_2]\!]^\rho = [\![M_1]\!]^\rho([\![M_2]\!]^\rho) \tag{D3}$$

The variable x is assigned its value when evaluated with ρ. We assign a function to abstraction, which for a given value v returns the denotation of the term M with the extension of the valuation ρ by mapping the variable x to the value v. The denotation for an application is defined recursively. It is an application of a function that is denoted by term M_1 to a value that is denoted by term M_2.

The denotational semantics for the constructors of the empty set and the set of atoms are quite obvious:

$$[\![\texttt{empty}]\!]^\rho = \emptyset \tag{D4}$$
$$[\![\texttt{atoms}]\!]^\rho = \mathcal{A} \tag{D5}$$

The denotations for the next constants are standard mathematical operations on sets:

$$[\![\texttt{insert}]\!]^\rho(v)(S) = \{v\} \cup S \tag{D6}$$
$$[\![\texttt{map}]\!]^\rho(f)(S) = \{f(v) \mid v \in S\} \tag{D7}$$
$$[\![\texttt{sum}]\!]^\rho(S) = \bigcup S \tag{D8}$$

The constant `insert`, which is intended to add an element to the set, denotes the operation of the union of the given set S and the singleton containing the value v. If `insert` is of a type $\tau \to \mathbb{S}\tau \to \mathbb{S}\tau$ for some type τ, then $v \in [\![\tau]\!]$, $S \in \mathcal{P}_{fs}([\![\tau]\!])$ and the result is also in the set $\mathcal{P}_{fs}([\![\tau]\!])$.

Denotation of the constant `map` is a function that applies element-wise a given function f to a given set S. If `map` has a type $(\tau_1 \to \tau_2) \to \mathbb{S}\tau_1 \to \mathbb{S}\tau_2$, then $f \in [\![\tau_1]\!] \to_{fs} [\![\tau_2]\!]$, $S \in \mathcal{P}_{fs}([\![\tau_1]\!])$ and the result is an element of the set $\mathcal{P}_{fs}([\![\tau_2]\!])$.

The meaning for the constant `sum` is the set-theoretic union of a set of sets. Therefore $S \in \mathcal{P}_{fs}(\mathcal{P}_{fs}([\![\tau]\!]))$ and the result belongs to $\mathcal{P}_{fs}([\![\tau]\!])$ if $\texttt{sum} : \mathbb{S}\mathbb{S}\tau \to \mathbb{S}\tau$ for some element type τ.

The denotations for constants operating on truth values are described in a straightforward manner:

$$[\![\texttt{true}]\!]^\rho = tt \tag{D9}$$

$$[\![\texttt{false}]\!]^\rho = ff \tag{D10}$$

$$[\![\texttt{not}]\!]^\rho(b) = \begin{cases} tt & \text{if } b = ff \\ ff & \text{otherwise} \end{cases} \tag{D11}$$

$$[\![\texttt{or}]\!]^\rho(b_1)(b_2) = \begin{cases} tt & \text{if } b_1 = tt \text{ or } b_2 = tt \\ ff & \text{otherwise} \end{cases} \tag{D12}$$

$$[\![\texttt{and}]\!]^\rho(b_1)(b_2) = \begin{cases} tt & \text{if } b_1 = tt \text{ and } b_2 = tt \\ ff & \text{otherwise} \end{cases} \tag{D13}$$

The denotational definitions for constants `isEmpty` and `if` are also not surprising:

$$[\![\texttt{isEmpty}]\!]^\rho(S) = \begin{cases} tt & \text{if } S = \emptyset \\ ff & \text{otherwise} \end{cases} \tag{D14}$$

$$[\![\texttt{if}]\!]^\rho(b)(v_1)(v_2) = \begin{cases} v_1 & \text{if } b = tt \\ v_2 & \text{otherwise} \end{cases} \tag{D15}$$

Finally, there are semantic clauses for constants derived from the signature of the relational structure $\mathcal{A}$:

$$[\![\texttt{eq}_{\mathbb{A}}]\!]^\rho(a)(b) = \begin{cases} tt & \text{if } a = b \\ ff & \text{otherwise} \end{cases} \tag{D16}$$

$$[\![\texttt{leq}_{\mathbb{A}}]\!]^\rho(a)(b) = \begin{cases} tt & \text{if } a \leq b \\ ff & \text{otherwise} \end{cases} \tag{D17}$$

The above denotations determine the equality of atoms or the relation $\leq$ if ordered atoms are considered.

3.4. Logic-based functional language

From the description of the language in Section 3.2 it may not be clear how to implement operations postulated in it. For example, how to implement the function `map` so that a function can be applied to every element of the infinite set of atoms in finite time? The denotational semantics in Section 3.3 formally defined what exactly each operation performs, but it does not answer the questions of how to do it. Therefore, in order to answer this question, a (small-step) reduction semantics of the core functional language will be introduced, which will precisely present the computational model.

The reduction rules will be presented in Section 3.7. Here we present the general ideas behind the semantics:

- Values of set types $\mathbb{S}\tau$ are represented not by enumerating their elements (that would be impossible, as usually they are infinite sets), but by set expressions as in Section 2.7.

- A new kind of variables will be introduced into the language, namely atom variables. They will only have type $\mathbb{A}$ and, unlike term variables, they can appear in first-order formulas and contexts.
- Values of type $\mathbb{B}$ are not just boolean values; they are rather first-order formulas over atom variables.
- Terms are evaluated in contexts that specify what relations hold between atom variables in them.
- Sometimes a condition ϕ in a conditional expression ($\texttt{if}\ \phi\ a\ b$) is neither tautologically true nor false. In such cases it is not clear whether the conditional should evaluate to a or b. In this case a *variant* is created that has value a or b, depending on the value of ϕ.

Formally, the grammar of terms is extended to the following form:

$$M ::= x \;\Big|\; \lambda x.M \;\Big|\; MM \;\Big|\; C \;\Big|\; a \;\Big|\; \phi \;\Big|\; \{M\colon \phi \text{ for } \sigma, \ldots, M\colon \phi \text{ for } \sigma\} \;\Big|\; a\colon \phi | \cdots | a\colon \phi$$

where

- C ranges over the same set of typed constants as in Section 3.2,
- a ranges over a fixed infinite set of atom variables (disjoint from the set of program variables such as x),
- ϕ ranges over the set of first-order formulas (with quantifiers allowed) over the signature of $\mathcal{A}$ and over atom variables,
- σ ranges over finite list of atom variables.

Variables

As already mentioned, the variables in the language are divided into term variables and atom variables. For the sake of clarity, the former will be denoted with the letters x, y, z, and the latter with the letters a, b, c.

The fundamental difference between these two groups of variables, and the reason why this division was introduced, stems from the process of creating first-order formulas. Atom variables (as opposed to term variables) are elements from which such formulas can be built. As we will see in the reduction rules, from the term

$$\texttt{eq}_{\mathbb{A}}\ a\ b$$

we can compute the formula $a = b$. On the other hand, the term

$$\texttt{eq}_{\mathbb{A}}\ x\ y$$

will not be reduced until the variables x and y are replaced with appropriate values. Allowing term variables to appear in formulas would complicate the substitution process and make it more difficult to present the computational model of the language.

As for the type of variables, an atom variable can only be of type $\mathbb{A}$, while a term variable can represent any term of any type.

Terms are considered up to α-equivalence, defined as expected on both term and atom variables. For example,

$$\{a\colon \neg(a = c) \text{ for } a\} \qquad \text{and} \qquad \{b\colon \neg(b = c) \text{ for } b\}$$

are α-equivalent. Formally, we are studying the set of terms quotiented by α-equivalence.

Both groups of variables can take the form of free or bound variables. Standard binding of term variables is by abstraction $\lambda x.M$. Atom variables can be bound in two ways: by quantifiers in first-order formulas and by the construction *for* in a set expression:

$$M\colon \phi \text{ for } a_1, \ldots, a_k$$

Barendregt's Variable Convention [Barendregt, 1984] is used in all definitions, theorems, and the proofs, which assumes that all bound variables are chosen to be different from the free variables.

A properly written Nλ program cannot have free term variables. Only subterms can contain free term variables, but in an overall program, each such variable must be bound by some λ. This is not the case with atom variables, which can also appear as free throughout the entire program.

The set of free term variables defined in a standard way naturally expands into new language constructions:

$$\begin{aligned}
\mathrm{FV_T}(x) &= \{x\} \\
\mathrm{FV_T}(\lambda x.M) &= \mathrm{FV_T}(M) \setminus \{x\} \\
\mathrm{FV_T}(MN) &= \mathrm{FV_T}(M) \cup \mathrm{FV_T}(N) \\
\mathrm{FV_T}(C) &= \emptyset \\
\mathrm{FV_T}(a) &= \emptyset \\
\mathrm{FV_T}(\phi) &= \emptyset \\
\mathrm{FV_T}(\{M_1 : \phi_1 \text{ for } \sigma_1, \ldots, M_n : \phi_n \text{ for } \sigma_n\}) &= \mathrm{FV_T}(M_1) \cup \cdots \cup \mathrm{FV_T}(M_n) \\
\mathrm{FV_T}(a_1 : \phi_1 | \cdots | a_n : \phi_n) &= \emptyset
\end{aligned}$$

As can be seen, terms that do not contain term variables are constants, atom variables, formulas, and variants.

A *closed term* M is one that contains no free term variables: $\mathrm{FV_T}(M) = \emptyset$.

Similarly, the process of replacing free term variables by expressions naturally extends the standard definition to the new terms. This process is called substitution and is necessary for β-reduction.

$$\begin{aligned}
x[N/x] &= N \\
y[N/x] &= y \quad \text{if } x \neq y \\
(\lambda y.M)[N/x] &= \lambda y.(M[N/x]) \quad \text{for } x \neq y \text{ and } y \notin \mathrm{FV_T}(N) \\
(M_1 M_2)[N/x] &= M_1[N/x]M_2[N/x] \\
C[N/x] &= C \\
a[N/x] &= a \\
\phi[N/x] &= \phi \\
\{M_1 : \phi_1 \text{ for } \sigma_1, \ldots, M_n : \phi_n \text{ for } \sigma_n\}[N/x] &= \{M_1[N/x] : \phi_1 \text{ for } \sigma_1, \ldots, M_n[N/x] : \phi_n \text{ for } \sigma_n\} \\
(a_1 : \phi_1 | \cdots | a_n : \phi_n)[N/x] &= a_1 : \phi_1 | \cdots | a_n : \phi_n
\end{aligned}$$

In the case of atom variables, the set of free variables is defined as follows:

$$\begin{aligned}
\mathrm{FV_A}(x) &= \emptyset \\
\mathrm{FV_A}(\lambda x.M) &= \mathrm{FV_A}(M) \\
\mathrm{FV_A}(MN) &= \mathrm{FV_A}(M) \cup \mathrm{FV_A}(N) \\
\mathrm{FV_A}(C) &= \emptyset \\
\mathrm{FV_A}(a) &= \{a\} \\
\mathrm{FV_A}(\{\ldots, M\colon \phi \text{ for } a_1, \ldots, a_k, \ldots\}) &= \cdots \;\cup\; \big(\mathrm{FV_A}(M) \cup \mathrm{FV_A}(\phi)\big) \smallsetminus \{a_1, \ldots, a_k\} \;\cup\; \cdots \\
\mathrm{FV_A}(a_1 : \phi_1 | \cdots | a_n : \phi_n) &= \{a_1, \ldots, a_n\} \cup \mathrm{FV_A}(\phi_1) \cup \;\cdots\; \cup \mathrm{FV_A}(\phi_n)
\end{aligned}$$

where $\mathrm{FV_A}(\phi)$ is the standard set of free atom variables in a first-order formula. The above is of course consistent with what has already been defined for set expressions (2.2).

For atom variables, the set of all atom variables $\mathrm{V_A}(M)$ in the term M will also be useful. It is defined analogously to the set of free atom variables up to the formula and the set expression, where bound variables are also taken into account:

$$\mathrm{V_A}(\{\ldots, M\colon \phi \text{ for } a_1, \ldots, a_k, \ldots\}) = \cdots \;\cup\; \mathrm{V_A}(M) \cup \mathrm{V_A}(\phi) \cup \{a_1, \ldots, a_k\} \;\cup\; \cdots$$

Note that this set is not well-defined for α-equivalent terms. But it will be used in a safe context as a side condition of a reducing rule to check for the need for α-conversion.

Internal representation

The terms from the syntax of the Nλ language explained so far, that is:

$$x \;\Big|\; \lambda x.M \;\Big|\; MM \;\Big|\; C \;\Big|\; a$$

are terms directly available to the programmer. So Nλ programs consist of exactly the above expressions.

Now we will discuss terms that are not available to the programmer, but appear only as final or intermediate values in the reduction semantics. These terms are an internal representation of formulas, sets, and variants:

$$\phi \;\Big|\; \{M\colon \phi \text{ for } \sigma, \ldots, M\colon \phi \text{ for } \sigma\} \;\Big|\; a\colon \phi | \cdots | a\colon \phi$$

ϕ represents a first-order formula over $\mathcal{A}$. We will use standard logical symbols to present such formulas: $\top, \bot, \vee, \wedge, \neg, \forall, \exists$. An example formula looks like this:

$$\forall a.\ \exists b.\ a = b \;\vee\; \neg(a = b)$$

It is worth emphasizing that for the sake of clarity for the formulas *true* and *false* we use a different notation than for the elements of the semantic domain $\mathbb{B}$ in denotational semantics: $\{\mathit{tt}, \mathit{ff}\}$.

Definable sets are represented in the syntax by expressions

$$\{M_1 : \phi_1 \text{ for } \sigma_1, \ldots, M_n : \phi_n \text{ for } \sigma_n\}$$

according to the idea presented in Section 2.7.

To limit the length of a set expression, tuples of bound atom variables $(a_1, \dots, a_k)$ in the syntax are denoted by σ. We denote the concatenations of two tuples of atom variables with $\cup$. If $\sigma_1 = (a_1, \dots, a_k)$ and $\sigma_2 = (b_1, \dots, b_n)$, then instead of writing

$$\dots \text{ for } a_1, \dots, a_k, b_1, \dots, b_n$$

we just write

$$\dots \text{ for } \sigma_1 \cup \sigma_2.$$

We omit "for σ" if σ is empty. We also omit ϕ if it is the always true formula $\top$.

Expressions of the form

$$a_1 \colon \phi_1 | \cdots | a_n \colon \phi_n$$

are called *variants*. They look syntactically similar to set expressions for sets with atom variables $\{a_1 \colon \phi_1, \dots, a_n \colon \phi_n\}$, but their meaning is very different. A variant as above does not denote a set of atom variables, but a single atom variable whose identity cannot be determined at the moment and will be fixed depending on which one of the formulas ϕ_1 to ϕ_n holds. By definition, exactly one of these formulas holds.

In the core language, variants contain only atom variables as values. The extension of the Nλ language (described in Chapter 8) and its implementation (Chapter 10) introduce variants for other types. Then any values of a primitive type, such as numbers or texts, can take the form of a variant, which then takes a more general form: $M \colon \phi | \cdots | M \colon \phi$.

3.5. Extended denotational semantics

Extending the basic functional language with new terms also requires defining denotation semantics for these terms. But first, it should be noted that dividing variables into two kinds: atom variables and term variables, also necessitates the existence of two valuations. The current denotational semantics have been denoted as $[\![M]\!]^\rho$, where ρ is the valuation of free variables. From now on, we will have two valuations:

- valuation ρ of free term variables that assigns terms to the variables from $\mathrm{FV_T}(M)$,
- valuation ν as a function $\nu \colon \mathrm{FV_A}(M) \to \mathcal{A}$, which assings atoms to the free atom variables.

The denotation of the term M will be denoted by $[\![M]\!]^\rho_\nu$.

The new valuation does not affect the semantics of the existing terms. Thus, the equations (D1) - (D17) can be left unchanged by adding only a subscript ν to them and also for recursive calls.

The denotations for additional terms are as follows:

$$[\![\mathtt{a}]\!]^\rho_\nu = \nu(a) \tag{D18}$$

The value of an atom variable results from the valuation of this kind of variables.

$$[\![\phi]\!]^{\rho}_{\nu} = \begin{cases} tt & \text{if } \mathcal{A}, \nu \vDash \phi \\ f\!f & \text{otherwise} \end{cases} \tag{D19}$$

Any first-order formula ϕ for the valuation ν of all its free atom variables is always true or false. Depending on which of the two situations occurs, the denotation for such a formula is the appropriate value from the set $\{tt, f\!f\}$.

The meaning of a set expression is as follows:

$$[\![\{M_1 \colon \phi_1 \text{ for } \sigma_1, \dots M_n \colon \phi_n \text{ for } \sigma_n\}]\!]^{\rho}_{\nu} = \bigcup_{1 \le i \le n} \{[\![M_i]\!]^{\rho}_{\nu_i} \mid v^i_1, \dots, v^i_{k_i} \in \mathcal{A} \text{ s.t. } \mathcal{A}, \nu_i \vDash \phi_i\} \tag{D20}$$

where $\sigma_i = (a^i_1, \dots, a^i_{k_i})$ and $\nu_i = \nu[a^i_j \to v^i_j]^{k_i}_{j=1}$ for $i = 1, \dots, n$. The meaning of a term in the form of a set expression is the sum of sets for subexpressions. The set for the i-th subexpression is the set of denotations of the subterm M_i for the valuation ν extended by the mapping of atom variables from σ_i with values $v^i_1, \dots, v^i_{k_i}$ satisfying the formula ϕ_i. Such a definition is consistent with the valuation of the set expression (2.3) defined in the previous chapter.

$$[\![a_1 \colon \phi_1 | \cdots | a_n \colon \phi_n]\!]^{\rho}_{\nu} = \begin{cases} \nu(a_1) & \text{if } \mathcal{A}, \nu \vDash \phi_1 \\ \cdots \\ \nu(a_n) & \text{if } \mathcal{A}, \nu \vDash \phi_n \end{cases} \tag{D21}$$

The value of the term in the form of a variant is the denotation of the atom variable a_i for which the formula ϕ_i is satisfied ($i \in \{1, \dots, n\}$). Recall that a correct variant is one in which exactly one formula is satisfied for every valuation ν and only for such variants the above denotational function is well-defined. Therefore, we can look at the denotational function as a partial function that assigns meanings only to well-formed terms.

3.6. Typing rules

It remains to assign types to each of the terms presented. This is done by defining a typing relation between terms and types, in an expected way.

First, we introduce the *typing context* Γ, which is a set of typing assumptions $\{x_1 : \alpha_1, \dots, x_n : \alpha_n\}$ where $x : \alpha$ means that a term variable x has a type α.

The *typing relation* $\Gamma \vdash M : \alpha$ indicates that M is a term of type α in context Γ. In this case M is said to be well-typed (having type α).

The typing rules for the grammar presented above are as follows:

$$\frac{x : \alpha \in \Gamma}{\Gamma \vdash x : \alpha} \qquad \frac{\Gamma \cup \{x : \alpha\} \vdash M : \beta}{\Gamma \vdash \lambda x.M : \alpha \to \beta}$$

$$\frac{\Gamma \vdash M : \alpha \to \beta \qquad \Gamma \vdash N : \alpha}{\Gamma \vdash MN : \beta} \qquad \frac{C \text{ is a constant of type } \alpha}{\Gamma \vdash C : \alpha}$$

$$\Gamma \vdash a : \mathbb{A} \qquad \Gamma \vdash \phi : \mathbb{B}$$

$$\frac{\Gamma \vdash M_1 \colon \tau \ \ldots \ \Gamma \vdash M_n \colon \tau}{\Gamma \vdash \{M_1 \colon \phi_1 \text{ for } \sigma_1, \ldots, M_n \colon \phi_n \text{ for } \sigma_n\} \colon \mathbb{S}\tau} \qquad \Gamma \vdash (a_1 \colon \phi_1 | \cdots | a_n \colon \phi_n) \colon \mathbb{A}$$

The first four rules are standard for simply typed lambda calculus. All atom variables are of type $\mathbb{A}$, while all first-order formulas are of type $\mathbb{B}$. All elements of the set must have the same element type τ that determines the type $\mathbb{S}\tau$ of the set. Variant terms always have type $\mathbb{A}$ because they represent one of the included atom variables.

3.7. Reduction rules

The reduction rules of the small-step operational semantics are evaluated in the context of a formula over atom variables. The basic semantic statements are of the form:

$$\psi \vdash M \to N$$

where ψ is a satisfiable formula and M, N are program terms. We just write $M \to N$, when the context is trivial, i.e. $\psi = \top$. We begin the process of reducing the entire program from the trivial context and a possible change of context may occur when reducing subterms. We write $\to^*$ for the reflexive and transitive closure of $\to$.

The rules are presented below divided into groups.

β-reduction

$$\frac{\psi \ \vdash \ M \ \to \ N}{\psi \ \vdash \ \lambda x.M \ \to \ \lambda x.N} \tag{R1}$$

$$\frac{\psi \vdash \ M \ \to \ M'}{\psi \ \vdash \ MN \ \to \ M'N} \tag{R2}$$

$$\frac{\psi \vdash \ N \ \to \ N'}{\psi \ \vdash \ MN \ \to \ MN'} \tag{R3}$$

$$\psi \ \vdash \ (\lambda x.M) \ N \ \to \ M[N/x] \tag{R4}$$

This is the standard infrastructure of the lambda calculus. We do not commit to any particular reduction strategy, allowing reductions both in functions and in their arguments. The exact definition of the substitution $M[N/x]$ from rule (R4) is given in Section 3.4.

Basic constants

$$\psi \vdash \texttt{empty} \to \{\ \} \tag{R5}$$

$$\psi \vdash \texttt{atoms} \to \{a\colon \text{for } a\} \tag{R6}$$

$$\begin{aligned}\psi \vdash \texttt{insert}\ M\ \{M_1\colon \phi_1 \text{ for } \sigma_1, \ldots, M_n\colon \phi_n \text{ for } \sigma_n\}\\ \to \{M, M_1\colon \phi_1 \text{ for } \sigma_1, \ldots, M_n\colon \phi_n \text{ for } \sigma_n\}\end{aligned} \tag{R7}$$

$$\begin{aligned}\psi \vdash \texttt{map}\ M\ \{M_1\colon \phi_1 \text{ for } \sigma_1, \ldots, M_n\colon \phi_n \text{ for } \sigma_n\}\\ \to \{MM_1\colon \phi_1 \text{ for } \sigma_1, \ldots, MM_n\colon \phi_n \text{ for } \sigma_n\}\end{aligned} \quad \text{if} \quad \mathrm{V_A}(M) \cap \bigcup_{i=1}^{n} \sigma_i = \emptyset \tag{R8}$$

$$\begin{aligned}\psi \vdash \texttt{sum}\ \{\ldots, \{M_1\colon \phi_1 \text{ for } \sigma_1, \ldots, M_n\colon \phi_n \text{ for } \sigma_n\}\colon \phi \text{ for } \sigma, \ldots\}\\ \to \{\ldots, M_1\colon \phi_1 \wedge \phi \text{ for } \sigma_1 \cup \sigma, \ldots,\ M_n\colon \phi_n \wedge \phi \text{ for } \sigma_n \cup \sigma, \ldots\}\end{aligned} \quad \text{if } \sigma \cap \bigcup_{i=1}^{n} \sigma_i = \emptyset \tag{R9}$$

The next group consists of rules that create and modify sets. The rule for `empty` in (R5) returns an empty set expression. The rule (R6) for `atoms` creates a set expression for the set of all atoms. What is worth noting is that this rule is the only place where a new bound atom variable is created. The rule for the constant `insert` (R7) extends the given set expression to include a new element M without any condition (that is, formally the condition is $\top$) and with no bound variables.

The rules (R8) and (R9) require a broader explanation.

The rule for `map` implements element-wise function application. Therefore, the term M representing the function is applied inside the set expression to each term M_i (for $i = 1, \ldots, n$) representing the elements of the set. This rule has a side condition because the function M can introduce atom variables into the set, which may have the same names as some bound variables in that set. Therefore, the condition says that all atom variables occurring in the term M must be different from any bound variable of the set expression[2] and terms may need to be α-converted before using the rule.

The rule for `sum` implements an operation of the set-theoretic union of a set of sets. This means reducing the level of nesting of set subexpressions and placing the elements of these set subexpressions directly in the main set expression. New conditions for these elements arise by conjunction of the conditions from the subexpressions and the condition from the main expression. In addition, lists of bound variables in the target term are concatenations of bound variables from subexpressions and bound variables from the main expression. So the rule (R6) introduces a bound variable, and the rule (R9) allows to increase the number of these variables in a single set expression. As with the `map` rule, the `sum` rule has a side condition that prevents name conflicts in bound variables. If such a conflict occurs then α-conversion must be applied. Finally, it is worth emphasizing that rule (R9) is presented as applied to a single subexpression, but the transformation is performed for all subexpressions simultaneously in one reduction step.

[2]Sigmas formally denote tuples of bound atom variables, but in the side conditions for rules (R8) and (R9), they are treated as sets to simplify notation.

Formula constructors

$$\psi \vdash \texttt{true} \to \top \qquad \psi \vdash \texttt{false} \to \bot \tag{R10}$$

$$\psi \vdash \texttt{not}\ \phi \ \to\ \neg\phi \tag{R11}$$

$$\psi \vdash \texttt{or}\ \phi_1\ \phi_2 \to \phi_1 \vee \phi_2 \qquad \psi \vdash \texttt{and}\ \phi_1\ \phi_2 \to \phi_1 \wedge \phi_2 \tag{R12}$$

$$\psi \vdash \texttt{isEmpty}\ \{M_1\colon \phi_1 \text{ for } \sigma_1, \ldots, M_n\colon \phi_n \text{ for } \sigma_n\} \to \bigwedge_{1\le i\le n} \forall\sigma_i.\neg\phi_i \tag{R13}$$

These rules naturally transform language expressions into logical formulas. Only the rule for checking the set emptiness (R13) is less obvious and may cause quantified first-order formulas to appear. At the same time, we introduce a notation that shortens the presentation of such a formula:

$$\forall\sigma \quad\equiv\quad \underbrace{\forall a_1 \forall a_2 \cdots \forall a_k}_{\sigma=(a_1,\ldots,a_k)}$$

Conditional expressions

$$\frac{\mathcal{A} \vDash \psi \Rightarrow \phi}{\psi \vdash \texttt{if}\ \phi\ M\ N \to M} \tag{R14}$$

$$\frac{\mathcal{A} \vDash \psi \Rightarrow \neg\phi}{\psi \vdash \texttt{if}\ \phi\ M\ N \to N} \tag{R15}$$

$$\frac{\mathcal{A} \nvDash \psi \Rightarrow \phi \quad \mathcal{A} \nvDash \psi \Rightarrow \neg\phi}{\psi \vdash (\texttt{if}\ \phi\ M_1\ M_2)\ N \ \to\ \texttt{if}\ \phi\ (M_1 N)\ (M_2 N)} \tag{R16}$$

$$\frac{\mathcal{A} \nvDash \psi \Rightarrow \phi \quad \mathcal{A} \nvDash \psi \Rightarrow \neg\phi}{\psi \vdash \texttt{if}\ \phi\ a\ b \to a\colon \phi \mid b\colon \neg\phi} \tag{R17}$$

$$\frac{\mathcal{A} \nvDash \psi \Rightarrow \phi \quad \mathcal{A} \nvDash \psi \Rightarrow \neg\phi}{\psi \vdash \texttt{if}\ \phi\ \phi_1\ \phi_2 \to (\phi_1 \wedge \phi) \vee (\phi_2 \wedge \neg\phi)} \tag{R18}$$

$$\frac{\mathcal{A} \nvDash \psi \Rightarrow \phi \quad \mathcal{A} \nvDash \psi \Rightarrow \neg\phi}{\begin{array}{l}\psi \vdash \texttt{if}\ \phi\ \{M_1\colon \phi_1 \text{ for } \sigma_1, \ldots, M_n\colon \phi_n \text{ for } \sigma_n\}\ \{N_1\colon \theta_1 \text{ for } \pi_1, \ldots, N_k\colon \theta_k \text{ for } \pi_k\} \\ \quad \to \{M_1\colon \phi_1 \wedge \phi \text{ for } \sigma_1, \ldots, M_n\colon \phi_n \wedge \phi \text{ for } \sigma_n, N_1\colon \theta_1 \wedge \neg\phi \text{ for } \pi_1, \ldots, N_k\colon \theta_k \wedge \neg\phi \text{ for } \pi_k\}\end{array}} \tag{R19}$$

$$\frac{\mathcal{A} \nvDash \psi \Rightarrow \phi \quad \mathcal{A} \nvDash \psi \Rightarrow \neg\phi}{\begin{array}{l}\psi \vdash \texttt{if}\ \phi\ (a_1\colon \phi_1| \cdots |a_n\colon \phi_n)\ (b_1\colon \theta_1| \cdots |b_k\colon \theta_k) \\ \quad \to a_1\colon \phi_1 \wedge \phi| \cdots |a_n\colon \phi_n \wedge \phi|b_1\colon \theta_1 \wedge \neg\phi| \ldots |b_k\colon \theta_k \wedge \neg\phi\end{array}} \tag{R20}$$

The conditional constant `if` is evaluated in a special way and it deserves a separate section of the semantics. A premise $\mathcal{A} \vDash \phi \Rightarrow \psi$ means that the formula $\phi \Rightarrow \psi$ holds in $\mathcal{A}$ under every valuation of its free atom variables. If some valuation falsifies the formula, we write $\mathcal{A} \nvDash \phi \Rightarrow \psi$.

Rules (R14) and (R15) apply where the value of the logical condition ϕ is determined by the ambient formula ψ. In such situations the condition ϕ behaves like a standard boolean value and the conditional expression is resolved as expected.

If the value of ϕ remains undetermined under the assumption of ψ, then both values to be chosen must be combined in the result of the conditional expression. The course of action depends on the type of those values, which means that the rules (R16) - (R20) are mutually exclusive. The general idea is to postpone the choice by pushing it down the structure of terms.

If the values of the conditional expression are functions as in rule (R16), then the argument is applied to both functions. If they are formulas or set expressions, rules (R18) and (R19) combine them in an expected way.

The most interesting case is a choice between atom variables: in rule (R17), a variant is created. It may be seen as an “ambiguous atom” equal to a or b depending on the value of ϕ. Formally, a separate rule (R20) for a choice between variants is required but it works as expected similarly to rule (R17).

Notice that rule (R17) is the only place where variants are created, and those variants are always built of atoms. As mentioned earlier, it is possible to expand the language with further basic values, such as texts and numbers. For each such basic type, a rule corresponding to (R17) would need to be added.

Set and variant reduction

$$\frac{\psi \wedge \phi \vdash\ M \to N}{\psi \vdash \{\ldots, M\colon \phi \text{ for } \sigma, \ldots\} \to \{\ldots, N\colon \phi \text{ for } \sigma, \ldots\}} \tag{R21}$$

$$\frac{\mathcal{A} \vDash \psi \Rightarrow \neg\phi}{\psi \vdash \{\ldots, M\colon \phi \text{ for } \sigma, \ldots\} \to \{\ldots, \ldots\}} \tag{R22}$$

$$\frac{\mathcal{A} \vDash \psi \Rightarrow \neg\phi}{\psi \vdash \ldots|a\colon \phi|\ldots \to \ldots|\ldots} \tag{R23}$$

Rule (R21) specifies how the reduction is done in the context of a set expression. This rule also shows how context formulas ψ are constructed.

Rules (R22) and (R23) allow us to remove components from the set and variant that do not affect the final result due to false conditions (in a given context) for them. Such contradictory conditions can appear after some reduction sequences despite the premises in conditional expressions.

Atom relations

$$\psi \vdash \mathtt{eq}_{\mathbb{A}}\ (a_1\colon \phi_1|\ldots|a_n\colon \phi_n)\ (b_1\colon \theta_1|\ldots|b_m\colon \theta_m) \to \bigvee_{\substack{1\le i\le n\\ 1\le j\le m}} (a_i = b_j \wedge \phi_i \wedge \theta_j) \tag{R24}$$

$$\psi \vdash \mathtt{leq}_{\mathbb{A}}\ (a_1\colon \phi_1|\ldots|a_n\colon \phi_n)\ (b_1\colon \theta_1|\ldots|b_m\colon \theta_m) \to \bigvee_{\substack{1\le i\le n\\ 1\le j\le m}} (a_i \le b_j \wedge \phi_i \wedge \theta_j) \tag{R25}$$

Rule (R24) specifies the behavior of the equality function used for equality atoms. This rule also applies to single atoms which are here understood as degenerated variants $a\colon \top$. If we work with ordered atoms, the function $\mathtt{leq}_{\mathbb{A}}$ is specified analogously.

Example of reduction

To illustrate how the reduction rules work, let us return to example (3.1):

$$\texttt{filter}\ (\lambda\texttt{x.or}\ (\texttt{leq}_{\mathbb{A}}\ \texttt{x a})\ (\texttt{leq}_{\mathbb{A}}\ \texttt{b x}))\ \texttt{atoms} \tag{3.2}$$

Example (3.1) referred to the language version without atom variables, so there are free variables y and z. In the above example we already have atom variables, so according to the convention their names were changed to a and b. As a result, the term can be evaluated without any valuation of free term variables.

The reduction process of the above program could look like this:

$\underline{\texttt{filter}}\ (\lambda\texttt{x.or}\ (\texttt{leq}_{\mathbb{A}}\ \texttt{x a})\ (\texttt{leq}_{\mathbb{A}}\ \texttt{b x}))\ \texttt{atoms}$	$\rightsquigarrow$ (`filter`)
$\texttt{sum}\ (\texttt{map}\ (\lambda\texttt{x.if}\ ((\lambda\underline{\texttt{x}}\texttt{.or}\ (\texttt{leq}_{\mathbb{A}}\ \underline{\texttt{x}}\ \texttt{a})\ (\texttt{leq}_{\mathbb{A}}\ \texttt{b}\ \underline{\texttt{x}}))\ \texttt{x})\ (\texttt{singleton x})\ \texttt{empty})\ \texttt{atoms})$	$\equiv$ (α-conversion)
$\texttt{sum}\ (\texttt{map}\ (\lambda\texttt{x.if}\ (\underline{(\lambda\texttt{y.or}\ (\texttt{leq}_{\mathbb{A}}\ \texttt{y a})\ (\texttt{leq}_{\mathbb{A}}\ \texttt{b y}))\ \texttt{x}})\ (\texttt{singleton x})\ \texttt{empty})\ \texttt{atoms})$	$\rightarrow$ (R4)
$\texttt{sum}\ (\texttt{map}\ (\lambda\texttt{x.if}\ (\texttt{or}\ (\texttt{leq}_{\mathbb{A}}\ \texttt{x a})\ (\texttt{leq}_{\mathbb{A}}\ \texttt{b x}))\ (\underline{\texttt{singleton}}\ \texttt{x})\ \texttt{empty})\ \texttt{atoms})$	$\rightsquigarrow$ (`singleton`)
$\texttt{sum}\ (\texttt{map}\ (\lambda\texttt{x.if}\ (\texttt{or}\ (\texttt{leq}_{\mathbb{A}}\ \texttt{x a})\ (\texttt{leq}_{\mathbb{A}}\ \texttt{b x}))\ (\texttt{insert x}\ \underline{\texttt{empty}})\ \texttt{empty})\ \texttt{atoms})$	$\rightarrow$ (R5)
$\texttt{sum}\ (\texttt{map}\ (\lambda\texttt{x.if}\ (\texttt{or}\ (\texttt{leq}_{\mathbb{A}}\ \texttt{x a})\ (\texttt{leq}_{\mathbb{A}}\ \texttt{b x}))\ (\underline{\texttt{insert}}\ \texttt{x}\ \{\})\ \texttt{empty})\ \texttt{atoms})$	$\rightarrow$ (R7)
$\texttt{sum}\ (\texttt{map}\ (\lambda\texttt{x.if}\ (\texttt{or}\ (\texttt{leq}_{\mathbb{A}}\ \texttt{x a})\ (\texttt{leq}_{\mathbb{A}}\ \texttt{b x}))\ \{\texttt{x}\}\ \underline{\texttt{empty}})\ \texttt{atoms})$	$\rightarrow$ (R5)
$\texttt{sum}\ (\texttt{map}\ (\lambda\texttt{x.if}\ (\texttt{or}\ (\texttt{leq}_{\mathbb{A}}\ \texttt{x a})\ (\texttt{leq}_{\mathbb{A}}\ \texttt{b x}))\ \{\texttt{x}\}\ \{\})\ \underline{\texttt{atoms}})$	$\rightarrow$ (R6)
$\texttt{sum}\ (\texttt{map}\ (\lambda\texttt{x.if}\ (\texttt{or}\ (\texttt{leq}_{\mathbb{A}}\ \texttt{x a})\ (\texttt{leq}_{\mathbb{A}}\ \texttt{b x}))\ \{\texttt{x}\}\ \{\})\ \{\underline{\texttt{a}}\colon \text{for}\ \underline{\texttt{a}}\})$	$\equiv$ (α-conversion)
$\texttt{sum}\ (\underline{\texttt{map}}\ (\lambda\texttt{x.if}\ (\texttt{or}\ (\texttt{leq}_{\mathbb{A}}\ \texttt{x a})\ (\texttt{leq}_{\mathbb{A}}\ \texttt{b x}))\ \{\texttt{x}\}\ \{\})\ \{\texttt{c}\colon \text{for}\ \texttt{c}\})$	$\rightarrow$ (R8)
$\texttt{sum}\ \{\underline{(\lambda\texttt{x.if}\ (\texttt{or}\ (\texttt{leq}_{\mathbb{A}}\ \texttt{x a})\ (\texttt{leq}_{\mathbb{A}}\ \texttt{b x}))\ \{\texttt{x}\}\ \{\})}\ \texttt{c}\colon \text{for}\ \texttt{c}\}$	$\rightarrow$ (R4)
$\texttt{sum}\ \{\texttt{if}\ (\texttt{or}\ (\underline{\texttt{leq}_{\mathbb{A}}}\ \texttt{c a})\ (\texttt{leq}_{\mathbb{A}}\ \texttt{b c}))\ \{\texttt{c}\}\ \{\}\colon \text{for}\ \texttt{c}\}$	$\rightarrow$ (R25)
$\texttt{sum}\ \{\texttt{if}\ (\texttt{or}\ (\texttt{c} \leq \texttt{a})\ (\underline{\texttt{leq}_{\mathbb{A}}}\ \texttt{b c}))\ \{\texttt{c}\}\ \{\}\colon \text{for}\ \texttt{c}\}$	$\rightarrow$ (R25)
$\texttt{sum}\ \{\texttt{if}\ (\underline{\texttt{or}}\ (\texttt{c} \leq \texttt{a})\ (\texttt{b} \leq \texttt{c}))\ \{\texttt{c}\}\ \{\}\colon \text{for}\ \texttt{c}\}$	$\rightarrow$ (R12)
$\texttt{sum}\ \{\underline{\texttt{if}}\ (\texttt{c} \leq \texttt{a}\ \vee\ \texttt{b} \leq \texttt{c})\ \{\texttt{c}\}\ \{\}\colon \text{for}\ \texttt{c}\}$	$\rightarrow$ (R19)
$\underline{\texttt{sum}}\ \{\{\texttt{c}\colon \texttt{c} \leq \texttt{a}\ \vee\ \texttt{b} \leq \texttt{c}\}\colon \text{for}\ \texttt{c}\}$	$\rightarrow$ (R9)
$\{\texttt{c}\colon \texttt{c} \leq \texttt{a}\ \vee\ \texttt{b} \leq \texttt{c}\ \text{for}\ \texttt{c}\}$	

In the example above, the leftmost reduction strategy was used. A different strategy could also be applied, but the final value obtained would be the same. At each step, a subterm that will be reduced has been underlined. For such a reduction of the subterm to take place, the rules that make it possible must first be applied: (R1, R2, R3, R21).

Two reductions result from the definition of the functions `filter` and `singleton`. Formally, the core language does not provide such a functionality as defining a function with a given name.[3] For the sake of brevity and readability, such a simplification has been used and denoted by $\rightsquigarrow$.

Twice in the example above, an α-conversion was needed. The first α-conversion was required before the β-reduction (R4), the second before applying the reduction for the constant `map`.

All of the above reductions took place in the trivial context $\top$. This is because in this example only terms of set expressions that had a trivial condition were reduced. But not every example has to have this property.

The obtained result of the above reduction process is consistent with the expected value for example (3.1).

[3] This functionality will be introduced in Section 8.2 along with the "let binding" construct.

3.8. Properties of semantics

The operational semantics presented in the previous section has a few expected properties proved in the following chapters:

- *subject reduction* holds, i.e., the reduction relation preserves types (Chapter 4),
- *equivalence of semantics*, i.e., the reduction relation preserves the meaning (denotation) of terms (Chapter 5),
- the *Church-Rosser property* holds up to first-order formula equivalence, i.e., if $M \to N$ and $M \to N'$ then there exist terms Q and Q' such that $N \to^* Q$ and $N' \to^* Q'$, where Q and Q' are equal up to replacing some first-order formulas with equivalent ones (Chapter 6),
- *strong normalization* holds, i.e., each well-typed term always reduces to an irreducible value (Chapter 7).

CHAPTER 4

Subject reduction

In the next few chapters, we will describe and prove some basic properties of the presented semantics.

First we will show that the reduction rules behave correctly with respect to types; more precisely, that term reduction does not change its type. This property is known as **subject reduction** [Curry and Feys, 1958]:

$$\Gamma \vdash M \colon \alpha \quad \wedge \quad \psi \vdash M \to N \qquad \Rightarrow \qquad \Gamma \vdash N \colon \alpha$$

4.1. Substitutions in typed terms

Before we come to the theorem we will prove a useful lemma about substitutions in typed terms.

Lemma 4.1. *If* $\Gamma \cup \{x \colon \alpha\} \vdash M \colon \beta$ *and* $\Gamma \vdash N \colon \alpha$ *for* $x \notin FV_T(N)$ *then* $\Gamma \vdash M[N/x] \colon \beta$.

Proof. By induction on the structure of the term M.

It is worth noting that if $x \notin \mathrm{FV}_\mathrm{T}(M)$ then $M[N/x] = M$ and in this case the result is obvious. This observation can be used for cases in which M is a constant, an atom variable, a formula or a variant.

For the other cases we have:

- **M is a term variable**

  If $M = x$ then $\alpha = \beta$ and $M[N/x] = N$ so we have $\Gamma \vdash M[N/x] \colon \beta$ by the assumption of the lemma.
  If $M = y$ where $x \neq y$, then $x \notin \mathrm{FV}_\mathrm{T}(M)$ and we can use the observation from the beginning of the proof.

- **M is an abstraction**

  We can assume by α-conversion that $M = \lambda y.M'$ where $x \neq y$, $y \notin \mathrm{FV}_\mathrm{T}(N)$ and context Γ does not contain a type assumption for the variable y.

  It must be that $\beta = \beta_1 \to \beta_2$ for some types β_1 and β_2, and the last rule used in the derivation for M looks like this:

$$\frac{\Gamma \cup \{x \colon \alpha, y \colon \beta_1\} \vdash M' \colon \beta_2}{\Gamma \cup \{x \colon \alpha\} \vdash \lambda y.M' \colon \beta_1 \to \beta_2} \tag{4.1}$$

We can add the variable y to the context Γ and obtain $\Gamma \cup \{y\colon \beta_1\} \vdash N : \alpha$. This combined with the premise of (4.1) and the induction hypothesis for M' gives us

$$\Gamma \cup \{y\colon \beta_1\} \vdash M'[N/x]\colon \beta_2$$

By applying the rule for abstraction again we get

$$\frac{\Gamma \cup \{y\colon \beta_1\} \vdash M'[N/x]\colon \beta_2}{\Gamma \vdash \lambda y.M'[N/x]\colon \beta_1 \to \beta_2} \tag{4.2}$$

and the above conclusion is equivalent to $\Gamma \vdash M[N/x]\colon \beta$.

➤ M is an application

If $M = M_1 M_2$ then the final step in the deduction for type of M must be

$$\frac{\Gamma \cup \{x\colon \alpha\} \vdash M_1\colon \gamma \to \beta \qquad \Gamma \cup \{x\colon \alpha\} \vdash M_2\colon \gamma}{\Gamma \cup \{x\colon \alpha\} \vdash M_1 M_2\colon \beta} \tag{4.3}$$

for some type γ. By applying the induction hypothesis to the premises of (4.3) and using the rule for the type of application, we obtain

$$\frac{\Gamma \vdash M_1[N/x]\colon \gamma \to \beta \qquad \Gamma \vdash M_2[N/x]\colon \gamma}{\Gamma \vdash M_1[N/x]M_2[N/x]\colon \beta} \tag{4.4}$$

Now by definition of substitution the conclusion of (4.4) is equal to $\Gamma \vdash (M_1 M_2)[N/x]\colon \beta$, which in turn is equivalent to $\Gamma \vdash M[N/x]\colon \beta$.

➤ M is a set

Suppose that $M = \{M_1\colon \phi_1 \text{ for } \sigma_1, \ldots, M_n\colon \phi_n \text{ for } \sigma_n\}$ and $\beta = \mathbb{S}\tau$. The final rule in the derivation for type of M must look like this:

$$\frac{\Gamma \cup \{x\colon \alpha\} \vdash M_1\colon \tau \quad \cdots \quad \Gamma \cup \{x\colon \alpha\} \vdash M_n\colon \tau}{\Gamma \cup \{x\colon \alpha\} \vdash \{M_1\colon \phi_1 \text{ for } \sigma_1, \ldots, M_n\colon \phi_n \text{ for } \sigma_n\}\colon \mathbb{S}\tau} \tag{4.5}$$

By the premises of (4.5) and the induction hypothesis we get

$$\Gamma \vdash M_i[N/x]\colon \tau \qquad \text{for } i = 1, \ldots, n$$

and applying the set rule to the above gives:

$$\frac{\Gamma \vdash M_1[N/x]\colon \tau \quad \cdots \quad \Gamma \vdash M_n[N/x]\colon \tau}{\Gamma \vdash \{M_1[N/x]\colon \phi_1 \text{ for } \sigma_1, \ldots, M_n[N/x]\colon \phi_n \text{ for } \sigma_n\}\colon \mathbb{S}\tau} \tag{4.6}$$

where above conclusion is equivalent to

$$\Gamma \vdash \{M_1\colon \phi_1 \text{ for } \sigma_1, \ldots, M_n\colon \phi_n \text{ for } \sigma_n\}[N/x]\colon \mathbb{S}\tau$$

as required.

□

4.2. The Subject Reduction Theorem

We can now proceed to the proof of the theorem.

Theorem 4.2 (Subject reduction)**.** *If* $\Gamma \vdash M : \alpha$ *and* $\psi \vdash M \to N$ *then* $\Gamma \vdash N : \alpha$

Proof. By induction on the derivation of $\psi \vdash M \to N$ for all contexts ψ. We perform case analysis for the final step of this derivation (see Section 3.7 for the list of rules).

- Rule R1

  We have $M = \lambda x.M'$, $N = \lambda x.N'$, $\alpha = \alpha_1 \to \alpha_2$ and the reduction rule of the form:

$$\frac{\psi \vdash M' \to N'}{\psi \vdash \lambda x.M' \to \lambda x.N'}$$

  The derivation for the type of M is as follows:

$$\frac{\dfrac{\vdots}{\Gamma \cup \{x : \alpha_1\} \vdash M' : \alpha_2}}{\Gamma \vdash \lambda x.M' : \alpha_1 \to \alpha_2}$$

  Thanks to the induction hypothesis the above premise can be replaced by

$$\Gamma \cup \{x : \alpha_1\} \vdash N' : \alpha_2$$

  so we get the derivation for $\Gamma \vdash \lambda x.N' : \alpha_1 \to \alpha_2$ which is the same as $\Gamma \vdash N : \alpha$.

- Rule R2

  Suppose that $M = M_1M_2$ and the rule $\psi \vdash M \to N$ is the reduction on the left of the application:

$$\frac{\psi \vdash M_1 \to M_1'}{\psi \vdash M_1M_2 \to M_1'M_2}$$

  Then the derivation for the type of M has the following form:

$$\frac{\dfrac{\vdots}{\Gamma \vdash M_1 : \beta \to \alpha} \qquad \dfrac{\vdots}{\Gamma \vdash M_2 : \beta}}{\Gamma \vdash M_1M_2 : \alpha}$$

  The inductive hypothesis applies to $\Gamma \vdash M_1 : \beta \to \alpha$ and yields $\Gamma \vdash M_1' : \beta \to \alpha$. This gives a type derivation for $N = M_1'M_2$:

$$\frac{\dfrac{\vdots}{\Gamma \vdash M_1' : \beta \to \alpha} \qquad \dfrac{\vdots}{\Gamma \vdash M_2 : \beta}}{\Gamma \vdash M_1'M_2 : \alpha}$$

- Rule R3

  Analogous to the case of Rule (R2).

➤ Rule R4

We assume that $M = (\lambda x.M_1)M_2$ and want to show that $\Gamma \vdash M_1[M_2/x]\colon \alpha$. The derivation this time has form:

$$\dfrac{\dfrac{\dfrac{\vdots}{\Gamma \cup \{x\colon \beta\} \vdash M_1\colon \alpha}}{\Gamma \vdash \lambda x.M_1\colon \beta \to \alpha} \qquad \dfrac{\vdots}{\Gamma \vdash M_2\colon \beta}}{\Gamma \vdash (\lambda x.M_1)M_2\colon \alpha}$$

So we have two subderivations for $\Gamma \cup \{x\colon \beta\} \vdash M_1\colon \alpha$ and $\Gamma \vdash M_2\colon \beta$. Applying Lemma 4.1 to conclusions of these derivations we get $\Gamma \vdash M_1[M_2/x]\colon \alpha$ as required. Note that the assumption of Lemma 4.1 is satisfied: $x \notin \mathrm{FV_T}(M_2)$ because we can assume that x is different from all free term variables due to Barendregt's Variable Convention.[1]

➤ Rule R5

For $M = \texttt{empty}$ where $\Gamma \vdash \texttt{empty}\colon \mathbb{S}\tau$ we want show that $\Gamma \vdash \{\}\colon \mathbb{S}\tau$. The type derivation for sets is as follows:

$$\dfrac{\dfrac{\vdots}{\Gamma \vdash M_1\colon \tau} \quad \cdots \quad \dfrac{\vdots}{\Gamma \vdash M_n\colon \tau}}{\Gamma \vdash \{M_1\colon \phi_1 \text{ for } \sigma_1, \dots, M_n\colon \phi_n \text{ for } \sigma_n\}\colon \mathbb{S}\tau}$$

Putting $n = 0$, we have $\Gamma \vdash \{\}\colon \mathbb{S}\tau$ for every τ.

➤ Rule R6

If $M = \texttt{atoms}$, then $\Gamma \vdash M\colon \mathbb{SA}$. The result of Rule (R6) has the same type because:

$$\dfrac{\Gamma \vdash a\colon \mathbb{A}}{\Gamma \vdash \{a\colon \text{for } a\}\colon \mathbb{SA}}$$

➤ Rule R7

Assume $M = \texttt{insert}\ M_0\ \{M_1\colon \phi_1 \text{ for } \sigma_1, \dots, M_n\colon \phi_n \text{ for } \sigma_n\}$ and a type derivation for M:

$$\dfrac{\dfrac{\Gamma \vdash \texttt{insert}\colon \tau \to \mathbb{S}\tau \to \mathbb{S}\tau \quad \dfrac{\vdots}{\Gamma \vdash M_0\colon \tau}}{\Gamma \vdash \texttt{insert}\ M_0\colon \mathbb{S}\tau \to \mathbb{S}\tau} \qquad \dfrac{\dfrac{\vdots}{\Gamma \vdash M_1\colon \tau} \quad \cdots \quad \dfrac{\vdots}{\Gamma \vdash M_n\colon \tau}}{\Gamma \vdash \{M_1\colon \phi_1 \text{ for } \sigma_1, \dots, M_n\colon \phi_n \text{ for } \sigma_n\}\colon \mathbb{S}\tau}}{\Gamma \vdash \texttt{insert}\ M_0\ \{M_1\colon \phi_1 \text{ for } \sigma_1, \dots, M_n\colon \phi_n \text{ for } \sigma_n\}\colon \mathbb{S}\tau}$$

Using derivations for $M_0, M_1, \dots, M_n$ we can create a derivation for:

$$\dfrac{\dfrac{\vdots}{\Gamma \vdash M_0\colon \tau} \quad \dfrac{\vdots}{\Gamma \vdash M_1\colon \tau} \quad \cdots \quad \dfrac{\vdots}{\Gamma \vdash M_n\colon \tau}}{\Gamma \vdash \{M_0, M_1\colon \phi_1 \text{ for } \sigma_1, \dots, M_n\colon \phi_n \text{ for } \sigma_n\}\colon \mathbb{S}\tau}$$

as required.

[1]See Variables subsection in Section 3.4.

➤ Rule R8

Suppose that $M = \texttt{map}\ M_0\ \{M_1\colon \phi_1 \text{ for } \sigma_1, \ldots, M_n\colon \phi_n \text{ for } \sigma_n\}$. Consider the type derivation of $\Gamma \vdash M\colon \alpha$, where $\alpha = \mathbb{S}\tau_2$:

$$\dfrac{\dfrac{\Gamma \vdash \texttt{map}\colon (\tau_1 \to \tau_2) \to \mathbb{S}\tau_1 \to \mathbb{S}\tau_2 \quad \dfrac{\vdots}{\Gamma \vdash M_0\colon \tau_1 \to \tau_2}}{\Gamma \vdash \texttt{map}\ M_0\colon \mathbb{S}\tau_1 \to \mathbb{S}\tau_2} \quad \dfrac{\dfrac{\vdots}{\Gamma \vdash M_1\colon \tau_1} \quad \cdots \quad \dfrac{\vdots}{\Gamma \vdash M_n\colon \tau_1}}{\Gamma \vdash \{M_1\colon \phi_1 \text{ for } \sigma_1, \ldots, M_n\colon \phi_n \text{ for } \sigma_n\}\colon \mathbb{S}\tau_1}}{\Gamma \vdash \texttt{map}\ M_0\ \{M_1\colon \phi_1 \text{ for } \sigma_1, \ldots, M_n\colon \phi_n \text{ for } \sigma_n\}\colon \mathbb{S}\tau_2}$$

So we have derivations for $M_0, M_1, \ldots, M_n$ and we can build from them a derivation for:

$$\dfrac{\dfrac{\dfrac{\vdots}{\Gamma \vdash M_0\colon \tau_1 \to \tau_2} \quad \dfrac{\vdots}{\Gamma \vdash M_1\colon \tau_1}}{\Gamma \vdash M_0 M_1\colon \tau_2} \quad \cdots \quad \dfrac{\dfrac{\vdots}{\Gamma \vdash M_0\colon \tau_1 \to \tau_2} \quad \dfrac{\vdots}{\Gamma \vdash M_n\colon \tau_1}}{\Gamma \vdash M_0 M_n\colon \tau_2}}{\Gamma \vdash \{M_0 M_1\colon \phi_1 \text{ for } \sigma_1, \ldots, M_0 M_n\colon \phi_n \text{ for } \sigma_n\}\colon \mathbb{S}\tau_2}$$

as required.

➤ Rule R9

In this case the term M is equal to

$$\texttt{sum}\ \{\ldots, \{M_1\colon \phi_1 \text{ for } \sigma_1, \ldots, M_n\colon \phi_n \text{ for } \sigma_n\}\colon \phi \text{ for } \sigma, \ldots\}$$

Take the type derivation for M:

$$\dfrac{\Gamma \vdash \texttt{sum}\colon \mathbb{S}\mathbb{S}\tau \to \mathbb{S}\tau \quad \dfrac{\cdots \quad \dfrac{\dfrac{\vdots}{\Gamma \vdash M_1\colon \tau} \quad \cdots \quad \dfrac{\vdots}{\Gamma \vdash M_n\colon \tau}}{\Gamma \vdash \{M_1\colon \phi_1 \text{ for } \sigma_1, \ldots, M_n\colon \phi_n \text{ for } \sigma_n\}\colon \mathbb{S}\tau} \quad \cdots}{\Gamma \vdash \{\ldots, \{M_1\colon \phi_1 \text{ for } \sigma_1, \ldots, M_n\colon \phi_n \text{ for } \sigma_n\}\colon \phi \text{ for } \sigma, \ldots\}\colon \mathbb{S}\mathbb{S}\tau}}{\Gamma \vdash \texttt{sum}\ \{\ldots, \{M_1\colon \phi_1 \text{ for } \sigma_1, \ldots, M_n\colon \phi_n \text{ for } \sigma_n\}\colon \phi \text{ for } \sigma, \ldots\}\colon \mathbb{S}\tau}$$

and notice that it contains of derivations for $M_1, \ldots, M_n$. We can rewrite them to get a derivation for the reduct term:

$$\dfrac{\cdots \quad \dfrac{\vdots}{\Gamma \vdash M_1\colon \tau} \quad \cdots \quad \dfrac{\vdots}{\Gamma \vdash M_n\colon \tau} \quad \cdots}{\Gamma \vdash \{\ldots, M_1\colon \phi_1 \wedge \phi \text{ for } \sigma_1 \cup \sigma, \ldots, M_n\colon \phi_1 \wedge \phi \text{ for } \sigma_n \cup \sigma, \ldots\}\colon \mathbb{S}\tau}$$

Now consider the rules regarding logical formulas.

➤ Rule R10

For M equal to `true` or `false` we have $\Gamma \vdash M\colon \mathbb{B}$. But N that is equal to $\top$ or $\bot$ has also type $\mathbb{B}$, because $\top$ and $\bot$ are formulas which have always type $\mathbb{B}$.

➤ Rule R11

The case of negation is similar. The conclusion of the relevant rule is a formula so it is enough to derive:

$$\dfrac{\Gamma \vdash \texttt{not}\colon \mathbb{B} \to \mathbb{B} \qquad \Gamma \vdash \phi\colon \mathbb{B}}{\Gamma \vdash \texttt{not}\ \phi\colon \mathbb{B}}$$

➤ Rule R12

The same works for alternative and conjunction. For $M = \texttt{or}\ \phi_1\ \phi_2$ we have:

$$\dfrac{\dfrac{\Gamma \vdash \texttt{or}\colon \mathbb{B} \to \mathbb{B} \to \mathbb{B} \qquad \Gamma \vdash \phi_1\colon \mathbb{B}}{\Gamma \vdash \texttt{or}\ \phi_1\colon \mathbb{B} \to \mathbb{B}} \qquad \Gamma \vdash \phi_2\colon \mathbb{B}}{\Gamma \vdash \texttt{or}\ \phi_1\ \phi_2\colon \mathbb{B}}$$

We can do the same reasoning for `and`.

➤ Rule R13

The last reduction rule where the reduct is a formula is the rule for the constant `isEmpty`. Here again we want to show that the redex term has a type $\mathbb{B}$. This is due to the following deduction:

$$\dfrac{\Gamma \vdash \texttt{isEmpty}\colon \mathbb{S}\tau \to \mathbb{B} \qquad \dfrac{\dfrac{\vdots}{\Gamma \vdash M_1\colon \tau} \quad \cdots \quad \dfrac{\vdots}{\Gamma \vdash M_n\colon \tau}}{\{M_1\colon \phi_1 \text{ for } \sigma_1, \ldots, M_n\colon \phi_n \text{ for } \sigma_n\}\colon \mathbb{S}\tau}}{\Gamma \vdash \texttt{isEmpty}\ \{M_1\colon \phi_1 \text{ for } \sigma_1, \ldots, M_n\colon \phi_n \text{ for } \sigma_n\}\colon \mathbb{B}}$$

Regardless of the type τ of elements in the set, the outcome of applying `isEmpty` to this set has type $\mathbb{B}$.

Next we consider reduction rules for conditional expressions.

➤ Rule R14

Suppose that $M = \texttt{if}\ \phi\ M_1\ M_2$ and $\mathcal{A} \vDash \psi \Rightarrow \phi$. It follows that $N = M_1$. But the type of M results from a derivation:

$$\dfrac{\dfrac{\dfrac{\Gamma \vdash \texttt{if}\colon \mathbb{B} \to \alpha \to \alpha \to \alpha \qquad \Gamma \vdash \phi\colon \mathbb{B}}{\Gamma \vdash \texttt{if}\ \phi\colon \alpha \to \alpha \to \alpha} \qquad \dfrac{\vdots}{\Gamma \vdash M_1\colon \alpha}}{\Gamma \vdash \texttt{if}\ \phi\ M_1\colon \alpha \to \alpha} \qquad \dfrac{\vdots}{\Gamma \vdash M_2\colon \alpha}}{\Gamma \vdash \texttt{if}\ \phi\ M_1\ M_2\colon \alpha}$$

This means that we have derivations for $\Gamma \vdash M_1\colon \alpha$ and $\Gamma \vdash M_2\colon \alpha$. So we proved that $\Gamma \vdash N\colon \alpha$.

➤ Rule R15

As for Rule (R14), with the difference that $\mathcal{A} \vDash \psi \Rightarrow \neg\phi$ and $N = M_2$.

For the other conditional expression rules we assume that $\mathcal{A} \nvDash \psi \Rightarrow \phi$ and $\mathcal{A} \nvDash \psi \Rightarrow \neg\phi$.

➤ Rule R16

Take $M = \texttt{if}\ \phi\ M_1\ M_2\ M_0$, $N = \texttt{if}\ \phi\ (M_1 M_0)\ (M_2 M_0)$ and assume that $\Gamma \vdash M\colon \alpha$. The derivation of the latter is as follows:

$$\dfrac{\dfrac{\dfrac{\dfrac{\Gamma \vdash \texttt{if}\colon \mathbb{B} \to \gamma \to \gamma \to \gamma \qquad \Gamma \vdash \phi\colon \mathbb{B}}{\Gamma \vdash \texttt{if}\ \phi\colon \gamma \to \gamma \to \gamma} \qquad \dfrac{\vdots}{\Gamma \vdash M_1\colon \gamma}}{\Gamma \vdash \texttt{if}\ \phi\ M_1\colon \gamma \to \gamma} \qquad \dfrac{\vdots}{\Gamma \vdash M_2\colon \gamma}}{\Gamma \vdash \texttt{if}\ \phi\ M_1\ M_2\colon \gamma} \qquad \dfrac{\vdots}{\Gamma \vdash M_0\colon \beta}}{\Gamma \vdash \texttt{if}\ \phi\ M_1\ M_2\ M_0\colon \alpha}$$

where $\gamma = \beta \to \alpha$ for some type β. So we get derivations showing that M_1 and M_2 have type $\beta \to \alpha$ and M_0 has type β. We use these derivations to show that $\Gamma \vdash N\colon \alpha$:

$$\dfrac{\dfrac{\dfrac{\Gamma \vdash \texttt{if}\colon \mathbb{B} \to \alpha \to \alpha \to \alpha \quad \Gamma \vdash \phi\colon \mathbb{B}}{\Gamma \vdash \texttt{if}\ \phi\colon \alpha \to \alpha \to \alpha} \quad \dfrac{\dfrac{\vdots}{\Gamma \vdash M_1\colon \gamma} \quad \dfrac{\vdots}{\Gamma \vdash M_0\colon \beta}}{\Gamma \vdash M_1 M_0\colon \alpha}}{\Gamma \vdash \texttt{if}\ \phi\ (M_1 M_0)\colon \alpha \to \alpha} \quad \dfrac{\dfrac{\vdots}{\Gamma \vdash M_2\colon \gamma} \quad \dfrac{\vdots}{\Gamma \vdash M_0\colon \beta}}{\Gamma \vdash M_2 M_0\colon \alpha}}{\Gamma \vdash \texttt{if}\ \phi\ (M_1 M_0)\ (M_2 M_0)\colon \alpha}$$

➤ Rule R17

In this case α must be equal to $\mathbb{A}$. This is due to the following derivations:

$$\dfrac{\dfrac{\dfrac{\Gamma \vdash \texttt{if}\colon \mathbb{B} \to \mathbb{A} \to \mathbb{A} \to \mathbb{A} \qquad \Gamma \vdash \phi\colon \mathbb{B}}{\Gamma \vdash \texttt{if}\ \phi\colon \mathbb{A} \to \mathbb{A} \to \mathbb{A}} \qquad \Gamma \vdash a\colon \mathbb{A}}{\Gamma \vdash \texttt{if}\ \phi\ a\colon \mathbb{A} \to \mathbb{A}} \qquad \Gamma \vdash b\colon \mathbb{A}}{\Gamma \vdash \texttt{if}\ \phi\ a\ b\colon \mathbb{A}}$$

$$\dfrac{\Gamma \vdash a\colon \mathbb{A} \qquad \Gamma \vdash b\colon \mathbb{A}}{\Gamma \vdash (a\colon \phi \mid b\colon \neg\phi)\colon \mathbb{A}}$$

➤ Rule R18

Note that $\alpha = \mathbb{B}$, because $N = (\phi_1 \wedge \phi) \vee (\phi_2 \wedge \neg\phi)$ is a formula. So we want to show that M has also type $\mathbb{B}$:

$$\dfrac{\dfrac{\dfrac{\Gamma \vdash \texttt{if}\colon \mathbb{B} \to \mathbb{B} \to \mathbb{B} \to \mathbb{B} \qquad \Gamma \vdash \phi\colon \mathbb{B}}{\Gamma \vdash \texttt{if}\ \phi\colon \mathbb{B} \to \mathbb{B} \to \mathbb{B}} \qquad \Gamma \vdash \phi_1\colon \mathbb{B}}{\Gamma \vdash \texttt{if}\ \phi\ \phi_1\colon \mathbb{B} \to \mathbb{B}} \qquad \Gamma \vdash \phi_2\colon \mathbb{B}}{\Gamma \vdash \texttt{if}\ \phi\ \phi_1\ \phi_2\colon \mathbb{B}}$$

➤ Rule R19

Here the conditional expression has the type $\mathbb{S}\tau$ for some type τ. So we have $\alpha = \mathbb{S}\tau$.

Consider the derivation for the expression M. We have $M = \texttt{if}\ \phi\ S_1\ S_2$, where:

$$S_1 = \{M_1\colon \phi_1 \text{ for } \sigma_1, \ldots, M_n\colon \phi_n \text{ for } \sigma_n\}$$
$$S_2 = \{N_1\colon \theta_1 \text{ for } \pi_1, \ldots, N_k\colon \theta_k \text{ for } \pi_k\}$$

with corresponding derivations:

$$\dfrac{\dfrac{\vdots}{\Gamma \vdash M_1 \colon \tau} \quad \cdots \quad \dfrac{\vdots}{\Gamma \vdash M_n \colon \tau}}{\Gamma \vdash \{M_1 \colon \phi_1 \text{ for } \sigma_1, \ldots, M_n \colon \phi_n \text{ for } \sigma_n\} \colon \mathbb{S}\tau}$$

$$\dfrac{\dfrac{\vdots}{\Gamma \vdash N_1 \colon \tau} \quad \cdots \quad \dfrac{\vdots}{\Gamma \vdash N_k \colon \tau}}{\Gamma \vdash \{N_1 \colon \theta_1 \text{ for } \pi_1, \ldots, N_k \colon \theta_k \text{ for } \pi_k\} \colon \mathbb{S}\tau}$$

From this it follows that $M_1, \ldots, M_n$ and $N_1, \ldots, N_k$ have the type τ.

The derivation of $\mathbb{S}\tau$ as a type for M looks like this:

$$\dfrac{\dfrac{\dfrac{\Gamma \vdash \texttt{if} \colon \mathbb{B} \to \mathbb{S}\tau \to \mathbb{S}\tau \to \mathbb{S}\tau \qquad \Gamma \vdash \phi \colon \mathbb{B}}{\Gamma \vdash \texttt{if}\ \phi \colon \mathbb{S}\tau \to \mathbb{S}\tau \to \mathbb{S}\tau} \qquad \dfrac{\vdots}{\Gamma \vdash S_1 \colon \mathbb{S}\tau}}{\Gamma \vdash \texttt{if}\ \phi\ S_1 \colon \mathbb{S}\tau \to \mathbb{S}\tau} \qquad \dfrac{\vdots}{\Gamma \vdash S_2 \colon \mathbb{S}\tau}}{\Gamma \vdash \texttt{if}\ \phi\ S_1\ S_2 \colon \mathbb{S}\tau}$$

where the details of the subderivations for S_1 and S_2 were presented above. We now have to prove that N also has the type $\mathbb{S}\tau$. It follows directly from derivations for $M_1, \ldots, M_n$ and $N_1, \ldots, N_k$ as premises:

$$\dfrac{\dfrac{\vdots}{\Gamma \vdash M_1 \colon \tau} \quad \cdots \quad \dfrac{\vdots}{\Gamma \vdash M_n \colon \tau} \qquad \dfrac{\vdots}{\Gamma \vdash N_1 \colon \tau} \quad \cdots \quad \dfrac{\vdots}{\Gamma \vdash N_k \colon \tau}}{\Gamma \vdash \{M_1 \colon \phi_1 \land \phi \text{ for } \sigma_1, \ldots, M_n \colon \phi_n \land \phi \text{ for } \sigma_n, N_1 \colon \theta_1 \land \neg\phi \text{ for } \pi_1, \ldots, N_k \colon \theta_k \land \neg\phi \text{ for } \pi_k\} \colon \mathbb{S}\tau}$$

➤ Rule R20

Take two wariants:

$$V_1 = a_1 \colon \phi_1 | \cdots | a_n \colon \phi_n$$
$$V_2 = b_1 \colon \theta_1 | \cdots | b_k \colon \theta_k$$

As we know from the typing rules, all variants have type $\mathbb{A}$. With this in mind, we can show that $M = \texttt{if}\ \phi\ V_1\ V_2$ has also type $\mathbb{A}$:

$$\dfrac{\dfrac{\dfrac{\Gamma \vdash \texttt{if} \colon \mathbb{B} \to \mathbb{A} \to \mathbb{A} \to \mathbb{A} \qquad \Gamma \vdash \phi \colon \mathbb{B}}{\Gamma \vdash \texttt{if}\ \phi \colon \mathbb{A} \to \mathbb{A} \to \mathbb{A}} \qquad \Gamma \vdash V_1 \colon \mathbb{A}}{\Gamma \vdash \texttt{if}\ \phi\ V_1 \colon \mathbb{A} \to \mathbb{A}} \qquad \Gamma \vdash V_2 \colon \mathbb{A}}{\Gamma \vdash \texttt{if}\ \phi\ V_1\ V_2\ \colon \mathbb{A}}$$

But the target term N, which is a variant, also has type $\mathbb{A}$:

$$\Gamma \vdash (a_1 \colon \phi_1 \land \phi | \cdots | a_n \colon \phi_n \land \phi | b_1 \colon \theta_1 \land \neg\phi | \cdots | b_k \colon \theta_k \land \neg\phi) \colon \mathbb{A}$$

We now consider the rules for reducing subexpressions of sets and variants.

➤ Rule R21

Suppose that $M = \{\ldots, M' \colon \phi \text{ for } \sigma, \ldots\}$. According to Rule (R21), M reduces to $N = \{\ldots, N' \colon \phi \text{ for } \sigma, \ldots\}$ in the context of the formula ψ, provided that M' reduces to N' in the context of the formula $\psi \wedge \phi$.

From the type derivation for M:

$$\dfrac{\cdots \qquad \dfrac{\vdots}{\Gamma \vdash M' \colon \tau} \qquad \cdots}{\Gamma \vdash \{\ldots, M' \colon \phi \text{ for } \sigma, \ldots\} \colon \mathbb{S}\tau}$$

we can conclude that $\alpha = \mathbb{S}\tau$ and $\Gamma \vdash M' \colon \tau$. By applying the induction hypothesis to M' we can say that $\Gamma \vdash N' \colon \tau$. The result follows:

$$\dfrac{\cdots \qquad \dfrac{\vdots}{\Gamma \vdash N' \colon \tau} \qquad \cdots}{\Gamma \vdash \{\ldots, N' \colon \phi \text{ for } \sigma, \ldots\} \colon \mathbb{S}\tau}$$

➤ Rule R22

For $M = \{\ldots, M' \colon \phi \text{ for } \sigma, \ldots\}$ this rule removes the subexpression with the subterm M' if the condition ϕ is unsatisfiable.

A derivation for N is obtained from the following derivation for M:

$$\dfrac{\cdots \qquad \dfrac{\vdots}{\Gamma \vdash M' \colon \tau} \qquad \cdots}{\Gamma \vdash \{\ldots, M' \colon \phi \text{ for } \sigma, \ldots\} \colon \mathbb{S}\tau}$$

by removing the premise and subderivation for M'. However, the remaining premises determine the same type of $\mathbb{S}\tau$ for N. If there are no other premises, then N is the empty set and has the type $\mathbb{S}\tau$ for each type τ.

➤ Rule R23

The reduct term N for this rule is derived from $M = \ldots|a \colon \phi|\ldots$ by removing the subexpression with atom variable a and false condition ϕ. Both term M and term N are variants, which according to the typing rules always have the same type $\mathbb{A}$.

Finally, we consider the two rules that reduce constants for atom relations.

➤ Rule R24

We have

$$\begin{aligned} M &= \mathtt{eq}_{\mathbb{A}}\ (a_1 \colon \phi_1 | \cdots | a_n \colon \phi_n)\ (b_1 \colon \theta_1 | \cdots | b_m \colon \theta_m) \\ N &= \bigvee_{\substack{1 \le i \le n \\ 1 \le j \le m}} (a_i = b_j \wedge \phi_i \wedge \theta_j) \end{aligned}$$

then M has only type $\mathbb{B}$:

$$\dfrac{\dfrac{\Gamma \vdash \mathtt{eq}_{\mathbb{A}} : \mathbb{A} \to \mathbb{A} \to \mathbb{B} \qquad \dfrac{\Gamma \vdash a_1 : \mathbb{A} \qquad \cdots \qquad \Gamma \vdash a_n : \mathbb{A}}{\Gamma \vdash (a_1 : \phi_1 | \cdots | a_n : \phi_n) : \mathbb{A}}}{\Gamma \vdash \mathtt{eq}_{\mathbb{A}}\ (a_1 : \phi_1 | \cdots | a_n : \phi_n) : \mathbb{A} \to \mathbb{B}} \qquad \dfrac{\Gamma \vdash b_1 : \mathbb{A} \qquad \cdots \qquad \Gamma \vdash b_m : \mathbb{A}}{\Gamma \vdash (b_1 : \theta_1 | \cdots | b_m : \theta_m) : \mathbb{A}}}{\Gamma \vdash \mathtt{eq}_{\mathbb{A}}\ (a_1 : \phi_1 | \cdots | a_n : \phi_n)\ (b_1 : \theta_1 | \cdots | b_m : \theta_m) : \mathbb{B}}$$

But N has also type $\mathbb{B}$ because it is a formula.

➤ Rule R25

As for Rule (R24).

□

CHAPTER 5

Equivalence of semantics

Chapter 3 presents two semantics of the Nλ language:

- **Denotational semantics**, which assigns the appropriate meaning (denotation) to grammar terms in the form of mathematical objects such as sets, functions, atoms, etc.
- **Operational semantics**, which through reduction rules defines a computational model of the language, and provides an algorithmic method to compute the result of a given program.

In this chapter we will show that the two semantics closely correspond to each other. More precisely, we prove that program reduction according to operational semantics does not change the denotational meaning of the program. That is, the result value of running the program is the same mathematical object as the denotation of the initial program. For this we prove that denotation is invariant under a single reduction step. Technically, for a given valuation ν of free atom variables occurring in a term M, a context ψ that is true in this valuation ($\mathcal{A}, \nu \vDash \psi$) and a valuation ρ of free term variables, the following implication holds:

$$\psi \vdash M \to N \qquad \Rightarrow \qquad [\![M]\!]^\rho_\nu = [\![N]\!]^\rho_\nu$$

where the reduction relation $\to$ is defined in Section 3.7, and denotational semantics $[\![\_]\!]^\rho_\nu$ in Sections 3.3 and 3.5.

5.1. Substitution in denotation

In the proof of the above implication, we will need the following substitution lemma:

Lemma 5.1. *For any terms M and N and valuations ρ and ν of free term and atom variables, the equality holds:*

$$[\![M]\!]^{\rho[x\mapsto v]}_\nu = [\![M[N/x]]\!]^\rho_\nu$$

where $v = [\![N]\!]^\rho_\nu$.

Proof. By induction on the structure of the term M.

First note that if $x \notin \mathrm{FV_T}(M)$ then $[\![M]\!]^{\rho[x\mapsto v]}_\nu = [\![M]\!]^\rho_\nu = [\![M[N/x]]\!]^\rho_\nu$. This completes the proof for cases where the term M is a constant, an atom variable, a formula or a variant. Let us analyze the remaining cases.

➤ **M is a term variable**

If $M = x$ then

$$[\![M]\!]_{\nu}^{\rho[x\mapsto v]} = [\![x]\!]_{\nu}^{\rho[x\mapsto v]} = v = [\![N]\!]_{\nu}^{\rho} = [\![x[N/x]]\!]_{\nu}^{\rho} = [\![M[N/x]]\!]_{\nu}^{\rho}$$

If $M = y \neq x$ then it is the same situation as for the other cases where $x \notin \mathrm{FV}_{\mathrm{T}}(M)$.

➤ **M is an abstraction**

Assume that $M = \lambda y.M'$, where $x \neq y$ and $y \notin \mathrm{FV}_{\mathrm{T}}(N)$. Then there is a chain of equalities:

$$\begin{array}{rcl} [\![M]\!]_{\nu}^{\rho[x\mapsto v]}(w) & = & [\![\lambda y.M']\!]_{\nu}^{\rho[x\mapsto v]}(w) \\ & \overset{(D2)}{=} & [\![M']\!]_{\nu}^{\rho[x\mapsto v,y\mapsto w]} \\ & \overset{\text{ind.}}{=} & [\![M'[N/x]]\!]_{\nu}^{\rho[y\mapsto w]} \\ & \overset{(D2)}{=} & [\![\lambda y.M'[N/x]]\!]_{\nu}^{\rho}(w) \\ & = & [\![(\lambda y.M')[N/x]]\!]_{\nu}^{\rho}(w) \\ & = & [\![M[N/x]]\!]_{\nu}^{\rho}(w) \end{array}$$

The second and fourth equality follow from the definition of denotation for abstraction (D2). The third equality is the induction hypothesis.

➤ **M is an application**

Suppose $M = M_1M_2$. Then:

$$\begin{array}{rcl} [\![M]\!]_{\nu}^{\rho[x\mapsto v]} & = & [\![M_1M_2]\!]_{\nu}^{\rho[x\mapsto v]} \\ & \overset{(D3)}{=} & [\![M_1]\!]_{\nu}^{\rho[x\mapsto v]}([\![M_2]\!]_{\nu}^{\rho[x\mapsto v]}) \\ & \overset{\text{ind.}}{=} & [\![M_1[N/x]]\!]_{\nu}^{\rho}([\![M_2[N/x]]\!]_{\nu}^{\rho}) \\ & \overset{(D3)}{=} & [\![(M_1M_2)[N/x]]\!]_{\nu}^{\rho} \\ & = & [\![M[N/x]]\!]_{\nu}^{\rho} \end{array}$$

As above, the second and fourth equality is the result of the definition of denotation (D3), and the third equality follows from the induction hypothesis.

➤ **M is a set**

If term M takes the form of a set expression, then we have:

$$\begin{aligned}
[\![M]\!]^{\rho[x\mapsto v]}_{\nu} &= [\![\{M_1\colon \phi_1 \text{ for } \sigma_1, \ldots, M_n\colon \phi_n \text{ for } \sigma_n\}]\!]^{\rho[x\mapsto v]}_{\nu} \\
&\overset{(D20)}{=} \bigcup_{1\le i\le n} \{[\![M_i]\!]^{\rho[x\mapsto v]}_{\nu_i} \mid v^i_1, \ldots, v^i_{k_i} \in \mathcal{A} \text{ s.t. } \mathcal{A}, \nu_i \vDash \phi_i\} \\
&\overset{\text{ind.}}{=} \bigcup_{1\le i\le n} \{[\![M_i[N/x]]\!]^{\rho}_{\nu_i} \mid v^i_1, \ldots, v^i_{k_i} \in \mathcal{A} \text{ s.t. } \mathcal{A}, \nu_i \vDash \phi_i\} \\
&\overset{(D20)}{=} [\![\{M_1[N/x]\colon \phi_1 \text{ for } \sigma_1, \ldots, M_n[N/x]\colon \phi_n \text{ for } \sigma_n\}]\!]^{\rho}_{\nu} \\
&= [\![\{M_1\colon \phi_1 \text{ for } \sigma_1, \ldots, M_n\colon \phi_n \text{ for } \sigma_n\}[N/x]]\!]^{\rho}_{\nu} \\
&= [\![M[N/x]]\!]^{\rho}_{\nu}
\end{aligned}$$

where $\sigma_i = (a^i_1, \ldots, a^i_{k_i})$ and $\nu_i = \nu[a^i_j \to v^i_j]^{k_i}_{j=1}$ for $i = 1, \ldots, n$, similar to the definition of denotation for the set expression (D20). Again, the second and fourth equality follow precisely from that definition. Thanks to the induction hypothesis, we have the third equality.

□

5.2. Equivalence theorem

We can now move on to the theorem about the preservation of term denotations by operational semantics.

Theorem 5.2. *For terms M, N and valuations ν, ρ of free atom and term variables occurring in these terms, the following implication holds:*

$$\mathcal{A}, \nu \vDash \psi \quad \wedge \quad \psi \vdash M \to N \qquad \Rightarrow \qquad [\![M]\!]^{\rho}_{\nu} = [\![N]\!]^{\rho}_{\nu}$$

Proof. By induction on the structure of term M and case analysis on the reducing rule for $\psi \vdash M \to N$.

➤ Rule R1 (abstraction)

Suppose $M = \lambda x.M'$ and $N = \lambda x.N'$. By rule (R1) we know that $\psi \vdash M' \to N'$. Therefore, for any value v:

$$\begin{aligned}
[\![M]\!]^{\rho}_{\nu}(v) &= [\![\lambda x.M']\!]^{\rho}_{\nu}(v) \\
&\overset{(D2)}{=} [\![M']\!]^{\rho[x\mapsto v]}_{\nu} \\
&\overset{\text{ind.}}{=} [\![N']\!]^{\rho[x\mapsto v]}_{\nu} \\
&\overset{(D2)}{=} [\![\lambda x.N']\!]^{\rho}_{\nu}(v) \\
&= [\![N]\!]^{\rho}_{\nu}(v)
\end{aligned}$$

➤ Rule R2 (application)

In this case, $M = M_1M_2$ and $N = M_1'M_2$. By rule (R2) we have $\psi \vdash M_1 \to M_1'$ and as a result:

$$\begin{aligned}
[\![M]\!]^\rho_\nu &= [\![M_1M_2]\!]^\rho_\nu \\
&\overset{(D3)}{=} [\![M_1]\!]^\rho_\nu([\![M_2]\!]^\rho_\nu) \\
&\overset{\text{ind.}}{=} [\![M_1']\!]^\rho_\nu([\![M_2]\!]^\rho_\nu) \\
&\overset{(D3)}{=} [\![M_1']\!]^\rho_\nu([\![M_2]\!]^\rho_\nu) \\
&= [\![N]\!]^\rho_\nu
\end{aligned}$$

➤ Rule R3 (application)

Analogous to the case for Rule (R2) above.

➤ Rule R4 (substitution)

This time $M = (\lambda x.M')N'$ and $N = M'[N'/x]$. Using Lemma 5.1, we obtain the following sequence of equalities:

$$\begin{aligned}
[\![M]\!]^\rho_\nu &= [\![(\lambda x.M')N']\!]^\rho_\nu \\
&\overset{(D3)}{=} [\![(\lambda x.M')]\!]^\rho_\nu([\![N']\!]^\rho_\nu) \\
&\overset{(D2)}{=} [\![M']\!]^{\rho[x\mapsto[\![N']\!]^\rho_\nu]}_\nu \\
&\overset{(5.1)}{=} [\![M'[N'/x]]\!]^\rho_\nu \\
&= [\![N]\!]^\rho_\nu
\end{aligned}$$

➤ Rule R5 (`empty`)

For the constant `empty` the deduction is simple:

$$[\![\texttt{empty}]\!]^\rho_\nu \overset{(D4)}{=} \emptyset \overset{(D20)}{=} [\![\{\}]\!]^\rho_\nu$$

➤ Rule R6 (`atoms`)

For the set of all atoms we have the following equalities:

$$\begin{aligned}
[\![\texttt{atoms}]\!]^\rho_\nu &\overset{(D5)}{=} \mathcal{A} \\
&= \{v \mid v \in \mathcal{A}\} \\
&\overset{(D18)}{=} \{[\![a]\!]^\rho_{\nu[a\mapsto v]} \mid v \in \mathcal{A}\} \\
&\overset{(D20)}{=} [\![\{a \colon \text{for } a\}]\!]^\rho_\nu
\end{aligned}$$

➤ Rule R7 (`insert`)

The first step in the proof for the constant `insert` is to use the definition of denotation (D6) for that constant. Then convert the term to a set according to the denotation definition (D20), and finally convert it back to a term using the same denotation definition (D20) for the set with an element already added.

$$\begin{aligned}
&[\![\texttt{insert}\ M_0\ \{M_1\colon \phi_1 \text{ for } \sigma_1, \ldots, M_n\colon \phi_n \text{ for } \sigma_n\}]\!]^\rho_\nu \\
&\overset{(D3)}{=} [\![\texttt{insert}]\!]^\rho_\nu([\![M_0]\!]^\rho_\nu)([\![\{M_1\colon \phi_1 \text{ for } \sigma_1, \ldots, M_n\colon \phi_n \text{ for } \sigma_n\}]\!]^\rho_\nu) \\
&\overset{(D6)}{=} \{[\![M_0]\!]^\rho_\nu\} \cup [\![\{M_1\colon \phi_1 \text{ for } \sigma_1, \ldots, M_n\colon \phi_n \text{ for } \sigma_n\}]\!]^\rho_\nu \\
&\overset{(D20)}{=} \{[\![M_0]\!]^\rho_\nu\} \cup \bigcup_{1\le i\le n} \{[\![M_i]\!]^\rho_{\nu_i} \mid v^i_1, \ldots, v^i_{k_i} \in \mathcal{A} \text{ s.t. } \mathcal{A}, \nu_i \vDash \phi_i\} \\
&\overset{(D20)}{=} [\![\{M_0, M_1\colon \phi_1 \text{ for } \sigma_1, \ldots, M_n\colon \phi_n \text{ for } \sigma_n\}]\!]^\rho_\nu
\end{aligned}$$

where $\sigma_i = \{a^i_1, \ldots, a^i_{k_i}\}$ and $\nu_i = \nu[a^i_j \to v^i_j]^{k_i}_{j=1}$ for $i = 1, \ldots, n$.

➤ Rule R8 (`map`)

The proof for the operation `map` uses the denotation definitions for this operation (D7) and for set expressions (D20) to transform a term into a set. Then this set, after appropriate modifications, is turned back into a term using definition (D20) again.

$$\begin{aligned}
&[\![\texttt{map}\ M_0\ \{M_1\colon \phi_1 \text{ for } \sigma_1, \ldots, M_n\colon \phi_n \text{ for } \sigma_n\}]\!]^\rho_\nu \\
&\overset{(D3)}{=} [\![\texttt{map}]\!]^\rho_\nu([\![M_0]\!]^\rho_\nu)([\![\{M_1\colon \phi_1 \text{ for } \sigma_1, \ldots, M_n\colon \phi_n \text{ for } \sigma_n\}]\!]^\rho_\nu) \\
&\overset{(D20)}{=} [\![\texttt{map}]\!]^\rho_\nu([\![M_0]\!]^\rho_\nu)(\bigcup_{1\le i\le n} \{[\![M_i]\!]^\rho_{\nu_i} \mid v^i_1, \ldots, v^i_{k_i} \in \mathcal{A} \text{ s.t. } \mathcal{A}, \nu_i \vDash \phi_i\}) \\
&\overset{(D7)}{=} \bigcup_{1\le i\le n} \{[\![M_0]\!]^\rho_\nu([\![M_i]\!]^\rho_{\nu_i}) \mid v^i_1, \ldots, v^i_{k_i} \in \mathcal{A} \text{ s.t. } \mathcal{A}, \nu_i \vDash \phi_i\} \\
&\overset{(*)}{=} \bigcup_{1\le i\le n} \{[\![M_0 M_i]\!]^\rho_{\nu_i} \mid v^i_1, \ldots, v^i_{k_i} \in \mathcal{A} \text{ s.t. } \mathcal{A}, \nu_i \vDash \phi_i\} \\
&\overset{(D20)}{=} [\![\{MM_1\colon \phi_1 \text{ for } \sigma_1, \ldots, MM_n\colon \phi_n \text{ for } \sigma_n\}]\!]^\rho_\nu
\end{aligned}$$

where $\sigma_i = \{a^i_1, \ldots, a^i_{k_i}\}$ and $\nu_i = \nu[a^i_j \to v^i_j]^{k_i}_{j=1}$ for $i = 1, \ldots, n$.
The equality marked with (*) is due to the fact:

$$[\![M]\!]^\rho_\nu([\![M_i]\!]^\rho_{\nu_i}) = [\![M]\!]^\rho_{\nu_i}([\![M_i]\!]^\rho_{\nu_i}) \overset{(D3)}{=} [\![MM_i]\!]^\rho_{\nu_i}$$

which results from the side condition for rule (R8): $\mathrm{V_A}(M_0) \cap \bigcup^n_{i=1} \sigma_i = \emptyset$, and the definition of denotation for application (D3).

➤ Rule R9 (`sum`)

The proof for sum of sets requires more steps. It begins by converting a term to a set using the denotation definitions for application (D3), sum (D8), and set (D20):

$$\begin{aligned}
&[\![\texttt{sum}\ \{\ldots, \{M_1\colon \phi_1 \text{ for } \sigma_1, \ldots, M_n\colon \phi_n \text{ for } \sigma_n\}\colon \phi \text{ for } \sigma, \ldots\}]\!]^\rho_\nu \\
&\overset{(D3)}{=} [\![\texttt{sum}]\!]^\rho_\nu [\![\{\ldots, \{M_1\colon \phi_1 \text{ for } \sigma_1, \ldots, M_n\colon \phi_n \text{ for } \sigma_n\}\colon \phi \text{ for } \sigma, \ldots\}]\!]^\rho_\nu \\
&\overset{(D8)}{=} \bigcup [\![\{\ldots, \{M_1\colon \phi_1 \text{ for } \sigma_1, \ldots, M_n\colon \phi_n \text{ for } \sigma_n\}\colon \phi \text{ for } \sigma, \ldots\}]\!]^\rho_\nu \\
&\overset{(D20)}{=} \bigcup (\cdots \cup \{[\![\{M_1\colon \phi_1 \text{ for } \sigma_1, \ldots, M_n\colon \phi_n \text{ for } \sigma_n\}]\!]^\rho_{\nu'} \mid v_1 \ldots, v_k \in \mathcal{A} \text{ s.t. } \mathcal{A}, \nu' \vDash \phi\} \cup \cdots)
\end{aligned}$$

Then we transform the set using basic properties of set-theoretic union:

$$
\begin{aligned}
&\bigcup(\cdots \cup \{[\![\{M_1\colon \phi_1 \text{ for } \sigma_1, \ldots, M_n\colon \phi_n \text{ for } \sigma_n\}]\!]^{\rho}_{\nu'} \mid v_1 \ldots, v_k \in \mathcal{A} \text{ s.t. } \mathcal{A}, \nu' \vDash \phi\} \cup \cdots) \\
&= \cdots \cup \bigcup \{[\![\{M_1\colon \phi_1 \text{ for } \sigma_1, \ldots, M_n\colon \phi_n \text{ for } \sigma_n\}]\!]^{\rho}_{\nu'} \mid v_1 \ldots, v_k \in \mathcal{A} \text{ s.t. } \mathcal{A}, \nu' \vDash \phi\} \cup \cdots \\
&= \cdots \cup \bigcup_{\substack{v_1 \ldots, v_k \in \mathcal{A} \\ \mathcal{A}, \nu' \vDash \phi}} [\![\{M_1\colon \phi_1 \text{ for } \sigma_1, \ldots, M_n\colon \phi_n \text{ for } \sigma_n\}]\!]^{\rho}_{\nu'} \cup \cdots
\end{aligned}
$$

We again use the denotation definition for the set (D20), this time applying it to elements of the set of sets:

$$
\begin{aligned}
&\cdots \cup \bigcup_{\substack{v_1 \ldots, v_k \in \mathcal{A} \\ \mathcal{A}, \nu' \vDash \phi}} [\![\{M_1\colon \phi_1 \text{ for } \sigma_1, \ldots, M_n\colon \phi_n \text{ for } \sigma_n\}]\!]^{\rho}_{\nu'} \cup \cdots \\
&\overset{(D20)}{=} \cdots \cup \bigcup_{\substack{v_1 \ldots, v_k \in \mathcal{A} \\ \mathcal{A}, \nu' \vDash \phi}} \bigcup_{1 \le i \le n} \{[\![M_i]\!]^{\rho}_{\nu_i} \mid v^i_1, \ldots, v^i_{k_i} \in \mathcal{A} \text{ s.t. } \mathcal{A}, \nu_i \vDash \phi_i\} \cup \cdots
\end{aligned}
$$

Then we change the union order, and reformulate:

$$
\begin{aligned}
&\cdots \cup \bigcup_{\substack{v_1 \ldots, v_k \in \mathcal{A} \\ \mathcal{A}, \nu' \vDash \phi}} \bigcup_{1 \le i \le n} \{[\![M_i]\!]^{\rho}_{\nu_i} \mid v^i_1, \ldots, v^i_{k_i} \in \mathcal{A} \text{ s.t. } \mathcal{A}, \nu_i \vDash \phi_i\} \cup \cdots \\
&= \cdots \cup \bigcup_{1 \le i \le n} \bigcup_{\substack{v_1 \ldots, v_k \in \mathcal{A} \\ \mathcal{A}, \nu' \vDash \phi}} \{[\![M_i]\!]^{\rho}_{\nu_i} \mid v^i_1, \ldots, v^i_{k_i} \in \mathcal{A} \text{ s.t. } \mathcal{A}, \nu_i \vDash \phi_i\} \cup \cdots \\
&= \cdots \cup \bigcup_{1 \le i \le n} \{[\![M_i]\!]^{\rho}_{\nu_i} \mid v^i_1, \ldots, v^i_{k_i}, v_1, \ldots, v_k \in \mathcal{A} \text{ s.t. } \mathcal{A}, \nu_i \vDash \phi_i \text{ and } \mathcal{A}, \nu' \vDash \phi\} \cup \cdots \\
&= \cdots \cup \bigcup_{1 \le i \le n} \{[\![M_i]\!]^{\rho}_{\nu_i} \mid v^i_1, \ldots, v^i_{k_i}, v_1, \ldots, v_k \in \mathcal{A} \text{ s.t. } \mathcal{A}, \nu_i \vDash \phi_i \wedge \phi\} \cup \cdots
\end{aligned}
$$

Finally, we come back to the term again using the denotation definition for the set (D20):

$$
\begin{aligned}
&\cdots \cup \bigcup_{1 \le i \le n} \{[\![M_i]\!]^{\rho}_{\nu_i} \mid v^i_1, \ldots, v^i_{k_i}, v_1, \ldots, v_k \in \mathcal{A} \text{ s.t. } \mathcal{A}, \nu_i \vDash \phi_i \wedge \phi\} \cup \cdots \\
&\overset{(D20)}{=} [\![\{\ldots, M_1\colon \phi_1 \wedge \phi \text{ for } \sigma_1 \cup \sigma, \ldots,\ M_n\colon \phi_n \wedge \phi \text{ for } \sigma_n \cup \sigma, \ldots\}]\!]^{\rho}_{\nu}
\end{aligned}
$$

In all the above steps, we used the following symbols for $i = 1, \ldots, n$:

$$
\begin{aligned}
\sigma &= (a_1, \ldots, a_k) & \nu' &= \nu[a_i \to v_i]^{k}_{i=1} \\
\sigma_i &= (a^i_1, \ldots, a^i_{k_i}) & \nu_i &= \nu'[a^i_j \to v^i_j]^{k_i}_{j=1}
\end{aligned}
$$

Due to the side condition of the rule (R9), each ν_i is an extension of ν' that does not change its current mapping.

➤ Rule R10 (`true`, `false`)

The theorem for the constants `true` and `false` follows from the equations:

$$\llbracket \texttt{true} \rrbracket_\nu^\rho \overset{(D9)}{=} tt = \begin{cases} tt & \text{if } \mathcal{A}, \nu \vDash \top \\ f\!f & \text{otherwise} \end{cases} \overset{(D19)}{=} \llbracket \top \rrbracket_\nu^\rho$$

$$\llbracket \texttt{false} \rrbracket_\nu^\rho \overset{(D10)}{=} f\!f = \begin{cases} tt & \text{if } \mathcal{A}, \nu \vDash \bot \\ f\!f & \text{otherwise} \end{cases} \overset{(D19)}{=} \llbracket \bot \rrbracket_\nu^\rho$$

➤ Rule R11 (`not`)

The case of negation follows mainly from the definition of denotation for the formula (D19) and negation (D11):

$$\begin{aligned}
\llbracket \texttt{not}\ \phi \rrbracket_\nu^\rho &\overset{(D3)}{=} \llbracket \texttt{not} \rrbracket_\nu^\rho(\llbracket \phi \rrbracket_\nu^\rho) \\
&\overset{(D19)}{=} \llbracket \texttt{not} \rrbracket_\nu^\rho\left(\begin{cases} tt & \text{if } \mathcal{A}, \nu \vDash \phi \\ f\!f & \text{otherwise} \end{cases} \right) \\
&= \begin{cases} \llbracket \texttt{not} \rrbracket_\nu^\rho(tt) & \text{if } \mathcal{A}, \nu \vDash \phi \\ \llbracket \texttt{not} \rrbracket_\nu^\rho(f\!f) & \text{otherwise} \end{cases} \\
&\overset{(D11)}{=} \begin{cases} f\!f & \text{if } \mathcal{A}, \nu \vDash \phi \\ tt & \text{otherwise} \end{cases} \\
&= \begin{cases} tt & \text{if } \mathcal{A}, \nu \vDash \neg\phi \\ f\!f & \text{otherwise} \end{cases} \\
&\overset{(D19)}{=} \llbracket \neg\phi \rrbracket_\nu^\rho
\end{aligned}$$

➤ Rule R12 (`or`, `and`)

The reasoning for logical disjunction relies mainly on the definition of denotation for the constant `or` (D12) and for the formula (D19):

$$\begin{aligned}
\llbracket \texttt{or}\ \phi_1\ \phi_2 \rrbracket_\nu^\rho &\overset{(D3)}{=} \llbracket \texttt{or} \rrbracket_\nu^\rho(\llbracket \phi_1 \rrbracket_\nu^\rho)(\llbracket \phi_2 \rrbracket_\nu^\rho) \\
&\overset{(D12)}{=} \begin{cases} tt & \text{if } \llbracket \phi_1 \rrbracket_\nu^\rho = tt \text{ or } \llbracket \phi_2 \rrbracket_\nu^\rho = tt \\ f\!f & \text{otherwise} \end{cases} \\
&\overset{(D19)}{=} \begin{cases} tt & \text{if } \mathcal{A}, \nu \vDash \phi_1 \text{ or } \mathcal{A}, \nu \vDash \phi_2 \\ f\!f & \text{otherwise} \end{cases} \\
&= \begin{cases} tt & \text{if } \mathcal{A}, \nu \vDash \phi_1 \vee \phi_2 \\ f\!f & \text{otherwise} \end{cases} \\
&\overset{(D19)}{=} \llbracket \phi_1 \vee \phi_2 \rrbracket_\nu^\rho
\end{aligned}$$

The reasoning for logical conjunction is analogous except that it uses the definition of denotation for the constant `and` (D13).

➤ Rule R13 (`isEmpty`)

The proof for the emptiness test begins with the use of the denotation definition for the application (D3) and for the constant `isEmpty` (D14):

$$\begin{aligned}
&[\![\texttt{isEmpty}\ \{M_1 \colon \phi_1 \text{ for } \sigma_1, \ldots, M_n \colon \phi_n \text{ for } \sigma_n\}]\!]^\rho_\nu \\
&\overset{(D3)}{=} [\![\texttt{isEmpty}]\!]^\rho_\nu([\![\{\mathtt{M_1} \colon \phi_1 \text{ for } \sigma_1, \ldots, \mathtt{M_n} \colon \phi_\mathtt{n} \text{ for } \sigma_\mathtt{n}\}]\!]^\rho_\nu) \\
&\overset{(D14)}{=} \begin{cases} tt & \text{if } [\![\{M_1 \colon \phi_1 \text{ for } \sigma_1, \ldots, M_n \colon \phi_n \text{ for } \sigma_n\}]\!]^\rho_\nu = \emptyset \\ f\!f & \text{otherwise} \end{cases}
\end{aligned}$$

Looking at the above condition in more detail, we obtain:

$$\begin{aligned}
&[\![\{M_1 \colon \phi_1 \text{ for } \sigma_1, \ldots, M_n \colon \phi_n \text{ for } \sigma_n\}]\!]^\rho_\nu = \emptyset \\
&\overset{(D20)}{\Leftrightarrow} \bigcup_{1 \le i \le n} \{[\![M_i]\!]^\rho_{\nu_i} \mid v^i_1, \ldots, v^i_{k_i} \in \mathcal{A} \text{ s.t. } \mathcal{A}, \nu_i \vDash \phi_i\} = \emptyset \\
&\Leftrightarrow \bigwedge_{1 \le i \le n} \bigwedge_{v^i_1, \ldots, v^i_{k_i} \in \mathcal{A}} \mathcal{A}, \nu_i \vDash \neg\phi_i \\
&\Leftrightarrow \bigwedge_{1 \le i \le n} \mathcal{A}, \nu \vDash \forall a^i_1 \ldots \forall a^i_{k_i}.\neg\phi_i \\
&\Leftrightarrow \bigwedge_{1 \le i \le n} \mathcal{A}, \nu \vDash \forall \sigma_i.\neg\phi_i \\
&\Leftrightarrow \mathcal{A}, \nu \vDash \bigwedge_{1 \le i \le n} \forall \sigma_i.\neg\phi_i
\end{aligned}$$

where $\sigma_i = \{a^i_1, \ldots, a^i_{k_i}\}$ and $\nu_i = \nu[a^i_j \to v^i_j]^{k_i}_{j=1}$ for $i = 1, \ldots, n$.

The first equivalence follows from the definition of set denotation, and the remaining ones are basic properties of first-order formulas. Using this, we conclude that the denotation for the term with the constant `isEmpty` is equal to:

$$\begin{aligned}
&\begin{cases} tt & \text{if } [\![\{M_1 \colon \phi_1 \text{ for } \sigma_1, \ldots, M_n \colon \phi_n \text{ for } \sigma_n\}]\!]^\rho_\nu = \emptyset \\ f\!f & \text{otherwise} \end{cases} \\
&= \begin{cases} tt & \text{if } \mathcal{A}, \nu \vDash \bigwedge_{1 \le i \le n} \forall \sigma_i.\neg\phi_i \\ f\!f & \text{otherwise} \end{cases} \\
&\overset{(D19)}{=} [\![\bigwedge_{1 \le i \le n} \forall \sigma_i.\neg\phi_i]\!]^\rho_\nu
\end{aligned}$$

➤ Rule R14

In this case, $M = \texttt{if}\ \phi\ M'\ N'$ and $N = M'$. Besides, we know from the premise of the rule (R14) that $\mathcal{A} \vDash \psi \Rightarrow \phi$. The premise of the theorem says that $\mathcal{A}, \nu \vDash \psi$, hence it follows that $\mathcal{A}, \nu \vDash \phi$. This brings us to the following conclusion:

$$\begin{aligned}
[\![M]\!]^{\rho}_{\nu} &= [\![\texttt{if}\ \phi\ M'\ N']\!]^{\rho}_{\nu} \\
&\overset{(D3)}{=} [\![\texttt{if}]\!]^{\rho}_{\nu}([\![\phi]\!]^{\rho}_{\nu})([\![M']\!]^{\rho}_{\nu})([\![N']\!]^{\rho}_{\nu}) \\
&\overset{(D15)}{=} \begin{cases} [\![M']\!]^{\rho}_{\nu} & \text{if } [\![\phi]\!]^{\rho}_{\nu} = tt \\ [\![N']\!]^{\rho}_{\nu} & \text{otherwise} \end{cases} \\
&\overset{(D19)}{=} \begin{cases} [\![M']\!]^{\rho}_{\nu} & \text{if } \mathcal{A}, \nu \vDash \phi \\ [\![N']\!]^{\rho}_{\nu} & \text{otherwise} \end{cases} \\
&= [\![M']\!]^{\rho}_{\nu} \\
&= [\![N]\!]^{\rho}_{\nu}
\end{aligned}$$

➤ Rule R15

As above, except that we first show that $\mathcal{A}, \nu \vDash \neg\phi$.

➤ Rule R16

For this rule, $M = (\texttt{if}\ \phi\ M_1\ M_2)\ M_0$ and $N = \texttt{if}\ \phi\ (M_1 M_0)\ (M_2 M_0)$. The proof in this case is as follows:

$$\begin{aligned}
[\![M]\!]^{\rho}_{\nu} &= [\![\texttt{if}\ \phi\ M_1\ M_2\ M_0]\!]^{\rho}_{\nu} \\
&\overset{(D3)}{=} [\![\texttt{if}]\!]^{\rho}_{\nu}([\![\phi]\!]^{\rho}_{\nu})([\![M_1]\!]^{\rho}_{\nu})([\![M_2]\!]^{\rho}_{\nu})([\![M_0]\!]^{\rho}_{\nu}) \\
&\overset{(D15)}{=} \left(\begin{cases} [\![M_1]\!]^{\rho}_{\nu} & \text{if } [\![\phi]\!]^{\rho}_{\nu} = tt \\ [\![M_2]\!]^{\rho}_{\nu} & \text{otherwise} \end{cases}\right)([\![M_0]\!]^{\rho}_{\nu}) \\
&= \begin{cases} [\![M_1]\!]^{\rho}_{\nu}([\![M_0]\!]^{\rho}_{\nu}) & \text{if } [\![\phi]\!]^{\rho}_{\nu} = tt \\ [\![M_2]\!]^{\rho}_{\nu}([\![M_0]\!]^{\rho}_{\nu}) & \text{otherwise} \end{cases} \\
&\overset{(D3)}{=} \begin{cases} [\![M_1 M_0]\!]^{\rho}_{\nu} & \text{if } [\![\phi]\!]^{\rho}_{\nu} = tt \\ [\![M_2 M_0]\!]^{\rho}_{\nu} & \text{otherwise} \end{cases} \\
&\overset{(D15)}{=} [\![\texttt{if}]\!]^{\rho}_{\nu}([\![\phi]\!]^{\rho}_{\nu})([\![M_1 M_0]\!]^{\rho}_{\nu})([\![M_2 M_0]\!]^{\rho}_{\nu}) \\
&\overset{(D3)}{=} [\![\texttt{if}\ \phi\ (M_1 M_0)\ (M_2 M_0)]\!]^{\rho}_{\nu} \\
&= [\![N]\!]^{\rho}_{\nu}
\end{aligned}$$

➤ Rule R17

The deduction for the conditional term with atom variables is straightforward:

$$\begin{aligned}
[\![\texttt{if}\ \phi\ a\ b]\!]^{\rho}_{\nu} &\overset{(D3)}{=} [\![\texttt{if}]\!]^{\rho}_{\nu}([\![\phi]\!]^{\rho}_{\nu})([\![a]\!]^{\rho}_{\nu})([\![b]\!]^{\rho}_{\nu}) \\
&\overset{(D15)}{=} \begin{cases} [\![a]\!]^{\rho}_{\nu} & \text{if } [\![\phi]\!]^{\rho}_{\nu} = tt \\ [\![b]\!]^{\rho}_{\nu} & \text{otherwise} \end{cases} \\
&\overset{(D19)}{=} \begin{cases} [\![a]\!]^{\rho}_{\nu} & \text{if } \mathcal{A}, \nu \vDash \phi \\ [\![b]\!]^{\rho}_{\nu} & \text{if } \mathcal{A}, \nu \vDash \neg\phi \end{cases} \\
&\overset{(D21)}{=} [\![a\colon \phi \mid b\colon \neg\phi]\!]^{\rho}_{\nu}
\end{aligned}$$

- Rule R18

  Reasoning for a conditional term with formulas requires the analysis of satisfiability of individual formulas:

$$
\begin{aligned}
[\![\texttt{if}\ \phi\ \phi_1\ \phi_2]\!]^\rho_\nu &\overset{(D3)}{=} [\![\texttt{if}]\!]^\rho_\nu([\![\phi]\!]^\rho_\nu)([\![\phi_1]\!]^\rho_\nu)([\![\phi_2]\!]^\rho_\nu) \\
&\overset{(D15)}{=} \begin{cases} [\![\phi_1]\!]^\rho_\nu & \text{if } [\![\phi]\!]^\rho_\nu = tt \\ [\![\phi_2]\!]^\rho_\nu & \text{otherwise} \end{cases} \\
&\overset{(D19)}{=} \begin{cases} \begin{cases} tt & \text{if } \mathcal{A},\nu \vDash \phi_1 \\ ff & \text{otherwise} \end{cases} & \text{if } [\![\phi]\!]^\rho_\nu = tt \\ \begin{cases} tt & \text{if } \mathcal{A},\nu \vDash \phi_2 \\ ff & \text{otherwise} \end{cases} & \text{otherwise} \end{cases} \\
&\overset{(D19)}{=} \begin{cases} tt & \text{if } \mathcal{A},\nu \vDash \phi_1 \text{ and } \mathcal{A},\nu \vDash \phi \\ ff & \text{if } \mathcal{A},\nu \nvDash \phi_1 \text{ and } \mathcal{A},\nu \vDash \phi \\ tt & \text{if } \mathcal{A},\nu \vDash \phi_2 \text{ and } \mathcal{A},\nu \nvDash \phi \\ ff & \text{if } \mathcal{A},\nu \nvDash \phi_2 \text{ and } \mathcal{A},\nu \nvDash \phi \end{cases} \\
&\overset{(D19)}{=} [\![(\phi_1 \wedge \phi) \vee (\phi_2 \wedge \neg\phi)]\!]
\end{aligned}
$$

- Rule R19

  The proof for this case begins as usual, with the use of the denotation definitions for the application (D3), conditional expression (D15), and set (D20).

$$
\begin{aligned}
&[\![\texttt{if}\ \phi\ \{M_1\colon \phi_1 \text{ for } \sigma_1, \ldots, M_n\colon \phi_n \text{ for } \sigma_n\}\ \{N_1\colon \theta_1 \text{ for } \pi_1, \ldots, N_k\colon \theta_k \text{ for } \pi_k\}]\!]^\rho_\nu \\
&\overset{(D3)}{=} [\![\texttt{if}]\!]^\rho_\nu[\![\phi]\!]^\rho_\nu[\![\{M_1\colon \phi_1 \text{ for } \sigma_1, \ldots, M_n\colon \phi_n \text{ for } \sigma_n\}]\!]^\rho_\nu[\![\{N_1\colon \theta_1 \text{ for } \pi_1, \ldots, N_k\colon \theta_k \text{ for } \pi_k\}]\!]^\rho_\nu \\
&\overset{(D15)}{=} \begin{cases} [\![\{M_1\colon \phi_1 \text{ for } \sigma_1, \ldots, M_n\colon \phi_n \text{ for } \sigma_n\}]\!]^\rho_\nu & \text{if } [\![\phi]\!]^\rho_\nu = tt \\ [\![\{N_1\colon \theta_1 \text{ for } \pi_1, \ldots, N_k\colon \theta_k \text{ for } \pi_k\}]\!]^\rho_\nu & \text{otherwise} \end{cases} \\
&\overset{(D20)}{=} \begin{cases} \bigcup_{1\le i\le n} \{[\![M_i]\!]^\rho_{\nu_i} \mid v^i_1, \ldots, v^i_{k_i} \in \mathcal{A} \text{ s.t. } \mathcal{A},\nu_i \vDash \phi_i\} & \text{if } \mathcal{A},\nu \vDash \phi \\ \bigcup_{1\le j\le k} \{[\![N_j]\!]^\rho_{\nu'_j} \mid w^j_1, \ldots, w^j_{n_j} \in \mathcal{A} \text{ s.t. } \mathcal{A},\nu'_j \vDash \theta_i\} & \text{if } \mathcal{A},\nu \vDash \neg\phi \end{cases}
\end{aligned}
$$

  where $\sigma_i = \{a^i_1, \ldots, a^i_{k_i}\}$ and $\nu_i = \nu[a^i_j \to v^i_j]^{k_i}_{j=1}$ for $i = 1, \ldots, n$
  and $\pi_i = \{b^i_1, \ldots, b^i_{n_i}\}$ and $\nu'_i = \nu[b^i_j \to w^i_j]^{n_i}_{j=1}$ for $i = 1, \ldots, k$.

  The above means that we obtained one of the two sets depending on the condition $\mathcal{A},\nu \vDash \phi$, which can be written equivalently as follows:

$$
\begin{aligned}
&\bigcup_{1\le i\le n} \{[\![M_i]\!]^\rho_{\nu_i} \mid v^i_1, \ldots, v^i_{k_i} \in \mathcal{A} \text{ s.t. } \mathcal{A},\nu_i \vDash \phi_i \text{ and } \mathcal{A},\nu \vDash \phi\} \\
\cup &\bigcup_{1\le j\le k} \{[\![N_j]\!]^\rho_{\nu'_j} \mid w^j_1, \ldots, w^j_{n_j} \in \mathcal{A} \text{ s.t. } \mathcal{A},\nu'_j \vDash \theta_j \text{ and } \mathcal{A},\nu \vDash \neg\phi\}
\end{aligned}
$$

  Reformulating these conditions slightly, we get:

$$
\begin{aligned}
&\bigcup_{1\le i\le n} \{[\![M_i]\!]^\rho_{\nu_i} \mid v^i_1, \ldots, v^i_{k_i} \in \mathcal{A} \text{ s.t. } \mathcal{A},\nu_i \vDash \phi_i \wedge \phi\} \\
\cup &\bigcup_{1\le j\le k} \{[\![N_j]\!]^\rho_{\nu'_j} \mid w^j_1, \ldots, w^j_{n_j} \in \mathcal{A} \text{ s.t. } \mathcal{A},\nu'_j \vDash \theta_j \wedge \neg\phi\}
\end{aligned}
$$

And this according to the definition of denotation for the set (D20) is equal to:

$$\begin{aligned}&[\![\{M_1\colon \phi_1 \wedge \phi \text{ for } \sigma_1, \ldots, M_n\colon \phi_n \wedge \phi \text{ for } \sigma_n,\\ &\quad N_1\colon \theta_1 \wedge \neg\phi \text{ for } \pi_1, \ldots\ N_k\colon \theta_k \wedge \neg\phi \text{ for } \pi_k\}]\!]^\rho_\nu\end{aligned}$$

➤ Rule R20

In the case of a conditional term with variants, the reasoning is similar to the case of rule (R18) and also requires an analysis of satisfiability of individual formulas:

$$\begin{aligned}
&[\![\texttt{if}\ \phi\ (a_1\colon \phi_1|\cdots|a_n\colon \phi_n)\ (b_1\colon \theta_1|\cdots|b_k\colon \theta_k)]\!]^\rho_\nu\\
\overset{(D3)}{=}\ & [\![\texttt{if}]\!]^\rho_\nu\ [\![\phi]\!]^\rho_\nu\ [\![a_1\colon \phi_1|\cdots|a_n\colon \phi_n]\!]^\rho_\nu\ [\![b_1\colon \theta_1|\cdots|b_k\colon \theta_k]\!]^\rho_\nu\\
\overset{(D15)}{=}\ & \begin{cases}[\![a_1\colon \phi_1|\cdots|a_n\colon \phi_n]\!]^\rho_\nu & \text{if } [\![\phi]\!]^\rho_\nu = tt\\ [\![b_1\colon \theta_1|\cdots|b_k\colon \theta_k]\!]^\rho_\nu & \text{otherwise}\end{cases}\\
\overset{(D21)}{=}\ & \begin{cases}\begin{cases}\nu(a_1) & \text{if } \mathcal{A},\nu \vDash \phi_1\\ \cdots & \\ \nu(a_n) & \text{if } \mathcal{A},\nu \vDash \phi_n\end{cases} & \text{if } [\![\phi]\!]^\rho_\nu = tt\\ \begin{cases}\nu(b_1) & \text{if } \mathcal{A},\nu \vDash \theta_1\\ \cdots & \\ \nu(b_k) & \text{if } \mathcal{A},\nu \vDash \theta_n\end{cases} & \text{otherwise}\end{cases}\\
=\ & \begin{cases}\nu(a_1) & \text{if } \mathcal{A},\nu \vDash \phi_1 \text{ and } \mathcal{A},\nu \vDash \phi\\ \cdots & \\ \nu(a_n) & \text{if } \mathcal{A},\nu \vDash \phi_n \text{ and } \mathcal{A},\nu \vDash \phi\\ \nu(b_1) & \text{if } \mathcal{A},\nu \vDash \theta_1 \text{ and } \mathcal{A},\nu \vDash \neg\phi\\ \cdots & \\ \nu(b_k) & \text{if } \mathcal{A},\nu \vDash \theta_k \text{ and } \mathcal{A},\nu \vDash \neg\phi\end{cases}\\
\overset{(D21)}{=}\ & [\![a_1\colon \phi_1 \wedge \phi|\cdots|a_n\colon \phi_n \wedge \phi|b_1\colon \theta_1 \wedge \neg\phi|\ldots|b_k\colon \theta_k \wedge \neg\phi]\!]^\rho_\nu
\end{aligned}$$

➤ Rule R21

Here $M = \{\ldots, M'\colon \phi \text{ for } \sigma, \ldots\}$ and $N = \{\ldots, N'\colon \phi \text{ for } \sigma, \ldots\}$. By the definition of denotation for the set (D20), we have:

$$\begin{aligned}
[\![M]\!]^\rho_\nu &= [\![\{\ldots, M'\colon \phi \text{ for } \sigma, \ldots\}]\!]^\rho_\nu\\
&\overset{(D20)}{=} \cdots \cup \{[\![M']\!]^\rho_{\nu'} \mid v_1, \ldots, v_k \in \mathcal{A} \text{ s.t. } \mathcal{A},\nu' \vDash \phi\} \cup \cdots
\end{aligned}$$

where $\sigma = \{a_1, \ldots, a_k\}$ and $\nu' = \nu[a_i \to v_i]^k_{i=1}$.

From the assumption of the theorem, we know that $\mathcal{A},\nu \vDash \psi$. Therefore, for $v_1, \ldots, v_k \in \mathcal{A}$ such that $\mathcal{A},\nu' \vDash \phi$ we have $\mathcal{A},\nu' \vDash \psi \wedge \phi$. Additionally, the premise of the rule (R21) shows that $\psi \wedge \phi \vdash\ M' \to N'$. Hence, we can use the induction hypothesis to obtain $[\![M']\!]^\rho_{\nu'} = [\![N']\!]^\rho_{\nu'}$. This allows:

$$\begin{aligned}
& \cdots \cup \{[\![M']\!]^\rho_{\nu'} \mid v_1, \ldots, v_k \in \mathcal{A} \text{ s.t. } \mathcal{A}, \nu' \vDash \phi\} \cup \cdots \\
\overset{\text{ind.}}{=} & \cdots \cup \{[\![N']\!]^\rho_{\nu'} \mid v_1, \ldots, v_k \in \mathcal{A} \text{ s.t. } \mathcal{A}, \nu' \vDash \phi\} \cup \cdots \\
\overset{(D20)}{=} & [\![\{\ldots, N' \colon \phi \text{ for } \sigma, \ldots\}]\!]^\rho_\nu \\
= & [\![N]\!]^\rho_\nu
\end{aligned}$$

➤ Rule R22

Assume that $M = \{\ldots, M' \colon \phi \text{ for } \sigma, \ldots\}$. The proof begins as usual using the definition of denotation:

$$\begin{aligned}
[\![M]\!]^\rho_\nu &= [\![\{\ldots, M' \colon \phi \text{ for } \sigma, \ldots\}]\!]^\rho_\nu \\
&\overset{(D20)}{=} \cdots \cup \{[\![M']\!]^\rho_{\nu'} \mid v_1, \ldots, v_k \in \mathcal{A} \text{ s.t. } \mathcal{A}, \nu' \vDash \phi\} \cup \cdots \qquad (5.1)
\end{aligned}$$

where $\sigma = \{a_1, \ldots, a_k\}$ and $\nu' = \nu[a_i \to v_i]^k_{i=1}$.

Rule (R22) eliminates a subexpression with an unsatisfiable condition. Suppose $M' \colon \phi$ for σ is a subexpression of this form. Then $\mathcal{A} \vDash \psi \Rightarrow \neg\phi$ and using the assumption of theorem $\mathcal{A}, \nu \vDash \psi$, we get $\mathcal{A}, \nu \vDash \neg\phi$. This means that $\mathcal{A}, \nu' \vDash \neg\phi$ for every $v_1, \ldots, v_k \in \mathcal{A}$. It follows that the set $\{[\![M']\!]^\rho_{\nu'} \mid v_1, \ldots, v_k \in \mathcal{A} \text{ s.t. } \mathcal{A}, \nu' \vDash \phi\}$ is empty. Therefore, this set can be removed from the union of the sets (5.1), and after reconverting to term denotation, we get the set expression without the subexpression $M' \colon \phi$ for σ, as required.

➤ Rule R23

Let $M = \cdots |a \colon \phi| \cdots$. The denotation of this term is equal to:

$$[\![M]\!]^\rho_\nu = [\![\cdots |a \colon \phi| \cdots]\!]^\rho_\nu \overset{(D21)}{=} \begin{cases} \cdots \\ [\![M]\!]^\rho_\nu & \text{if } \mathcal{A}, \nu \vDash \phi \\ \cdots \end{cases} \qquad (5.2)$$

Rule (R23) removes a subexpression with an unsatisfiable condition. So suppose that $\mathcal{A} \vDash \psi \Rightarrow \neg\phi$. Combined with the condition $\mathcal{A}, \nu \vDash \psi$, this means that $\mathcal{A}, \nu \vDash \neg\phi$. So, from the list of cases above (5.2), we can remove the one containing the condition $\mathcal{A}, \nu \vDash \phi$. Transforming this list of cases back to a variant, we get the result without the subexpression containing the atom variable a as required.

➤ Rule R24

The proof for the atom equality ($\texttt{eq}_{\mathbb{A}}$) relation is as follows:

$$
\begin{aligned}
&[\![\texttt{eq}_{\mathbb{A}}\ (a_1\colon \phi_1|\dots|a_n\colon \phi_n)\ (b_1\colon \theta_1|\dots|b_m\colon \theta_m)]\!]^{\rho}_{\nu} \\
\overset{(D3)}{=}\ & [\![\texttt{eq}_{\mathbb{A}}]\!]^{\rho}_{\nu}[\![(a_1\colon \phi_1|\dots|a_n\colon \phi_n)]\!]^{\rho}_{\nu}[\![(b_1\colon \theta_1|\dots|b_m\colon \theta_m)]\!]^{\rho}_{\nu} \\
\overset{(D21)}{=}\ & [\![\texttt{eq}_{\mathbb{A}}]\!]^{\rho}_{\nu}\left(\begin{cases}\nu(a_1) & \text{if } \mathcal{A},\nu\vDash\phi_1\\ \dots & \\ \nu(a_n) & \text{if } \mathcal{A},\nu\vDash\phi_n\end{cases}\right)\left(\begin{cases}\nu(b_1) & \text{if } \mathcal{A},\nu\vDash\theta_1\\ \dots & \\ \nu(b_m) & \text{if } \mathcal{A},\nu\vDash\theta_m\end{cases}\right) \\
\overset{(D16)}{=}\ & \begin{cases}tt & \text{if } \nu(a_1)=\nu(b_1) \text{ and } \mathcal{A},\nu\vDash\phi_1 \text{ and } \mathcal{A},\nu\vDash\theta_1\\ \dots & \\ tt & \text{if } \nu(a_n)=\nu(b_m) \text{ and } \mathcal{A},\nu\vDash\phi_n \text{ and } \mathcal{A},\nu\vDash\theta_m\\ ff & \text{otherwise}\end{cases} \\
\overset{(*)}{=}\ & \begin{cases}tt & \text{if } \mathcal{A},\nu\vDash \bigvee_{\substack{1\le i\le n\\ 1\le j\le m}}(a_i=b_j\wedge\phi_i\wedge\theta_j)\\ ff & \text{otherwise}\end{cases} \\
\overset{(D19)}{=}\ & [\![\bigvee_{\substack{1\le i\le n\\ 1\le j\le m}}(a_i=b_j\wedge\phi_i\wedge\theta_j)]\!]^{\rho}_{\nu}
\end{aligned}
$$

First, we use the definition of denotation for the application (D3), variant (D21), and constant $\texttt{eq}_{\mathbb{A}}$ (D16). Then, in step $(*)$, multiple conditions are combined into one. Finally, using the definition of denotation for formulas (D19), we obtain the expected result.

➤ Rule R25

For ordered atoms, for the constant $\texttt{leq}_{\mathbb{A}}$ analogical reasoning can be made as for $\texttt{eq}_{\mathbb{A}}$, with definition (D17) used and the equality symbols in comparing atoms replaced with $\leq$.

$\square$

CHAPTER 6

Church-Rosser Property

In this chapter, we will prove another important property of semantics, namely the Church-Rosser property. This property is important for several reasons. First of all, it shows that the order in which the reduction is performed does not affect the ultimate result. Secondly, it helps to describe the term equality generated by the reduction relation (the reflexive, symmetric, transitive closure of this relation). Thanks to this property, we can show that two terms are equal if they can be reduced to the same term. Thirdly, it implies the uniqueness of the normal form (more on this in Chapter 7). Finally, it ensures that the λ-theory is consistent (it does not equate all terms).

In the case of $N\lambda$ semantics, we consider terms to be identical up to first-order formula equivalence. Strictly speaking, we are studying the set of terms quotiented by the equivalence relation defined as follows.

Definition 6.1. *Two terms M and N are equal in context ψ (written $M \sim_\psi N$) in the following cases:*

$$\begin{aligned}
x &\sim_\psi x \\
\lambda x.M &\sim_\psi \lambda x.N && \textit{if } M \sim_\psi N \\
M_1 M_2 &\sim_\psi N_1 N_2 && \textit{if } M_1 \sim_\psi N_1 \textit{ and } M_2 \sim_\psi N_2 \\
C &\sim_\psi C \\
a &\sim_\psi a \\
\phi &\sim_\psi \theta && \textit{if } \mathcal{A}, \psi \vDash \phi \Leftrightarrow \theta
\end{aligned}$$

$$\{M_1 \colon \phi_1 \textit{ for } \sigma_1, \dots, M_n \colon \phi_n \textit{ for } \sigma_n\} \sim_\psi \{N_1 \colon \theta_1 \textit{ for } \sigma_1, \dots, N_n \colon \theta_n \textit{ for } \sigma_n\}$$
$$\textit{if } \phi_i \sim_\psi \theta_i \textit{ and } M_i \sim_{\psi \wedge \phi_i} N_i \textit{ for } i = 1, \dots, n$$

$$a_1 \colon \phi_1 | \dots | a_n \colon \phi_n \sim_\psi a_1 \colon \theta_1 | \dots | a_n \colon \theta_n$$
$$\textit{if } \phi_i \sim_\psi \theta_i \textit{ for } i = 1, \dots, n$$

In other words, the terms M and N have exactly the same structure but differ only in formulas that are equivalent in the given context. This context is passed down in the recursive definition, except for sets where a formula that conditions a subterm is added to the corresponding context. If the context is a tautology, we just write $M \sim N$.

Recall that terms are already considered up to α-equivalence.[1] Therefore, $\lambda x.x \sim \lambda y.y$ even though $x \not\sim y$.

[1]See Variables subsection in Section 3.4

For example, in the settings of ordered atoms, the following two terms are equal in context $b < c$:

$$\lambda x.\{b \leq a \colon a = c \text{ for } a\} \quad \sim_{b<c} \quad \lambda x.\{\top \colon c = a \text{ for } a\}$$

A simple lemma about substitution can be proved for this relation:

Lemma 6.2. *If $M_1 \sim_\psi M_2$ and $N_1 \sim_\psi N_2$ then $M_1[N_1/x] \sim_\psi M_2[N_2/x]$.*

Proof. By induction on the structure of M_1 (equivalently, M_2). Note that for any formula ϕ if $M \sim_\psi N$ then $M \sim_{\psi\wedge\phi} N$. The substitution above is a replacement of subterms x which are in relation ($x \sim_\psi x$) with terms that are also in relation ($N_1 \sim_\psi N_2$). In the case of the subterms of the set expression, these relations also hold within a narrow context ($\psi \wedge \phi$). □

The definition that we will need to describe the Church-Rosser property is:

Definition 6.3. *The reduction relation $\rightarrow$ satisfies the **diamond property** if for $\psi \vdash M \rightarrow M'$ and $\psi \vdash M \rightarrow M''$ exist N' and N'' such that $\psi \vdash M' \rightarrow N'$, $\psi \vdash M'' \rightarrow N''$ and $N' \sim_\psi N''$.*

We can now go to the main definition.

Definition 6.4. *The reduction relation $\rightarrow$ is **Church-Rosser** if $\rightarrow^*$ satisfies the diamond property, where $\rightarrow^*$ is the reflexive and transitive closure of $\rightarrow$.*

We will show that the reduction relation from Chapter 3 is Chuch-Rosser. For classical λ-calculus this was proved in 1936 by Alonzo Church and J. Barkley Rosser, after whom it is named [Church and Rosser, 1936].

There are two known proof techniques of the theorem: tracing the residuals of redexes along a sequence of reductions [Church and Rosser, 1936, Barendregt, 1984, Hindley and Seldin, 2008], and working with a parallel reduction [Curry et al., 1974, Barendregt, 1984, Hindley and Seldin, 2008, Takahashi, 1995]). The proof presented below will be based on the latter method by Martin-Löf and Tait.

6.1. Parallel reduction

The basic idea behind parallel reduction is that its rules allow subterms during the main reduction also to be reduced. This allows simultaneous subterm reductions in one step.

Definition 6.5. *The binary relation over terms called **parallel reduction**, written $\rightarrow_\parallel$, is defined by the following rules.*

β-reduction:

$$\frac{}{\psi \vdash M \rightarrow_\parallel M} \tag{P1}$$

$$\frac{\psi \vdash M \rightarrow_\parallel M'}{\psi \vdash \lambda x.M \rightarrow_\parallel \lambda x.M'} \tag{P2}$$

$$\frac{\psi \vdash M \rightarrow_{\|} M' \qquad \psi \vdash N \rightarrow_{\|} N'}{\psi \vdash MN \rightarrow_{\|} M'N'} \tag{P3}$$

$$\frac{\psi \vdash M \rightarrow_{\|} M' \qquad \psi \vdash N \rightarrow_{\|} N'}{\psi \vdash (\lambda x.M)N \rightarrow_{\|} M'[N'/x]} \tag{P4}$$

Basic constants:

Rules for `empty` and `atoms` are exactly the same as for $\rightarrow$ (R5, R6).

$$\frac{\psi \vdash M \rightarrow_{\|} M' \quad \psi \wedge \phi_1 \vdash M_1 \rightarrow_{\|} M_1' \quad \cdots \quad \psi \wedge \phi_n \vdash M_n \rightarrow_{\|} M_n'}{\begin{array}{l}\psi \vdash \texttt{insert}\ M\ \{M_1 \colon \phi_1 \text{ for } \sigma_1, \ldots, M_n \colon \phi_n \text{ for } \sigma_n\} \\ \quad \rightarrow_{\|} \{M', M_1' \colon \phi_1 \text{ for } \sigma_1, \ldots, M_n' \colon \phi_n \text{ for } \sigma_n\}\end{array}} \tag{P5}$$

$$\frac{\begin{array}{c}\psi \wedge \phi_1 \vdash M \rightarrow_{\|} M_1' \quad \cdots \quad \psi \wedge \phi_n \vdash M \rightarrow_{\|} M_n' \\ \psi \wedge \phi_1 \vdash M_1 \rightarrow_{\|} M_1'' \quad \cdots \quad \psi \wedge \phi_n \vdash M_n \rightarrow_{\|} M_n''\end{array}}{\begin{array}{l}\psi \vdash \texttt{map}\ M\ \{M_1 \colon \phi_1 \text{ for } \sigma_1, \ldots, M_n \colon \phi_n \text{ for } \sigma_n\} \\ \quad \rightarrow_{\|} \{M_1'M_1'' \colon \phi_1 \text{ for } \sigma_1, \ldots, M_n'M_n'' \colon \phi_n \text{ for } \sigma_n\}\end{array}} \tag{P6}$$

$$\text{if} \quad \bigcup_{i=1}^{n} \mathrm{V_A}(M_i') \cap \bigcup_{i=1}^{n} \sigma_i = \emptyset$$

$$\frac{\cdots \quad \psi \wedge \phi_1 \wedge \phi \vdash M_1 \rightarrow_{\|} M_1' \quad \cdots \quad \psi \wedge \phi_n \wedge \phi \vdash M_n \rightarrow_{\|} M_n' \quad \cdots}{\begin{array}{l}\psi \vdash \texttt{sum}\ \{\ldots, \{M_1 \colon \phi_1 \text{ for } \sigma_1, \ldots, M_n \colon \phi_n \text{ for } \sigma_n\} \colon \phi \text{ for } \sigma, \ldots\} \\ \quad \rightarrow_{\|} \{\ldots, M_1' \colon \phi_1 \wedge \phi \text{ for } \sigma_1 \cup \sigma, \ldots, \ M_n' \colon \phi_n \wedge \phi \text{ for } \sigma_n \cup \sigma, \ldots\}\end{array}} \tag{P7}$$

$$\text{if} \quad \sigma \cap \bigcup_{i=1}^{n} \sigma_i = \emptyset$$

Formula constructors:

Rules for constants `true`, `false`, `not`, `or`, `and`, `isEmpty` are exactly the same as for $\rightarrow$ (R10, R11, R12, R13).

Conditional expressions:

Rules for conditional expressions with atoms, formulas and variants are exactly the same as for $\rightarrow$ (R17, R18, R20).

$$\frac{\mathcal{A} \vDash \psi \Rightarrow \phi \qquad \psi \vdash M \rightarrow_{\|} M'}{\psi \vdash \texttt{if}\ \phi\ M\ N \rightarrow_{\|} M'} \tag{P8}$$

$$\frac{\mathcal{A} \vDash \psi \Rightarrow \neg\phi \qquad \psi \vdash N \rightarrow_{\|} N'}{\psi \vdash \texttt{if}\ \phi\ M\ N \rightarrow_{\|} N'} \tag{P9}$$

$$\frac{\mathcal{A} \nvDash \psi \Rightarrow \phi \quad \mathcal{A} \nvDash \psi \Rightarrow \neg\phi \qquad \psi \vdash M_1 \to_{\parallel} M_1' \quad \psi \vdash M_2 \to_{\parallel} M_2' \quad \psi \vdash N \to_{\parallel} N'}{\psi \vdash \texttt{if}\ \phi\ M_1\ M_2\ N \ \to_{\parallel}\ \texttt{if}\ \phi\ (M_1'N')\ (M_2'N')} \tag{P10}$$

$$\frac{\mathcal{A} \nvDash \psi \Rightarrow \phi \quad \mathcal{A} \nvDash \psi \Rightarrow \neg\phi \quad \begin{array}{ccc} \psi \wedge \phi_1 \wedge \phi \vdash M_1 \to_{\parallel} M_1' & \cdots & \psi \wedge \phi_n \wedge \phi \vdash M_n' \to_{\parallel} M_n' \\ \psi \wedge \theta_1 \wedge \neg\phi \vdash N_1 \to_{\parallel} N_1' & \cdots & \psi \wedge \theta_k \wedge \neg\phi \vdash N_k \to_{\parallel} N_k' \end{array}}{\begin{array}{l} \psi \vdash \texttt{if}\ \phi\ \{M_1 \colon \phi_1 \text{ for } \sigma_1, \ldots, M_n \colon \phi_n \text{ for } \sigma_n\}\ \{N_1 \colon \theta_1 \text{ for } \pi_1, \ldots, N_k \colon \theta_k \text{ for } \pi_k\} \\ \quad \to_{\parallel} \{M_1' \colon \phi_1 \wedge \phi \text{ for } \sigma_1, \ldots, M_n' \colon \phi_n \wedge \phi \text{ for } \sigma_n, \\ \qquad N_1' \colon \theta_1 \wedge \neg\phi \text{ for } \pi_1, \ldots, N_k' \colon \theta_k \wedge \neg\phi \text{ for } \pi_k\} \end{array}} \tag{P11}$$

Set reduction:

$$\frac{\cdots \quad \psi \wedge \phi_1 \vdash M_1 \to_{\parallel} M_1' \quad \cdots \quad \mathcal{A} \vDash \psi \Rightarrow \neg\phi_2 \quad \cdots}{\begin{array}{c} \psi \vdash \{\ldots, M_1 \colon \phi_1 \text{ for } \sigma_1, \ldots, M_2 \colon \phi_2 \text{ for } \sigma_2, \ldots\} \\ \to_{\parallel} \{\ldots, M_1' \colon \phi_1 \text{ for } \sigma_1, \ldots, \ldots\} \end{array}} \tag{P12}$$

Rule (P12) is a combination of Rules (R21) and (R22). For each subexpression, the subterm is reduced, and if the condition is unsatisfiable then the subexpression can be removed from the set. It is worth emphasizing that subexpressions with false conditions may or may not be removed, the rule removes a subset of such subexpressions. Therefore, we are dealing with a nondeterminism in which a reduction does not directly follow from a premise. This construction results from the requirement that each reduction rule $\to$ (in this case Rules (R21) and (R22)) can be described with some parallel rule $\to_{\parallel}$ (in this case (P12)). This property will be used in the proof of Lemma 6.12.

Variant reduction:

The rule for variant reduction is exactly the same as for $\to$ (R23).

Atom relations:

Rules for atom relations are exactly the same as for $\to$ (R24, R25).

6.2. A substitution lemma for parallel reduction

After defining the concept of parallel reduction, let us look at its properties related to substitution. We will begin with recalling a standard lemma about nested substitution:

Lemma 6.6 (Nested substitution)**.** *For any variable x distinct from y, any terms M, N and term Q not containing x as a free variable,*

$$M[N/x][Q/y] = M[Q/y][N[Q/y]/x]$$

Proof. Straightforward induction on the structure of M. □

And now we look at the lemma that composes parallel reduction with substitution.

Lemma 6.7 (Substitution for parallel reduction)**.** *If* $\psi \vdash M \to_{\parallel} M'$ *and* $\psi \vdash N \to_{\parallel} N'$ *then* $\psi \vdash M[N/x] \to_{\parallel} M'[N'/x]$*.*

Proof. The proof is by induction on the structure of M for all M', N, N' and contexts ψ. Suppose that the result holds for all subterms of M and that $\psi \vdash M \to_{\parallel} M'$. We perform a case analysis on the rule showing this reduction.

➤ Rule P1 (reflexivity)

Then $M = M'$ and analyze how $M[N/x]$ and $M'[N'/x] = M[N'/x]$ look depending on the form of the term M:

M	$M[N/x]$	$M'[N'/x]$
x	N	N'
y	y	y
$\lambda y.M_1$	$\lambda y.M_1[N/x]$	$\lambda y.M_1[N'/x]$
M_1M_2	$M_1[N/x]M_2[N/x]$	$M_1[N'/x]M_2[N'/x]$
C	C	C
a	a	a
ϕ	ϕ	ϕ
$\{\ldots, M_i\colon \phi_i \text{ for } \sigma_i, \ldots\}$	$\{\ldots, M_i[N/x]\colon \phi_i \text{ for } \sigma_i, \ldots\}$	$\{\ldots, M_i[N'/x]\colon \phi_i \text{ for } \sigma_i, \ldots\}$
$a_1\colon \phi_1 \vert \cdots \vert a_n\colon \phi_n$	$a_1\colon \phi_1 \vert \cdots \vert a_n\colon \phi_n$	$a_1\colon \phi_1 \vert \cdots \vert a_n\colon \phi_n$

From the table, it can be seen that for some cases the thesis is trivial, and for others, it immediately follows from the induction hypothesis for M_i, where $i = 1, \ldots, n$.

➤ Rule P2 (abstraction)

In this case, we have $M = \lambda y.M_1$ and $M' = \lambda y.M_2$ and we know from Rule (P2) that $\psi \vdash M_1 \to_{\parallel} M_2$ and $\psi \vdash M_1[N/x] \to_{\parallel} M_2[N'/x]$ from inductive hypothesis. Hence in context ψ:

$$M[N/x] \;=\; \lambda y.M_1[N/x] \;\to_{\parallel}\; \lambda y.M_2[N'/x] \;=\; M'[N'/x]$$

➤ Rule P3 (application)

Let us assume that $M = M_1M_2$ and $M' = M_1'M_2'$. From Rule (P3) we have $\psi \vdash M_1 \to_{\parallel} M_1'$ and $\psi \vdash M_2 \to_{\parallel} M_2'$. By applying the induction hypothesis to these terms, we obtain

$$\psi \vdash M_1[N/x] \to_{\parallel} M_1'[N'/x] \quad \text{and} \quad \psi \vdash M_2[N/x] \to_{\parallel} M_2'[N'/x].$$

And using Rule (P3) again it follows that in context ψ:

$$M[N/x] = M_1[N/x]M_2[N/x] \to_{\parallel} M_1'[N'/x]M_2'[N'/x] = M'[N'/x]$$

- Rule P4 (substitution)

  To apply Rule (P4), we must have $M = (\lambda y.M_1)N_1$ and $M' = M_2[N_2/y]$ for $\psi \vdash M_1 \to_{\parallel} M_2$ and $\psi \vdash N_1 \to_{\parallel} N_2$. The induction hypothesis applied to M_1 and N_1 leads to

  $$\psi \vdash M_1[N/x] \to_{\parallel} M_2[N'/x] \quad \text{and} \quad \psi \vdash N_1[N/x] \to_{\parallel} N_2[N'/x].$$

  Using the above, we can make the following reasoning:

  $$\begin{aligned} M[N/x] &= ((\lambda y.M_1)N_1)[N/x] \\ &= (\lambda y.M_1[N/x])(N_1[N/x]) && \text{by definition of substitution} \\ &\to_{\parallel} M_2[N'/x][N_2[N'/x]/y] && \text{by Rule (P4) in context } \psi \\ &= M_2[N_2/y][N'/x] && \text{by Lemma 6.6} \\ &= M'[N'/x] \end{aligned}$$

  Note that the assumption of Lemma 6.6 is satisfied: $y \notin \mathrm{FV_T}(N')$ because we can assume that y is different from all free term variables.[2]

- `empty` and `atoms`

  These constants have no term variables, so the result comes directly from the assumptions of the lemma.

- Rule P5 (`insert`)

  Here we have

  $$\begin{aligned} M &= \texttt{insert}\ M_0\ \{M_1 \colon \phi_1 \text{ for } \sigma_1, \ldots, M_n \colon \phi_n \text{ for } \sigma_n\} \\ M' &= \{M'_0, M'_1 \colon \phi_1 \text{ for } \sigma_1, \ldots, M'_n \colon \phi_n \text{ for } \sigma_n\} \end{aligned}$$

  where $\psi \vdash M_0 \to_{\parallel} M'_0$ and $\psi \wedge \phi_i \vdash M_i \to_{\parallel} M'_i$ for $i = 1, \ldots, n$.

  By inductive hypothesis $\psi \vdash M_0[N/x] \to_{\parallel} M'_0[N'/x]$ and $\psi \wedge \phi_i \vdash M_i[N/x] \to_{\parallel} M'_i[N'/x]$ for $i = 1, \ldots, n$. Hence in context ψ:

  $$\begin{aligned} M[N/x] &= \texttt{insert}\ M_0[N/x]\ \{M_1[N/x] \colon \phi_1 \text{ for } \sigma_1, \ldots, M_n[N/x] \colon \phi_n \text{ for } \sigma_n\} \\ &\to_{\parallel} \{M'_0[N'/x], M'_1[N'/x] \colon \phi_1 \text{ for } \sigma_1, \ldots, M'_n[N'/x] \colon \phi_n \text{ for } \sigma_n\} \\ &= M'[N'/x] \end{aligned}$$

- Rules P6 (`map`) and P7 (`sum`)

  They can be proved in a similar way as for Rule (P5) using the inductive hypothesis for elements of the set.

- `true`, `false`, `not`, `or`, `and`

  Obvious because rules for these constants do not contain term variables neither in the redex nor in the reduct.

- `isEmpty`

  The reduction rule for this constant has no term variables in the reduct. Substitution on the redex side does not affect the reduction rule.

[2]See Variables subsection in Section 3.4.

- Conditional expressions with atoms, formulas and variants

  As above, obvious due to the lack of term variables.

- Rules P8, P9, P10, P11 (other conditional expressions)

  For all these rules, the proof looks similar, so we will present it only for Rule (P10).

  Let $M = \texttt{if}\ \phi\ M_1\ M_2\ M_3$ and $M' = \texttt{if}\ \phi\ (M'_1 M'_3)\ (M'_2 M'_3)$, where $\psi \vdash M_i \rightarrow_{\parallel} M'_i$ for $i = 1, 2, 3$. Therefore in context ψ:

$$\begin{aligned} M[N/x] &= (\texttt{if}\ \phi\ M_1\ M_2\ M_3)[N/x] \\ &= \texttt{if}\ \phi\ M_1[N/x]\ M_2[N/x]\ M_3[N/x] \\ &\rightarrow_{\parallel} \texttt{if}\ \phi\ (M'_1[N'/x] M'_3[N'/x])\ (M'_2[N'/x] M'_3[N'/x]) \\ &= (\texttt{if}\ \phi\ (M'_1 M'_3)\ (M'_2 M'_3))[N'/x] \\ &= M'[N'/x] \end{aligned}$$

  by Rule (P10) and the inductive hypothesis applied to M_1, M_2 and M_3.

- Rule P12 (set reduction)

  The proof is similar to earlier rules for set expressions. The inductive hypothesis is applied to the elements of the set. If some subexpressions are removed by $\psi \vdash M \rightarrow_{\parallel} M'$, then $\psi \vdash M[N/x] \rightarrow_{\parallel} M'[N'/x]$ removes the same subexpressions.

- Variant reduction

  Obvious, because a variant has no term variables.

- Atoms relations

  Obvious, because these rules contain no term variables.

□

6.3. Parallel reduction in restricted context

We will show now how parallel reduction can be transferred to a more restrictive context. For a given parallel reduction in context ψ, we prove that both redex and reduct terms can be reduced to the same term (up to the relation $\sim_{\psi'}$) in any restricted context ψ'.

Lemma 6.8. *Let $\psi \vdash M \rightarrow_{\parallel} M'$. For every formula ψ' such that $\psi' \Rightarrow \psi$ there are reductions $\psi' \vdash M \rightarrow_{\parallel} N$ and $\psi' \vdash M' \rightarrow_{\parallel} N'$ such that $N \sim_{\psi'} N'$.*

Proof. By induction on the structure of the term M and by case analysis on the rule showing $\psi \vdash M \rightarrow_{\parallel} M'$.

- Rules for which context does not matter

  The context has no influence on rules P1 (reflexivity), for `empty` and `atoms` constants, formula constructors and atom relations. Therefore, we can assume that $N = N' = M'$, and the reduction $\psi' \vdash M \rightarrow_{\parallel} N$ is the same as the reduction $\psi \vdash M \rightarrow_{\parallel} M'$.

➤ Rules P2 (abstraction), P3 (application) and P4 (substitution)

All these rules can be proved in a similar way by the inductive hypothesis for the subterms. The most interesting case is for Rule (P4), therefore it will be presented here.

Let $M = (\lambda x.M_1)M_2$ and $M' = M_1'[M_2'/x]$, where $\psi \vdash M_1 \rightarrow_{\parallel} M_1'$ and $\psi \vdash M_2 \rightarrow_{\parallel} M_2'$.

By inductive hypothesis for some $N_1 \sim_{\psi'} N_1'$ and $N_2 \sim_{\psi'} N_2'$ we have reductions:

$$\begin{aligned} &\psi' \vdash M_1 \rightarrow_{\parallel} N_1 \qquad && \psi' \vdash M_1' \rightarrow_{\parallel} N_1' \\ &\psi' \vdash M_2 \rightarrow_{\parallel} N_2 && \psi' \vdash M_2' \rightarrow_{\parallel} N_2' \end{aligned}$$

As a result, we get the following diagram:

$$\begin{tikzcd}
(\lambda x.M_1)M_2 \arrow[rr, "\psi", "\parallel"'] \arrow[d, "\psi'", "\text{by Rule (P4)}"'] & & M_1'[M_2'/x] \arrow[d, "\psi'", "\text{by Lemma 6.7}"'] \\
N_1[N_2/x] \arrow[rr, phantom, "\sim_{\psi'}", "\text{by Lemma 6.2}"'] & & N_1'[N_2'/x]
\end{tikzcd}$$

➤ Rule P5 (`insert`)

This case is also proved by induction, but for different contexts. We begin with

$$\begin{aligned} M &= \texttt{insert}\ M_0\ \{M_1 \colon \phi_1 \text{ for } \sigma_1, \dots, M_n \colon \phi_n \text{ for } \sigma_n\} \\ M' &= \{M_0', M_1' \colon \phi_1 \text{ for } \sigma_1, \dots, M_n' \colon \phi_n \text{ for } \sigma_n\} \end{aligned}$$

From the premises of Rule (P5) and the inductive hypothesis, we have:

$$\begin{tikzcd}
M_0 \arrow[r, "\psi", "\parallel"'] \arrow[d, "\psi'"] & M_0' \arrow[d, "\psi'"] & M_i \arrow[r, "\psi\wedge\phi_i", "\parallel"'] \arrow[d, "\psi'\wedge\phi_i"] & M_i' \arrow[d, "\psi'\wedge\phi_i"] \\
N_0 \arrow[r, phantom, "\sim_{\psi'}"] & N_0' & N_i \arrow[r, phantom, "\sim_{\psi'\wedge\phi_i}"] & N_i'
\end{tikzcd}$$

for some $N_0, N_1, \dots, N_n$ and $N_0', N_1', \dots, N_n'$ where $i = 1, \dots, n$. This is enough to show that:

$$\begin{tikzcd}
M \arrow[r, "\psi", "\parallel"'] \arrow[d, "\psi'", "\text{by Rule (P5)}"'] & M' \arrow[d, "\psi'", "\text{by Rule (P12)}"'] \\
N \arrow[r, phantom, "\sim_{\psi'}"] & N'
\end{tikzcd}$$

where

$$\begin{aligned} N &= \{N_0, N_1 \colon \phi_1 \text{ for } \sigma_1, \dots, N_n \colon \phi_n \text{ for } \sigma_n\} \\ N' &= \{N_0', N_1' \colon \phi_1 \text{ for } \sigma_1, \dots, N_n' \colon \phi_n \text{ for } \sigma_n\} \end{aligned}$$

➤ Rules P6 (`map`) and P7 (`sum`)

Analogous to the case of Rule (P5).

➤ Rule P8

Here we have $M = \texttt{if}\ \phi\ M_1\ M_2$ and $M' = M_1'$ where $\psi \vdash M_1 \to_{\parallel} M_1'$. Additionally, the premise of the rule guarantees that $\mathcal{A} \vDash \psi \Rightarrow \phi$. This means that also for context ψ' it holds that $\mathcal{A} \vDash \psi' \Rightarrow \phi$. As a result, Rule (P8) can also be applied to the term M in context ψ'. Hence:

$$\begin{tikzcd}
\texttt{if}\ \phi\ M_1\ M_2 \arrow[r, "\psi", "\parallel"'] \arrow[d, "\psi'", "\text{by Rule (P8)}\ ="'] & M_1' \arrow[d, "\psi'", "="'] \\
N_1 \arrow[r, phantom, "\sim_{\psi'}"] & N_1'
\end{tikzcd}$$

where the reductions $\psi' \vdash M_1 \to_{\parallel} N_1$ and $\psi' \vdash M_1' \to_{\parallel} N_1'$ are thanks to the inductive hypothesis.

➤ Rule P9

As for Rule (P8).

➤ Rules for conditional expresions

All the rules for conditional expressions are proved in a similar way. Therefore, we will only present here the proof for Rule (P10) where $M = (\texttt{if}\ \phi\ M_1\ M_2)\ M_3$ and $M' = \texttt{if}\ \phi\ (M_1'M_3')\ (M_2'M_3')$. From the inductive hypothesis we have:

$$\begin{tikzcd}
M_1 \arrow[r, "\psi", "\parallel"'] \arrow[d, "\psi'", "="'] & M_1' \arrow[d, "\psi'", "="'] & M_2 \arrow[r, "\psi", "\parallel"'] \arrow[d, "\psi'", "="'] & M_2' \arrow[d, "\psi'", "="'] & M_3 \arrow[r, "\psi", "\parallel"'] \arrow[d, "\psi'", "="'] & M_3' \arrow[d, "\psi'", "="'] \\
N_1 \arrow[r, phantom, "\sim_{\psi'}"] & N_1' & N_2 \arrow[r, phantom, "\sim_{\psi'}"] & N_2' & N_3 \arrow[r, phantom, "\sim_{\psi'}"] & N_3'
\end{tikzcd}$$

for some terms $N_1, N_2, N_3, N_1', N_2', N_3'$. Let us consider three cases for context ψ' and for each of them, using the above, we prove the correctness of the following diagrams.

➢ $\mathcal{A} \vDash \psi' \Rightarrow \phi$

$$\begin{tikzcd}
(\texttt{if}\ \phi\ M_1\ M_2)\ M_3 \arrow[r, "\psi", "\parallel"'] \arrow[d, "\psi'", "\text{by Rule (P3) and (P8) for subterm}\ ="'] & \texttt{if}\ \phi\ (M_1'M_3')\ (M_1'M_3') \arrow[d, "\psi'\ \text{by Rule (P8)}", "="'] \\
N_1N_3 \arrow[r, phantom, "\sim_{\psi'}"] & N_1'N_3'
\end{tikzcd}$$

➢ $\mathcal{A} \vDash \psi' \Rightarrow \neg\phi$

$$\begin{tikzcd}
(\texttt{if}\ \phi\ M_1\ M_2)\ M_3 \arrow[r, "\psi", "\parallel"'] \arrow[d, "\psi'", "\text{by Rule (P3) and (P9) for subterm}\ ="'] & \texttt{if}\ \phi\ (M_1'M_3')\ (M_1'M_3') \arrow[d, "\psi'\ \text{by Rule (P9)}", "="'] \\
N_2N_3 \arrow[r, phantom, "\sim_{\psi'}"] & N_2'N_3'
\end{tikzcd}$$

- $\mathcal{A} \not\models \psi' \Rightarrow \phi$ and $\mathcal{A} \not\models \psi' \Rightarrow \neg\phi$

$$\begin{tikzcd}
(\texttt{if}\ \phi\ M_1\ M_2)\ M_3 \arrow[r, "\psi", "\parallel"'] \arrow[d, "\psi'", "\text{by Rule (P10)}"'] & \texttt{if}\ \phi\ (M_1'M_3')\ (M_1'M_3') \arrow[d, "\psi'"', "\text{by Rule (P3)}"] \\
\texttt{if}\ \phi\ (N_1N_3)\ (N_1N_3) \arrow[r, phantom, "\sim_{\psi'}"] & \texttt{if}\ \phi\ (N_1'N_3')\ (N_1'N_3')
\end{tikzcd}$$

- Rule P12 (set reduction)

  In this case $M = \{M_1 \colon \phi_1 \text{ for } \sigma_1, \ldots M_n \colon \phi_n \text{ for } \sigma_n\}$. The reduction $\psi \vdash M \to_{\parallel} M'$ reduces some subterms M_i for $1 \leq i \leq n$. Additionally, it can remove some subterms M_j for $1 \leq j \leq n$ if $\mathcal{A} \models \psi \Rightarrow \neg\phi_j$. One can construct reductions $\psi' \vdash M \to_{\parallel} N$ and $\psi' \vdash M' \to_{\parallel} N'$, using Rule (P12), that satisfy the thesis. Subterms are reduced as in the inductive hypothesis. If certain subterms are removed in context ψ, they are also removed in context ψ' (if $\mathcal{A} \models \psi \Rightarrow \neg\phi_j$, then also $\mathcal{A} \models \psi' \Rightarrow \neg\phi_j$ for $1 \leq j \leq n$).

- Variant reduction

  The proof is a simplified version of that for Rule (P12).

□

6.4. Diamond property of parallel reduction

We want to show that parallel reduction satisfies the diamond property. For this purpose, we will prove the lemma with a more general assumption for contexts. It is necessary to use induction in the proof of the lemma.

Lemma 6.9. *For $\psi_1 \vdash M \to_{\parallel} M'$ and $\psi_2 \vdash M \to_{\parallel} M''$, there exist N' and N'' such that $\psi_1 \wedge \psi_2 \vdash M' \to_{\parallel} N'$, $\psi_1 \wedge \psi_2 \vdash M'' \to_{\parallel} N''$ and $N' \sim_{\psi_1 \wedge \psi_2} N''$.*

Proof. By induction on the structure of M and by case analysis on the rules showing $\psi_1 \vdash M \to_{\parallel} M'$ and $\psi_2 \vdash M \to_{\parallel} M''$.

- Both rules are the same

  If both reductions of M use the same rule from Definition 6.5, then M' and M'' have the same form (term variable, abstraction, application, atom, formula, constant, set or variant) and differ at most in subterms.

  We apply the inductive hypothesis to these subterms and use subterms reduction rules:

  - Rule (P2) for abstraction,
  - Rule (P3) for application,
  - Rule (P12) for set

  we achieve the required outcome. For Rule (P4) additionally Lemma 6.7 is needed.

  The only subtlety is when the set/variant is reduced by Rule (P12)/(R23). In these cases, we obtain two terms where some subexpressions may be deleted in one term and not in the other. But by reapplying the same rule, any subexpression removed in one term can be removed in the other term as well.

When terms M' and M'' are equal, we apply Rule (P1).

- One of the rules does not change the term M

  A rule does not change a term if it is Rule (P1) or if it delegates a reduction to subterms that are not changed.

  Suppose $M = M'$. Then for reduction $\psi_2 \vdash M \to_{\|} M''$ we use Lemma 6.8, which ends the proof of this case.

For the rest of the proof, we assume that both reductions use different rules and that both reductions change the term M. For the following cases, we will analyze situations in which one of the reductions will use a specific rule. Let us assume without loss of generality that this rule reduces to M'.

- One of the rules is P2 (abstraction)

  In this case, M is an abstraction that cannot be reduced by two different rules.

- One of the rules is P4 (substitution)

  So $M = (\lambda x.M_1)M_2$, $M' = M_1'[M_2'/x]$ with $\psi_1 \vdash M_1 \to_{\|} M_1'$ and $\psi_1 \vdash M_2 \to_{\|} M_2'$. The second rule can only be (P3). Then $M'' = (\lambda x.M_1'')M_2''$ with $\psi_2 \vdash M_1 \to_{\|} M_1''$ and $\psi_2 \vdash M_2 \to_{\|} M_2''$.

  By applying the induction hypothesis to M_1 and M_2, there are terms $N_1' \sim_{\psi_1 \wedge \psi_2} N_1''$ and $N_2' \sim_{\psi_1 \wedge \psi_2} N_2''$ with

$$\begin{array}{ll} \psi_1 \wedge \psi_2 \vdash M_1' \to_{\|} N_1' & \qquad \psi_1 \wedge \psi_2 \vdash M_1'' \to_{\|} N_1'' \\ \psi_1 \wedge \psi_2 \vdash M_2' \to_{\|} N_2' & \qquad \psi_1 \wedge \psi_2 \vdash M_2'' \to_{\|} N_2'' \end{array}$$

  Take $N' = N_1'[N_2'/x]$ and $N'' = N_1''[N_2''/x]$. Then $N' \sim_{\psi_1 \wedge \psi_2} N''$ by Lemma 6.2, $\psi_1 \wedge \psi_2 \vdash M' \to_{\|} N'$ by Substitution Lemma 6.7 and $\psi_1 \wedge \psi_2 \vdash M'' \to_{\|} N''$ by Rule (P4).

- M is `empty` or `atoms`

  These constants cannot be reduced by different rules.

- One of the rules is P5 (`insert`)

  Then M has a form: `insert` M_0 $\{M_1\colon \phi_1$ for $\sigma_1, \ldots, M_n\colon \phi_n$ for $\sigma_n\}$. After reduction, it takes the form $M' = \{M_0', M_1'\colon \phi_1$ for $\sigma_1, \ldots, M_n'\colon \phi_n$ for $\sigma_n\}$, where $\psi_1 \vdash M_0 \to_{\|} M_0'$ and $\psi_1 \wedge \phi_i \vdash M_i \to_{\|} M_i'$ for $i = 1, \ldots, n$.

  The second reduction rule can be only the application reducing rule (P3). Thus applying this rule twice and reducing the set by Rule (P12), we get

$$M'' = \texttt{insert}\ M_0''\ \{M_1''\colon \phi_1 \text{ for } \sigma_1, \ldots, M_n''\colon \phi_n \text{ for } \sigma_n\},$$

  where $\psi_2 \vdash M_0 \to_{\|} M_0''$ and $\psi_2 \wedge \phi_i \vdash M_i \to_{\|} M_i''$ for $i = 1, \ldots, n$. Additionally, some set subexpressions with unsatisfiable conditions could be removed by reduction $\psi \vdash M \to_{\|} M''$.

By inductive hypothesis, there exist $N'_0 \sim_{\psi_1 \wedge \psi_2} N''_0$ and $N'_i \sim_{\psi_1 \wedge \psi_2 \wedge \phi_i} N''_i$ for $i = 1, \ldots, n$ such that

$$\begin{array}{rr} \psi_1 \wedge \psi_2 \vdash M'_0 \to_{\parallel} N'_0 & \qquad \psi_1 \wedge \psi_2 \vdash M''_0 \to_{\parallel} N''_0 \\ \psi_1 \wedge \psi_2 \wedge \phi_i \vdash M'_i \to_{\parallel} N'_i & \qquad \psi_1 \wedge \psi_2 \wedge \phi_i \vdash M''_i \to_{\parallel} N''_i \end{array}$$

Using these reductions we choose

$$N' = \{N'_0, N'_1 \colon \phi_1 \wedge \phi \text{ for } \sigma, \ldots, N'_n \colon \phi_n \wedge \phi \text{ for } \sigma \ \ldots\}$$
$$N'' = \{N''_0, N''_1 \colon \phi_1 \wedge \phi \text{ for } \sigma, \ldots, N''_n \colon \phi_n \wedge \phi \text{ for } \sigma \ \ldots\}$$

in which we remove respectively subexpressions as in M''. Then $N' \sim_{\psi_1 \wedge \psi_2} N''$ by Definition 6.1.

It remains to show that $\psi_1 \wedge \psi_2 \vdash M' \to_{\parallel} N'$ and $\psi_1 \wedge \psi_2 \vdash M'' \to_{\parallel} N''$. The former results from applying Rule (P12) to the term M', and the latter is derived by applying Rule (P5) to the term M''.

➤ One of the rules is P6 (`map`) or P7 (`sum`)

Similar reasoning as for the rule with the constant `insert`.

➤ `true`, `false`, `not`, `or`, `and`

Terms with these constants (and possibly formulas) can be meaningfully reduced only in one way.

➤ `isEmpty`

In this case,

$$M = \texttt{isEmpty} \ \underbrace{\{M_1 \colon \phi_1 \text{ for } \sigma_1, \ldots, M_n \colon \phi_n \text{ for } \sigma_n\}}_{S}$$

M' is the outcome of parallel equivalent of Rule (R13):

$$M' = \bigwedge_{1 \le i \le n} \forall \sigma_i . \neg \phi_i$$

M'' is the result of Rule (P3). So $M'' = \texttt{isEmpty}\ S''$ for some term S'', which was created by reducing the set S by Rule (P12).

Let us take $N' = M'$ and reduce M'' by Rule (R13). We want to show that the reduct obtained in this way is equivalent to the formula N'.

Let us analyze how the reduction $\psi_1 \wedge \psi_2 \vdash S \to_{\parallel} S''$ could look like. Rule (P12) either does not change formulas in the set or removes formulas that are unsatisfiable in context ψ_2. But in such a case, they are also unsatisfiable in context $\psi_1 \wedge \psi_2$ and their negations are tautologies in this context. If the term N'' does not have these formulas compared to the term N', it will still be equivalent to it in context $\psi_1 \wedge \psi_2$.

➤ One of the rules is the reduction of a conditional expression with an undetermined condition

We assume that the condition in the expression is undetermined in context ψ_1, that is $\mathcal{A} \nvDash \psi_1 \Rightarrow \phi$ and $\mathcal{A} \nvDash \psi_1 \Rightarrow \neg\phi$. So the term M' is a reduct of one of the rules (P10), (P11) or parallel equivalents of rules (R17), (R18) (R20).

Depending on context ψ_2, the second reduction may be due to one of the rules (P8), (P9) or (P3).

The proofs for this kind of expressions look similar, so let us consider here the proof for the most difficult case, which is the reduction of the conditional expression with sets.

The term M has the following form:

$$\begin{aligned} M = \texttt{if}\ \phi\ &\{M_1\colon \phi_1 \text{ for } \sigma_1, \dots, M_n\colon \phi_n \text{ for } \sigma_n\} \\ &\{N_1\colon \theta_1 \text{ for } \pi_1, \dots, N_k\colon \theta_k \text{ for } \pi_k\} \end{aligned}$$

M' is the result of Rule (P11)

$$\begin{aligned} M' = \{&M'_1\colon \phi_1 \wedge \phi \text{ for } \sigma_1, \dots, M'_n\colon \phi_n \wedge \phi \text{ for } \sigma_n, \\ &N'_1\colon \theta_1 \wedge \neg\phi \text{ for } \pi_1, \dots, N'_k\colon \theta_k \wedge \neg\phi \text{ for } \pi_k\} \end{aligned} \tag{6.1}$$

Consider the following cases:

➢ $\mathcal{A} \vDash \psi_2 \Rightarrow \phi$
Then the second reduction is based on Rule (P8) and M'' is a set expression that contains only reducts M''_i of subterms M_i for $i = 1, \dots, n$. The proof uses the inductive hypothesis and the fact that $\mathcal{A} \vDash \psi_1 \wedge \psi_2 \Rightarrow \phi$. The set expressions M' and M'' are reduced in context $\psi_1 \wedge \psi_2$ by Rule (P12). During the reduction of the term M' subterms N'_j for $j = 1, \dots, k$ are removed (because the condition $\theta_j \wedge \neg\phi$ is unsatisfiable in context $\psi_1 \wedge \psi_2$).

➢ $\mathcal{A} \vDash \psi_2 \Rightarrow \neg\phi$
Proof similar to the previous case.

➢ $\mathcal{A} \nvDash \psi_2 \Rightarrow \phi$ and $\mathcal{A} \nvDash \psi_2 \Rightarrow \neg\phi$
In this case, M'' is the outcome of the rules (P3) and (P12):

$$\begin{aligned} M'' = \texttt{if}\ \phi\ &\{M''_1\colon \phi_1 \text{ for } \sigma_1, \dots, M''_n\colon \phi_n \text{ for } \sigma_n\} \\ &\{N''_1\colon \theta_1 \text{ for } \pi_1, \dots, N''_k\colon \theta_k \text{ for } \pi_k\} \end{aligned} \tag{6.2}$$

with the proviso that some subexpressions in the sets may have been removed during the reduction.

We know from the inductive hypothesis that:

$$\begin{tikzcd} & M_i \arrow[dl, "\psi_1\wedge\phi_i\wedge\phi" description] \arrow[dr, "\psi_2\wedge\phi_i" description] & \\ M'_i \arrow[d, "\psi_1\wedge\psi_2\wedge\phi_i\wedge\phi" description] & & M''_i \arrow[d, "\psi_1\wedge\psi_2\wedge\phi_i\wedge\phi" description] \\ P'_i & \sim_{\psi_1\wedge\psi_2\wedge\phi_i\wedge\phi} & P''_i \end{tikzcd}$$

for $i = 1, \ldots, n$ and some terms P'_i and P''_i.

We can draw a similar conclusion for the subterms of the second set expression:

$$\begin{tikzcd}
& N_j \arrow[dl, Rightarrow, "\psi_1\wedge\theta_j\wedge\neg\phi"'] \arrow[dr, Rightarrow, "\psi_2\wedge\theta_j"] & \\
N'_j \arrow[d, Rightarrow, "\psi_1\wedge\psi_2\wedge\theta_j\wedge\neg\phi"'] & & N''_j \arrow[d, Rightarrow, "\psi_1\wedge\psi_2\wedge\theta_j\wedge\neg\phi"'] \\
Q'_j & \sim_{\psi_1\wedge\psi_2\wedge\theta_j\wedge\neg\phi} & Q''_j
\end{tikzcd}$$

for $j = 1, \ldots, k$ and some terms Q'_i and Q''_i.

The further strategy again depends on which of the options holds:

* $\mathcal{A} \models \psi_1 \wedge \psi_2 \Rightarrow \phi$,
* $\mathcal{A} \models \psi_1 \wedge \psi_2 \Rightarrow \neg\phi$,
* $\mathcal{A} \not\models \psi_1 \wedge \psi_2 \Rightarrow \phi$ and $\mathcal{A} \not\models \psi_1 \wedge \psi_2 \Rightarrow \neg\phi$.

Let us consider the last one, the other two are similar. Then the term M' in (6.1) can be reduced by Rule (P12), and M'' in (6.2) by Rule (P11).

It all leads us to the set expressions:

$$\begin{aligned}
N' = \{&P'_1\colon \phi_1 \wedge \phi \text{ for } \sigma_1, \ldots, P'_n\colon \phi_n \wedge \phi \text{ for } \sigma_n,\\
&Q'_1\colon \theta_1 \wedge \neg\phi \text{ for } \pi_1, \ldots, Q'_k\colon \theta_k \wedge \neg\phi \text{ for } \pi_k\}\\
N'' = \{&P''_1\colon \phi_1 \wedge \phi \text{ for } \sigma_1, \ldots, P''_n\colon \phi_n \wedge \phi \text{ for } \sigma_n,\\
&Q''_1\colon \theta_1 \wedge \neg\phi \text{ for } \pi_1, \ldots, Q''_k\colon \theta_k \wedge \neg\phi \text{ for } \pi_k\}
\end{aligned}$$

in which we remove respectively subexpressions as in term M'' in (6.2) was removed by Rule (P12). It is not difficult to verify that $N' \sim_{\psi_1\wedge\psi_2} N''$.

➤ One of the rules is the reduction of a conditional expression with a determined condition

It means that $\mathcal{A} \models \psi_1 \Rightarrow \phi$ or $\mathcal{A} \models \psi_1 \Rightarrow \neg\phi$ and term M' is a reduct of Rule (P8) or (P9). Most of the scenarios of this case have already been considered in the previous case. Only the scenario in which term M'' is the result of Rule (P3) remains. This scenario is solved by a proof by induction on the structure of M.

➤ One of the rules is P12 (set reduction)

The set can be reduced only by this rule.

➤ Variant reduction

The variant can be reduced only by one rule.

➤ Atoms relations

There are no two different rules that can reduce such expressions.

- One of the rules is P3 (application reduction)

  All cases related to this rule have already been discussed in previous points.

□

Corollary 6.10. $\to_{\|}$ *satisfies the diamond property from Definition 6.3.*

Proof. It follows from Lemma 6.9 with the assumption that

$$\psi \;=\; \psi_1 \;=\; \psi_2 \;=\; \psi_1 \wedge \psi_2.$$

□

Additionally, it can be shown that the relation $\to_{\|}$ meets a slightly modified diamond property.

Lemma 6.11. *For any $M_1 \sim_{\psi} M_2$ such that $\psi \vdash M_1 \to_{\|} M_1'$ and $\psi \vdash M_2 \to_{\|} M_2'$ exist terms N_1 and N_2 such that $\psi \vdash M_1' \to_{\|} N_1$, $\psi \vdash M_2' \to_{\|} N_2$ and $N_1 \sim_{\psi} N_2$.*

Proof. It is enough to use Corollary 6.10 and prove by induction that for any $M \sim_{\psi} M'$ and reduction $\psi \vdash M \to_{\|} N$ there is a reduction $\psi \vdash M' \to_{\|} N'$ such that $N \sim_{\psi} N'$. □

6.5. Church–Rosser theorem

Before we come to the main theorem we will prove one more lemma.

Lemma 6.12. $\to^*$ *is the reflexive transitive closure of* $\to_{\|}$.

Proof. This proof is standard. Note that

$$\to \;\subseteq\; \to_{\|} \;\subseteq\; \to^*$$

because:

- all rules of the relation $\to$ can be described as some rule from the relation $\to_{\|}$,
- all rules of the relation $\to_{\|}$ are valid for the relation $\to^*$.

$\to^*$ is the reflexive transitive closure of $\to$. But the operation of taking the reflexive transitive closure of a relation is monotonous with respect to inclusion. So taking the reflexive transitive closure of all three relations gives $\to^*$ which completes the proof.

□

We now have sufficient tools to prove the Church-Rosser theorem.

Theorem 6.13. $\to$ *is Church-Rosser.*

Proof. By Corollary 6.10 we know that the relation $\to_{\|}$ satisfies the diamond property. Moreover, Lemma 6.12 says that $\to^*$ is the reflexive transitive closure of the $\to_{\|}$. So it remains to show that the diamond property is preserved by the operation of taking the transitive closure.

Lemma 6.11 makes the following diagram commute:

$$\begin{tikzcd}
 & & M \arrow[dl, two heads] \arrow[dr, two heads] & & \\
 & M_1 \arrow[dl, two heads] \arrow[dr, two heads] & & M_2 \arrow[dl, two heads] \arrow[dr, two heads] & \\
M_3 \arrow[dr, two heads] & & N_1 \sim N_2 \arrow[dl, two heads] \arrow[dr, two heads] & & M_4 \arrow[dl, two heads] \\
 & N_3 \sim N_1' \arrow[dr, two heads] & & N_2' \sim N_4 \arrow[dl, two heads] & \\
 & & N \sim N' & &
\end{tikzcd}$$

This implies the required result.

□

CHAPTER 7

Normalization

We will now consider what the reduction of a term leads to. One would like the reduction not only to replace one term with another, but that the reduction process should converge to the value that we want to calculate using a given program represented by semantic definitions. We will prove that such a process is always finite and leads to a certain *normal* form. Finally, we will analyze how this normal form looks like.

Definition 7.1. *Let ψ be a first-order formula, $\rightarrow$ a binary relation, and $\rightarrow^*$ its reflexive transitive closure.*

(i) A term M is ***in normal form*** *if there is no term N such that $\psi \vdash M \rightarrow N$.*

(ii) A term M ***has a normal form*** *if there is a term N in normal form with $\psi \vdash M \rightarrow^* N$. In this case, we also say that M is* ***normalizable*** *and that N* ***is a normal form for*** *M.*

(iii) A term M is ***strongly normalizable*** *if there is no infinite sequence $M_1, M_2, \dots$ such that $\psi \vdash M \rightarrow M_1 \rightarrow M_2 \rightarrow \cdots$. In other words, each reduction sequence that begins at term M is finite.*

(iv) $\rightarrow$ is ***weakly normalizing*** *if every term is normalizable.*

(v) $\rightarrow$ is ***strongly normalizing*** *if every term is strongly normalizable.*

(vi) $\rightarrow$ has the ***unique normal form*** *property if $\psi \vdash M \rightarrow^* N_1$ and $\psi \vdash M \rightarrow^* N_2$ for terms N_1 and N_2 in normal form implies that $N_1 \sim_\psi N_2$ (see Definition 6.1).*

The above definition is standard in the study of λ-calculi (see [Ker, 2009, Section 2.4]), but it is extended to a context ψ in which the reduction takes place. As it will turn out in this chapter, all of the above concepts (except Definition 7.1(i)) are context-independent. That is, each valid term has the above properties for each context. However, whether term is in normal form depends on the context. For example, sets or variants may or may not be reduced by rules (R22) and (R23), respectively, depending on the context (see Section 3.7).

We write that a term is strongly normalizable in $\leq\psi$ if it is strongly normalizable in all contexts ψ' such that $\mathcal{A} \vDash \psi' \Rightarrow \psi$.

Thanks to the results from Chapter 6, we can immediately deduce the unique normal form property.

Corollary 7.2. *$N\lambda$ reduction relation has the unique normal form property.*

Proof. Suppose that $\psi \vdash M \to^* N_1$ and $\psi \vdash M \to^* N_2$. According to the Church-Rosser Theorem 6.13 there exist $N_1' \sim_\psi N_2'$ such that $\psi \vdash N_1 \to^* N_1'$ and $\psi \vdash N_2 \to^* N_2'$. Because N_1 and N_2 are in normal form, so $N_1 = N_1'$ and $N_2 = N_2'$, hence $N_1 \sim_\psi N_2$. □

In this chapter, we will prove that the reduction relation from Chapter 3 is strongly normalizing (in every context) and this also implies weak normalization. At the end, we will present the structure of the term in normal form.

We begin with a simple lemma about set expressions that will be useful during the reasoning that follows.

Lemma 7.3. *$\{M_1 : \phi_1$ for $\sigma_1, \ldots, M_n : \phi_n$ for $\sigma_n\}$ is strongly normalizable in a context ψ if and only if terms M_i are strongly normalizable in a context $\psi \wedge \phi_i$ for $i = 1, \ldots, n$.*

Proof. The set expression can only be reduced by rules (R21) and (R22). The latter rule can be applied only finitely many times. The number of possible applications of the former rule depends on the strong normalization of the subterms. □

7.1. Reducibility

First normalization results were established by Turing in 1941 (published in [Gandy, 1980a]). The technique of proving strong normalization from (weak) normalization also comes from [Gandy, 1980b] and [Nederpelt, 1994].

The following proof of strong normalization is closely based on the method originally presented in [Tait, 1967]. A similar version of this method can be found in [Hankin, 1994]. The reasoning presented here is an adaptation of the version of this proof presented in [Ker, 2009]. The adaptation consists of extending the relevant concepts to the context of reduction and proving theorems for all terms in the Nλ grammar.

Let us first introduce an auxiliary definition of a property called reducibility.

Definition 7.4. *Suppose that $\Gamma \vdash M : \alpha$ for some term M and type α. We say that M is **reducible** in a context ψ if the following conditions are met:*

(i) If α is an element type (not a function type), then M is strongly normalizable in $\leq\psi$.

(ii) If $\alpha = \beta \to \gamma$, then for every context ψ' such that $\mathcal{A} \vDash \psi' \Rightarrow \psi$ and for all terms N reducible in ψ' such that $\Gamma' \vdash MN : \gamma$ for some Γ', MN must be reducible in ψ'.

Note that each type α can be represented in the form:

$$\alpha = \alpha_1 \to \cdots \to \alpha_n \to \tau,$$

where τ is an element type (possibly $n = 0$). Then Definition 7.4 can be formulated as follows: M is reducible in ψ if, for all terms $N_1, \ldots, N_n$ reducible in $\psi_1, \ldots, \psi_n$ ($\mathcal{A} \vDash \psi_n \Rightarrow \cdots \Rightarrow \psi_1 \Rightarrow \psi$) of types $\alpha_1, \ldots, \alpha_n$ respectively, $MN_1 \ldots N_n$ is strongly normalizable in $\leq\psi_n$.

From Definition 7.4, a simple lemma follows.

Lemma 7.5. *If a term M is reducible in ψ, it is also reducible in ψ' for $\mathcal{A} \vDash \psi' \Rightarrow \psi$.*

Proof. Directly by Definition 7.4. □

The plan for the proof of strong normalization is as follows:

➤ Section 7.2: prove that every reducible term is also strongly normalizable.

➤ Section 7.3: show that every well-typed term is reducible.

7.2. Reducibility to strong normalization

The implication from reducibility to strong normalization follows from item (i) of the theorem below.

Theorem 7.6. *For every type α:*

(i) If $\Gamma \vdash M : \alpha$ and M is reducible in ψ, then M is strongly normalizable in $\leq\psi$.

(ii) If $\Gamma \vdash xM_1 \dots M_n : \alpha$ (for $n \geq 0$), where M_i is strongly normalizable in $\leq\psi_i$ for $i = 1, \dots, n$ (where $\mathcal{A} \vDash \psi_n \Rightarrow \dots \Rightarrow \psi_1$), then $xM_1 \dots M_n$ is reducible in ψ_n (or in $\top$ for $n = 0$).

Proof. By induction on the type α for all contexts Γ and ψ.

➤ α is an element type

(i) It follows directly from item (i) of Definition 7.4.

(ii) All terms M_i are strongly normalizable in $\leq \psi_i$. Additionally, term $xM_1 \dots M_n$ can only be a redex of rules (R2) and (R3). These rules only reduce subterms, which means that $xM_1 \dots M_n$ is strongly normalizable in $\leq\psi_n$ and by definition also reducible in ψ_n.

➤ α is a function type ($\alpha = \beta \to \gamma$)

(i) Suppose that $\Gamma \vdash M : \beta \to \gamma$ and M is reducible in ψ.
Take a fresh variable x of type β. Since β is a subtype of α, we can apply the inductive hypothesis (ii) for $n = 0$. Therefore x is reducible in $\top$ so in ψ by Lemma 7.5. But we know that M is reducible in ψ, so Mx must also be reducible in ψ.
The type of Mx is γ, which is a subtype of α. So using induction again, this time for (i) we conclude that Mx is strongly normalizable in $\leq\psi$. And therefore so is M.

(ii) Let $\Gamma \vdash xM_1 \dots M_n : \beta \to \gamma$ for all M_i strongly normalizable in $\leq\psi_i$. Also, let $\Gamma' \vdash xM_1 \dots M_n N : \gamma$ for some term N reducible in ψ' (where $\mathcal{A} \vDash \psi' \Rightarrow \psi_n$). We want to show that $xM_1 \dots M_n N$ is reducible in ψ'.
We know that N has a type β. So by induction from (i) it is strongly normalizable in $\leq\psi'$.
But for type γ we can also use induction, now from (ii). Since all M_i and N are strongly normalizable so $xM_1 \dots M_n N$ is reducible in ψ'.

□

7.3. Universality of reducibility

In this step, we show that every valid term is reducible for every context ψ by induction on the structure of terms. Let us begin with atoms, formulas and variants.

Lemma 7.7. *Atoms, formulas and variants are reducible for all contexts.*

Proof. Atoms, formulas and variants are strongly normalizable. There are no rules to reduce an atom or a formula. A variant can be reduced by Rule (R23) only a finite number of times. Because these expressions types are not function types, by Definition 7.4(i) they are reducible. □

The same can be proved for all term variables and constants.

Lemma 7.8. *All term variables x are reducible for all contexts.*

Proof. Follows from Theorem 7.6(ii) for $n = 0$. □

Lemma 7.9. *All constants are reducible for all contexts.*

Proof. Let us analyze each constant in turn.

➤ `empty`, `atoms`, `true`, `false`

These can only be reduced by a single rule to terms in normal form, so they are strongly normalizable. In addition, they have no function type, so they are reducible by Definition 7.4(i).

➤ `and`, `or`, `not`

These are functions that take only formulas (i.e., reducible terms by Lemma 7.7) as arguments, and return formulas as values. So on the basis of Definition 7.4(ii) they are also reducible.

➤ `insert`

Assume that M and S (with types τ and $\mathbb{S}\tau$) are reducible in ψ and ψ', respectively (where $\mathcal{A} \vDash \psi' \Rightarrow \psi$), so by Theorem 7.6(i) strongly normalizable in $\leq\psi$ and $\leq\psi'$. We want to show that the expression `insert` M S is strongly normalizable in $\leq\psi'$, therefore reducible in ψ' (because the type of this expression is $\mathbb{S}\tau$).

Let us take some reduction sequence for the term `insert` M S. If this sequence does not contain Rule (R7), it must be finite because it can only involve reductions inside M and S.

If this sequence contains Rule (R7), then assume that it applies to reducts M_0 and $\{M_1 \colon \phi_1 \text{ for } \sigma_1, \ldots, M_n \colon \phi_n \text{ for } \sigma_n\}$ of M and S, respectively. The result is a set $\{M_0, M_1 \colon \phi_1 \text{ for } \sigma_1, \ldots, M_n \colon \phi_n \text{ for } \sigma_n\}$. M_0 is strongly normalizable in $\leq\psi'$, and $M_1, \ldots, M_n$ are strongly normalizable in $\leq\psi' \wedge \phi_1, \ldots, \leq\psi' \wedge \phi_n$ by Lemma 7.3. The set containing these elements is strongly normalizable in $\leq\psi'$ again by Lemma 7.3. So the rest of the reduction sequence is finite, and term `insert` M S is strongly normalizable in $\leq\psi'$ and reducible in ψ'.

➤ `map`

Suppose that M and S are reducible in ψ and in ψ' (where $\mathcal{A} \vDash \psi' \Rightarrow \psi$), and thus strongly normalizable in $\leq\psi$ and $\leq\psi'$ (Theorem 7.6(i)). To show that the `map` is reducible, we must prove that `map` $M\ S$ is reducible in ψ'. And this will be a consequence of the fact that `map` $M\ S$ is strongly normalizable in $\leq\psi'$ (because this term has an element type).

To show this fact, take any reduction sequence. If this does not reduce the redex with `map`, then it must be finite (as in the case of `insert`). Otherwise, terms M and S must be reduced to M_0 and $\{M_1\colon \phi_1 \text{ for } \sigma_1, \dots, M_n\colon \phi_n \text{ for } \sigma_n\}$, where M_i is strongly normalizable in $\leq\psi' \wedge \phi_i$ for $i = 1, \dots, n$ (by Lemma 7.3). The application of Rule (R8) to these terms gives

$$\{M_0 M_1\colon \phi_1 \text{ for } \sigma_1, \dots, M_0 M_n\colon \phi_n \text{ for } \sigma_n\}$$

It remains to show that the latter term is strongly normalizable in $\leq \psi'$. Note that M_i is reducible in $\psi' \wedge \phi_i$ by Definition 7.4(i). It follows that MM_i is reducible in $\psi' \wedge \phi_i$ by Definition 7.4(ii). Again, by Definition 7.4(i) MM_i is strongly normalizable in $\leq \psi' \wedge \phi_i$, and so is $M_0 M_i$ as a reduct of strongly normalizable term. And finally from Lemma 7.3 it follows that $\{M_0 M_1\colon \phi_1 \text{ for } \sigma_1, \dots, M_0 M_n\colon \phi_n \text{ for } \sigma_n\}$ is strongly normalizable in $\leq\psi'$.

➤ `sum`

The proof is analogous to the cases with `insert` and `map` constants.

➤ `isEmpty`

This time we want to show that `isEmpty` S is reducible (and equivalently strongly normalizable) for a reducible (equivalently, strongly normalizable) set S. This is since each reduction sequence is finite because it reduces only the set S or applies Rule (R13), which ends in a term in normal form.

➤ `if`

Recall that `if` has the type $\mathbb{B} \to \alpha \to \alpha \to \alpha$. We will prove that this constant is reducible by induction on the type α.

Take M_1 of type $\mathbb{B}$ reducible in ψ_1 and terms M_2 and M_3 of type α reducible in ψ_2 and ψ_3, where $\mathcal{A} \vDash \psi_3 \Rightarrow \psi_2 \Rightarrow \psi_1$.

- α is an element type

  In this case, it is enough to show that `if` $M_1\ M_2\ M_3$ is strongly normalizable in $\leq\psi_3$.

  This is quite simple in the case where the term M_1 reduces to a formula that is tautologically true or false.

  This is also simple for $\alpha = \mathbb{A}$ or $\alpha = \mathbb{B}$ (atoms, variants, formulas), since each reduction sequence reduces subterms or ends with a reduction rule for the conditional expression (with a possible further finite number of reductions using Rule (R23)).

  For sets, the proof looks similar to the constants `insert`, `map` and `sum`, using Lemma 7.3.

➢ α is a function type

Let us assume that $\alpha = \alpha_1 \to \cdots \to \alpha_n \to \tau$, where τ is an element type. Take $N_1, \ldots, N_n$ with types $\alpha_1, \ldots, \alpha_n$ reducible in $\theta_1, \ldots, \theta_n$ (where $\mathcal{A} \vDash \theta_n \Rightarrow \ldots \Rightarrow \theta_1 \Rightarrow \psi_3$). We will show that

$$\texttt{if}\ M_1\ M_2\ M_3\ N_1\ \cdots\ N_n$$

is strongly normalizable in $\leq\theta_n$.

Consider some reduction sequence. If it does not reduce the redex with `if` then it must be finite because it only involves reductions inside M_1, M_2, M_3, $N_1, \ldots, N_n$, which are strongly normalizable by Theorem 7.6(i). Otherwise, this redex may be reduced by rules (R14), (R15) and (R16). Rules (R14) and (R15) end the proof by reducibility of M_2 and M_3. After applying Rule (R16) we have

$$\texttt{if}\ \phi\ (M_2'N_1')\ (M_3'N_1')\ N_2'\ \cdots\ N_n' \tag{7.1}$$

for some reducts $\phi, M_2', M_3', N_1', \ldots, N_n'$ of $M_1, M_2, N_1, \ldots, N_n$.

We can apply the induction hypothesis to a term

$$\texttt{if}\ \phi\ (M_2N_1)\ (M_3N_1)\ N_2\ \cdots\ N_n \tag{7.2}$$

because the type of M_2N_1 and M_3N_1 is smaller than α and these terms are reducible in θ_1 (by Definition 7.4(ii)). Thus, term (7.2) is strongly normalizable in $\leq\theta_n$. The term (7.1) as a reduct of (7.2) is also strongly normalizable in $\leq\theta_n$, which ends the proof.

➤ $\texttt{eq}_{\mathbb{A}}$, $\texttt{leq}_{\mathbb{A}}$

These constants are only applied to variants or atoms that are reducible. As a result, they give formulas, which are also reducible (see Lemma 7.7).

□

One more lemma is needed regarding the reducibility of abstraction.

Lemma 7.10. *For a given context ψ, if $\Gamma, x\colon \alpha \vdash M\colon \beta$ and for all term N with type α reducible in ψ' (where $\mathcal{A} \vDash \psi' \Rightarrow \psi$), $M[N/x]$ is reducible in ψ', then $\lambda x.M$ is reducible in ψ.*

Proof. Note that $\Gamma \vdash \lambda x.M\colon \alpha \to \beta$. Note also that M is reducible in ψ (take $N = x$, which is reducible by Lemma 7.8).

Suppose that $\beta = \beta_1 \to \cdots \to \beta_n \to \tau$ for some element type τ. Take $N, N_1, \ldots, N_n$ with types $\alpha, \beta_1, \ldots, \beta_n$ reducible in $\psi', \psi_1, \ldots, \psi_n$ ($\mathcal{A} \vDash \psi_n \Rightarrow \cdots \Rightarrow \psi_1 \Rightarrow \psi' \Rightarrow \psi$). We want to show that $(\lambda x.M)NN_1 \ldots N_n$ is strongly normalizable in $\leq\psi_n$.

Take some reduction sequence of the above term. If it does not reduce the reduct with the initial abstraction, then it must be finite because it only reduces $M, N, N_1, \ldots, N_n$, which are reducible so strongly normalizable by Theorem 7.6(i).

Otherwise, the reduction sequence is as follows

$$(\lambda x.M)NN_1 \ldots N_n \ \to^* \ (\lambda x.M')N'N_1' \ldots N_n' \ \to \ M'[N'/x]N_1' \ldots N_n' \ \to \ \cdots$$

Recall that by assumption $M[N/x], N_1, \ldots, N_n$ are reducible in $\psi', \psi_1, \ldots, \psi_n$. So a term $M[N/x]N_1 \ldots N_n$ is strongly normalizable in $\leq\psi_n$. We conclude that also its reduct: $M'[N'/x]N'_1 \ldots N'_n$ is strongly normalizable, the reduction sequence is finite, and therefore $\lambda x.M$ is reducible in ψ. □

And now an important theorem, which will easily imply the main result.

Theorem 7.11. *Suppose that $x_1\colon \alpha_1, \ldots, x_n\colon \alpha_n \vdash M\colon \alpha$, $FV_T(M) \subseteq \{x_1, \ldots, x_n\}$. For every $N_1, \ldots, N_n$ with types $\alpha_1, \ldots, \alpha_n$ reducible in $\psi_1, \ldots, \psi_n$, the term $M[N_1/x_1, \ldots, N_n/x_n]$ is reducible in $\psi_1 \wedge \cdots \wedge \psi_n$.*

Proof. We write $M[\overrightarrow{N}/\overrightarrow{x}]$ for the simultaneous substitution $M[N_1/x_1, \ldots, N_n/x_n]$.

Proof by induction on the structure of M:

➤ ***M* is a term variable**

Then $M = x_i$ for some $i \in \{1, \ldots, n\}$. Therefore $M[\overrightarrow{N}/\overrightarrow{x}] = N_i$, which is reducible in ψ_i, so also in $\psi_1 \wedge \cdots \wedge \psi_n$ (by Lemma 7.5).

➤ ***M* is an abstraction**

Let $M = \lambda x.M'$. We may assume that x is not one of $x_1, \ldots, x_n$. Note that $x_1\colon \alpha_1, \ldots, x_n\colon \alpha_n, x\colon \beta \vdash M'\colon \gamma$ with $\alpha = \beta \to \gamma$.

Based on the induction hypothesis, we know that $M'[\overrightarrow{N}/\overrightarrow{x}][N/x]$ is reducible in $\psi_1 \wedge \cdots \wedge \psi_n \wedge \psi'$ for every N with type β reducible in ψ'.

But this means that $\lambda x.M'[\overrightarrow{N}/\overrightarrow{x}]$ is reducible in $\psi_1 \wedge \cdots \wedge \psi_n$ thanks to Lemma 7.10, so $M[\overrightarrow{N}/\overrightarrow{x}]$ is also reducible in this context.

➤ ***M* is an application**

We have $M = M_1 M_2$. The induction hypothesis applies to M_1 and M_2, telling that $M_1[\overrightarrow{N}/\overrightarrow{x}]$ and $M_2[\overrightarrow{N}/\overrightarrow{x}]$ are reducible in $\psi_1 \wedge \cdots \wedge \psi_n$. Hence $M[\overrightarrow{N}/\overrightarrow{x}]$ is reducible in $\psi_1 \wedge \cdots \wedge \psi_n$ by Definition 7.4(ii).

➤ ***M* is a constant, an atom, a formula or a variant**

These terms do not contain term variables, so $M[\overrightarrow{N}/\overrightarrow{x}] = M$ is reducible by Lemma 7.7 and 7.9.

➤ ***M* is a set**

Let $M = \{M_1\colon \phi_1 \text{ for } \sigma_1, \ldots, M_n\colon \phi_n \text{ for } \sigma_n\}$. By the induction hypothesis $M_i[\overrightarrow{N}/\overrightarrow{x}]$ is reducible in $\psi_1 \wedge \cdots \wedge \psi_n$ for $i = 1, \ldots, n$. Theorem 7.6(i) indicates that these terms are also strongly normalizable in $\leq\psi_1 \wedge \cdots \wedge \psi_n$. From Lemma 7.3 it follows that the set

$$\{M_1[\overrightarrow{N}/\overrightarrow{x}]\colon \phi_1 \text{ for } \sigma_1, \ldots, M_n[\overrightarrow{N}/\overrightarrow{x}]\colon \phi_n \text{ for } \sigma_n\} = M[\overrightarrow{N}/\overrightarrow{x}]$$

is strongly normalizable in $\leq\psi_1 \wedge \cdots \wedge \psi_n$, therefore $M[\overrightarrow{N}/\overrightarrow{x}]$ is reducible in $\psi_1 \wedge \cdots \wedge \psi_n$ by Definition 7.4(i).

□

Corollary 7.12. *$N\lambda$ reduction relation is strongly normalizing in every context.*

Proof. Take any term M with $\mathrm{FV_T}(M) = \{x_1, \ldots, x_n\}$. Then for some type α we have $x_1\colon \alpha_1, \ldots, x_n\colon \alpha_n \vdash M\colon \alpha$. Variables $x_1, \ldots, x_n$ are reducible by Lemma 7.8. Thus $M = M[x_1/x_1, \ldots, x_n/x_n]$ is reducible by Theorem 7.11. In turn, from Theorem 7.6(i) we know that M is strongly normalizable. □

7.4. Normal form

The purpose of programming languages and the reduction system is to calculate final values, not structures containing functions and applications. In turn, strong normalization means that any term can be reduced to normal form. That is why it is worth considering what the normal form looks for a term whose type is not functional.

Theorem 7.13. *A closed term M in normal form with a element type is an atom, formula, variant or set containing: atom, formula, variant or another set.*

Proof. By induction on the structure of term M:

- ***M* is a term variable**

  M cannot be a variable because then this variable would be free and the term M is closed.

- ***M* is an abstraction**

  If M were an abstraction, it would not have an element type.

- ***M* is an application**

  Suppose $M = M_1M_2$. M_1 has a function type, therefore it can only be a variable, an abstraction, an application or a constant. We can exclude a variable and an abstraction due to the assumption that M is a closed term in normal form. Therefore, as we move left on the structure of the term M while we encounter the application we will eventually reach some constant C.

  Because term M has an element type, so the constant C is applied to the maximum possible number of arguments resulting from the type of C. Consider all possible cases:

  - C is `insert` or `map`

    Then the second argument must be in the form of a set (by the induction hypothesis). And that would mean that M is not a term in normal form (see rules (R7) and (R8)).

  - C is `sum` or `isEmpty`

    By the induction hypothesis, we know that the argument of constant C must be a set (in the case of `sum` a set containing other sets). So in this case we could also apply reduction rules ((R9) or (R13)), which contradicts the assumption of the theorem.

  - C is `or`, `and` or `not`

    In this case, the arguments must be formulas, which contradicts normal form.

  - $C =$ `if`

    For a conditional expression, the first argument must be a formula, while the next two depend on the type of constant C. Applying the induction hypothesis to these arguments leads us to a term that can be reduced by one of the rules (R14)–(R20).

- C is `eq`$_{\mathbb{A}}$ or `leq`$_{\mathbb{A}}$

  Arguments must be atoms or variants of atoms, which allows to apply rules (R24) or (R25).

The above analysis means that M cannot be an application.

- ***M* is a constant**

  The only constants that have an element type are `empty`, `atoms`, `true` and `false`. But in this case, M would not be in normal form (see rules (R5), (R6) and (R10)).

- ***M* is an atom, formula or variant**

  Then the statement of the theorem holds.

- ***M* is a set**

  By the induction hypothesis, the elements of the set are atoms, formulas, variants or sets.

□

7.5. Consequences of strong normalization

Strong normalization implies several desirable properties of semantics.

First of all, Corollary 7.12 ensures that every valid program always terminates with a term in normal form. In addition, thanks to the unique normal form property (Corollary 7.2), regardless of the order in which the term is reduced, this reduction always ends with the same term (up to first-order formula equivalence).

Let us define an equality relation on terms that is a reflexive, symmetric and transitive closure of the reduction relation $\to$. This relation tells whether two terms can be reduced to the same term in normal form (up to the equivalence relation $\sim$). Then strong normalization implies:

Corollary 7.14. *Equality of terms is decidable.*

Proof. To determine if two terms are equal, just reduce both to the normal form and compare whether they are identical up to formula equivalence. The satisfiability of formulas can be computed based on the decidability of the first-order theory (see assumptions in Section 2.1). □

Moreover, terms that are contained in the above equality relation have equal values (see definition of denotation in Section 3.5). For a fixed valuation of free atom variables, the denotation values of these terms are equal, which follows from Theorem 5.2.

.

CHAPTER 8

Language extensions

The Nλ language presented in Chapter 3 allows one to perform calculations on sets with atoms, according to the theory presented in Chapter 2. In the previous chapters we showed that the semantics of the language is correct and has basic desirable properties. But it is still far from a complete programming language. The missing elements include: defining functions and values under a given name, operations on numbers and texts, the ability to create popular data structures such as lists, arrays, tuples or records. A very important omission is the lack of recursion. Features such as pattern matching or tagged unions can also be considered. All of this is linked to more general concepts like algebraic data types or polymorphism.

In this chapter, we will present various approaches to extending the current semantics. We will also consider how these extensions affect the already proven properties and expressivity of the language.

8.1. Atom constants

The core language presented in Chapter 3 represents atoms through atom variables. It allows the programmer to create variables with a selected name, and then operate on these atom variables, e.g. by computing the relations (from a relational structure) with other atom variables. What this language does not allow is the use of specific atom values (e.g. natural numbers for equality atoms, rational numbers for ordered atoms). One can imagine a situation where the user would like to calculate the results for specific numbers, e.g. a set of atoms from the interval $[1, 2]$ for ordered atoms:

$$\texttt{filter}\ (\lambda\texttt{x.and}\ (\texttt{leq}_{\mathbb{A}}\ \texttt{1 x})\ (\texttt{leq}_{\mathbb{A}}\ \texttt{x 2}))\ \texttt{atoms}$$

Of course, a programmer can introduce two atom variables a and b into the program and finally substitute $a \mapsto 1, b \mapsto 2$ in the result. However, the use of specific values, apart from convenience, has another advantage. Formulas that use values instead of variables can be greatly simplified, which means that their processing is also simpler; for example, conditional expressions often can only evaluate one case instead of creating variants. The above example is very simple, but in more complex calculations it may be of great importance in terms of the calculation time and the complexity of the resulting structure.

Introducing atom constants into the language is not difficult. This is because in most cases they behave the same as atom variables. The term containing an atom constant, just like for atom variables, is a closed term of type $\mathbb{A}$. Just like the variable, the constant can be passed to the same functions and relations. Both constants and variables can appear in formulas or variants. And most importantly, for atom constants, exactly the same reasoning can be used in all lemmas and theorems from Chapters 4 - 7. Therefore, the language extended by atom constants will have all the properties proved in these chapters.

The two important differences between them are:

➤ a different denotation – while valuation is required to evaluate a variable, a constant is equal to its denotation, therefore the denotation function is the identity function in this case;

➤ a formula containing a constant instead of a variable may turn out to be a tautology or a contradiction and resulting in a possible simplification of the calculations, as we mentioned earlier.

8.2. Explicit extension

Introducing new elements to the language will involve extending its syntax and semantics: adding types, reduction and typing rules, and defining denotation for new terms. Below are examples of how to add numbers, texts, tuples, unit type, records, and sums to a language. Finally, we consider how such extensions affect language properties.

Primitive data types

Practically all popular programming languages have built-in primitive types for numbers. This is due to the fact that they are directly supported by computer hardware and therefore the operations on them are very efficient. Most often, numeric types are divided into the following categories: integers, floating-point numbers, fixed-point numbers.

To introduce integers into Nλ one may add a new base type

$$\tau \ ::= \ \dots \ | \ \mathbb{N}$$

and literals for individual integers containing sequences of digits with an optional negative sign at the beginning. Constants representing basic operations are also needed:

$$\begin{aligned} (+) &: \mathbb{N} \to \mathbb{N} \to \mathbb{N} \\ (-) &: \mathbb{N} \to \mathbb{N} \to \mathbb{N} \\ (*) &: \mathbb{N} \to \mathbb{N} \to \mathbb{N} \end{aligned}$$

Additional operations like division with remainder, negation, or absolute value can also be introduced. For the new constants, we will not list reduction rules because they are built-in arithmetic operations. But we will add a new typing rule:

$$\frac{n \text{ is an integer literal}}{\Gamma \vdash n \colon \mathbb{N}}$$

In order to execute conditional expressions for numbers, a reduction rule for `if` must be added, analogous to Rule (R17). Moreover, a variant construction analogous to that for atoms $(a_1 \colon \phi_1 | \cdots | a_n \colon \phi_n)$ must be added to the grammar. Additionally, an equality on numbers will be useful, which will be the counterpart of the `eq` function for atoms.

Some programming languages support text strings as primitive values while others treat a text string as an array of characters. In the case of Nλ, we can introduce a new type $\mathbb{C}$ for characters, extending the semantics in a similar way as for integers. Strings will be available after introducing lists to the language (in the same way as it is done in Haskell).

Let binding

A very important functionality of a language is the ability to assign names to expressions. This avoids duplication and increases readability. To this end, we may introduce a new term construction with the keyword *let*.

$$M \ ::= \ \ldots \ \Big| \ \text{let } x = M \text{ in } M$$

Such a term can be defined as syntactic sugar for the lambda abstraction applied to a term:

$$\text{let } x = M \text{ in } N \qquad \equiv \qquad (\lambda x.N)\ M$$

Importantly, the above structure is not recursive, meaning that the variable x cannot be used in the term M. The recursive version of *let* will be described later (Section 8.4).

Tuples

Many programming languages provide a variety of ways of building compound data structures. The simplest example is a pair, and more generally a tuple. In other words, it is an ordered list of elements of any element type. The type of tuple is the product type, which, following Haskell's notation, will be written as a tuple of types.[1]

$$\tau \ ::= \ \ldots \ | \ (\tau, \ldots, \tau)$$

A degenerated form of product type (with zero types) is the unit type. It is a singleton type with only one element – the empty tuple ().

Tuples are represented by a new kind of term $(M_1, \ldots, M_n)$ which, like abstraction, is available directly to the programmer in the source code (as opposed to sets and variants that arise as results of calculations). Additionally, projections of tuples onto their individual elements are available, written with π_i for projection on the i-th element.

$$M \ ::= \ \ldots \ \Big| \ (M, \ldots, M) \ \Big| \ \pi M$$

[1] Another common notation is $\tau \times \cdots \times \tau$, but Haskell notation makes it easier to represent the unit type.

The reduction rules for tuples include the reduction of tuple elements, the reduction of a projected tuple, and the projection itself:

$$\frac{\psi \ \vdash \ M \ \rightarrow \ N}{\psi \ \vdash \ (\dots, M, \dots) \ \rightarrow \ (\dots, N, \dots)}$$

$$\frac{\psi \ \vdash \ M \ \rightarrow \ N}{\psi \ \vdash \ \pi_i M \ \rightarrow \ \pi_i N}$$

$$\psi \ \vdash \ \pi_i(M_1, \dots, M_n) \ \rightarrow \ M_i \qquad \text{for } 1 \leq i \leq n$$

An additional rule is needed to execute conditional expressions on products.

$$\frac{\mathcal{A} \not\models \psi \Rightarrow \phi \quad \mathcal{A} \not\models \psi \Rightarrow \neg\phi}{\psi \vdash \texttt{if}\ \phi\ (M_1, \dots, M_n)\ (N_1, \dots, N_n) \ \rightarrow \ (\texttt{if}\ \phi\ M_1 N_1, \dots, \texttt{if}\ \phi\ M_n N_n)}$$

The typing rules for tuples and projections are straightforward.

$$\frac{\Gamma \vdash M_1 \colon \tau_1 \ \dots \ \Gamma \vdash M_n \colon \tau_n}{\Gamma \vdash (M_1, \dots, M_n) \colon (\tau_1, \dots, \tau_n)} \qquad \frac{\Gamma \vdash M \colon (\tau_1, \dots, \tau_n)}{\Gamma \vdash \pi_i M \colon \tau_i} \ \text{for } 1 \leq i \leq n$$

Tuples can be generalized to records. Records are very convenient structures that increase the readability of code. While a tuple is an ordered list of values, a record is a map of developer-defined labels to values. Labels are used here in three ways: to denote values, types, and as projection operations.

For example, one could define a complete graph with vertices that are atoms with the following record:

$$\{\texttt{vertices} = \texttt{atoms},\ \texttt{edges} = \texttt{atomsPairs}\}$$

of the following type

$$\{\texttt{vertices} : \mathbb{A},\ \texttt{edges} : (\mathbb{A}, \mathbb{A})\}$$

with operations of graph projection into its individual components: `vertices` and `edges`.

Sums

A tuple is a structure that stores values of many types at once. On the other hand, the sum (also called disjoint union or coproduct) is used to select one of many types. Therefore, sum types are dual to product types.

At first glance, disjoint union is similar to a variant expression, but there are key differences:

- a variant consists of elements of the same type, which is at the same time the type of the whole variant, while the disjoint union may contain elements of different types,
- a variant stores all possible values together with formulas which satisfiability determines which value is returned, while the coproduct contains only one target value.

A sum type is denoted by a list of component types separated by a plus sign:

$$\tau \ ::= \ \dots \ \mid \ \tau + \dots + \tau$$

Coproducts come equipped with injections, the index of which indicates which type corresponds to a given value. Injection ι_i indicates that the given value is of type τ_i, where the type of the whole sum is $\tau_1 + \dots + \tau_n$. The exact value of a sum element is identified through case analysis similar to pattern matching. The *case* command is used for this purpose:

$$M \ ::= \ \dots \ \Big| \ \iota M \ \Big| \ \text{case } M \text{ of } x \Rightarrow M; \dots ; x \Rightarrow M$$

Reduction is possible both for the subterms of a sum and for *case* terms. An additional reduction rule realizes case matching:

$$\frac{\psi \ \vdash \ M \ \to \ N}{\psi \ \vdash \ \iota_i M \ \to \ \iota_i N}$$

$$\frac{\psi \ \vdash \ M \ \to \ N}{\psi \ \vdash \ \text{case } M \text{ of } x_1 \Rightarrow M_1; \dots ; x_n \Rightarrow M_n \ \to \ \text{case } N \text{ of } x_1 \Rightarrow M_1; \dots ; x_n \Rightarrow M_n}$$

$$\psi \ \vdash \ \text{case } \iota_i M \text{ of } x_1 \Rightarrow M_1; \dots ; x_n \Rightarrow M_n \ \to \ M_i[M/x_i] \quad \text{for } 1 \leq i \leq n$$

The typing rules for the sum and case expression are as follows:

$$\frac{\Gamma \vdash M : \tau_i}{\Gamma \vdash \iota_i M : \tau_1 + \dots + \tau_n}$$

$$\frac{\Gamma \vdash M : \tau_1 + \dots + \tau_n \quad \Gamma \vdash x_1 : \tau_1 \quad \dots \quad \Gamma \vdash x_n : \tau_n \quad \Gamma \vdash M_1 : \tau \quad \dots \quad \Gamma \vdash M_n : \tau}{\Gamma \vdash \text{case } M \text{ of } x_1 \Rightarrow M_1; \dots ; x_n \Rightarrow M_n : \tau}$$

Sum types are problematic for conditional expressions. Such an expression cannot be delegated downwards as in the case of a product. This can be solved in two ways.

One way is not to define a rule for the conditional expression for the sum type. But then Theorem 7.13 will no longer holds about closed value term in normal form. Our implementation (see Chapter 10) handles this so that only types belonging to a certain class support conditional expressions.[2] Sum types do not belong to this class, so a conditional expression over a sum type is not well-typed.

Another solution is to add a rule similar to (R17).[3]

Just as a record is an extension of a tuple by allowing values to be labeled, a tagged union is a version of a disjoint union in which values are also labeled. Then the names of the injections come from the labels.

[2]The type class is a Haskell construct. The one that handles conditional expressions is called `Conditional`. See Section 10.9.

[3]There is a variant version of the sum type in the implementation. It is called `NominalEither`. See Section 10.10.

Basic properties

Nλ semantics enriched with additional base types, let bindings, products, and sums still satisfies basic properties: subject reduction (Theorem 4.2), Church-Rosser property (Theorem 6.13) and strong normalization (Corollary 7.12). We will not present proofs for extended semantics in this thesis, because they are straightforward modification of the basic proofs and would add little to the story.

Proof that the simply typed lambda calculus extended with products and sums has the Church-Rosser property and strong normalization can be found in [Dougherty, 1993] or [Di Cosmo and Kesner, 1995].

In the examples above, products and coproducts were only possible for values. The same structures can also be created for functions. For this purpose, function types must be extended analogously.[4]

8.3. Construction of sets with atoms

Adding product and sum types to Nλ is very important because this makes it possible to compute any set definable by a set expression (see Section 2.7). So far, the greatest difficulty has been to construct a set that has elements of different types. Set expressions as presented in Definition 2.10 enable, for example, a set containing an atom and a set of atoms. Meanwhile, Nλ semantics only allows sets with elements of the same type. Sum types provide a solution to this problem: such a set will then be of type $\mathbb{S}(\mathbb{A} + \mathbb{S}\mathbb{A})$.

In this way we can type any set expression, assuming that atom variables have the type $\mathbb{A}$, while the expression $\{\xi_1, \dots, \xi_n\}$ has the type $\mathbb{S}(\tau_1 + \cdots + \tau_n)$, where ξ_i describes the elements of the type τ_i for $i = 1, \dots, n$.

Futhermore, product types allow us to construct sets containing more complex types. For example,

$$\texttt{atomsPairs} = \texttt{sum}\ (\texttt{map}\ (\lambda x.\texttt{map}\ (\lambda y.(x,y))\ \texttt{atoms})\ \texttt{atoms}) : \mathbb{S}(\mathbb{A},\mathbb{A}) \tag{8.1}$$

evaluates to the set of all pairs of atoms, and

$$\texttt{filter}\ (\lambda x.\texttt{not}\ (\texttt{eq}_{\mathbb{A}}\ \ \pi_1 x\ \ \pi_2 x))\ \texttt{atomsPairs} : \mathbb{S}(\mathbb{A},\mathbb{A})$$

to the set of all distinct pairs of atoms.

The proof of the following theorem provides a technique for constructing any set definable by a set expression.

Theorem 8.1. *Every definable set can be computed by Nλ with product and sum types.*

Proof. Take any set expression $e \colon \mathbb{S}\tau$ (as presented in Definition 2.10) with free atom variables $a_1, \dots, a_k$. We will show how to construct a term $\texttt{set}_e \colon \mathbb{A}^k \to \mathbb{S}\tau$ that evaluates to a function from $\mathbb{A}^k$ that, when applied to atom values $v_1, \dots, v_k$, returns $[\![e]\!]_{[a_i \to v_i]_{i=1}^k}$.

[4]In the Nλ implementation, there is no distinction between element and function types, but sets can only contain types with a specific `Nominal` class that does not contain functions or types composed of functions. See Section 10.11.

We perform an inductive construction on the structure of a set expression. Atom variables are represented in an obvious way. Let $e = \{\xi_1, \dots, \xi_n\}$. First let us create a term representing the set $\{\xi_i\}$ for each $i = 1, \dots, n$.

Suppose

$$\xi_i \;=\; e_i \;:\; \phi \;\text{ for }\; a_{k+1}, \dots, a_m \in \mathcal{A}$$

for some $e_i \colon \tau_i$ and ϕ such that $FV_A(e_i),\; FV_A(\phi) \subseteq\; \{a_1, \dots, a_m\}$.

It is easy to generalize the term `atomsPairs` from (8.1) to a function

$$\texttt{atomTuples}_{k,m} \;:\; \mathbb{A}^k \to \mathbb{S}\mathbb{A}^m$$

that extends a given k-tuple of atoms to the set of all m-tuples that arise by putting arbitrary atoms on the remaining $m-k$ components.

Then $\{\xi_i\}$ is represented by term $\texttt{set}_{\{\xi_i\}} \colon \mathbb{A}^k \to \mathbb{S}\tau_i$ defined as follows

$$\texttt{set}_{\{\xi_i\}} \;=\; \lambda t.\, \texttt{map}\ \texttt{set}_{e_i}\ (\texttt{filter}\ \texttt{form}_\phi\ (\texttt{atomTuples}_{k,m}\ t))$$

where $\texttt{set}_{e_i} \colon \mathbb{A}^m \to \tau_i$ exists by the inductive assumption, and $\texttt{form}_\phi \colon \mathbb{A}^m \to \mathbb{B}$ is a term that encodes the first-order formula ϕ. Such a term exists since $\mathcal{A}$ has quantifier elimination (see assumptions in Section 2.1), and so without loss of generality we may assume that ϕ is quantifier-free.

Before we combine functions $\texttt{set}_{\{\xi_i\}}$, we need to inject them into a common type.

$$\texttt{set}'_{\{\xi_i\}} \;=\; \lambda t.\, \texttt{map}\ (\lambda x.\iota_i x)\ (\texttt{set}_{\{\xi_i\}}\ t)$$

Of course, if all τ_i are equal, this step is redundant. Now it is enough to calculate the sum of the individual sets.

$$\begin{aligned}\texttt{set}_e \;=\; \lambda t.\, \texttt{sum}\ &(\texttt{insert}\ (\texttt{set}'_{\{\xi_1\}}\ t)\\ &(\texttt{insert}\ (\texttt{set}'_{\{\xi_2\}}\ t)\\ &\vdots\\ &(\texttt{insert}\ (\texttt{set}'_{\{\xi_n\}}\ t)\ \texttt{empty}) \cdots))\end{aligned}$$

where $\texttt{set}_e \colon \mathbb{A}^k \to \mathbb{S}(\tau_1 + \cdots + \tau_n)$. $\square$

8.4. Recursion

Recursion is a popular method of solving computational problems, especially in functional languages. Most often it consists of breaking the task into smaller parts and calling the same function on smaller data. This requires the function to be called by itself. The *let* expression presented in Section 8.2 does not allow this kind of function call. Below we present a recursive version of this expression.

In lambda calculus, recursion is most often handled by fixed-point combinators [Peyton Jones, 1987]. In a typed setting, such a combinator is a higher-order function `fix`: $(\alpha \to \alpha) \to \alpha$ that returns the fixed point of a given function. If the function f has a fixed point, then equality holds

$$\texttt{fix}\ f \;=\; f\ (\texttt{fix}\ f)$$

The classic untyped lambda calculus allows to implement the function `fix` using the paradoxical combinator `Y` (first discussed in [Curry and Feys, 1958], in the typed context in [Scott, 1969] and [Platek, 1966]).

$$\texttt{Y} \;=\; \lambda f.\,(\lambda x.f(x\,x))\,(\lambda x.f(x\,x))$$

However, in a simply-typed setting, this is not a valid term. Especially within $(x\,x)$, we cannot assign a unique type to both occurrences of x. Nevertheless, the function `fix` can be added to Nλ as a constant with its reduction rule:

$$\psi \;\vdash\; \texttt{fix}\;M \;\rightarrow\; M\;(\texttt{fix}\;M)$$

The type of this function is $(\alpha \rightarrow \alpha) \rightarrow \alpha$. The function `fix` can be used to define a recursive version of the *let* expression:

$$\text{let rec } x = M \text{ in } N \qquad \equiv \qquad \text{let } x = \texttt{fix}\;(\lambda x.M) \text{ in } N$$

The expression *let rec*, unlike the expression *let*, allows the occurrence of the variable x in term M.

The Nλ language enriched with recursion obviously ceases to be strongly normalizing. This is due to the fact that one can define a recursive function that will never terminate. Denotational semantics of a language equipped with recursion is also vastly more complicated than the simplistic framework that we used in Chapter 3. But recursion, in addition to the convenience and readability of the code, greatly increases the computing power of the language.

8.5. Polymorphism

The semantics presented so far used the Curry-style type system (sometimes called extrinsic or implicitly typed) [Curry and Feys, 1958]. In short, it means that terms are not explicitly marked with types. The type inference method for a given term is based on context and typing rules (see Section 3.6).

The opposite approach is the Church-style type system (also called intrinsic or explicitly typed) [Church, 1940]. It requires explicit type annotations for term variables. For example, the abstraction looks like $\lambda x\colon \alpha.M$. Thus, the variable x is immediately assigned the type α.

The differences between the two approaches are very easy to understand on generic Nλ functions such as `if` or `eq` (Section 3.2). In the Curry approach, these functions can be of many types depending on the arguments to which they are applied.

$$\begin{array}{ll}
\texttt{if}\colon \mathbb{B} \rightarrow \mathbb{A} \rightarrow \mathbb{A} \rightarrow \mathbb{A} & \qquad \texttt{eq}\colon \mathbb{A} \rightarrow \mathbb{A} \rightarrow \mathbb{B} \\
\texttt{if}\colon \mathbb{B} \rightarrow \mathbb{B} \rightarrow \mathbb{B} \rightarrow \mathbb{B} & \qquad \texttt{eq}\colon \mathbb{B} \rightarrow \mathbb{B} \rightarrow \mathbb{B} \\
\texttt{if}\colon \mathbb{B} \rightarrow \mathbb{SA} \rightarrow \mathbb{SA} \rightarrow \mathbb{SA} & \qquad \texttt{eq}\colon \mathbb{SA} \rightarrow \mathbb{SA} \rightarrow \mathbb{B} \\
\quad\vdots & \qquad\quad \vdots
\end{array}$$

In the second case, we would have to define a separate function for each type: $\texttt{if}_{\mathbb{A}}, \texttt{if}_{\mathbb{B}}, \texttt{if}_{\mathbb{SA}}$ and $\texttt{eq}_{\mathbb{A}}, \texttt{eq}_{\mathbb{B}}, \texttt{eq}_{\mathbb{SA}}$, etc.

So in the first approach we have functions that can be applied to arguments of many types, and in the second, the uniqueness of the type. To get both these two advantageous properties one needs a richer type system such as System F, a typed lambda calculus with universal quantification over types [Girard, 1972, Reynolds, 1974]. This system extends simply typed lambda calculus with variables ranging over types, and binders for them. Thus, we get types such as:

$$\texttt{if}\colon \forall\alpha.\, \mathbb{B} \to \alpha \to \alpha \to \alpha \qquad\qquad \texttt{eq}\colon \forall\alpha.\, \alpha \to \alpha \to \mathbb{B}$$

where α is a type variable. Before using the above functions, we must first specify the type that will be the value of variable α.

System F (which does not include recursion in its definition) is strongly normalizing, but type inference (without explicit type annotations) is undecidable. It is used in many statically typed functional programming languages such as Haskell 98 [Jones, 2002] and the ML family. Due to the undecidability of type inference, these languages actually used a restriction of System F called the Hindley-Milner type system [Hindley, 1969, Milner, 1978].

System F extension has been used as the type system in the Nλ implementation as described below.

8.6. Towards an implementation

As we mentioned in the previous section, the initial version of the Haskell language was based on System F. Over time, this system became insufficient and therefore the language developers designed an extension of this system called System FC [Sulzmann et al., 2007]. Haskell's most popular compiler, Glasgow Haskell Compiler [Hall et al., 1992, Haskell.org, 2022], is based on this system.

Since Nλ is implemented in Haskell (Chapter 10) and compiled with GHC, System FC describes the actual semantics of Nλ. Let us list some of the basic functionalities of this system.

First of all, we have *let* expressions in both the non-recursive (Section 8.2) and recursive (Section 8.4) versions.

In addition, we have algebraic data types, which are an extension of the product and sum types (Section 8.2) [Burstall, 1969, Burstall and Darlington, 1977]. This allows to define not only such structures as tuples or disjoint unions, but also lists, trees and many others.

Algebraic data types enable another important feature of functional languages that greatly enhances their readability, namely pattern matching. It is available through a more general version of the *case* expression.

Another feature worth mentioning is type classes [Hall et al., 1996].[5] It allows to mark types for which certain operations will be available. In the case of Nλ implementation we use this mechanism to determine which types can be element types, for which we can create conditional expressions or for which equality functions are defined.

[5]Type classes are not directly available in System FC, but can be translated into this system, and are available in Haskell.

8.7. Relational structures

So far in this chapter, we have discussed language extensions as extensions of its basic semantics. But Nλ can also be expanded by adding more relational structures of atoms to it. In Section 2.1 we considered such structure in general and we said that we will primarily consider two structures: equality and ordered atoms. Also, our implementation at the moment is limited to these two structures. But the design of the language is open to considering other structures.

Recall that the structures we are considering must have certain properties: decidability of the first order theory, oligomorphism, and homogeneity. It turns out that the last two properties are satisfied by so-called Fraïssé limits (for details see [Hodges, 1993]). Briefly, for a class of finite relational structures satisfying certain properties, one can construct a countable structure called the Fraïssé limit, that contains all the elements of the class as substructures.

Equality atoms is a limit for the class of finite sets, and ordered atoms is a limit for the class of finite total orders. We can also construct limits for other classes of finite structures:

- partial orders,
- equivalence relations,
- directed and undirected graphs,
- trees and forests.

A detailed presentation of the cases of the random graph (the limit of all finite undirected graphs), bit vectors and universal trees can be found in [Bojańczyk, 2019, Chapter 7].

CHAPTER 9

Alternative approaches

In this chapter we briefly sketch some alternative approaches to designing programming languages for manipulating sets with atoms.

Sets with atoms can be represented not only by set expressions. As it is described in this chapter, they can also be represented by specific elements of the orbits or sizes of the smallest supports of these orbits. An example here is the predecessor of the Nλ language. Additionally, a language that uses the same representation of sets with atoms, but is based on the imperative programming paradigm will be characterized. Finally, other alternative approaches to the broad topic of computing with infinite objects will be discussed, both using nominal techniques and using other methods.

9.1. Hulls, supports and orbits

The direct predecessor of the Nλ language presented in this thesis is a toy functional language of the same name described in [Bojańczyk et al., 2012]. In that approach, a different internal representation of infinite sets was used. To construct such sets a programming construction `hull` was provided, which, given a finite list C of atoms and a set of values X of some type (possibly built of atoms), returns the closure of X under all automorphisms of atoms that fix every element of C.

For example, the expression

$$\texttt{hull}\ []\ \{1\}$$

evaluates to the set of all atoms, because each atom can be obtained by applying some automorphism of $\mathcal{A}$ to the atom 1. Any automorphism can be considered, since the list C is empty.

If equality atoms are considered, the expressions

$$\texttt{hull}\ [2]\ \{3\} \qquad\qquad \texttt{hull}\ [4]\ \{(4,5)\}$$

evaluate respectively to the set of all atoms other than 2, and to the set of all pairs of atoms such that the first atom is equal to 4 and the second atom is not 4. For ordered atoms, the expression

$$\texttt{hull}\ []\ \{(6,7)\}$$

evaluates to the set of all pairs of atoms where the second component is strictly greater than the first.

Sets constructed in this way can be then manipulated by functions such as `map`, `sum`, `filter`, etc. This means that programs can be implemented in a similar way to the language presented here. The main difference, however, is that the internal representation of the sets was not based on first-order formulas. Rather, the `hull` construct was used as a basic semantic construct in computed values of set types (see [Bojańczyk et al., 2012] for details).

Representing infinite sets with the `hull` construction has significant disadvantages. The most important of them is that the size of the representation of an orbit-finite set is proportional to the number of orbits of this set. For example, the set of all triples of atoms is constructed by the following expression:

$$\texttt{hull}\ []\ \{(1,1,1),(1,1,2),(1,2,1),(2,1,1),(1,2,3)\}$$

and in general, the set of ordered n-tuples requires an internal representation of size exponential with respect to n. This is very inefficient, as can be seen from the prototype implementation of language from [Bojańczyk et al., 2012], where only very simple programs terminated within a reasonable time. It is worth noting that in the current semantics of the language, the set of all triples of atoms is represented by the more concise form:

$$\{(a_1,a_2,a_3) : \text{for } a_1,a_2,a_3\}.$$

Another problem with the `hull`-based definition of sets is the need to use constants that denote particular atoms, even if the mathematical definitions of these sets do not require it. For example, although the mathematical definition of triples of natural numbers does not refer to specific numbers, in the definition based on the `hull` of this set, given above, as many as three numbers are used. This is not a significant difficulty when considering equality or ordered atoms, but can be a serious problem for more sophisticated structures of atoms. For example, although the universal partial order [Hubička and Nešetřil, 2005] is a legal and well-behaved structure of atoms, no easy and natural representation of it is known and it is not clear how to denote its particular elements in a convenient way.

For these reasons, `hull`-based semantics has been replaced with logic-based semantics presented in Section 3.4. The theorems in the previous chapters, and in particular Theorem 8.1, show the lack of an explicit `hull` construction does not limit the expressive power of the language.

However, `hull` function and other support and orbit operations can be useful for a developer. Therefore, in our implementation the following functions have been added to the Nλ language:

`hull` : $[\mathbb{A}] \to \mathbb{S}\tau \to \mathbb{S}\tau$	(closure of set under automorphisms that fix atoms)
`groupAction` : $(\mathbb{A} \to \mathbb{A}) \to \tau \to \tau$	(renames free atoms in an argument)
`supports` : $[\mathbb{A}] \to \tau \to \mathbb{B}$	(checks if a list of atoms supports the argument)
`support` : $\tau \to [\mathbb{A}]$	(returns some finite support of the argument; efficient)
`leastSupport` : $\tau \to [\mathbb{A}]$	(returns the least support of the argument; less efficient)
`setOrbit` : $\mathbb{S}\tau \to \tau \to \mathbb{S}\tau$	(returns the orbit of an element in a set)
`setOrbits` : $\mathbb{S}\tau \to \mathbb{S}\mathbb{S}\tau$	(returns the (finite) set of orbits of a given set)

In [Bojańczyk et al., 2012], most of these functions or their minor variations were derived from `hull`. For example, one may write:

```
isSingleton s = exists (λx. forall (eq x) s) s
  supports c x = isSingleton (hull c (singleton x))
```

But as mentioned above, such definitions are rather inefficient. Therefore, in the current version of the language, all the above functions have been derived from the two efficient functions `groupAction` and `support`, which were added to the core language. These functions were realized in such a way that they both iterate over a given element and, respectively, map the atoms in it with a given action, or collect free atoms.

9.2. While programs with atoms

Nλ has its counterpart in the world of imperative languages called *while programs with atoms*. It allows the programmer to iterate over sets with atoms and execute instructions inside loops with possible side effects.

The concept of such a language was first introduced in [Bojańczyk and Toruńczyk, 2012]. The initial version (as [Bojańczyk et al., 2012] for Nλ) described sets with atoms through representatives of individual orbits. A more efficient design appeared in [Kopczyński and Toruńczyk, 2016, Kopczyński and Toruńczyk, 2017], where the idea of set expressions was first presented (see Definition 2.10).

The following language description is based on [Bojańczyk, 2019, Chapter 8].

While programs with atoms over a relational structure $\mathcal{A}$ are constructed using the following syntax:

- Assignment:

  $$x := \emptyset \qquad x := \mathcal{A} \qquad x := R \qquad x := y \cup z \qquad x := \{y\}$$

  The above expressions mean assigning a value to a variable. This value can be the empty set, the set of all atoms, or a relation derived from structure $\mathcal{A}$. Additionally, values can be composed by the set union or a singleton set with given element.

- Sequential composition:

  $$I; J$$

  For the given programs I and J, the above expression means a sequential call of program I, and then program J.

- Control flow:

  $$\texttt{if } x \ \delta \ y \texttt{ then } I \qquad \texttt{while } x \ \delta \ y \texttt{ do } I \qquad \texttt{for } x \texttt{ in } y \texttt{ do } I$$

  where x, y are variables, I is an already defined program, and δ is one of the operations: $\in$, $\subseteq$, $\subsetneq$, $=$, $\neq$. The semantics for `if` and `while` are as expected and follow commonly known imperative languages. But `for` requires more explanation. One can imagine that for each element x of the set y, a thread is created that executes the program I. After each thread computes the new state of the variables, the results are aggregated using set union (except for some special cases, see [Bojańczyk, 2019] for details). Note that the above description of threads is more logical than computational, due to the possible infinite number of threads. Comparing this construction to Nλ, one can see that it is a combination of our functions `map` and `sum`. It is also worth adding that the `while` loop may cause the program to diverge.

Using this syntax, it is possible to write a program that calculates any definable set. For example, a set of bounded open intervals

$$X \ = \ \{\{z : x < z \wedge z < y \text{ for } z \in \mathcal{A}\} : x < y \text{ for } x, y \in \mathcal{A}\}$$

is stored under variable X after executing the following code:

```
$X := \emptyset$
for $x$ in $\mathbb{A}$ do
  for $y$ in $\mathbb{A}$ do
    if $x < y$ then
      $Z := \emptyset$
      for $z$ in $\mathbb{A}$ do
        if $x < z \ \wedge \ z < y$ then
          $Z := Z \cup \{z\}$
      $X := X \cup \{Z\}$
```

It is worth noting that for finite sets a program written in a language such as Pascal, would look very similar. For comparison, the same set can be computed in Nλ by the program:

```
map
  (λ(x, y). (filter
              (λz. (and (x < z) (z < y)))
              atoms))
  atomsPairs
```

Now let us look at a more complex program that computes all the vertices of a directed graph (V, E) that can be reached from a given set $S \subseteq V$.

```
$Reach := S$
$New := \emptyset$
while $New \neq Reach$ do
  $Reach := New$
  for $v$ in $Reach$ do
    for $w$ in $V$ do
      if $(v, w) \in E$ then
        $New := New \cup \{w\}$
```

The above example implements the breadth-first search algorithm. And again, for a comparison, an implementation of the same algorithm in Nλ:

$$\begin{aligned}
\texttt{succesors}\ (V,E)\ S\ &= \texttt{map}\ \pi_2\ (\texttt{filter}\\
&\qquad\qquad (\lambda(v,w).\ \texttt{contains}\ E\ (v,w))\\
&\qquad\qquad (\texttt{pairs}\ S\ V))\\
\\
\texttt{reachable}\ (V,E)\ S\ &= \texttt{let}\ S' = \texttt{union}\ S\ (\texttt{succesors}\ (V,E)\ S)\\
&\quad\ \texttt{in if}\ (\texttt{eq}\ S\ S')\ S\ (\texttt{reachable}\ (V,E)\ S')
\end{aligned}$$

where π_2 is the projection of a pair onto the second component, and `pairs` is a generalization of the term (8.1) to a function that computes the Cartesian product for any two sets.

The implementation of the concept of *while programs with atoms* was presented in [Kopczyński and Toruńczyk, 2016, Kopczyński and Toruńczyk, 2017] and is called LOIS (Looping Over Infinite Sets). LOIS is a C++ library that operates on sets with atoms, including non-oligomorphic structures. It represents sets by first-order formulas over an underlying logical structure. To effectively handle such formulas, the language uses and implements SMT solvers for various first-order theories.

The difference between Nλ and *while programs with atoms* largely comes down to the difference between functional and imperative languages. It is worth noting that the imperative version is not strongly typed, which makes it easier to construct sets containing both atoms and other sets. But it does not have the advantage of strong typing in terms of greater security and protection from programming errors. Additionally, Nλ relies on pure functions, unlike *while programs* which allow side effects.

On the implementation level, the difference is in the choice of language to implement programs with atoms: Haskell or C++. Haskell is a statically typed purely functional programming language with type inference and lazy evaluation. It provides a concise, elegant, mathematical-like notation. On the other hand, C++ as an extension of the C language, object-oriented with some functional features, and it focuses on performance and memory management. Programs implemented in LOIS are usually more efficient than in Nλ.

9.3. Other alternatives

Computing with infinite objects has a long tradition in computer science. The most popular form of such calculations results from *lazy evaluation*. This idea, first proposed in [Wadsworth, 1971], is to delay the evaluation of an expression until it is needed. The benefits of this approach include the possibility of calculating the value of a function even if one of its arguments cannot be determined, and an increase in performance by avoiding the multiple calculation of the same value. Another advantage is the definition of infinite, or more precisely, potentially infinite structures. This means that logically certain structures are treated as infinite objects, but at a given moment only a finite part is actually constructed. For example, one can filter and map infinite lists, and then enumerate values on selected indexes. But the operation of checking membership or comparing two such lists can cause the program to diverge.

The approach used in Nλ and LOIS does not have these limitations, so all the operations mentioned above will return the result in a finite time. This is due to the representation of sets by formulas that are transformed by successive functions, but are always finite and can be evaluated. However here the limitations are that only *definable* sets can be handled. It is worth adding that the Haskell implementation of Nλ allows to use both concepts: lazy evaluation and sets with atoms.

A different way to represent sets with atoms (nominal sets) was used in a library called ONS (Ordered Nominal Sets) [Venhoek et al., 2019]. This tool allows to create and modify sets, but only for ordered atoms. Additionally, it does not rely on first-order formulas, but uses the representation theorem [Bojańczyk et al., 2014, Theorem 9.17]. Each orbit in the set is coded by a natural number, which is the size of the least support of any element of this orbit, and the orbit-finite set is coded by a multiset of such natural numbers. The representation is extended to equivariant maps and products. ONS, unlike Nλ and LOIS, is not a programming language that allows the programmer to operate naturally on infinite sets. Instead, it is a library that enables calculations on sets with ordered atoms (in particular, minimisation of automata and active automata learning) and achieves much better running times on certain classes of inputs.

Another tool that uses nominal sets (but only for equality atoms) is Mihda [Ferrari et al., 2005], which is an implementation of the minimization algorithm for history-dependent automata (HD-automata [Pistore, 1999]) defined using a polymorphic λ-calculus with dependent types. Mihda is implemented in OCaml and is based on *named sets*, which in turn are categorical equivalent to nominal sets [Gadducci et al., 2006].

Nominal techniques work very well for modeling name binding and α-conversion. A good example can be the results of the FreshML project [Pitts and Gabbay, 2000, Gabbay and Pitts, 2002, Shinwell et al., 2003]. FreshML is an extension of the functional programming language ML that facilitates metaprogramming by convenient and elegant constructs for declaring and manipulating syntactic structures of an object-level language.

When a program containing bind operations is analyzed in a metalanguage, the developer must implement the mechanisms for resolving variable name conflicts, creating fresh variable names and capture-avoiding substitution. All this to operate on equivalence classes of α-equivalence relation. Building these mechanisms from scratch is tedious and error prone.

FreshML, and then its extensive successor Fresh OCaml [Shinwell, 2005], introduces a binary type constructor $\langle\langle\, \alpha \,\rangle\rangle\, \beta$ for expressions of type β with bound names of type α. Values of this type are abstract and can be deconstructed by corresponding pattern matching. The language ensures that such an operation returns a result where each bound entity has a fresh name in the context of use and that there are no name conflicts. This construct is a built-in first-class citizen and allows the user to focus on the algorithmic side of metaprogramming. The whole setting relies on properties of sets with equality atoms.

A similar technique was used in the formalization of the lambda calculus in Nominal Isabelle – a definitional extension of the Isabelle/HOL, a proof assistant for higher-order logic [Urban, 2008, Huffman and Urban, 2010]. Also in this case equality atoms are used for representing variables that might be bound.

Finally, it is worth mentioning a different approach to computing on infinite structures, not related to nominal sets, but to codata [Downen et al., 2019]. This method also has limitations, and for data structures to be finitely represented, it is limited to regular coinductive types. An example is CoCaml [Jeannin et al., 2017], a functional programming language that extends OCaml and allows to define recursive functions on infinite structures if an equation solver is provided.

CHAPTER 10

Implementation

The previous chapters covered the theoretical part of Nλ. They described the syntax and semantics of the language, proved its properties, and showed how the language can be extended.

This chapter is a more practical part of the thesis: it describes an implementation of the language. Nλ was implemented in Haskell as a package that introduces the appropriate types, constants, functions and constructs described in Chapter 3 [Klin and Szynwelski, 2016]. Most of the language extensions described in Chapter 8 are provided by Haskell itself.

As for types, the package introduces basic element types presented in Section 3.1:

$$\tau ::= \mathbb{A} \mid \mathbb{B} \mid \mathbb{S}\tau$$
$$\alpha, \beta ::= \tau \mid \alpha \to \beta$$

In the implementation, element types appear as `Atom`, `Formula` and `Set a` (for some element type `a`), respectively. Haskell itself provides function types: `a -> b`.

The language constants listed in Section 3.2 are present in the implementation as follows:

```
empty :: Set a
atoms :: Set Atom
insert :: Nominal a => a -> Set a -> Set a
map :: (Nominal a, Nominal b) => (a -> b) -> Set a -> Set b
sum :: Nominal a => Set (Set a) -> Set a
true :: Formula
false :: Formula
not :: Formula -> Formula
(/\) :: Formula -> Formula -> Formula
(\/) :: Formula -> Formula -> Formula
isEmpty :: Set a -> Formula
ite :: Conditional a => Formula -> a -> a -> a
```

In addition, the constants that result from the relational structure of atoms are:

```
eq :: Nominal a => a -> a -> Formula
le :: Atom -> Atom -> Formula
```

The last constant only appears for ordered atoms.

Why some types for the above constants must be instances of the `Nominal` or `Conditional` classes? Why the relation constant `le` operates only on atoms, and the equality relation `eq` on more general types? All of this will be explained in the following sections. We will also describe formula solving, variant construction, and other examples or technical issues related to the language implementation.

But first, let us look at a simple example expression in Nλ, used as an example of reduction in (3.2):

```
Nλ> filter (\x -> le x a \/ le b x) atoms
{a₁ : b ≤ a₁ ∨ a ≥ a₁ for a₁ ∈ 𝔸}
```

Prompt `Nλ>` in the above example means that the program was run in an interactive Haskell environment (called GHCi) where language expressions can be interpreted and evaluated.

All such expressions are interpreted in the context of some logical structure (Section 2.1). Unless otherwise indicated, we assume that all subsequent examples are interpreted for ordered atoms.

10.1. Features of Haskell

Why was Haskell chosen as the language in which to implement Nλ? The most important reason, of course, is that Haskell is a functional language. Therefore, it provides the machinery of the lambda calculus that is the basis for Nλ. Additionally, Haskell provides many functionalities described in Chapter 8.

But beyond the above reasons, Haskell is a polymorphically statically typed, lazy, purely functional language of general use that has many advantages and stands out in many ways in the programming world. Many of the concepts commonly used today in other languages have been propagated by Haskell. The main reasons why is it worth using are listed below.

Purely functional

Programming in Haskell focuses on functions that are not only first-class citizens but also *pure*. This means that they are functions in a mathematical sense, i.e. their result depends only on the given inputs and they produce no side effects. Moreover, all data objects are *immutable* by default. This approach makes the code more readable and predictable, easier to test, more fault-tolerant and less error-prone.

Operations that are impure in nature are performed using *monads* [Lane, 1998, Moggi, 1991]. This allows us to still operate on pure functions, but in isolation from the impure context. Monads are a popular design pattern that is widely used to define calculations that contain additional processing rules, such as state transformation, concurrency, I/O or exception handling.

In addition to monads, Haskell offers many other higher-order functional constructs that are naturally built into the language.

Type system

One of Haskell's strengths is its *strong*, *static* type system. This means that each expression has a type that is determined at compile time. As a result, many errors are caught before the program starts running. An additional advantage is *type inference* which allows the user to decide whether to declare the type of a given expression or leave it to the system.

But that's not all, as the Haskell type system is very extensive. Enough to mention the support for parametric polymorphism [Milner et al., 1975], generalized algebraic data types [Xi et al., 2003, Cheney and Hinze, 2003], type classes (ad hoc polymorphism) [Wadler and Blott, 1997], type families [Schrijvers et al., 2008], type equality constraints [Orchard and Schrijvers, 2010], existential quantification [Läufer and Odersky, 1994], higher-rank polymorphism [Peyton Jones et al., 2007] or kind polymorphism [Yorgey et al., 2012].

Lazy evaluation

Lazy evaluation is one of Haskell's hallmarks. It relies on delaying the calculation of values until it is needed (non-strict evaluation) and in remembering the calculated results (sharing). The latter is possible due to the purity of functions. In addition to increased efficiency and the ability to operate on potentially infinite structures[1], this approach allows the programmer to easily define control structures and improves composability.

However, there is a cost associated with lazy evaluation: memory allocation becomes hard to predict and sometimes errors are more difficult to resolve due to the nondeterministic order in which the operations are performed.

Elegant and concise

Well-written Haskell code is concise and elegant. As it contains a lot of high-level concepts, it is short and therefore easier to understand and maintain. Much credit for this goes to the inspiration by mathematical notation:

`\x->x`	`f . g . h`	`[101 .. 200]`	`[3*x | x <- [0..], x^2 > 10]`
$\lambda x.x$ $x \mapsto x$	$f \circ g \circ h$	$\{101, \ldots, 200\}$	$\{3 \cdot x \mid x \in \mathbb{N},\ x^2 > 10\}$

Functions written in Haskell very often capture the essence of the algorithm at first glance:

```
quicksort :: (Ord a) => [a] -> [a]
quicksort []     = []
quicksort (p:xs) = quicksort less ++ [p] ++ quicksort more
    where less = filter (<p)  xs
          more = filter (>=p) xs
```

Ecosystem and community

Language development has a scientific basis. Haskell was invented by a committee of scientists, and its evolution is based on successive publications. Haskell has an active community that meets regularly at conferences. Many useful libraries and tools have been developed over the years. Many of them can be found in the central Hackage repository [Hackage, 2022]. The software can be easily searched using the Hoogle search engine [Hoogle, 2022]. Thanks to the extensive system of types, the desired function is often found by type. GHC [Haskell.org, 2022], the most popular language compiler, provides extensions to the standard version of the Haskell language.

[1]See also Section 9.3

10.2. Atom constants and atom variables

The basic element of the Nλ language is the atom. The core version of the language described in Chapter 3 operated on free and bound atom variables. The variables were bound by set expressions and quantifiers in first-order formulas. The latter binding does not occur in the language implementation, due to the elimination of quantifiers (see Section 10.5). Section 8.1 describes an extension of the basic language with atom constants.

Three types of atoms can be found in the language implementation:

- atom constants,
- user-defined atom variables,
- bound atom variables in set expressions.

Atom constants

An atom constant is a specific value from the domain of the relational structure of atoms. Depending on the selected structure, it can be a natural number (equality atoms) or a rational number (ordered atoms). Nλ provides a function called `constant` which creates an atom constant with the given value:

```
constant :: Constant -> Atom
```

As mentioned, type `Constant` depends on the selected logic[2]:

```
type Constant = Integer         -- for equality atoms
type Constant = Rational        -- for ordered atoms
```

So creating a constant is done by giving the value:

```
Nλ> constant 1
1
Nλ> constant (1/2 + 1/4)
3/4
```

User-defined atom variables

The user can declare an atom variable with a given name, which can take any value from the relational structure domain. A variable is declared using the function:

```
atom :: String -> Atom
```

For readability, variable names are limited to a certain character set:

```
Nλ> atom "a"
a
Nλ> atom "1"
*** Exception: variable name must start with a letter and contain only
↪  alphanumeric characters, dot, underscore or hyphen
```

[2]For equality atoms, integers were used, not natural numbers. But in this case, the structure only contains an equality relation, so any countable set is appropriate, and Haskell doesn't have a default type just for natural numbers.

Naturally, variables with the same name are always equal, and variables with different names can be equal or not. This is shown by the result of the equality relation:

```
Nλ> eq (atom "a") (atom "a")
true
Nλ> eq (atom "a") (atom "b")
a = b
```

The `eq` function returns a formula rather than a boolean value; this formula may hold or not, depending on the values of atom variables `a` and `b`.

It is possible to automatically generate fresh atom names using a type `AtomsSpace`. This is a monad that involves a state with the last used variable name. With the `newAtom` function, a new atom variable is produced and the monad state is updated. This allows us to implement programs that generate any number of fresh atoms. Such a program is then run using the `runNLambda` function:

```
type AtomsSpace = State [String]
newAtom :: AtomsSpace Atom
runNLambda :: AtomsSpace a -> a
```

A simple example of such a program looks like this[3]:

```
Nλ> :{
Nλ| runNLambda
Nλ| (do
Nλ|    a <- newAtom
Nλ|    s <- fmap singleton newAtom
Nλ|    return (a,s))
Nλ| :}
(a,{b})
```

From now on, when presenting code snippets that use atom variables `a`, `b`, `c`, we assume that such atoms have already been defined as follows:

```
Nλ> let a = atom "a"
Nλ> let b = atom "b"
Nλ> let c = atom "c"
```

and so on.

Bound atom variables in set expressions

The last type of atoms are bound variables in sets with atoms. These variables cannot be created directly by the user, but are generated automatically when a set is created.

```
Nλ> atoms
{a_1 : for a_1 ∈ 𝔸}
```

Bound variables are also given automatic names that have a special format that will not conflict with user-defined variables. The name consists a letter with an additional number as subscript. Letters correspond to the nesting level of the set, and numbers are used to enumerate bound variables in the set.

```
Nλ> map (\a -> map (\(b,c) -> (a,b,c)) atomsPairs) atoms
{{(a_1,b_1,b_2) : for b_1,b_2 ∈ 𝔸} : for a_1 ∈ 𝔸}
```

[3]Notation $:\{\ldots:\}$ defines commands with multiple lines in GHCi.

10.3. Relational structures and logic

The basic parameter of the Nλ language is a relational structure of atoms. Currently, support for two structures is implemented: equality atoms and ordered atoms. The choice of a structure affects two areas:

- a relational signature – values of atoms and relations between them,
- the logic of solving first-order formulas.

The first area covers the definition of the type `Constant` mentioned in the previous section. Additionally, it requires providing an instance of the following class, describing the signature of the relational structure:

```haskell
class AtomsSignature where

    -- Minimum list of relations from signature of atoms type
    minRelations :: [Relation]

    -- Returns text representation of given constant
    showConstant :: Constant -> String

    -- Returns constant for text representation
    readConstant :: ReadPrec Constant

    -- Default constant value
    defaultConstant :: Constant
```

The first method of this class returns a minimal list of relations that describe the structure: for equality atoms it will be $=$, and for ordered atoms: $=$ and $<$[4]. The next two functions are for showing atoms in the output and reading (and then parsing) them from the input. The last operation returns a default element from the set of atoms (zero for both structures)[5].

The second area is directly related to a first order formulas. The chosen relational structure should provide an instance of the following class:

```haskell
class AtomsLogic where
    -- Creates a formula representing ∃x.f
    existsVar :: Variable -> Formula -> Formula

    -- Creates a formula representing ∀x.f
    forAllVars :: Variable -> Formula -> Formula

    -- Returns conditions for all orbits for given list of variables
    exclusiveConditions :: [Variable] -> [Formula]

    -- Checks whether the formula is a tautology.
    isTrue :: Formula -> Bool

    -- Checks whether the formula is a contradiction.
    isFalse :: Formula -> Bool
```

[4]Needed to calculate the orbit in the `Nominal.Orbit` module (see Section 10.13).
[5]Used by the `element` function in the `Nominal.Set` module (see Section 10.12).

```
    -- Simplify given formula.
    simplifyFormula :: Formula -> Formula

    -- Returns a model for a given formula or report an error
    -- when formula is unsatisfied.
    model :: Formula -> Map Variable Variable
```

The `exclusiveConditions` function creates a list of formulas for a given list of atom variables that fully describe the relations between the variables and are mutually exclusive [6]. For example:

```
Nλ> exclusiveConditions [variable "a", variable "b", variable "c"]
[a ≠ b ∧ a ≠ c ∧ b ≠ c, b = c ∧ a ≠ b, a = b ∧ a ≠ c, a = c ∧ a ≠ b,
↪  a = b ∧ b = c]
```

Other class methods provide a set of tools for creating and resolving formula decision problems. The details of these operations are described in the following sections.

10.4. Logical formulas

Logical formulas are a key element of the Nλ language. Several language constructs rely on formulas: variants (Section 10.8), contexts (Section 10.7), conditional expressions (Section 10.9), sets (Section 10.12), etc.

The names of all formula modules begin with `Nominal.Formula`. The data type representing formulas is located in the module `Nominal.Formula.Definition` and is called `Formula`:

```
data Formula = Formula {isSimplified :: Bool,
                        formula :: FormulaStructure}
```

Each formula holds a value `isSimplified`, which indicates whether the given formula has been simplified by the SMT solver. This is to prevent the re-simplification of a formula that has already been simplified, which affects performance, especially with large and complex formulas.

Additionally, the type contains the structure of the formula itself. The structure of the formula is represented as follows:

```
data FormulaStructure
    = F
    | T
    | Constraint Relation Variable Variable
    | And (Set Formula)
    | Or (Set Formula)
    | Not Formula deriving (Eq, Ord)
```

As it can be seen from the structure above, the formulas are stored without quantifiers and therefore take the form of standard propositional logic. An atomic formula can be built from relation and atom variables.

Formulas can be constructed using functions provided by two modules: `Nominal.Formula.Constructors` and `Nominal.Formula.Operators`. There, we have constants with tautology and contradiction:

[6] Used to compute the size of a nominal set in the `Nominal.Set` module (see Section 10.12).

```
-- Represents the tautology formula
true :: Formula

-- Represents the contradiction formula
false :: Formula
```

We also have functions for creating relation constraints. Starting with the equality relation:[7]

```
-- Checks equivalence of two given elements
eq :: Nominal a => a -> a -> Formula
```

and the order relations available only for ordered atoms:

```
-- Creates a formula that describes the "<" relation between given atoms
lt :: Atom -> Atom -> Formula

-- Creates a formula that describes the "≤" relation between given atoms
le :: Atom -> Atom -> Formula

-- Creates a formula that describes the ">" relation between given atoms
gt :: Atom -> Atom -> Formula

-- Creates a formula that describes the "≥" relation between given atoms
ge :: Atom -> Atom -> Formula
```

Note that, although the signature of ordered atoms in theory provides only the relation $\leq$, the implementation for convenience, brevity and readability provides the full set of relations: $\leq, <, =, >, \geq$. This is possible thanks to the fact that each such relation is equivalent to some formula that contains only $\leq$.

Finally, logical operators are also available[8]:

```
-- Creates a logical conjunction of two given formulas
(/\) :: Formula -> Formula -> Formula

-- Creates a logical disjunction of two given formulas
(\/) :: Formula -> Formula -> Formula

-- Creates a negation of a given formula
not :: Formula -> Formula

-- Creates an implication of given formulas
(==>) :: Formula -> Formula -> Formula
(<==) :: Formula -> Formula -> Formula

-- Creates a formula representing biconditional relation
(<==>) :: Formula -> Formula -> Formula
```

Below are examples of building simple logical formulas:

```
Nλ> eq a b /\ lt a b
false
```

[7]The `eq` function allows one to compare not only atoms. More on this in Section 10.11.

[8]The precedence of these operators is defined with fixity declaration according to the standard precedence of logical operators in Haskell.

```
Nλ> eq a b <==> eq b a
true
Nλ> lt a b \/ eq c (constant 3) ==> not (gt a c)
a ≤ c ∨ (c ≠ 3 ∧ a ≥ b)
```

These examples show that already at the moment of creating formulas, some basic simplification takes place (without the use of an SMT solver).

10.5. Elimination of quantifiers

Recall that the logical structures considered in the context of Nλ are oligomorphic and homogeneous. As explained e.g. in [Bojańczyk, 2019, Theorem 7.6], such structures has quantifier elimination, i.e., every first-order formula over $\mathcal{A}$ is equivalent to a quantifier-free formula.

The implementation of Nλ provides support for two structures: equality atoms and ordered atoms. Both have quantifier elimination. If support for other structures is added, such elimination will be implemented by providing an instance of `AtomsLogic` class. Elimination of quantifiers for currently supported structures is implemented in the `Nominal.Formula.Quantification` module.

Elimination of quantifiers when building first-order formulas is important for two reasons. First, it ensures that we do not have atom variables bound by quantifiers, and consequently also the problem of resolving name conflicts of such variables does not occur. In addition, experience with the SMT solver used in the implementation shows that it does not deal well with quantified formulas that do not involve arithmetic.

The quantifier elimination algorithm used for ordered atoms used in Nλ implementation is based on the method of infinitesimals for linear real arithmetic proposed by [Loos and Weispfenning, 1993] and adapted by [Nipkow, 2008] to dense linear order. The same algorithm was used for equality atoms, only in a simpler form.

Suppose ϕ is a quantifier free formula with a free variable x. Additionally, assume that ϕ is in NNF (negation normal form), which in practice means no negation. The negation is pushed inwards to relation constraints, which means the relations are changed to symmetric ones. Let E be the set of those variables y which, due to some constraint in the formula ϕ, can be equal to x (e.g. $x = y, y = x, x \leq y, y \leq x$, etc.). Furthermore, let L be the set of those variables y which, due to a certain constraint, can be smaller than x ($x > y, y < x, x \geq y$ or $y \leq x$). Symmetrically, U is the set of variables possibly greater than x. With such assumptions, the following equality holds:

$$\exists x.\phi \quad = \quad \phi[-\infty/x] \;\vee\; \bigvee_{y \in E} \phi[y/x] \;\vee\; \bigvee_{y \in L} \phi[(y+\epsilon)/x]$$

where $-\infty$ is a fictitious value less than any other, and ϵ is infinitesimal. Therefore $y + \epsilon$ is less than any value greater than y. We assume the following constraint equivalence for every z:

$$\begin{array}{rclcrcl}
-\infty < z & \equiv & \top & \qquad & y+\epsilon < z & \equiv & y < z \\
-\infty \leq z & \equiv & \top & & y+\epsilon \leq z & \equiv & y < z \\
-\infty = z & \equiv & \bot & & y+\epsilon = z & \equiv & \bot \\
-\infty \geq z & \equiv & \bot & & y+\epsilon \geq z & \equiv & y \geq z \\
-\infty > z & \equiv & \bot & & y+\epsilon > z & \equiv & y \geq z,
\end{array}$$

and analogously when the values $-\infty$ and $y+\epsilon$ are on the right side of the relation. The left column with infinites are obvious. The equivalences in the right column can intuitively be understood to say that if the relation holds for all small enough epsilons, then the equivalent formula without epsilon also holds.

The quantifier elimination algorithm consists in replacing the variable x in formula ϕ with $-\infty$, y or $y+\epsilon$, and then replacing the constraints by equivalent ones. Its proof of correctness can be found in [Nipkow, 2008].

An analogous equation, and a corresponding algorithm, exist for the other side:

$$\exists x.\phi \quad = \quad \phi[\infty/x] \ \vee \ \bigvee_{y\in E} \phi[y/x] \ \vee \ \bigvee_{y\in U} \phi[(y-\epsilon)/x]$$

Thus, we can choose an option corresponding to a smaller set of lower L or upper U bounds.

For equality atoms, the sets U and L are empty, and infinities are fictitious values different from any other. In this case the algorithm is considerably simplified.

10.6. SMT Solver

A fundamental element of the Nλ computational model is the first-order formula satisfiability check; more precisely, checking whether a given formula is tautologically true or false. This type of check is required by the reduction rules (R14) - (R20), (R22) and (R23) of the language operational semantics.

The research field concerned with the satisfiability of formulas with respect to some formal theory is called *Satisfiability Modulo Theories* (SMT) [Barrett et al., 2009]. It is a generalization of the *Boolean satisfiability problem* (SAT) to first-order formulas interpreted within (“modulo”) a certain formal theory, such as the theory of real numbers or the theory of integer arithmetic (without multiplication). The number of theories under consideration is constantly growing and includes certain theories of arrays and strings, several variants of the theory of finite sets or multisets, the theories of finite, regular and infinite trees, the theories of several classes of lattices, lists, tuples, records, queues, hash tables, and bit-vectors. Tools that try to solve the SMT problem are called *SMT solvers* and have many practical applications in computer science, including in automated theorem proving, program analysis, program verification, and software testing.

One of the most popular and efficient SMT solvers is Z3, developed by Microsoft Research [De Moura and Bjørner, 2008]. Z3 is open source and is available on GitHub [Bjørner et al., 2021] under the MIT license. Z3 supports arithmetic, fixed-size bit-vectors, extensional arrays, datatypes, uninterpreted functions, and quantifiers.

Z3 solver was used in the implementation of Nλ, which intensively interacts with this checker to analyse formulas that arise in representations of infinite data structures. The integration uses system calls to the solver and is based on the SMT-LIB format [Barrett et al., 2010].

Depending on the selected relational structure, solving formulas is performed using the following logics:

- For equality atoms: `QF_LIA`

  Unquantified linear integer arithmetic. In essence, Boolean combinations of inequations between linear polynomials over integer variables.

- For ordered atoms: `QF_LRA`

  Unquantified linear real arithmetic. In essence, Boolean combinations of inequations between linear polynomials over real variables.

These logics are used in the version without quantifiers, thanks to the procedure of quantifier elimination described in Section 10.5.

The SMT solver is used by Nλ in four ways:

- to check if the formula is a tautology ($\mathcal{A} \vDash \phi$) – whether the negation of the formula is unsatisfiable,
- to check if the formula is a contradiction ($\mathcal{A} \nvDash \phi$) – whether it is unsatisfiable,
- to simplify the formula,
- to find a model for the formula, i.e. a valuation of free atom variables for which the formula is true.

The formula simplification process is important for readability of results and for program efficiency. Model finding is needed for a function that returns a chosen element of a set with atoms.[9]

An example of a program for equality atoms that calls the SMT solver is as follows:

```
Nλ> isTrue (eq a b /\ eq a c /\ neq b c)
False
```

Such a program checks whether a given formula is a tautology and sends a query to the SMT solver in the SMT-LIB format:

```
(set-logic QF_LIA)
(declare-const a Int)
(declare-const b Int)
(declare-const c Int)
(assert (not (and (= a b)(= a c)(not (= b c)))))
(check-sat)
```

The query indicates the logic to use, defines the three variables, and the formula that contains them. As one can see, the query checks satisfiability for the negation of the formula indicated in the program. The response below:

```
sat
```

means that the negation is satisfiable, therefore the formula is not a tautology.

[9]The `element` function in the `Nominal.Set` module (see Section 10.12).

In the next example, this time for ordered atoms, it is checked whether the formula is a contradiction:

```
Nλ> isFalse (lt a b /\ lt b c /\ lt c a)
True
```

The above program results in the following query to Z3:

```
(set-logic QF_LRA)
(declare-const a Real)
(declare-const b Real)
(declare-const c Real)
(assert (and (< a b)(< b c)(> a c)))
(check-sat)
```

In this case, we use the logic of real numbers and check the satisfiability of the formula given in the program. Since the formula is not satisfiable, we get the result:

```
unsat
```

Many Nλ functions simplify the formula when checking the satisfiability of a formula. But there is also the option of invoking the simplification procedure on its own with the `simplifyFormula` function:

```
Nλ> simplifyFormula (lt a b /\ lt b c ==> eq a c)
a ≥ b ∨ b ≥ c
```

The query that is sent is similar to the previous ones, except that it uses the `ctx-solver-simplify` tactic.

```
(set-logic QF_LRA)
(declare-const a Real)
(declare-const b Real)
(declare-const c Real)
(assert (or (= a c)(>= a b)(>= b c)))
(apply ctx-solver-simplify)
```

As a result of using the tactic, Z3 returns a list of formulas, which are called *goals*. In the case of simplification, it will always be one goal containing the simplified formula:

```
(goals
  (goal
    (or (>= a b) (>= b c))
    :precision precise :depth 1))
```

Of course, all interactions with SMT solver take place under the hood and the developer is not aware of them. However, if one would like to debug this integration, one can use the `Nominal.Formula.SmtLogger` module. After calling the `showSmtInfo` function, all requests and responses in SMT-LIB format are printed in the standard error stream. But one can also set a different way to log this information using the `addSmtHandler`, `removeSmtHandler` and `setSmtHandlers` functions.

Finally, it is worth mentioning that the integration with SMT solver relies on the following standard Haskell function:

```
unsafePerformIO :: IO a -> a
```

In Haskell, all I/O operations take place inside the `IO` monad. In this way, operations that, by their nature, may cause side effects are handled. The `unsafePerformIO` function removes this monad wrapper and therefore its use requires special caution. But in the case of the SMT solver this is a safe solution, because Z3 works in a deterministic way and thus can be treated as a pure function. Foreign function interface (FFI) is a common place when `unsafePerformIO` is used because Haskell does not know the properties of any external services used, and treats them as impure by default.

10.7. Contexts

The reduction rules of our operational semantics always evaluate in a certain context:

$$\psi \vdash M \rightarrow N$$

At the beginning, the context is trivial: $\psi = \top$. Only the reduction of subterms in a set expression expands the context (Rule (R21)).

The implementation of Nλ allows the programmer to define the context. If she wants to run her program in the context of a given formula, she can do it using the `when` function:

```
Nλ> when (lt a b) (ite (eq a b) atoms empty)
{}
```

The same program evaluated without a declared context would return the following result:

```
Nλ> ite (eq a b) atoms empty
{a₁ : a = b for a₁ ∈ 𝔸}
```

The function `when` is provided by instances of the `Contextual` class:

```
-- Class of types of expressions to evaluating with a given context.
class Contextual a where
    -- Evaluates an expression in the context of a given formula.
    when :: Formula -> a -> a
```

Thus, `when` takes a context formula and a value whose type is an instance of the `Contextual` class. The result of the function is a given value that can be simplified in a given context.

`Contextual` class instances are all common types in Haskell, from simple types to function types, lists, tuples, etc. For simple types, `when` just returns its second argument. For most complex types, the implementation of an instance is done automatically via the `deriving` statement, just as Haskell often does with classes such as `Eq` or `Show`. This is possible because the `Contextual` class provides a default implementation through the *Generics* mechanism [Magalhães et al., 2010].

The instance implementation for the `Formula` type is as follows::

```
instance Contextual Formula where
    when ctx f
        | isTrue ctx = simplifyFormula f
        | isTrue (ctx ==> f) = true
        | isTrue (ctx ==> not f) = false
        | otherwise = simplifyFormula $ mapFormula (when ctx) f
```

For a trivial context, the same formula is returned, but in a simplified form. If in the given context the formula is always true or false, it is replaced by the corresponding trivial formula. In other cases, the sub-formulas are processed in the given context, and finally the formula is simplified.

This means that running the function `when`, even with a trivial context, recursively simplifies all the formulas in the structure. Therefore, the following function works as expected:

```
-- Evaluates an expression in the context of a 'true' formula. In practice
↪  all formulas in expressions are simplified.
simplify :: Contextual a => a -> a
simplify = when true
```

10.8. Variants

The core language in Chapter 3 contains a variant construct that is limited to atoms:

$$a_1 : \phi_1 | \cdots | a_n : \phi_n$$

Chapter 8, in turn, describes the extension of the language with primitive data types, which can also be used in conditional expressions, and therefore there is a need to generalize Rule (R17) and to extend the variant with such types. An implementation of this more general construct can be found in the `Nominal.Variants` module.

```
-- Storing values under various mutually exclusive conditions.
newtype Variants a = Variants (Map a Formula) deriving (Eq, Ord)
```

As can be seen from this definition, the type that can be stored in a variant is not limited to simple types only. Technically, the above definition restricts type `a` to types that are instances of the `Eq` or `Ord` classes. This is due to the requirements for operations on `Data.Map`. But in fact, variants are mainly created in *if... then... else...* statements, and therefore there is no need to create variants for types like functions, formulas, sets with atoms, or tuples. How to handle such types is described in the next section.

The existence of the type `Variants a` is somewhat incompatible with the semantics of the Nλ language. The typing rules (Section 3.6) assume that a variant has a type equal to the type of the stored values. However, there is no easy way to implement this concept in Haskell without interfering with the compiler.

One may wonder why the variant is stored as a map from type `a` to `Formula`, and not as a list of (value, condition) pairs. The map facilitates handling of cases where we have two variants with the same value but different conditions: $M : \phi'$ and $M : \phi''$. In this case, we can combine these two variants into one: $M : \phi' \vee \phi''$.

The `Atom` type in the Nλ implementation is also defined as a variant:

```
type Atom = Variants Variable
```

Even for a single atom variable, a degenerated variant $(a : \top)$ is created.

```
-- Creates atom with the given name.
atom :: String -> Atom
atom = variant . variable
```

where the function `variant` is defined as follows:

```
-- Creates a single variant.
variant :: a -> Variants a
variant x = Variants $ Map.singleton x true
```

The same is true for atom constants:

```
-- Creates atom representing given constant
constant :: Constant -> Atom
constant = variant . constantVar
```

10.9. Conditional expressions

In classical programming, a conditional expression is a simple statement in which one of two values (in typed languages: of the same type) is chosen depending on the value of a boolean condition. This is consistent with the denotation of the constant `if` (D15) in denotational semantics.

On the other hand, in Nλ, conditional expressions require as many as six reduction rules (R14 – R20) in operational semantics and their behavior largely depends on the type of possible values. This is because the formula condition in this expression can be other than true or false. If this formula is true or false, then regardless of type, the behavior is the same as the classical conditional expression (Rule (R14) and Rule (R15)). This can be seen in the definition of the `ite` function [10], which is equivalent to the `if` constant in operational semantics.[11]

```
-- if ... then ... else... with condition solving.
ite :: Conditional a => Formula -> a -> a -> a
ite c x1 x2
    | f == true  = x1
    | f == false = x2
    | otherwise  = cond f x1 x2
  where f = simplifyFormula c
```

In the implementation of this function, the formula is simplified first. If the formula is trivial after the simplification, the first or second value of the conditional expression is returned, respectively. Otherwise, functionality is delegated to the function `cond`, which is an operation of the class `Conditional`, and its behavior depends on the type `a`.

```
-- Class of types implementing conditional statements.
class Conditional a where
    -- Join two values or returns two variants for undetermined condition
    ↪  in conditional statement.
    cond :: Formula -> a -> a -> a
```

The `Conditional` class, like `Contextual`, provides a default implementation through the *Generics* mechanism. Therefore, instances of this class for most algebraic data types are made through the `deriving` functionality. The default implementation delegates the call of the function `cond` to subtypes. If we wanted to limit such a procedure to a pair, the following code would come out:

[10] The name `ite` comes from the first letters of the *if ... then ... else ...* statement.
[11] The keyword `if` is reserved in Haskell and functions cannot be called that.

```
instance (Conditional a, Conditional b) => Conditional (a,b) where
    cond c (x1,x2) (y1,y2) = (cond c x1 y1, cond c x2 y2)
```

The implementation of the function `cond` for the basic types of the Nλ language follows the reduction rules (R16 – R20). For example, for the function type (Rule (R16)) we have:

```
instance Conditional b => Conditional (a -> b) where
    cond c f1 f2 x = cond c (f1 x) (f2 x)
```

In the case of a formula, the behavior of the function follows directly from Rule (R18):

```
instance Conditional Formula where
    cond f1 f2 f3 = (f1 /\ f2) \/ (not f1 /\ f3)
```

However, it is not possible to implement the function `cond` in such a simple way for all types. For example, there is no obvious implementation of the function `cond` for lists. In the case of lists of the same length, a tactic similar to the pair example above can be used. But when the function is called, we are not sure if these lists will actually be of the same length. Moreover, simple types, as has already been mentioned many times, need to create a variant structure. Therefore, numeric types or a list type are not an instance of the `Conditional` class, and the `ite` function cannot be applied to them.

Of course, the type `Variants` is an instance of the class `Conditional` (according to Rule (R20)) and therefore for atoms (which, as a reminder, are always variants of atom variables), we can also use the `ite` function:

```
Nλ> ite (eq a b) a b
a : a = b | b : a ≠ b
```

But for types that do not have a `Variants` type constructor in their definition, a special version of the `ite` conditional function was created:

```
-- If ... then ... else... for types that are not instances of
↪  'Conditional' class.
iteV :: Ord a => Formula -> a -> a -> Variants a
iteV c x1 x2 = ite c (variant x1) (variant x2)
```

The `iteV` function wraps the given values with variants before calling `ite`. And now it is possible to create conditional expressions also for simple types:

```
Nλ> iteV (eq a b) 1 2
1 : a = b | 2 : a ≠ b
```

By default, Haskell handles a conditional expression not as a function, but through the following syntax:

```
if <condition> then <true-value> else <false-value>
```

where `<condition>` is of type `Bool`.

Fortunately, Haskell is a language that allows changing the default behavior through its numerous extensions. One such extension is `RebindableSyntax` [Haskell.org, 2022], which allows to rewrite own version of the operators in the `Prelude`, even with changing types. In order to override the *if... then... else...* statement, one need to define `ifThenElse` function. The definition of such a function can be found in the module `Nominal.If`.

```
-- Class of types allowed in a conditional expression
class IfThenElse c a where
  ifThenElse :: c -> a -> a -> a

-- Conditional expression with a formula condition
instance Conditional a => IfThenElse Formula a where
  ifThenElse = ite

-- Conditional expression with a bool condition
instance IfThenElse Bool a where
  ifThenElse True a _ = a
  ifThenElse False _ b = b
```

After enabling the `RebindableSyntax` extension, one can use a conditional construction also if the condition is a formula. Below is an example with the type of set with atoms according to Rule (R19):

```
Nλ> :set -XRebindableSyntax
Nλ> if eq a b then atoms else empty
{a₁ : a = b for a₁ ∈ 𝔸}
```

10.10. Maybe, Either

Haskell, apart from basic primitive types, has a whole range of data types that wrap certain values. The most popular of them are `Maybe` and `Either`. The former is used for an optional values and the latter is used for storing one of the two possible values of different types. However, these types cannot be directly used in conditional expressions because they are not instances of the `Conditional` class.

There are two ways to deal with it: use the `iteV` function (see Section 10.9), which wraps the given values into variants, or use types prepared in Nλ, which are already variants:

```
type NominalMaybe a = Variants (Maybe a)
type NominalEither a b = Variants (Either a b)
```

Additionally, functions for creating and operating on these types are provided:

```
nothing :: NominalMaybe a
just :: a -> NominalMaybe a
left :: a -> NominalEither a b
right :: b -> NominalEither a b
```

An example of using the `NominalMaybe` type in a conditional expression may look like this:

```
Nλ> ite (eq a b) (just c) nothing
Nothing : a ≠ b | Just c : a = b
```

Similar nominal versions of types can be created for other data types available from Haskell or user-defined.

10.11. Nominal types

Recall that in the core language Nλ we divided types into element types and function types (Section 3.1):

$$\tau ::= \mathbb{A} \mid \mathbb{B} \mid \mathbb{S}\tau$$
$$\alpha, \beta ::= \tau \mid \alpha \to \beta$$

Element types were denoted with the letter τ and sets with atoms could only contain elements of such types.

The reason for this division was mainly technical and was due to duplicate detection capability and performance considerations. In Haskell, items in collections such as `Data.Set` or `Data.Map` must be an instance of the `Ord` class that has methods to compare items. In the Nλ implementation, objects that can be elements of a set with atoms (that is, element type objects) must also be an instance of a certain class. This class is called `Nominal` and its definition is as follows:

```
-- Basic type in 'NLambda' required by most of functions in the module.
-- The Ord instance is used for efficiency.
class Ord a => Nominal a where
    -- Checks equivalence of two given elements.
    eq :: a -> a -> Formula
    -- If "a" is a variant type then returns variants values.
    variants :: a -> Variants a
    -- Map all variables from a given scope.
    mapVariables :: MapVarFun -> a -> a
    -- Fold all variables form a given scope.
    foldVariables :: FoldVarFun b -> b -> a -> b
```

Most importantly, for a given type to be an instance of the `Nominal` class, it must also be an `Ord` instance. The implementation of sets with atoms is based on the `Data.Set` and `Data.Map` structures, and the requirements of the `Ord` class are inherited from there. Comparing elements allows to perform many operations in these structures, such as searching for elements in logarithmic time instead of linear time.[12] As we know, a function type is not an instance of the `Ord` class, so it cannot be an instance of the `Nominal` class either.

Let us move on to the methods of the `Nominal` class. The first method is the function `eq :: Nominal a => a -> a -> Formula`, which is an implementation of the function of the same name from the core language. Recall that in Section 3.2 we had function definitions of $\mathtt{eq}_{\mathbb{A}}$, $\mathtt{eq}_{\mathbb{B}}$ and $\mathtt{eq}_{\mathbb{S}\tau}$, which together gave us the functionality to check for equality for all element types:

$$\mathtt{eq} : \tau \to \tau \to \mathbb{B}$$

The implementation of `eq` method of the `Nominal` class for the types listed in the core language is the same as the definitions of the `eq` function in Section 3.2. For example, for formulas we have:

[12]The `Data.Set` and `Data.Map` structures have logarithmic time complexity. There are hashed structures that have amortized constant average complexity, but for Nλ, the cost of these operations is negligible compared to the α-conversion and calls to the SMT solver.

```
instance Nominal Formula where
    eq = iff
    ...
```

that is, the method `eq` in this case is a logical equivalence relation ("if and only if") available in the Nλ implementation also as an operator (`<==>`). Whereas for sets with atoms, the function `eq` is checking the subset relation in both directions:

```
instance Nominal a => Nominal (Set a) where
    eq s1 s2 = isSubsetOf s1 s2 /\ isSubsetOf s2 s1
    ...
```

For simple types that do not contain atoms, the function `eq` is identical to the method `(==)` of the standard class `Eq`. On the other hand, for complex types, the function `eq` is often defined recursively and is based on results of this function for substructures.

As for the `Contextual` and `Conditional` classes, the `Nominal` class uses the *Generics* mechanism to provide a default implementation (including the function `eq`), so we can provide an instance for most of the data types in Haskell thanks to the `deriving` functionality.

Another method of the `Nominal` class is the `variants` function. This function exists for internal purposes and is used when implementing operations on sets with atoms. It allows the following reduction rule to be implemented, which could be added to the operational semantics of the core language:

$$\begin{aligned}&\psi \vdash \{\ldots, (M_1 \colon \phi_1 | \cdots | M_n \colon \phi_n) \colon \phi \text{ for } \sigma, \ldots\}\\ &\quad \rightarrow \{\ldots,, M_1 \colon \phi_1 \wedge \phi \text{ for } \sigma_1, \ldots, M_n \colon \phi_n \wedge \phi \text{ for } \sigma_n, \ldots\}\end{aligned}$$

Thanks to this rule, variants are dissolved as soon as they emerge as elements of set expressions. Such a reduction increases the readability of the result because this results in fewer nested structures.

Almost all instances of the `Nominal` class can use the default implementation of the `variants` function, which for a given term M returns a singleton variant $M \colon \top$. Only the data type (`Variants a`) for this function returns a list of all its variants.

The last two methods of the `Nominal` class are:

```
mapVariables :: Nominal a => MapVarFun -> a -> a
foldVariables :: Nominal a => FoldVarFun b -> b -> a -> b
```

The types `MapVarFun` and `FoldVarFun` used here have the following definition:

```
-- Variables scope.
data Scope
    -- All variables.
    = All
    -- Only free variables.
    | Free

-- Map function for variables from a scope.
type MapVarFun = (Scope, Variable -> Variable)
-- Fold function for variables from a scope.
type FoldVarFun b = (Scope, Variable -> b -> b)
```

These two methods of the Nominal class are needed for α-conversion of atom variables. Some reduction rules require that sets of atom variables in reduced terms be disjoint. This is the case of the reduction rule for the map function (R8) and for the sum function (R9). Therefore, we need functions that will allow us to rename atom variables and retrieve a collection of atom variables.

The mapVariables function enables the first of these functionalities. Through the second element of the MapVarFun pair, we can indicate a function that will replace the given atom variables in the object or structure. The scope of this change can be limited to free atom variables or performed on all variables (using the Scope data type).

The foldVariables function allows to aggregate atom variables of an object or structure into a selected accumulator. The operation of this function can also be limited to a certain scope. Therefore, this function can be used to compute the $\mathrm{FV}_{\mathbb{A}}(M)$ and $\mathrm{V}_{\mathbb{A}}(M)$ sets for a given term M as defined in Section 3.4. For example, the set of free atom variables can be computed as follows:

```haskell
freeVariables :: Nominal a => a -> Set Variable
freeVariables = foldVariables (Free, insert) empty
```

10.12. Sets

The implementation of sets with atoms can be found in the Nominal.Set module. Recall that a set with atoms for a given relational structure $\mathcal{A}$ is a set from the cumulative hierarchy over $\mathcal{A}$ which is hereditarily finitely supported (Definition 2.4) and is represented by a set expression of the following form:

$$\{M_1 : \phi_1 \text{ for } \sigma_1, \ldots, M_n : \phi_n \text{ for } \sigma_n\}$$

So the representation takes the form of a list of values that are conditioned by some formula, and a list of bound atom variables. The type denoting such a pair (set of bound variables σ, formula ϕ) is defined in the implementation as follows:

```haskell
type SetElementCondition = (Data.Set.Set Variable, Formula)
```

where Data.Set.Set is the standard finite set from the Data.Set module. The use of the qualified name Data.Set.Set is due to a name conflict with the type Set, which in the module Nominal.Set means a set with atoms:

```haskell
newtype Set a = Set {setElements :: Map a SetElementCondition}
```

As can be seen from the above definition, the set with atoms is represented as a map (from the module Data.Map) from the values of the elements of the set to their conditions. Due to the fact that the map's keys are the values of the element, one can be sure that the set with atoms does not contain duplicates.

The newtype construct is used here because the set type has exactly one constructor and exactly one field. In that case, newtype is faster than data because after the type is checked at compile time, compiler can remove type constructor Set and there is no wrap or unwrap overhead at run time.

In the Nominal.Set module, the Set type has been implemented as an instance of the following classes: Eq, Ord, Show, Read, Conditional, Contextual, Nominal.

The functions that create and modify the above structure of a set in the form of a map are:

```
-- Returns an empty set.
empty :: Set a

-- Returns the set of all atoms.
atoms :: Set Atom

-- Insert an element to a set.
insert :: Nominal a => a -> Set a -> Set a

-- Delete an element from a set.
delete :: Nominal a => a -> Set a -> Set a

-- Applies function to all elements of a set and returns a new set.
map :: (Nominal a, Nominal b) => (a -> b) -> Set a -> Set b

-- Filter all elements that satisfy the predicate.
filter :: Nominal a => (a -> Formula) -> Set a -> Set a

-- For a set of sets returns the union of these sets.
sum :: Nominal a => Set (Set a) -> Set a

-- Checks whether the set is not empty.
isNotEmpty :: Set a -> Formula
```

Almost all other set functions do not have direct access to the internal representation of the set, but are defined using the above functions. This is consistent with the language definition in Section 3.2. The definition of the core language included the above functions in its grammar, while the remaining functions used the basic ones. The above is true up to the following differences.

- The `delete` function was not part of the core grammar, but it was defined with `filter` and `eq` functions. In the implementation, this function first tries to remove an element directly from the map, looking for that element based on a comparison function from the `Ord` class. And then it filters with the `eq` function.

- The `filter` function is also not part of the core language grammar, it is defined using core functions. However, implementing this frequently used function directly on the internal representation of the set improves performance.

- The implementation of the `isNotEmpty` function instead of `isEmpty` as a core function is due to the prosaic reason that the formula $\bigvee_{1 \le i \le n} \exists \sigma_i . \phi_i$ is slightly simpler than $\bigwedge_{1 \le i \le n} \forall \sigma_i . \neg \phi_i$.

Using the above functions, the module includes functions for calculating intersection, sum, difference, membership, subset relation, partition, creating a set from a list etc. It also provides functions to build a set of tuples or lists. So, by analogy to the standard `zip`, `zipWith`, `replicate`, we have the following functions and their different variants:

```
-- Creates a set of pairs of elements from two sets.
pairs :: (Nominal a, Nominal b) => Set a -> Set b -> Set (a, b)
```

```
-- Creates a set of results of applying a function to elements from two
↪  sets.
pairsWith :: (Nominal a, Nominal b, Nominal c) => (a -> b -> c) -> Set a ->
↪  Set b -> Set c

-- Creates a set of triples of elements from three sets.
triples :: (Nominal a, Nominal b, Nominal c) => Set a -> Set b -> Set c ->
↪  Set (a, b, c)

-- Creates a set of results of applying a function to elements from three
↪  sets.
triplesWith :: (Nominal a, Nominal b, Nominal c, Nominal d)
    => (a -> b -> c -> d) -> Set a -> Set b -> Set c -> Set d

-- Creates a set of lists of a given length of elements from a set, e.g.
replicateSet :: Nominal a => Int -> Set a -> Set [a]
```

The above functions can be used for example as follows:

```
Nλ> pairs atoms atoms
{(a₁,a₂) : for a₁,a₂ ∈ 𝔸}
Nλ> replicateSet 3 atoms
{[a₁,a₂,a₃] : for a₁,a₂,a₃ ∈ 𝔸}
```

We also have a `size` function analogous to the `length` function for a list.

```
-- Returns a variants of numbers of the size of a set.
-- It is an inefficient function for large sets and will not return the
↪  answer for the infinite sets.
size :: (Contextual a, Nominal a) => Set a -> Variants Int
```

And just as the `length` function does not return a result (it runs forever) for infinite lists (`length [1..]`), the `size` function does not return a result for infinite sets (`size atoms`). The `size` function returns the result as variants, because the size of the set can depend on formulas, for example:

```
Nλ> size $ fromList [a, b, c]
1 : a = c ∧ b = c | 2 : (b = c ∧ a ≠ c) ∨ (b ≠ c ∧ (a = b ∨ a = c)) | 3 :
↪  a ≠ b ∧ a ≠ c ∧ b ≠ c
```

Since the `size` function does not return a number, the `maxSize` function was also created for convenience:

```
-- Returns the maximum size of a set for all free atoms constraints.
maxSize :: (Contextual a, Nominal a) => Set a -> Int
```

Another interesting function is the `element` function, which returns some element of a given set.

```
-- Returns some given element of a set or 'Nothing' if the set is empty.
-- The function report error if the set has condition with constraint
↪  between free and bound atom variable
element :: (Contextual a, Nominal a) => Set a -> NominalMaybe a
```

This function was added to the language because it is useful for implementing orbit representatives (see Section 10.13).

The mechanism of computing the set element is based on the `model` function, which queries the SMT solver for a model for a given formula (see Section 10.6). The following are examples of using the function:

```
Nλ> element empty
Nothing
Nλ> element $ singleton a
Just a
Nλ> element atoms
Just 0
Nλ> element differentAtomsPairs
Just (0,1)
Nλ> element $ filter (lt a) atoms
*** Exception: Cannot get element from set with constraint free and bound
↪  atom variable
```

As one can see from the last example, the function has some limitations. In this case, the specific element cannot be returned without evaluating the atom variable `a`. If we have a specific value instead of a variable, the function returns a result:

```
Nλ> let a = constant 7
Nλ> element $ filter (lt a) atoms
Just 8
```

Finally, it is worth mentioning that the module `Nominal.Set` for a set that contains only atoms, i.e. of the type `Set Atom`, provides functions for computing ranges, checking maximum, minimum, infimum, supremum or whether a set is open, closed, connected or compact.

10.13. Supports and orbits

The theory of sets with atoms is strongly based on two concepts: support (see Section 2.3) and orbit (see Section 2.6). The concept of support is needed to define a set with atoms. Recall that by Definition 2.4 every set with atoms has finite support, its elements have finite supports, and so on recursively. Thanks to the concept of orbit, we define the property of hereditarily orbit-finiteness (as defined in Section 2.6). And this property is required for a set with atoms to be represented by a set expression (Theorem 2.11).

Functions that enable the calculation of supports and orbits can be found in the `Nominal.Orbit` module. First, we have a function that returns a certain finite support of an element.

```
-- Returns all free atoms of an element. The result list is also support of
↪  an element, but not always the least one.
support :: Nominal a => a -> [Atom]
```

This function only calls the `freeVariables` function (see end of Section 10.11) and returns the result as a list.

It was mentioned in Section 2.3 that equality atoms and ordered atoms have the property that every finitely supported set has the least finite support with respect to inclusion. The following function is used to calculate such the least support:

```
-- Returns the least support of an element. From the result of 'support'
↪  function it removes atoms and check if it is still support.
leastSupport :: Nominal a => a -> [Atom]
```

The implementation begins with the output of the `support` function and removes subsequent atoms to check if the list is still a support after this removal. Whether the list is a support can be checked using the function:

```
-- Checks whether a list of atoms supports an element.
supports :: Nominal a => [Atom] -> a -> Formula
```

The implementation of this function checks, for a given support candidate S, whether the S-orbit of a given element is a singleton. If an element is supported by the empty list, it is called equivariant (see Section 2.5). This can be checked using the function:

```
-- Checks whether an element is equivariant, i.e. it has empty support.
isEquivariant :: Nominal a => a -> Formula
```

Let us apply the above functions in a few simple examples. Since all sets from Example 2.5 are equivalent, the set of pairs of atoms in particular also has an empty support.

```
Nλ> leastSupport atomsPairs
[]
```

But the least support for a set of atoms that belong to some interval must contain elements that are at the end of that interval.

```
Nλ> range a b
{a₁ : a ≤ a₁ ∧ b ≥ a₁ for a₁ ∈ 𝔸}
Nλ> leastSupport (range a b)
[a,b]
```

We know that a single atom is only supported by the list that contains that atom. Hence:

```
Nλ> supports [a] b
a = b
```

The support calculation is based on the `freeVariables` function, which is based on the `foldVariables` method from the `Nominal` class (see Section 10.11). The second method of this class, `mapVariables`, is used to calculate the orbit. To do this, we first implement a function that applies an automorphism to this element:

```
-- Applies permutations of atoms to all atoms in an element.
groupAction :: Nominal a => (Atom -> Atom) -> a -> a
```

This is an auxiliary function for the function that returns the S-orbit of a given element for a given finite list S:

```
-- Returns an orbit of an element with a given support.
orbit :: Nominal a => [Atom] -> a -> Set a
```

The algorithm that computes the result of the function for a given support S and an element e is as follows:

- First calculate the support S_e of the element e. Suppose S_e is a list of length n.
- Create the set of all n-element lists of atoms. For example for $n = 5$:

  ```
  Nλ> replicateAtoms 5
  {[a₁,a₂,a₃,a₄,a₅] : for a₁,a₂,a₃,a₄,a₅ ∈ 𝔸}
  ```

- Filter this set of lists in such a way that only those lists of atoms remain for which, for any two indices i and j, we have the same relation between a_i and a_j as between atoms on the same positions in the support S_e.[13]
- Additionally, filter this set to obtain lists in which the same relations hold between each atom a_i and the elements of the support S, as between the i-th atom from the support S_e and elements from S.
- For a given list of atoms, let define an automorphism that will replace the atoms from the support S_e occuring in the element e with the corresponding atoms from this list.
- The result of the function `orbit` is the set of elements that arise by applying automorphisms for each list of atoms from the filtered set to the element e.

We can see how the function `orbit` works on a few examples.

```
Nλ> orbit [a] a
{a}
Nλ> orbit [] (constant 1, constant 2)
{(a₁,a₂) : a₁ < a₂ for a₁,a₂ ∈ 𝔸}
Nλ> orbit [constant 2] [constant 1, constant 3]
{[a₁,a₂] : a₁ < a₂ ∧ a₁ < 2 ∧ a₂ > 2 for a₁,a₂ ∈ 𝔸}
```

The first example shows that if S is a support for the element e, we obtain a singleton set. In the second example, the `orbit` function for a pair of atoms where the first atom is smaller than the second atom generates all pairs of this type. In the third example, we have a two-atom list where the first atom is less than 2 and the second is greater. Since $S = [2]$, the result is the set of all two-atom lists with this property.

The remaining functions from the module `Nominal.Orbit` return the partition of a given set into orbits, the number of orbits, equivariant subsets or check whether two elements are in the same orbit. We also have an implementation of the function `hull` that was the basis of the predecessor language to Nλ (see Section 9.1).

Finally, it is worth mentioning the `setOrbitsRepresentatives` function, which using the `element` function (see Section 10.12) returns a list of representatives of all orbits of the set.

```
Nλ> setOrbitsRepresentatives atomsTriples
{Just (-1,-1,0), Just (-1,0,-1), Just (-1,0,0), Just (-1,0,1), Just
↪  (0,-1,-2), Just (0,-1,1), Just (0,1,-1), Just (0,2,1), Just (1,0,0),
↪  Just (1,0,1), Just (1,1,0), Just (1,1,1), Just (2,0,1)}
```

10.14. Show, Read

All data types available in Nλ are instances of the `Show` and `Read` classes. The first of these classes provides the `show` function, which returns a textual representation of the given value. The second one uses the `read` function to parse the text presentation and return the original value.

The `show` function is implicitly used every time the result is displayed in the console. Examples of using the `read` function for a set or variant are as follows:

[13] The function `minRelations` of the `AtomsSignature` class is used for this operation (see Section 10.3).

```
Nλ> read "{(a₁,a₂) : a₁ ≠ a₂ for a₁,a₂ ∈ 𝔸}" :: Set (Atom, Atom)
{(a₁,a₂) : a₁ ≠ a₂ for a₁,a₂ ∈ 𝔸}
Nλ> read "Nothing : a ≠ b | Just c : a = b" :: NominalMaybe Atom
Nothing : a ≠ b | Just c : a = b
```

There are non-ASCII characters in the textual presentation of atom relations, formulas or sets. In particular, there are symbols: ≤, ≠, ≥, ¬, ∨, ∧, ∈, 𝔸. If in a certain runtime there is a problem with displaying Unicode characters, one can compile the program with the `DISABLE_UNICODE` flag, which will cause the results to be presented only as ASCII characters:

```
Nλ> differentAtomsPairs
{(a_1,a_2) : a_1 /= a_2 for a_1,a_2 in A}
Nλ> le a b \/ neq c d
a <= b \/ c /= d
```

10.15. Installation Guide

The Nλ source code is available in the GitHub repository [Szynwelski, 2022b]. The language is implemented as the `NLambda` package in Haskell. The easiest way to get the language source code is to clone the repository with the command:

```
git clone https://github.com/szynwelski/nlambda.git
```

To compile the code one need to install the GHC compiler [Haskell.org, 2022] and to install the package, one can use the `cabal` software [Cabal, 2022].

To build and install the `NLambda` package, execute the following commands in the repository root directory:

```
cabal v2-configure -fTOTAL_ORDER
cabal v2-build
cabal v2-install
```

The `TOTAL_ORDER` flag means that the package will be installed for ordered atoms. The lack of the flag means that the equality atoms are selected.

The above installation process can also be performed using dedicated scripts:

- `install-equality.sh` – for equality atoms
- `install-total-order.sh` – for ordered atoms

Additionally, SMT solver Z3 installation is required for full functionality of Nλ [Bjørner et al., 2021]. After installing the solver, it should be added to the `PATH` environment variable.

The fastest way to check the functionality of an installed package is to use the GHCi interactive environment. There is a `.ghci` configuration file in the main repository directory, which imports the `NLambda` package and hides functions from the default `Prelude` module whose names conflict with the function names in Nλ. Option `RebindableSyntax` is also set which allows the use of the *if... then... else...* construction for conditions that are formulas. Additionally, the `Nλ>` prompt is set and example atoms for the letters 'a' to 'e' are defined.

The full content of the `.ghci` file looks like this:

```
:set -XNoImplicitPrelude
:set -XRebindableSyntax
:m NLambda
:set prompt "Nλ> "
import Prelude hiding (or, and, not, sum, map, filter, maybe)
let [a,b,c,d,e] = fmap atom ["a","b","c","d","e"]
```

If the package installation was successful, then after running the `ghci` command in the main directory of the repository, one should be in an environment in which one can evaluate Nλ expressions.

Additional information on the language can be found on the dedicated Nλ website [Szynwelski, 2022c]. The page contains a reference to the full documentation of `NLambda` package [Szynwelski, 2022a].

CHAPTER 11

Use cases

Previous chapters have described how the language Nλ looks, how it works and how it was implemented. The last chapter will show some examples of how the language can be used.

We will focus here on extensions of standard graphs and automata to their infinite versions. But Nλ can also be used for algorithms on other computation models like register automata, pushdown automata or context-free grammars. More examples that can be also implemented can be found in [Bojańczyk, 2019, Chapter 5] or [Kopczyński and Toruńczyk, 2017, Section 8].

Unless otherwise indicated, we assume that all examples in this chapter are interpreted for ordered atoms.

11.1. Graphs

In the standard definition, a graph is described as a pair (V, E) of finite sets of vertices and edges. This definition naturally extends to hereditarily orbit-finite sets with atoms (see Section 2.6) or equivalently, definable sets (see Section 2.7 and Theorem 2.11). Therefore, instead of finite graphs, we will deal with *definable graphs* that can have an infinite number of vertices and edges but a finite number of orbits.

The definition of a structure that represents such a graph in the directed version is as follows:[1]

```
-- A directed graph with vertices of type "a" and set of pairs representing
↪  edges.
data Graph a = Graph {vertices :: Set a, edges :: Set (a,a)}
  deriving (Eq, Ord, Show, Read, Generic, Nominal, Conditional, Contextual)
```

The definition includes a set of vertices of any type and a set of edges between them. The graph is automatically an instance of the standard Haskell type classes and Nλ type classes described in Chapter 10.

Let us create some simple graphs that will serve as examples for graph algorithms. One of the simpler examples of a graph is the atom clique:

```
Nλ> let atomsClique = Graph atoms atomsPairs
Nλ> atomsClique
Graph {vertices = {a₁ : for a₁ ∈ 𝔸}, edges = {(a₁,a₂) : for a₁,a₂ ∈ 𝔸}}
```

[1]Source code related to graphs can be found in the `Nominal.Graph` module.

It is a graph in which the vertices are atoms and the edges are all possible pairs of atoms. With the help of the `filterEdges` auxiliary function, one can easily create other atom graphs.

```
-- Returns a graph with filtered edges
filterEdges :: Nominal a => (a -> a -> Formula) -> Graph a -> Graph a
filterEdges f (Graph vs es) = Graph vs (filter (uncurry f) es)
```

Below is a graph where the edges only go from smaller to greater atoms.

```
Nλ> let monotonicGraph = filterEdges lt atomsClique
Nλ> monotonicGraph
Graph {vertices = {a₁ : for a₁ ∈ 𝔸}, edges = {(a₁,a₂) : a₁ < a₂ for a₁,a₂ ∈ 𝔸}}
```

Another example is a graph that is a clique without loops, i.e. one in which only different atoms are connected by an edge.

```
Nλ> let differentAtomsGraph = filterEdges neq atomsClique
Nλ> differentAtomsGraph
Graph {vertices = {a₁ : for a₁ ∈ 𝔸}, edges = {(a₁,a₂) : a₁ ≠ a₂ for a₁,a₂ ∈ 𝔸}}
```

In the following algorithms, we will mainly operate on directed graphs. If some properties naturally relate to undirected graphs, we will continue to use the above graph representation in which for each edge we will add an opposite edge using the function:

```
-- Adds all reverse edges to existing edges in a graph.
undirected :: Nominal a => Graph a -> Graph a
undirected (Graph vs es) = Graph vs (union es (map swap es))
```

Under this assumption, `differentAtomsGraph` is an undirected version of `monotonicGraph`. Of course, another approach is to represent the edges of undirected graphs as pair sets.

11.2. Graph algorithms

For hereditarily orbit-finite graphs, one can implement many algorithms derived from the world of finite graphs. The first example is a transitive closure of the graph. For the implementation of this algorithm, we will use an auxiliary function that computes the composition of the set of edges treated as a binary relation:

```
-- Produces a set of edges that are obtained by composition of edges from
↪  given sets.
composeEdges :: Nominal a => Set (a,a) -> Set (a,a) -> Set (a,a)
composeEdges = pairsWithFilter (\(a, b) (c, d) -> maybeIf (eq b c) (a, d))
```

The above function for any two pairs `(a, b)` and `(c, d)` returns `(a, d)` only if `b` and `c` are equal. The function `maybeIf` returns a value only if the condition is met (otherwise it returns `Nothing`), and `pairsWithFilter` returns the filtered set of pairs. Given a function for composition, one can compute transitive closure recursively:

```
-- Returns a transitive closure of a graph.
transitiveClosure :: Nominal a => Graph a -> Graph a
transitiveClosure (Graph vs es) = Graph vs (edgesClosure es)
    where edgesClosure es = let es' = union es (composeEdges es es)
                            in if eq es es' then es else edgesClosure es'
```

The algorithm for the set of edges adds those that are created by the composition of the existing edges. If the obtained result differs from the current set, the procedure is called recursively.

Below are examples of using this function for previously defined graphs:

```
Nλ> transitiveClosure atomsClique
Graph {vertices = {a₁ : for a₁ ∈ 𝔸}, edges = {(a₁,a₂) : for a₁,a₂ ∈ 𝔸}}
Nλ> transitiveClosure monotonicGraph
Graph {vertices = {a₁ : for a₁ ∈ 𝔸}, edges = {(a₁,a₂) : a₁ < a₂ for a₁,a₂ ∈ 𝔸}}
Nλ> transitiveClosure differentAtomsGraph
Graph {vertices = {a₁ : for a₁ ∈ 𝔸}, edges = {(a₁,a₂) : for a₁,a₂ ∈ 𝔸}}
```

Many properties of graphs can be computed by using transitive closure. For example, reachability and existence of paths can be checked:

```
-- Checks whether there is a path from one vertex to the second one in a
↪  graph.
existsPath :: Nominal a => Graph a -> a -> a -> Formula
existsPath g v1 v2 = contains (edges (transitiveClosure g)) (v1,v2)
```

One can also examine connectivity properties:

```
-- Checks whether a graph is strongly connected.
isStronglyConnected :: Nominal a => Graph a -> Formula
isStronglyConnected g = eq (transitiveClosure g) (clique (vertices g))
-- Checks whether a graph is weakly connected.
isWeaklyConnected :: Nominal a => Graph a -> Formula
isWeaklyConnected = isStronglyConnected . undirected
```

A graph is strongly connected if its transitive closure is a clique. A graph is weakly connected if treated as an undirected graph it is connected. All our example graphs are weakly connected, and only the monotonic graph is not strongly connected.

```
Nλ> let examples = [atomsClique, monotonicGraph, differentAtomsGraph]
Nλ> fmap isWeaklyConnected examples
[true,true,true]
Nλ> fmap isStronglyConnected examples
[true,false,true]
```

Transitive closure also helps in finding cycles in a graph:

```
-- Checks whether a graph has a cycle.
hasCycle :: Nominal a => Graph a -> Formula
hasCycle = hasLoop . transitiveClosure
```

The function `hasLoop` checks for a loop (an edge that starts and ends on the same vertex). Additionally, one can test the parity of cycles:

```
-- Checks whether a graph has a even-length cycle.
hasEvenLengthCycle :: Nominal a => Graph a -> Formula
hasEvenLengthCycle g = hasCycle (composeGraph g g)
```

```
-- Checks whether a graph has a odd-length cycle.
hasOddLengthCycle :: Nominal a => Graph a -> Formula
hasOddLengthCycle g = edges (reverseEdges g) `intersect` edges
↪  (transitiveClosure (composeGraph g g))
```

A graph has an even-length cycle if there is a cycle in the graph created by combining the graph with itself. To find an odd-length cycle, it is enough to check if there is an even-length path that starts and ends at the same vertices as some reversed edge. Knowing whether the graph has an odd-length cycle, one can deduce whether it is bipartite (if treated as undirected):

```
-- Checks whether a graph (treated as undirected) is bipartite in the sense
↪  that it does not have odd-length cycle
isBipartite :: Nominal a => Graph a -> Formula
isBipartite = not . hasOddLengthCycle . undirected
```

None of our example graphs are bipartite.

```
Nλ> fmap isBipartite examples
[false,false,false]
```

An example of a bipartite graph can be a graph in which all atoms larger than some fixed atom `a` are connected only to atoms that are not larger than `a`.

```
Nλ> let bipartiteGraph = filterEdges (\v1 v2 -> gt v1 a `xor` gt v2 a)
↪  atomsClique

Nλ> bipartiteGraph
Graph {vertices = {a₁ : for a₁ ∈ 𝔸}, edges = {(a₁,a₂) : (a < a₁ ∧ a ≥ a₂) ∨
↪  (a < a₂ ∧ a ≥ a₁) for a₁,a₂ ∈ 𝔸}}

Nλ> isBipartite bipartiteGraph
true
```

11.3. Graph coloring

The algorithms presented so far naturally extended from finite sets. Which means that a source code for infinite sets looked the same as for finite ones. The situation is not always so comfortable and sometimes the programmer needs to be aware of the orbital structure in the case of infinite sets. Graph coloring is a good example.

Recall that a graph coloring is a valuation of its nodes such that no two adjacent vertices share the same value. The verification of whether a given function is a valid coloring looks as follows:

```
isColoringOf :: (NominalType a, NominalType b) => (a -> b) -> Graph a -> Formula
isColoringOf c g = forAll (\(v1,v2) -> c v1 `neq` c v2) (edges g)
```

A k-coloring is a graph coloring with k colors. In order to check whether a graph is k-colorable in the finite setting, one could generate all k-partitions of a set of n vertices:

```
partitions :: Int -> Int -> Set [Int]
partitions n 1 = singleton (replicate n 0)
partitions n k | k < 1 || n < k = empty
partitions n k | n == k = singleton [0..n-1]
partitions n k = union (map (k-1:) $ partitions (n-1) (k-1))
                       (pairsWith (:) (fromList [0..k-1]) (partitions (n-1) k))
```

For example, we have three 2-partitions of three vertices:

```
Nλ> partitions 3 2
{[0,0,1], [1,0,0], [1,0,1]}
```

For each such partition, one could examine if the valuation that arises from it is a valid coloring.

This works for finite sets and graphs. However, in the world of definable sets, the situation is much more complicated. One cannot enumerate and collect all partitions because the set of partitions of a definable set might not be first-order definable or even countable. Indeed, at first sight, it is not clear that the colorability of definable graphs is a decidable problem. For example, consider the undirected graph:

```
Graph {vertices = {(a₁,a₂) : a₁ ≠ a₂ for a₁,a₂ ∈ 𝔸}, edges =
↪  {((a₁,a₂),(a₂,a₃)) : a₁ ≠ a₂ ∧ a₁ ≠ a₃ ∧ a₂ ≠ a₃ for a₁,a₂,a₃ ∈ 𝔸,
↪  ((a₁,a₂),(a₃,a₁)) : a₁ ≠ a₂ ∧ a₁ ≠ a₃ ∧ a₂ ≠ a₃ for a₁,a₂,a₃ ∈ 𝔸}}
```

This graph, used as an example in [Klin et al., 2015], is not 3-colorable. However, its smallest finite non-3-colorable subgraph has as many as 10 vertices and 20 edges. One may try to check larger and larger finite subgraphs of a given definable graph and check their colorability using the standard code above, but it is not clear when one can stop and declare the entire graph colorable.

One may make some additional assumptions, for example, consider only equivariant colorings, where nodes in the same orbit must get the same color (for example, the above graph has no equivariant colorings, as it only has one orbit of vertices and it has edges). The problem then reduces to coloring the finite set of orbits. For a given list of orbits and a list of its partitions (generated by the above function `partitions`), one can create a coloring function that determines which orbit contains a given element and returns the color assigned to such an orbit.[2]

```
coloring :: NominalType a => [Set a] -> [Int] -> a -> Variants Int
coloring [] [] _ = variant 0
coloring (o:os) (p:ps) a = if member a o then variant p else coloring os ps a
```

Then it remains to be checked whether a coloring function created by a partition of orbits is a proper coloring of the graph. This can be implemented as follows:

```
hasEquivariantColoring :: NominalType a => Graph a -> Int -> Formula
hasEquivariantColoring g k = member true $
    pairsWith (\os ps -> (coloring os ps) `isColoringOf` g)
              (replicateSet n orbits)
              (partitions n k)
    where orbits = setOrbits (vertices g)
               n = maxSize orbits
```

[2]The returned color has the form of a variant because for a given element it may be ambiguous which orbit it belongs to.

where the auxiliary function was used:

```
-- Creates a set of lists of a given length of elements from a set
replicateSet :: Nominal a => Int -> Set a -> Set [a]
replicateSet n s = mapList id (replicate n s)
```

This solves the problem of finding equivariant colorings of definable graphs. As it turns out, it solves the problem of general k-colorability as well: in [Klin et al., 2015], it was proved that over ordered atoms a definable graph has a k-coloring if and only it has an equivariant one. That result relies on deep theorems in topological dynamics. As we can see, the programmer needs to know the mathematics of first-order definable structures not only to write the program for k-colorability, but even more so to prove its correctness.

It is worth noting that the problem of finding an equivariant k-coloring may have different solutions depending on the structure of atoms. For example, the graph `g`:

```
Nλ> let g = filterEdges (\(a1,b1) (a2,b2) -> eq a1 b2 /\ eq a2 b1) (clique
↪  differentAtomsPairs)

Nλ> g
Graph {vertices = {(a₁,a₂) : a₁ ≠ a₂ for a₁,a₂ ∈ 𝔸}, edges =
↪  {((a₁,a₂),(a₂,a₁)) : a₁ ≠ a₂ for a₁,a₂ ∈ 𝔸}}
```

does not have an equivariant 2-coloring when equality atoms are considered. But for ordered atoms, a function

```
uncurry lt :: (Atom, Atom) -> Formula
```

is a correct coloring. So for these two structures of atoms the expression `(hasEquivariantColoring g 2)` will evaluate to `false` and `true` respectively.

Note that 2-colorings can be looked for in a way very similar to the one used for finite graphs; indeed, a graph is 2-colorable if and only if it is bipartite, and an Nλ program to check that was shown above. The expression `(isBipartite g)` will evaluate to `true` both over equality and ordered atoms, indicating that a 2-coloring (not necessarily equivariant) of `g` exists.

11.4. Automata

Automata are a well-established computational model with a wide range of applications. The definition of an automaton for sets with atoms is similar to the standard one, only all sets are definable:

$$(Q, \quad \Sigma, \quad \delta \subseteq Q \times \Sigma \times Q, \quad I \subseteq Q, \quad F \subseteq Q)$$

Such an automaton is called a *definable automaton*. We will represent a nondeterministic definable automaton as follows:[3]

[3]The source code related to automata can be found in the `Nominal.Automaton.Base`, `Nominal.Automaton.Deterministic` and `Nominal.Automaton.Nondeterministic` modules.

```
-- An automaton with a set of state with type "q" accepting/rejecting words
↪  from an alphabet with type "a".
data Automaton q a = Automaton {states :: Set q,
                                alphabet :: Set a,
                                delta :: Set (q, a, q),
                                initialStates :: Set q,
                                finalStates :: Set q}
  deriving (Eq, Ord, Show, Read, Generic, Nominal, Contextual, Conditional)
```

For a deterministic automaton, we will use the same representation but in this case, additional conditions must be met:

- the set of initial states is a singleton,
- the delta set represents a well-defined function (`q -> a -> q`).

The problem of checking whether a given automaton represented in the above manner is deterministic is decidable and can be implemented as follows:

```
-- Checks whether an automaton is deterministic.
isDeterministic :: (Nominal q, Nominal a) => Automaton q a -> Formula
isDeterministic aut = isSingleton (initialStates aut) /\
  forAll (`hasSizeLessThan` 2)
         (pairsWith (transit aut) (states aut) (alphabet aut))
```

What is worth emphasizing is that nondeterministic definable automata are strictly more expressive than deterministic ones. The standard determination algorithm does not work because the power set does not have to be a set with atoms (see Example 2.6).

However, what works analogously to finite automata is:

- checking if an automaton accepts a given word

  ```
  accepts :: (Nominal q, Nominal a) => Automaton q a -> [a] -> Formula
  accepts aut = intersect (finalStates aut) . foldl (transitSet aut)
  ↪  (initialStates aut)
  ```

- emptiness problem

  ```
  isEmptyAutomaton :: (Nominal q, Nominal a) => Automaton q a -> Formula
  isEmptyAutomaton = isEmpty . finalStates . onlyReachable
  ```

- union and intersection (with the standard implementation)

  ```
  unionAutomaton :: (Nominal q1, Nominal q2, Nominal a) =>
      Automaton q1 a -> Automaton q2 a -> Automaton (NominalEither q1 q2) a

  intersectionAutomaton :: (Nominal q1, Nominal q2, Nominal a) =>
      Automaton q1 a -> Automaton q2 a -> Automaton (q1, q2) a
  ```

In the case of deterministic definable automata, we can additionally handle the following features:

- a deterministic automaton that accepts the complementation of a language

  ```
  complementDA :: Nominal q => Automaton q a -> Automaton q a
  complementDA (Automaton q a d i f) = Automaton q a d i (q \\ f)
  ```

- universality checking of a deterministic automaton

```
isUniversalDA :: (Nominal q, Nominal a) => Automaton q a -> Formula
isUniversalDA = isEmptyAutomaton . complementDA
```

- building a deterministic automaton recognizing the difference of languages

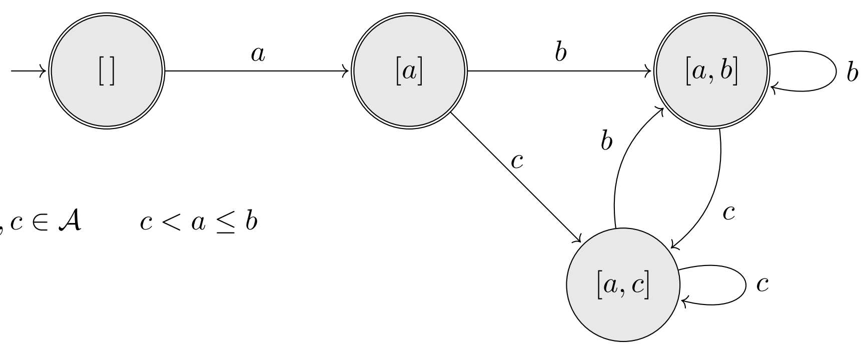
```
differenceDA :: (Nominal q1, Nominal q2, Nominal a) =>
    Automaton q1 a -> Automaton q2 a -> Automaton (q1, q2) a
differenceDA aut1 aut2 = intersectionAutomaton aut1 (complementDA aut2)
```

- checking equivalence of deterministic automata

```
equivalentDA :: (Nominal q1, Nominal q2, Nominal a) =>
    Automaton q1 a -> Automaton q2 a -> Formula
equivalentDA aut1 aut2 = isEmptyAutomaton (differenceDA aut1 aut2)
    /\ isEmptyAutomaton (differenceDA aut2 aut1)
```

Languages recognized by nondeterministic definable automata are not closed under complementation. Moreover, universality is undecidable for these automata.

As an example, let us create a deterministic automaton that accepts words whose last letter is not less than the first one. The alphabet in our example will be the set of atoms. The states are lists of at most two atoms (for the first and last letter read). The only non-final states are two-atom lists where the second atom is smaller than the first one. The automaton diagram is as follows:



Each node with a non-empty list represents an infinite number of states for any atoms $a, b, c \in \mathcal{A}$ such that $c < a \leq b$. Likewise, each arrow represents an infinite number of transitions in the automaton.

Let us create this automaton in Nλ. First, we generate a set of states:

```
Nλ> let q = replicateAtomsUntil 2
Nλ> q
{[], [a₁] : for a₁ ∈ 𝔸, [a₁,a₂] : for a₁,a₂ ∈ 𝔸}
```

Then we implement the transition function that after reading a letter creates a singleton list (if it was empty so far), or puts this letter in the second position of a list:

```
Nλ> let transition list a = if null list then [a] else [head list,a]
```

The initial state will be an empty list and the set of final states will contain all states except for lists where the second atom is smaller than the first one.

```
Nλ> let init = []
Nλ> let final = q \\ mapFilter (\(a,b) -> maybeIf (gt a b) [a,b]) atomsPairs
```

Finally, we create a deterministic automaton using the helper function `da`.

```
Nλ> let aut = da q atoms transition init final
Nλ> aut
Automaton {states = {[], [a₁] : for a₁ ∈ 𝔸, [a₁,a₂] : for a₁,a₂ ∈ 𝔸},
           alphabet = {a₁ : for a₁ ∈ 𝔸},
           delta = {([],a₁,[a₁]) : for a₁ ∈ 𝔸, ([a₁],a₂,[a₁,a₂]) : for a₁,a₂
           ↪  ∈ 𝔸, ([a₁,a₂],a₃,[a₁,a₃]) : for a₁,a₂,a₃ ∈ 𝔸},
           initialStates = {[]},
           finalStates = {[], [a₁] : for a₁ ∈ 𝔸, [a₁,a₂] : a₁ ≤ a₂ for
           ↪  a₁,a₂ ∈ 𝔸}}
```

Simple tests confirm that the automaton recognizes words from the expected language:

```
Nλ> accepts aut []
true
Nλ> accepts aut [a]
true
Nλ> accepts aut [a,b]
a ≤ b
Nλ> accepts aut [a,b,c]
a ≤ c
Nλ> accepts aut [a,b,c,d]
a ≤ d
```

11.5. Minimization of deterministic automata

Recall that for a given language $L \subseteq \Sigma^*$, the Myhill-Nerode equivalence relation $\sim_L$ is defined as follows:

$$u \sim_L w \qquad \Leftrightarrow_{\text{def}} \qquad \forall_{v \in \Sigma^*}\ uv \in L \Leftrightarrow wv \in L$$

for any $u, w \in \Sigma^*$.

The language L is recognized by a deterministic definable automaton if and only if the relation $\sim_L$ is orbit-finite [Bojańczyk et al., 2014, Theorem 3.8]. It follows that for definable automata there is a standard construction of a minimal automaton with states being equivalence classes of the Myhill-Nerode relation.

The minimization algorithm of a deterministic definable automaton can be implemented in Nλ based on Moore's algorithm [Moore, 1956]. Let us follow the main points of this algorithm.

Initially, unreachable states from the input automaton are removed using the graph reachability algorithm.

Then we calculate which states of the automaton are equivalent according to the relation $\sim_L$. For this purpose, we build a directed graph whose nodes are pairs of states from the automaton. Two pairs are adjacent if there is a transition from the states of the source pair to the states of the target pair by some letter.

```
statePairsGraph :: (Nominal q, Nominal a) => Automaton q a -> Graph (q, q)
```

Then, by inverting the edges of this graph, we compute all non-equivalent state pairs i.e. reachable from the set:

$$(Q \setminus F) \times F \quad \cup \quad F \times (Q \setminus F)$$

We obtain pairs of equivalent states by the differences of the set of all pairs and pairs of non-equivalent states.

```
equivalentStates :: (Nominal q, Nominal a) => Automaton q a -> Set (q, q)
equivalentStates aut@(Automaton q a d i f) = square q \\ nonEquivalent
    where reverseGraph = reverseEdges (statePairsGraph aut)
          nonEquivalentInitial = pairs (q \\ f) f `union` pairs f (q \\ f)
          nonEquivalent = reachableFromSet reverseGraph nonEquivalentInitial
```

The states of the resulting automaton are connected components of the graph with the vertices being the states of the input automaton and the edges being the relation calculated above. Based on this, the remaining elements of the automat can be calculated.

```
-- Returns a minimal automaton accepting the same language as a given automaton.
minimize :: (Nominal q, Nominal a) => Automaton q a -> Automaton (Set q) a
minimize aut = Automaton q' a d' i' f'
    where reachAut@(Automaton q a d i f) = onlyReachable aut
          equiv = equivalentStates reachAut
          q' = map vertices (stronglyConnectedComponents (graph q equiv))
          d' = filter
                 (\(ss1,ll,ss2) ->
                   exists
                     (\(s1,l,s2)-> member s1 ss1 /\ eq l ll /\ member s2 ss2)
                     d)
                 (triples q' a q')
          i' = filter (intersect i) q'
          f' = filter (intersect f) q'
```

Let us analyze the algorithm result for the example automaton from the previous section. The equivalence relation for this automaton is as follows:

```
Nλ> equivalentStates aut
{([],[]),
 ([a₁],[a₁]) : for a₁ ∈ 𝔸,
 ([a₁],[a₁,a₂]) : a₁ ≤ a₂ for a₁,a₂ ∈ 𝔸,
 ([a₁,a₂],[a₁]) : a₁ ≤ a₂ for a₁,a₂ ∈ 𝔸,
 ([a₁,a₂],[a₁,a₃]) : (a₁ ≤ a₂ ∨ a₁ > a₃) ∧ (a₁ ≤ a₃ ∨ a₁ > a₂) for a₁,a₂,a₃
 ↪  ∈ 𝔸}
```

In particular, the relation connects singleton lists with two-atom lists if they have the same prefix, and in the long list, the second atom is greater than or equal to the first one. Additionally, the relation connects two-atom lists if the first atoms are equal (a_1) and the relation between the atoms in the lists is consistent ($a_1 \leq a_2 \Leftrightarrow a_1 \leq a_3$). It is as expected because the information about the last letter read is redundant. It is enough to store information about the first letter of a word and whether the last letter read is smaller than it.

Finally, let us take a look at the resulting minimal automaton.

```
Nλ> simplify (minimize aut)
Automaton {states = {{[]}, {[a1], [a1,b1] : a1 ≤ b1 for b1 ∈ 𝔸} : for a1 ∈
        ↪  𝔸, {[a1] : a1 ≤ a2, [b1,b2] : a1 = b1 ∧ (a1 ≤ a2 ∨ a1 > b2)
        ↪  ∧ (a1 ≤ b2 ∨ a1 > a2) for b1,b2 ∈ 𝔸} : for a1,a2 ∈ 𝔸},
          alphabet = {a1 : for a1 ∈ 𝔸},
          delta = {({[]},a1,{[a1], [a1,b1] : a1 ≤ b1 for b1 ∈ 𝔸}) : for a1
        ↪  ∈ 𝔸, ({[]},a1,{[a1], [b1,b2] : (a1 ≤ b2 ∧ a1 = b1) ∨ (a1 =
        ↪  b1 ∧ a2 = b2) for b1,b2 ∈ 𝔸}) : a1 ≤ a2 for a1,a2 ∈ 𝔸,
        ↪  ({[a1], [a1,b1] : a1 ≤ b1 for b1 ∈ 𝔸},a2,{[a1], [a1,b1] : a1
        ↪  ≤ b1 for b1 ∈ 𝔸}) : a1 ≤ a2 for a1,a2 ∈ 𝔸, ({[a1], [a1,b1] :
        ↪  a1 ≤ b1 for b1 ∈ 𝔸},a2,{[a3] : a3 ≤ a4, [b1,b2] : a3 = b1 ∧
        ↪  (a3 ≤ a4 ∨ a3 > b2) ∧ (a3 ≤ b2 ∨ a3 > a4) for b1,b2 ∈ 𝔸}) :
        ↪  a1 = a3 ∧ (a3 ≤ a4 ∨ a3 > a2) ∧ (a2 ≥ a3 ∨ a3 > a4) for
        ↪  a1,a2,a3,a4 ∈ 𝔸, ({[a1] : a1 ≤ a2, [b1,b2] : a1 = b1 ∧ (a1 ≤
        ↪  a2 ∨ a1 > b2) ∧ (a1 ≤ b2 ∨ a1 > a2) for b1,b2 ∈ 𝔸},a3,{[a4],
        ↪  [a4,b1] : a4 ≤ b1 for b1 ∈ 𝔸}) : a1 = a4 ∧ a3 ≥ a4 for
        ↪  a1,a2,a3,a4 ∈ 𝔸, ({[a1] : a1 ≤ a2, [b1,b2] : a1 = b1 ∧ (a1 ≤
        ↪  a2 ∨ a1 > b2) ∧ (a1 ≤ b2 ∨ a1 > a2) for b1,b2 ∈ 𝔸},a3,{[a4]
        ↪  : a4 ≤ a5, [b1,b2] : a4 = b1 ∧ (a4 ≤ a5 ∨ a4 > b2) ∧ (a4 ≤
        ↪  b2 ∨ a4 > a5) for b1,b2 ∈ 𝔸}) : a1 = a4 ∧ (a4 ≤ a5 ∨ a4 >
        ↪  a3) ∧ (a3 ≥ a4 ∨ a4 > a5) for a1,a2,a3,a4,a5 ∈ 𝔸},
          initialStates = {{[]}},
          finalStates = {{[]}, {[a1], [a1,b1] : a1 ≤ b1 for b1 ∈ 𝔸} : for
        ↪  a1 ∈ 𝔸, {[a1], [b1,b2] : (a1 ≤ b2 ∧ a1 = b1) ∨ (a1 = b1 ∧ a2
        ↪  = b2) for b1,b2 ∈ 𝔸} : a1 ≤ a2 for a1,a2 ∈ 𝔸}}
```

The resulting automaton is quite complicated and difficult for human interpretation, even after simplifying formulas with the `simplify` function. Unfortunately, this is a disadvantage of using Nλ, as calculation results often contain large formulas. The correctness of our minimized automaton should rather be verified with another function, for example by checking whether the received automaton recognizes the same language:

```
Nλ> equivalentDA aut (minimize aut)
true
```

11.6. Learning automata

Automata learning is the problem of constructing an automaton for some language $L \subseteq \Sigma^*$ based on observations. There are two parties: *learner* and *teacher*. The learner tries to infer the language from the sequence of two types of queries to the teacher (who knows the language):

- Membership

  The query contains a single word $w \in \Sigma^*$ for which the teacher replies whether the word belongs to the language L.

- Equivalence

  The query contains a hypothetical language represented by an automaton, and the teacher replies if this language is equal to the target language. If not, it additionally indicates a counterexample, i.e. a word where these languages disagree.

The original algorithm L^* by Dana Angluin solves the problem for deterministic finite automata [Angluin, 1987]. The algorithm is based on the construction of the observation table. A table correctly represents a deterministic finite automaton if it is *closed* and *consistent* (see [Angluin, 1987] for details). Starting with a trivial table, it is expanded (by adding rows and columns) using membership queries. When a valid automaton can be constructed from the table, an equivalence query is sent. If the response is negative, then based on the obtained counterexample, subsequent rows are added to the table. The algorithm runs until it receives a positive answer to the equivalence query. The correctness and termination of the algorithm have been proven.

Later, this algorithm was extended to other types of automata, including more useful ones for practical applications: Mealy Machines [Shahbaz and Groz, 2009], Register Automata [Howar et al., 2012] or Büchi Automata [Maler et al., 1995].

The generalization of the algorithm to definable automata has been described in the article “Learning Nominal Automata” [Moerman et al., 2016]. The work describes two algorithms. The first, called νL^*, works for deterministic definable automata and differs from the original Angluin algorithm in that the rows and columns represent not only individual words, but word orbits.

The second algorithm is intended for nondeterministic definable automata. It is the first learning algorithm for nondeterministic automata over infinite alphabets. It is an extension of the NL^* algorithm for finite equivalents [Bollig et al., 2009]. The algorithm is called νNL^* and supports a strict subclass of nondeterministic definable automata. This subclass includes all languages recognized by deterministic automata, but due to nondeterminism, they are much more succinct.

Both algorithms have been implemented in Nλ and are available in the library `nominal-lstar` [Moerman, 2016]. The algorithm for deterministic automata has been developed in two versions (one adds a counterexample to the observation table as columns, and the other as rows).

```
learnAngluin :: (Nominal i, Contextual i, Show i) =>
    Teacher i -> Automaton (Set [i]) i
learnAngluinRows :: (Nominal i, Contextual i, Show i) =>
    Teacher i -> Automaton (Set [i]) i
learnBollig :: (Nominal i, Contextual i, Show i) =>
    Int -> Int -> Teacher i -> Automaton (Set [i]) i
```

All these algorithms need a teacher data structure to handle queries:

```
data Teacher i = Teacher {
    membership :: Set [i] -> Set ([i], Formula)
    equivalent :: forall q. (Show q, Nominal q) => Automaton q i -> Maybe (Set [i])
    alphabet   :: Set i
}
```

The library provides a teacher who will handle queries based on a given automaton, and a teacher who will delegate the queries to the terminal.

Experiments on the implemented algorithms were run for several sample automata available in the library. For example, an automaton that checks if a given sequence of commands `put` and `push` is valid for a FIFO queue (first in, first out) with a limited size. Results are reported in [Moerman et al., 2016].

Bibliography

[Angluin, 1987] Angluin, D. (1987). Learning regular sets from queries and counterexamples. *Information and Computation*, 75(2):87–106.

[Barendregt, 1984] Barendregt, H. P. (1984). *The Lambda Calculus Its Syntax and Semantics*, volume 103. North Holland, revised edition.

[Barendregt, 1993] Barendregt, H. P. (1993). *Lambda Calculi with Types*, page 117–309. Oxford University Press, Inc., USA.

[Barrett et al., 2009] Barrett, C., Sebastiani, R., Seshia, S., and Tinelli, C. (2009). Satisfiability modulo theories. In Biere, A., Heule, M., Maaren, H., and Walsh, T., editors, *Handbook of Satisfiability: Volume 185 Frontiers in Artificial Intelligence and Applications*, chapter 12, page 825–885. IOS Press.

[Barrett et al., 2010] Barrett, C., Stump, A., and Tinelli, C. (2010). The SMT-LIB Standard: Version 2.0. In Gupta, A. and Kroening, D., editors, *Proceedings of the 8th International Workshop on Satisfiability Modulo Theories (Edinburgh, UK)*.

[Bjørner et al., 2021] Bjørner, N., De Moura, L., et al. (2021). Z3. `https://github.com/Z3Prover/z3`. [Online; accessed March, 2022].

[Bojańczyk, 2019] Bojańczyk, M. (2019). *Slightly Infinite Sets*. University of Warsaw. `https://www.mimuw.edu.pl/~bojan/paper/atom-book` [Online; accessed March, 2022].

[Bojańczyk et al., 2012] Bojańczyk, M., Braud, L., Klin, B., and Lasota, S. (2012). Towards nominal computation. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '12, page 401–412. Association for Computing Machinery.

[Bojańczyk et al., 2014] Bojańczyk, M., Klin, B., and Lasota, S. (2014). Automata theory in nominal sets. *Logical Methods in Computer Science*, 10.

[Bojańczyk et al., 2013] Bojańczyk, M., Klin, B., Lasota, S., and Toruńczyk, S. (2013). Turing machines with atoms. In *2013 28th Annual ACM/IEEE Symposium on Logic in Computer Science*, pages 183–192.

[Bojańczyk and Toruńczyk, 2012] Bojańczyk, M. and Toruńczyk, S. (2012). Imperative programming in sets with atoms. 18:4–15.

[Bollig et al., 2009] Bollig, B., Habermehl, P., Kern, C., and Leucker, M. (2009). Angluin-style learning of nfa. pages 1004–1009.

[Burstall, 1969] Burstall, R. M. (1969). Proving Properties of Programs by Structural Induction. *The Computer Journal*, 12(1):41–48.

[Burstall and Darlington, 1977] Burstall, R. M. and Darlington, J. (1977). A transformation system for developing recursive programs. *J. ACM*, 24(1):44–67.

[Cabal, 2022] Cabal (2022). Cabal: Common architecture for building applications and libraries. https://www.haskell.org/cabal/. [Online; accessed March, 2022].

[Cheney and Hinze, 2003] Cheney, J. and Hinze, R. (2003). First-class phantom types.

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

[Church and Rosser, 1936] Church, A. and Rosser, J. B. (1936). Some properties of conversion. *Transactions of the American Mathematical Society*, 39(3):472 – 482.

[Curry and Feys, 1958] Curry, H. B. and Feys, R. (1958). *Combinatory Logic Vol. 1*. North-Holland Publishing Company.

[Curry et al., 1974] Curry, H. B., Feys, R., and Craig, W. (1974). *Combinatory logic*, volume 1. North-Holland Publ. Co., third edition.

[De Moura and Bjørner, 2008] De Moura, L. and Bjørner, N. (2008). Z3: An efficient SMT solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, page 337–340, Berlin, Heidelberg. Springer-Verlag.

[Di Cosmo and Kesner, 1995] Di Cosmo, R. and Kesner, D. (1995). A confluent reduction for the extensional typed lambda-calculus with pairs, sums, recursion and terminal object.

[Dougherty, 1993] Dougherty, D. (1993). Some lambda calculi with categorical sums and products.

[Downen et al., 2019] Downen, P., Sullivan, Z., Ariola, Z., and Peyton Jones, S. (2019). *Codata in Action*, pages 119–146.

[Engeler, 1959] Engeler, E. (1959). A characterisation of theories with isomorphic denumerable models. *Notices Amer. Math. Soc.*, 6:161.

[Ferrari et al., 2005] Ferrari, G., Montanari, U., and Tuosto, E. (2005). Coalgebraic minimization of hd-automata for the π-calculus using polymorphic types. *Theoretical Computer Science*, 331(2):325–365. Formal Methods for Components and Objects.

[Fraenkel, 1922] Fraenkel, A. (1922). Der Begriff "definit" und die Unabhängigkeit des Auswahlaxioms. *Berl. Ber.*, 1922:253–257.

[Gabbay and Pitts, 2002] Gabbay, M. and Pitts, A. (2002). A new approach to abstract syntax with variable binding. *Formal Asp. Comput.*, 13:341–363.

[Gadducci et al., 2006] Gadducci, F., Miculan, M., and Montanari, U. (2006). About permutation algebras, (pre)sheaves and named sets. *Higher-Order and Symbolic Computation*, 19:283–304.

[Gandy, 1980a] Gandy, R. (1980a). An early proof of normalization. In *To H.B.Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 457–478. Academic Press.

[Gandy, 1980b] Gandy, R. (1980b). Proofs of strong normalisation. In *To H.B.Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 457–478. Academic Press.

[Girard, 1972] Girard, J.-Y. (1972). *Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur.* PhD thesis.

[Hackage, 2022] Hackage (2022). Hackage: The Haskell Package Repository. https://hackage.haskell.org/. [Online; accessed March, 2022].

[Hall et al., 1992] Hall, C., Hammond, K., Partain, W., Jones, S. P., and Wadler, P. (1992). The Glasgow Haskell Compiler: A Retrospective. pages 62–71.

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

[Hankin, 1994] Hankin, C. (1994). *Lambda Calculi: A Guide for Computer Scientists.* Graduate texts in computer science. Clarendon Press.

[Haskell.org, 2022] Haskell.org (2022). Glasgow Haskell Compiler User's Guide. https://downloads.haskell.org/~ghc/latest/docs/html/users_guide/. [Online; accessed March, 2022].

[Hindley and Seldin, 2008] Hindley, J. R. and Seldin, J. P. (2008). *Lambda-Calculus and Combinators: An Introduction.* Cambridge University Press, USA, 2 edition.

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

[Hodges, 1993] Hodges, W. (1993). *Model Theory.* Encyclopedia of Mathematics and its Applications. Cambridge University Press.

[Hoogle, 2022] Hoogle (2022). Hoogλe: The Haskell API search engine. https://hoogle.haskell.org/. [Online; accessed March, 2022].

[Howar et al., 2012] Howar, F., Steffen, B., Jonsson, B., and Cassel, S. (2012). Inferring canonical register automata. *Lecture Notes in Computer Science*, 7148:251–266.

[Hubička and Nešetřil, 2005] Hubička, J. and Nešetřil, J. (2005). Universal partial order represented by means of oriented trees and other simple graphs. *European Journal of Combinatorics*, 26(5):765–778.

[Huffman and Urban, 2010] Huffman, B. and Urban, C. (2010). Proof pearl: A new foundation for nominal isabelle. volume 6172, pages 35–50.

[Jeannin et al., 2017] Jeannin, J.-B., Kozen, D., and Silva, A. (2017). CoCaml: Functional Programming with Regular Coinductive Types. *Fundamenta Informaticae*, 150:347–377.

[Jones, 2002] Jones, S. P., editor (2002). *Haskell 98 Language and Libraries: The Revised Report*. Haskell.org.

[Ker, 2009] Ker, A. D. (2009). Lambda calculus and types. Lecture Notes, Oxford University.

[Klin et al., 2015] Klin, B., Kopczyński, E., Ochremiak, J., and Toruńczyk, S. (2015). Locally finite constraint satisfaction problems. In *2015 30th Annual ACM/IEEE Symposium on Logic in Computer Science*, pages 475–486.

[Klin et al., 2014] Klin, B., Lasota, S., Ochremiak, J., and Toruńczyk, S. (2014). Turing machines with atoms, constraint satisfaction problems, and descriptive complexity.

[Klin and Szynwelski, 2016] Klin, B. and Szynwelski, M. (2016). SMT Solving for Functional Programming over Infinite Structures. *Electronic Proceedings in Theoretical Computer Science*, 207:57–75.

[Kopczyński and Toruńczyk, 2016] Kopczyński, E. and Toruńczyk, S. (2016). LOIS: an Application of SMT Solvers.

[Kopczyński and Toruńczyk, 2017] Kopczyński, E. and Toruńczyk, S. (2017). LOIS: syntax and semantics. *ACM SIGPLAN Notices*, 52:586–598.

[Kuratowski, 1921] Kuratowski, C. (1921). Sur la notion de l'ordre dans la théorie des ensembles. *Fundamenta Mathematicae*, 2(1):161–171.

[Lane, 1998] Lane, S. (1998). *Categories for the working mathematician. 2nd ed.*

[Loos and Weispfenning, 1993] Loos, R. and Weispfenning, V. (1993). Applying linear quantifier elimination. *Comput. J.*, 36:450–462.

[Läufer and Odersky, 1994] Läufer, K. and Odersky, M. (1994). Polymorphic type inference and abstract data types. *ACM Trans. Program. Lang. Syst.*, 16:1411–1430.

[Magalhães et al., 2010] Magalhães, J. P., Dijkstra, A., Jeuring, J., and Löh, A. (2010). A Generic Deriving Mechanism for Haskell. *Proceedings of the ACM SIGPLAN International Conference on Functional Programming, ICFP*, 45:37–48.

[Maler et al., 1995] Maler, Oded, Pnueli, and Amir (1995). On the learnability of infinitary regular sets. *Information and Computation*, 118(2):316–326.

[Milner, 1978] Milner, R. (1978). A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3):348 – 375.

[Milner et al., 1975] Milner, R., Morris, L., and Newey, M. (1975). A logic for computable functions with reflexive and polymorphic types.

[Moerman, 2016] Moerman, J. (2016). Learning nominal automata repository on GitHub. https://github.com/Jaxan/nominal-lstar. [Online; accessed March, 2022].

[Moerman et al., 2016] Moerman, J., Sammartino, M., Silva, A., Klin, B., and Szynwelski, M. (2016). Learning nominal automata. *ACM SIGPLAN Notices*, 52.

[Moggi, 1991] Moggi, E. (1991). Notions of computation and monads. *Information and Computation*, 93:55–92.

[Montanari and Pistore, 1999] Montanari, U. and Pistore, M. (1999). Finite state verification for the asynchronous pi-calculus. In *Proceedings of the 5th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, TACAS '99, page 255–269, Berlin, Heidelberg. Springer-Verlag.

[Montanari and Pistore, 2005] Montanari, U. and Pistore, M. (2005). History-dependent automata: An introduction. volume 3465, pages 1–28.

[Moore, 1956] Moore, E. (1956). *Gedanken-Experiments on Sequential Machines*, volume 34, pages 129–153.

[Mostowski, 1938] Mostowski, A. (1938). Über den begriff einer endlichen menge. In *Comptes rendus des séances de la Société des Sciences et des Lettres de Varsovie*, 31, pages 13–20.

[Nederpelt, 1994] Nederpelt, R. (1994). *Strong normalization in a typed lambda calculus with lambda structured types*, pages 389–468. Studies in Logic. North-Holland Publishing Company, Netherlands.

[Nipkow, 2008] Nipkow, T. (2008). Linear quantifier elimination. *Journal of Automated Reasoning*, 45:189–212.

[Ochremiak, 2016] Ochremiak, J. (2016). *Extended constraint satisfaction problems*. PhD thesis, University of Warsaw.

[Orchard and Schrijvers, 2010] Orchard, D. and Schrijvers, T. (2010). Haskell type constraints unleashed. volume 6009, pages 56–71.

[Peyton Jones, 1987] Peyton Jones, S. (1987). *The Implementation of Functional Programming Languages*. Prentice Hall.

[Peyton Jones et al., 2007] Peyton Jones, S., Vytiniotis, D., Weirich, S., and Shields, M. (2007). Practical type inference for arbitrary-rank types. *J. Funct. Program.*, 17:1–82.

[Pistore, 1999] Pistore, M. (1999). *History dependent automata*. PhD thesis, PhD thesis, Computer Science Department, Universita di Pisa.

[Pitts and Gabbay, 2000] Pitts, A. and Gabbay, M. (2000). A metalanguage for programming with bound names modulo renaming.

[Pitts, 2013] Pitts, A. M. (2013). *Nominal Sets: Names and Symmetry in Computer Science*. Cambridge University Press, USA.

[Platek, 1966] Platek, R. (1966). *Foundations of Recursion Theory.* Stanford University.

[Reynolds, 1974] Reynolds, J. C. (1974). Towards a theory of type structure. In *Programming Symposium, Proceedings Colloque Sur La Programmation*, page 408–423, Berlin, Heidelberg. Springer-Verlag.

[Ryll-Nardzewski, 1959] Ryll-Nardzewski, C. (1959). On the categoricity on power $\leq \aleph_0$. *Bull. Acad. Pol. Sci., Sér. Sci. Math. Astron. Phys.*, 7:545–548.

[Schrijvers et al., 2008] Schrijvers, T., Peyton Jones, S., Chakravarty, M., and Sulzmann, M. (2008). Type checking with open type functions. volume 43, pages 51–62.

[Scott, 1969] Scott, D. S. (1969). A type-theoretical alternative to ISWIM, CUCH, OWHY. *Oxford notes.*

[Shahbaz and Groz, 2009] Shahbaz, M. and Groz, R. (2009). Inferring mealy machines. In *World Congress on Formal Methods*, pages 207–222.

[Shinwell et al., 2003] Shinwell, M., Pitts, A., and Gabbay, M. (2003). FreshML: Programming with Binders Made Simple. *SIGPLAN Notices*, 38:263–274.

[Shinwell, 2005] Shinwell, M. R. (2005). *The Fresh Approach: functional programming with names and binders.* PhD thesis, University of Cambridge.

[Sulzmann et al., 2007] Sulzmann, M., Chakravarty, M. M. T., Jones, S. P., and Donnelly, K. (2007). System F with Type Equality Coercions. In *Proceedings of the 2007 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation*, TLDI '07, page 53–66.

[Svenonius, 1959] Svenonius, L. (1959). No-categoricity in first-order predicate calculus. *Theoria*, 25(2):82–94.

[Szynwelski, 2022a] Szynwelski, M. (2022a). Nλ documentation. https://www.mimuw.edu.pl/~szynwelski/nlambda/doc/. [Online; accessed March, 2022].

[Szynwelski, 2022b] Szynwelski, M. (2022b). Nλ repository on GitHub. https://github.com/szynwelski/nlambda. [Online; accessed March, 2022].

[Szynwelski, 2022c] Szynwelski, M. (2022c). Nλ website. https://www.mimuw.edu.pl/~szynwelski/nlambda/. [Online; accessed March, 2022].

[Tait, 1967] Tait, W. W. (1967). Intensional interpretations of functionals of finite type i. *J. Symb. Log.*, 32:198–212.

[Takahashi, 1995] Takahashi, M. (1995). Parallel reductions in lambda-calculus. *Inf. Comput.*, 118(1):120–127.

[Urban, 2008] Urban, C. (2008). Nominal techniques in isabelle/hol. *Journal of Automated Reasoning*, 40:327–356.

[Venhoek et al., 2019] Venhoek, D., Moerman, J., and Rot, J. (2019). Fast computations on ordered nominal sets. *CoRR*, abs/1902.08414.

[Wadler and Blott, 1997] Wadler, P. and Blott, S. (1997). How to make ad-hoc polymorphism less ad hoc.

[Wadsworth, 1971] Wadsworth, C. (1971). *Semantics and Pragmatics of the Lambda-calculus*. University of Oxford.

[Xi et al., 2003] Xi, H., Chen, C., and Chen, G. (2003). Guarded recursive datatype constructors.

[Yorgey et al., 2012] Yorgey, B., Weirich, S., Cretin, J., Peyton Jones, S., Vytiniotis, D., and Magalhaes, J. (2012). Giving haskell a promotion. *Types in Language Design and Implementation*.

[Zermelo, 1930] Zermelo, E. (1930). Über Grenzzahlen und Mengenbereiche: Neue Untersuchungen über die Grundlagen der Mengenlehre. *Fundamenta Mathematicæ*, 16:29–47.