Technical Report

UCAM-CL-TR-860
ISSN 1476-2986

Number 860

UNIVERSITY OF CAMBRIDGE
Computer Laboratory

Program equivalence in functional metaprogramming via nominal Scott domains

Steffen Loesch

October 2014

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500

http://www.cl.cam.ac.uk/

This technical report is based on a dissertation submitted May 2014 by the author for the degree of Doctor of Philosophy to the University of Cambridge, Trinity College.

Technical reports published by the University of Cambridge Computer Laboratory are freely available via the Internet:

http://www.cl.cam.ac.uk/techreports/

ISSN 1476-2986

ABSTRACT

A prominent feature of *metaprogramming* is to write algorithms in one programming language (the meta-language) over structures that represent the programs of another programming language (the object-language). Whenever the object-language has binding constructs (and most programming languages do), we run into tedious issues concerning the semantically correct manipulation of binders.

In this thesis we study a semantic framework in which these issues can be dealt with automatically by the meta-language. Our framework takes the user-friendly 'nominal' approach to metaprogramming in which bound objects are named.

Specifically, we develop mathematical tools for giving logical proofs that two metaprograms (of our framework) are equivalent. We consider two programs to be equivalent if they always give the same observable results when they are run as part of any larger codebase. This notion of program equivalence, called *contextual equivalence*, is examined for an extension of Plotkin's archetypal functional programming language PCF with nominal constructs for metaprogramming, called PNA.

Historically, PCF and its denotational semantics based on Scott domains [47] were hugely influential in the study of contextual equivalence. We mirror Plotkin's classical results with PNA and a denotational semantics based on a variant of Scott domains that is modelled within the logic of *nominal sets*. In particular, we prove the following *full abstraction* result: two PNA programs are contextually equivalent if and only if they denote equal elements of the nominal Scott domain model. This is the first full abstraction result we know of for languages combining higher-order functions with some form of locally scoped names, which uses a domain theory based on ordinary extensional functions, rather than using the more intensional approach of game semantics.

To obtain full abstraction, we need to add two new programming language constructs to PNA, one for *existential quantification over names* and one for *'definite description' over names*. Adding only one of them is insufficient, as we give proofs that full abstraction fails if either is left out.

To Niklas,
the most tangible outcome of my PhD,
and my coauthor Kristen.

Acknowledgements

First and foremost I wish to thank my supervisor, Glynn Winskel, whose guidance throughout my PhD years has been invaluable to me. I could not have asked for a more supportive, accessible and kind mentor, and it has been an honour for me to work with such a pioneer of the underlying principles of programming languages. His mathematical intuition and insights helped me countless times to gain a higher-level understanding of the concrete constructions in my work.

Andrew Pitts, my second supervisor, deserves huge amounts of gratitude. I thank him for the manifold discussions we had on nominal sets, among other things, and for his endless patience in answering even the most obvious questions. I hope that the mathematical formulations in my work come even close to being as tasteful and clean as those in his.

I would also like to thank Andrew as well as Sam Staton and Marcelo Fiore for assessing my first and second year reports, thereby providing me with extremely helpful feedback for my ensuing years of research. I especially thank Ian Stark and Marcelo Fiore for being both thorough and fair examiners of my PhD. Their expert feedback lead to tremendous improvements in this thesis.

Over the years I have had meaningful discussions with several individuals on the topic of this thesis, far too many, unfortunately, to mention here. One whose contribution stands out in particular, however, is Nikos Tzevekelos, who shed light on the failure of full abstraction problem . I have also thoroughly benefited from the time spent with the participants of the weekly ECSYM meetings, members of the Programming, Logic and Semantics Group, and other colleagues at the Computer Laboratory. These groups have been a source of both academic inspiration and friendship, and I would particularly like to extend thanks to my office mates Jannis Bulian, Alex Katovsky, Rok Strniša, Nik Sultana and Matej Urbas, as well as to the members of the Rainbow group, whose coffee machine sustained me throughout the course of my research. Michael D. Adams, Julian Chou-Lambert, Leszek Świrski and Janina Voigt have all provided much-needed feedback on my academic writing, for which I am very grateful.

This work would not have been possible without the financial support of the Gates Cambridge Trust, throughout both my Master's degree and my PhD study. In this vein I also acknowledge Trinity College, the Computer Laboratory, die Studienstiftung des deutschen Volkes, the Oregon Programming Languages Summer School, and the ACM SIGPLAN Professional Activities Committee, without whom attending conferences and gaining valuable experiences abroad would have been far more

difficult.

I am indebted to Gert Smolka for first encouraging me to apply to Cambridge – a path that seemed so impossible that it never previously occurred to me.

In my years as a student here, I have been lucky enough to be surrounded by good friends and wonderful communities, all of whom kept me sane during the tougher stretches of my PhD. I thank Trinity College for providing a wonderful (and extremely picturesque) backdrop to my social life and the Gates Cambridge community for too many exciting events and opportunities to mention. I thank Will for the type-theory-filled gym sessions, my fellow runners George and Doug, as well as the other Steffen, for ensuring that there has never been a dull moment.

Finally, I am eternally grateful to my family: my parents, Renate and Gerhard, and my siblings, Daniela, Carolin and Manuel, for their unwavering support of my student career, and to my fiancée Kristen, whose love and encouragement helped me cross the finish line. I also thank Niklas, who only made his debut in 2013 but whose influence on my PhD has been immeasurable. Without all of you, this thesis and the years I have spent on it would be neither possible nor worthwhile. Thank you.

CONTENTS

CHAPTER 1

INTRODUCTION

When are two programs equivalent? This question is difficult to answer. Probably every programmer is familiar with the following situation: you implement a small code change, which was not supposed to affect the overall behaviour, and subsequently your program crashes. The original and the changed program were not equivalent and you overlooked the reason why. But how can we check if two programs are equivalent? Simply running them on different inputs and checking if the outputs are the same is arguably not enough, especially for security critical code running, for example, a business server with confidential information or a nuclear power plant. This thesis examines the application of formal methods to programming languages, in particular to the verification of program equivalences. In other words, we develop mathematical tools for giving proof that two programs are equivalent.

1.1 Background and overview

Contextual equivalence Before we can prove program equivalences, we first need to identify the notion of equivalence we are working with, that is the widespread notion of *contextual equivalence*.

> Two programs p_1, p_2 are **contextually equivalent** $p_1 \cong p_2$ if they give the same *observable results* when put in any context that forms a *complete program*.

What we mean exactly by 'observable results' and 'complete program' may vary. For giving the observable results, it is usually insufficient to say 'I executed the program on my machine and it printed 5 on the screen', because such results are machine- and compiler-dependent. In our formal setting this problem is solved by giving a so-called *operational semantics* (see Plotkin [48]) to the programming language in question: we define how programs are executed on an abstract mathematical machine and the observable results are the results of this abstract machine. Note that with this notion of observable result, we (usually) do not consider performance measurements, such as the computational complexity or actual runtimes of programs.

Contextual equivalence is a very natural notion of program equivalence, as it captures exactly the intuition that one should always be able to replace two equivalent programs in a larger codebase without changing any results. As such, it is of high interest in the programming language community. However, it is hard to prove directly that two programs are contextually equivalent, because we have to consider 'any context' for that. Therefore a wide range of alternative techniques for proving contextual equivalences are being developed, such as logical relations [40] or bisimulations [49], and in this thesis we use a denotational semantics based on domain theory.

Denotational semantics The field of semantics is concerned with giving a formal meaning to phrases of a language. In the operational semantics of programming languages, meaning is given in terms of program execution on an abstract machine. In *denotational semantics*, the meaning of a program p is given in terms of a single mathematical object containing all the necessary information about the program, usually written $[\![p]\!]$ and called the *denotation* of p. What the exact nature of denotations are depends on the application; popular choices are continuous functions between domains (in domain-based denotational semantics) or game strategies between arenas (in game-based denotational semantics).

The distinguishing feature of denotational semantics is that it should be *compositional*, in the sense that the meaning of a program only depends on the meaning of its subprograms. For example, the meaning of a conditional $[\![\texttt{if}\ p_1\ \texttt{then}\ p_2\ \texttt{else}\ p_3]\!]$ should only depend on $[\![p_1]\!]$, $[\![p_1]\!]$ and $[\![p_3]\!]$. Compositionality simplifies the reasoning about programs, because with it an argument about a large program can be broken down into arguments about smaller subprograms, which is not always possible with operational semantics.

For a programming language with denotational semantics, the equality relation between denotations $[\![\_]\!] = [\![\_]\!]$ gives us another natural notion of equivalence between programs. If it can be connected to contextual equivalence, then denotational arguments can be used to reason about contextual equivalence. The connection of this kind with the highest practical relevance is the property of computational adequacy.

> A programming language is **computationally adequate** if denotational equality implies contextual equivalence for any two programs.

Often computational adequacy is considered to be a baseline result that any denotational semantics ought to achieve (in particular to be useful). In a computationally adequate programming language, we can derive $p_1 \cong p_2$ from knowing $[\![p_1]\!] = [\![p_2]\!]$. However, from knowing $[\![p_1]\!] \neq [\![p_2]\!]$ we cannot derive anything about $\cong$ if we only have computational adequacy. The stronger property that the two notions of equivalence coincide, so in particular $[\![p_1]\!] \neq [\![p_2]\!]$ implies $p_1 \ncong p_2$, is called full abstraction.

A programming language is **fully abstract** if denotational equality implies and is implied by contextual equivalence for any two programs.

Obviously full abstraction is the stronger result compared to computational adequacy. As such it is harder to achieve and additionally it often turns out to be very brittle, in the sense that it quickly fails if the language or the semantics are changed. Yet it is only the case in a fully abstract language that contextual equivalence is exactly captured by the denotational semantics.

If a programming language is not fully abstract, but we want it to be, there are two things we can try:

1. Change the denotational semantics to match the programming language.

2. Change the programming language to match the denotational semantics.

In his hugely influential paper "LCF Considered as a Programming Language" [47] Plotkin follows the second approach. He starts with the archetypal functional programming language PCF and develops a denotational semantics for it in terms of the domain-theoretic notion of Scott domain. The Scott domain model turns out not to be fully abstract for PCF. He then proceeds to prove that PCF becomes fully abstract once we add a 'parallel-or' construct to the language. In this thesis we mirror these results with a domain theory based on nominal sets and an extension of PCF that adds some facilities for metaprogramming.

Metaprogramming Whenever one wants to express computations not solely with numbers, but also with structures representing the programs of a programming language, or the formulas of a logic, then one is in a setting called *metaprogramming*. It arises for example in compiler implementation, mechanised theorem proving, or in domain specific languages. We distinguish between an *object-language* and a *meta-language*: we write algorithms in the meta-language for manipulating object-language syntax. For example, if we use the `Coq` system [`http://coq.inria.fr/`] for proving theorems about `C` programs [`http://www.open-std.org/jtc1/sc22/wg14/`], then `C` is the object-language and `Coq` is the meta-language.

In general, 'metaprogramming' can be an umbrella term for many features, such as using the meta-language as an object-language (known as reflection), or exposing certain attributes of the run-time system to the programmer. In this thesis, the term 'metaprogramming' exclusively stands for the representation of and computation over object-language syntax as data, with a clear separation between meta-language and object-language.

Most programming languages have some sort of name-binding construct in their syntax. If we want to use such a language as an object-language, then we run into tedious issues regarding α-equivalence (renaming of bound names) that programmers like to gloss over, but that have to be dealt with in the design of a meta-language. These α-equivalence issues seem trivial at first glance, but become quite intricate once

considered in detail. For example, Aydemir *et al.* [6] write about machine-checked proofs of properties of object-languages in systems like Coq:

> "However, constructing these proofs remains a black art (...) The representation and manipulation of terms with variable binding is a key issue."

Consequently, several approaches have been developed to deal with the issue of binding in object-languages, see our overview in Section 6.1.1. Here we take the user-friendly 'nominal' approach in which bound objects are named, yet α-equivalence is still dealt with by the meta-language.

Nominal sets Nominal sets provide a theory for mathematical structures involving atomic names[1] based on name permutations and the notion of *finite support*. They have been used to develop the semantic properties of binders and locally scoped names, with applications to functional and logic programming, to equational logic and rewriting, to type theory and to interactive theorem proving; see Gabbay [17] and Pitts [44] for recent surveys.

Nominal sets allow us to devise meta-languages that represent object-language binders by using the permutation-based notion of *name abstraction*. Our main motivation to remodel domain theory with nominal sets in this thesis is to gain access to name abstractions. Furthermore, the nominal structures arising in the domain theory inspire the syntax and operational semantics of the binding-related constructs of our meta-language PNA.

Programming with name abstractions In Chapter 4 we define the language PNA (Programming with Name Abstractions), an extension of PCF with name abstractions and Odersky-style [36] locally scoped names. We consider it to be a case study of how to do functional metaprogramming with nominal sets. To exercise its usefulness for metaprogramming, PNA features a representative datatype for programming language syntax, namely a type for α-equivalence classes of λ-calculus syntax. We choose to fix the object-language to be the λ-calculus, in order to have a simple showcase of our approach. It would be possible to extend PNA with generic constructs for user-defined object-languages, as the techniques in this thesis apply to any 'nominal algebraic signature' (see Pitts [44, Definition 8.2]).

Using the nominal constructs, we can avoid tedious α-equivalence issues when defining computation over syntax with binding. For example, in (4.1) we give a PNA program for capture-avoiding substitution, in which we entirely avoid binding-related special conditions that programmers often have to write.

Nominal domain theory As mentioned above, the denotational semantics of PNA is based on a domain theory done in nominal sets. We develop such a nominal domain theory in Chapter 3 up to the notion of nominal Scott domain. Our domain theory is useful for characterising contextual equivalence in PNA, as we derive computational

[1] Names whose only attribute is their identity; Harper [21, Part XII] calls them 'symbols'.

adequacy and full abstraction results in Chapter 5. We observe that a key concept underlying the automata-theoretic research programme of Bojańczyk *et al.* [8], that of being an *orbit-finite subset*, turns out to subsume a notion of topological compactness introduced, for quite different purposes, by Turner and Winskel [63] in their work on nominal domain theory for concurrency. We explain the connection and use it to develop a version of the classic notion of Scott domain within nominal sets[2]. The situation can be summarised as follows:

$$\frac{\text{finite}}{\text{directed}}\text{sets} \sim \frac{\text{orbit-finite}}{\text{uniform-directed}}\text{nominal sets} \tag{1.1}$$

Orbit-finite subsets are an instance of the general idea of considering structures that are finite up to some suitable notion of symmetry.

Finite modulo symmetry Various forms of symmetry are used in many branches of mathematics and computer science. The results in this thesis have to do with using symmetry to extend the reach of computation theory from finite data structures and algorithms to ones that, although they are infinite, *become finite when quotiented by a suitable notion of symmetry*.

In the nominal setting we take symmetry to be permutations of atomic names. Structures that are finite only modulo permutation arise frequently when dealing with nominal sets. Orbit-finite sets (finite sets modulo permutation, Theorem 2.3.38) and name abstractions (can be represented in infinitely many ways, Lemma 2.3.21), are examples of this phenomenon.

1.2 Main results

We outline the main results of this thesis.

- We show that the notion of compactness used in Turner-Winskel nominal domain theory [63] coincides with the notion of orbit-finite subset used by Bojańczyk *et al.* [8]. Specifically, we prove (Theorem 3.3.6) that a finitely supported subset of a nominal set is finite if and only if it is compact with respect to unions that are uniform-directed in the sense of Turner and Winskel.

- We develop a domain theory with nominal sets up to the notion of nominal Scott domain (Definition 3.4.1). We prove that the category of nominal Scott domains is cartesian closed (Theorem 3.4.28), has least fixed points (Proposition 3.4.30) and is closed under forming domains of name abstractions (Theorem 3.6.1). Although there are infinitely many names, the nominal Scott domain of names has some strong finiteness properties. In particular, we show that denotationally the functionals for existential quantification over names and definite description of names are uniform-compact elements of their function

[2]Previous work on denotational semantics with nominal sets [56, 62] has focussed on less sophisticated notions of domain, analogous either to ω-chain complete posets, or to algebraic lattices.

domains (Examples 3.5.2 and 3.5.3) and can be given a structural operational semantics (Sections 5.2.1 and 5.2.2).

- We design the programming language PNA (Programming with Name Abstractions) that extends the well-known language PCF of Plotkin [47] with atomic names that can be abstracted, concreted, swapped, locally scoped and checked for equality. In order to illustrate these facilities for metaprogramming, PNA has a datatype for representing λ-calculus syntax. PNA's operational semantics is inspired by [25, 43]; in particular, its method for deconstructing name abstractions makes use of Odersky-style, 'scope-intrusive' local names [36]. We give a simple denotational semantics for PNA using nominal Scott domains and prove that it is computationally adequate (Theorem 5.3.19) via a suitable logical relation (Definition 5.3.1). Our proof of the fundamental property of the logical relation (Proposition 5.3.5) requires us to define a notion of weak Kleene preorder (Definition 5.3.8), because the standard Kleene preorder (Definition 5.3.6) turns out to be too strong in the presence of Odersky-style local names.

- We show by counter-examples that full abstraction can fail for PNA, even if it is extended with operational versions of either existential quantification over names or definite descriptions over names (Theorems 5.4.5 and 5.4.10). For showing that certain contextual preorder relations hold, the counter-examples rely on extensionality properties of contextual equivalence (Theorem 5.3.25) and a frame-stack operational semantics in the style of Felleisen and Hieb [14].

- We prove that the extension of PNA with both definite description and existential quantification over names is fully abstract with respect to the nominal Scott domain model (Theorem 5.5.20). Our proof of full abstraction is novel (as far as we can tell) compared with other proofs of similar full abstraction results in the literature [12], as we avoid the need to show definability of compact elements at all types. We prove definability only for compact elements of a certain subset of 'simple types' (Defintion 5.5.2) and this suffices for full abstraction because every type is a definable retract of some simple type (Proposition 5.5.12). Our proof of definability at simple types in principle follows the classical argument for PCF and Scott domains, but our nominal setting requires some additional (and non-trivial) arguments concerning permutations (Lemmas 5.5.16 and 5.5.17).

In the literature [2, 23, 34, 64], there are full abstraction results based on game semantics for languages combining higher-order functions with some form of locally scoped names. To the best of our knowledge, our Theorem 5.5.20 is the first full abstraction result for such languages which uses a domain-theoretic semantics based on ordinary extensional functions.

1.3 Contribution details

The content of this thesis is based on conference paper [26] and journal paper [27], both being coauthored with Andrew M. Pitts. The initial line of research of investigating a nominal version of PCF and Scott domain was proposed by the author. The general approach and proof techniques of this thesis were developed in collaboration with Pitts, and the details were worked out by the author. Additionally the counter-examples for full abstraction from Example 5.1.4 were proposed (in a slightly different version) by Nikos Tzevelekos in private communication and the proof technique of Lemmas 5.4.2 and 5.4.7 was developed by Tzevelekos, Pitts and the author. Some material from the nominal domain theory from Chapter 3 also appears in the Pitts' textbook [44, Chapter 11].

CHAPTER 2

TECHNICAL BACKGROUND

In the literature, many publications follow the pattern of 'doing x with nominal sets'. For example, authors replaced x by structural induction [41], unification [66], game semantics [64] or automata theory [9]. In this thesis we replace x by domain theory. In order to provide the necessary background for the rest of this thesis, we introduce some basic notions of domain theory in Section 2.2 and give an extended introduction to nominal sets in Section 2.3. We additionally introduce category theory in Section 2.1, as it allows us to identify some common structure in different areas of this thesis.

2.1 Category theory

Category theory often serves as a unifying framework between theories that helps to highlight similarities between what appear to be at first glance very different approaches. It also can help to identify constructions that 'should be done' in a theory, because they have certain universal properties. Such universal constructions from category theory, such as the 'exponential' from Definition 2.1.5, are sometimes perceived unintuitive in the concrete theory, yet they provide pleasant theoretical properties.

Very little category theory is required for the understanding of this thesis. For later developments, we need the notion of cartesian closed category (Definition 2.1.5) and introduce the category theory leading to it. For a more comprehensive account of category theory we recommend the classical textbook by Mac Lane [28] or the more recent book by Awodey [5].

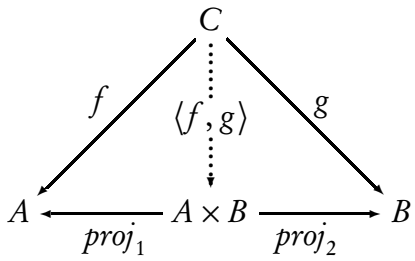
Definition 2.1.1 (**category**)**.** By definition, a *category* $\mathbf{C}$ consists of a collection of *objects* and a collection of morphisms. For each morphism f there are objects $\operatorname{dom} f$ and $\operatorname{cod} f$ called the *domain* and the *codomain* of f. We write $A, B, C, \ldots \in \mathbf{C}$ for objects of $\mathbf{C}$, we write $f, g, h \ldots : A \to B$ for morphisms of $\mathbf{C}$ with domain A and codomain B, and we write $\mathbf{C}(A, B)$ for the collection of all such morphisms. For every object A there is a morphism $\mathrm{id}_A : A \to A$, called the *identity* of A and abbreviated as 'id' if A is clear from the context. For all morphisms $f : A \to B$ and $g : B \to C$ (with matching domain and codomain $\operatorname{cod} f = \operatorname{dom} g$) there is a morphism $g \circ f : A \to C$,

called the *composite* and the operation $\_ \circ \_$ is called *composition*. Each identity and composition must satisfy the following associativity and unit laws:

$$h \circ (g \circ f) = (h \circ g) \circ f$$
$$f \circ \mathrm{id} = f = \mathrm{id} \circ f\,.$$

Definition 2.1.2 (**terminal object**)**.** A *terminal object* in a category $\mathbf{C}$ is an object $1 \in \mathbf{C}$ such that for every object $A \in \mathbf{C}$ there is a unique morphism $*_A : A \to 1$. Terminal objects are unique up to isomorphism, and a proof of that can be found, for example, in Awodey [5, Proposition 2.10].

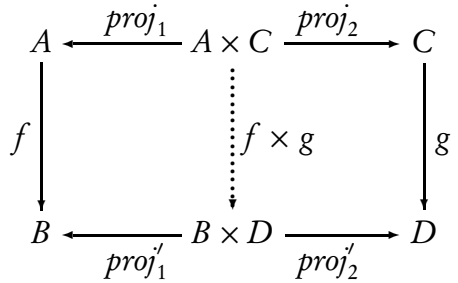
Definition 2.1.3 (**products**)**.** In a category $\mathbf{C}$, the *binary product* of two objects $A, B \in \mathbf{C}$ is an object $A \times B$ (the product) with morphisms $proj_1 : A \times B \to A$ (the first projection) and $proj_2 : A \times B \to B$ (the second projection), such that for every object $C \in \mathbf{C}$ with morphisms $f : C \to A$ and $g : C \to B$ there is a unique morphism $\langle f, g \rangle : C \to A \times B$ satisfying $proj_1 \circ \langle f, g \rangle = f$ and $proj_2 \circ \langle f, g \rangle = g$. The situation can be visualised via the following diagram:



This can be generalised from two to finitely many objects in the obvious way: The *finite product* (of size $n \in \{0, 1, \ldots\}$) of the objects $A_1, \ldots, A_n \in \mathbf{C}$ is an object $A_1 \times \ldots \times A_n$ with morphisms $proj_i : A_1 \times \ldots \times A_n \to A_i$ for $i \in \{1, \ldots, n\}$, such that for every object $C \in \mathbf{C}$ with morphisms $f_i : C \to A_i$ there is a unique morphism $\langle f_1, \ldots, f_n \rangle : C \to A_1 \times \ldots \times A_n$ satisfying $proj_i \circ \langle f_1, \ldots, f_n \rangle = f_i$. Products are unique up to isomorphism, as for example Awodey [5, Proposition 2.17] shows.

We say that a category has binary products if there is a binary product for each pair of objects, and we say that it has finite products if there is a finite product for each finite collection of objects.

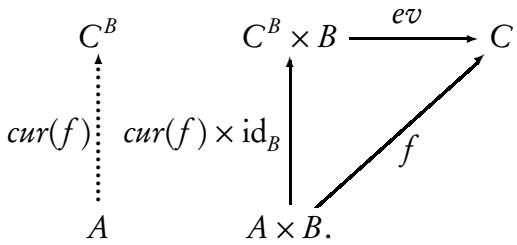
Given a category that has binary products and morphisms $f : A \to B$, $g : C \to D$ we define the morphism $f \times g : A \times C \to B \times D$ by $f \times g \triangleq \langle f \circ proj_1, g \circ proj_2 \rangle$, where $proj_1$ and $proj_2$ are the projections of $A \times C$. The following diagram illustrates the construction.



Proposition 2.1.4 (**finite products = terminal object + binary products**). *A category has finite products if and only if it has a terminal object and binary products.*

Proof. For the 'only if'-direction note that a product of size 0 is a terminal object. For the 'if'-direction we proceed by induction on n to show that $A_1 \times \ldots \times A_n = (\ldots(A_1 \times A_2) \times \ldots) \times A_n$, where for the $n = 1$ case we have that the product of A is A. See, for example, Awodey [5, Section 2.6] for another presentation of this well-known result. □

Definition 2.1.5 (**cartesian closed category**). Let $\mathbf{C}$ be a category with binary products. An *exponential* (or alternatively function space) of two objects $B, C \in \mathbf{C}$ is an object $C^B \in \mathbf{C}$ with a morphism $ev : C^B \times B \to C$ such that for any object $A \in \mathbf{C}$ and morphism $f : A \times B \to C$ there is a unique morphism $cur(f) : A \to C^B$ such that $ev \circ (cur(f) \times \mathrm{id}_B) = f$. This property can be expressed diagrammatically by



A category is said to have exponentials if they exist for all pairs of objects, and it is called *cartesian closed* if it has finite products and exponentials.

Cartesian closed categories are used for giving a denotational semantics to programming languages with higher-order functions, as for example Awodey [5, Section 6.6] describes. Accordingly, the category **Nsd**, which is used for modelling our language PNA from Chapter 4, is cartesian closed (Theorem 3.4.28).

2.2 Domain theory

A key idea behind domain theory in general is to give a formal meaning to a program with potentially infinite behaviour as a limit of approximations. For domain theory based on approximation via a partial order (rather than a metric), limits are joins of chains, or more generally, joins of subsets that are directed. This section gives the relevant definitions of this kind of domain theory and repeats some classical results.

Definition 2.2.1 (**basic sets**). We work in a naive set theory, where we assume that some simple notions such as the membership relation $\in$, the subset relation $\subseteq$, pairs of elements (x, y) or the notion of a finite set are known and can be used.

For any set X, we write $\mathrm{P}X$ for its set of subsets, so $\mathrm{P}X \triangleq \{S \mid S \subseteq X\}$ and call it the *powerset* of X. Similarly, let the *finite powerset* $\mathrm{P_f}X$ be the set of finite subsets of X, that is $\mathrm{P_f}X \triangleq \{S \mid S \subseteq X \wedge S \text{ is finite}\}$, and let $\subseteq_{\mathrm{f}}$ be the corresponding subset relation.

The *cartesian product* (or just product) of two sets X, Y is given by

$$X \times Y \triangleq \{(x,y) \mid x \in X \wedge y \in Y\} \tag{2.1}$$

where (x,y) is called a pair. A *binary relation* R between the sets X and Y is just a subset of their product, so $R \subseteq X \times Y$. A binary relation on X is a relation of the form $R \subseteq X \times X$. This extends to finite products and relations in the obvious way.

As usual, we define a *function* with *domain* X and *codomain* Y, where X and Y are sets, to be a relation $f \subseteq X \times Y$ that satisfies $(\forall x \in X)(\exists y \in Y)\ (x,y) \in f$ and $(\forall x \in X)(\forall y_1, y_2 \in Y)\ (x,y_1) \in f \wedge (x,y_2) \in f \Rightarrow y_1 = y_2$. We write $X \to Y$ or Y^X for the set of all such functions, and $f\,x$ or $f(x)$ for the unique $y \in Y$ such that $(x,y) \in f$. Notationally, functions can also be given as mappings $x \mapsto f\,x$ or as lambda-abstractions $\lambda x \in X \to f\,x$.

A function f is called *injective* if $(\forall x_1, x_2 \in X)\ f\,x_1 = f\,x_2 \Rightarrow x_1 = x_2$, *surjective* if $(\forall y \in Y)(\exists x \in X)\ f\,x = y$ and *bijective* if it is injective as well as surjective. An *endofunction* is a function of the form $f \in X \to X$, that is a function where the domain and codomain are identical. For any sets X_1, X_2, x_3 and functions $f \in X_1 \to X_2$ and $g \in X_2 \to X_3$, the *function composition* of f and g is defined by $g \circ f \triangleq \lambda x \in X_1 \to g(f\,x)$ and this gives a function $g \circ f \in X_1 \to X_3$. The identity function id_X on a set X (where the subscript is often omitted) is given by $\mathrm{id}_X \triangleq \lambda x \in X \to x$. Two sets X, Y are said to be *isomorphic*, written as $X \cong Y$, if there are functions $f \in X \to Y$ and $g \in Y \to X$ satisfying $f \circ g = \mathrm{id}_X$ and $g \circ f = \mathrm{id}_Y$ (and in this case f and g must be bijective).

Definition 2.2.2 (posets and chains). A binary relation $\sqsubseteq$ on a set D is called a *partial order* if it is *reflexive* $(\forall d \in D)\ d \sqsubseteq d$, *transitive* $(\forall d, d', d'' \in D)\ d \sqsubseteq d' \wedge d' \sqsubseteq d'' \Rightarrow d \sqsubseteq d''$ and *antisymmetric* $(\forall d, d' \in D)\ d \sqsubseteq d' \wedge d' \sqsubseteq d \Rightarrow d = d'$. A partial order that is also *total* $(\forall d, d' \in D)\ d \sqsubseteq d' \vee d' \sqsubseteq d$ is called a *total order*.

A *poset* (or partially ordered set) is a set equipped with a partial order and a *chain* (or totally ordered set) is a set equipped with a total order. We often refer to posets and chains by their underlying set and leave the partial order implicit, so we write D instead of $(D, \sqsubseteq)$. A function between posets $f \in D_1 \to D_2$ is called *monotone*, if $d \sqsubseteq d' \Rightarrow (f\,d) \sqsubseteq (f\,d')$ holds for all $d, d' \in D_1$.

Definition 2.2.3 (upper bound). In a poset D, an *upper bound* of a subset $S \subseteq D$ is an element $d \in D$ satisfying

$$(\forall s \in S)\ s \sqsubseteq d\,. \tag{2.2}$$

Subsets that have an upper bound are called *bounded*.

Definition 2.2.4 (joins and least elements). Given a poset D, a *join* (often also called supremum, sup, least upper bound, or lub) of a subset $S \subseteq D$ is an element $d \in D$, which is an upper bound for S (it satisfies (2.2)) and it is additionally the least such

$$(\forall d' \in D)((\forall s \in S)\ s \sqsubseteq d') \Rightarrow d \sqsubseteq d'\,.$$

A join of a subset S, if it exists, is unique and we write it as $\bigsqcup S$. If S is a two-element set $S = \{s_1, s_2\}$ then we write its join as $s_1 \sqcup s_2$.

A *least element* of a poset D is an element $d \in D$ that satisfies

$$(\forall d' \in D)\, d \sqsubseteq d' .$$

It is unique if it exists, we call it *bottom* and write it as $\bot_D$, where the subscript is omitted if D is clear from the context. Least elements can be alternatively defined as the join of the empty set, so $\bot = \bigsqcup \emptyset$. A poset that possesses a least element is called *pointed*.

Definition 2.2.5 (**directed set**)**.** A subset of a poset $S \subseteq D$ is *directed* if it is non-empty and every pair of elements has an upper bound in the subset S:

$$S \neq \emptyset \;\wedge\; ((\forall s_1, s_2 \in S)(\exists s_3 \in S)\; s_1 \sqsubseteq s_3 \wedge s_2 \sqsubseteq s_3) .$$

A *cpo* is a poset where every directed subset has a join.

Example 2.2.6 (**powerset cpo**)**.** The powerset $\mathrm{P}X$ of any set X, endowed with the subset relation $\subseteq$ as partial order, is a poset. It has joins of all subsets, which are given by union, so in particular $\mathrm{P}X$ is a cpo.

Proposition 2.2.7 (**joins of chains = joins of directed sets**)**.** *A poset has joins of all directed sets if and only if it has joins of all chains.*

Proof. This is given in Abramsky and Jung [4, Proposition 2.1.15] and further historic references to this result can be found therein. □

Definition 2.2.8 (**compact element**)**.** We call an element u of a cpo D *compact* if for all directed subsets S of D we have that

$$u \sqsubseteq \bigsqcup S \;\Rightarrow\; (\exists s \in S)\; u \sqsubseteq s . \tag{2.3}$$

A cpo D is called *algebraic* if each element of D is the join of a directed set that only contains compact elements of D. D is called *ω-algebraic* if additionally its set of compact elements is countable.

Proposition 2.2.9 (**compact subsets**)**.** *For any set X, the compact elements of the powerset cpo $\mathrm{P}X$ (see Example 2.2.6) are exactly the finite subsets of X.*

Proof. This is a well-known classical result. Let $E \in \mathrm{P}X$, so $E \subseteq X$ be given. To prove that finiteness of E implies compactness, assume that $E = \{x_1, \ldots, x_n\}$ and $E \subseteq \bigcup S$ for a directed $S \subseteq \mathrm{P}X$. By definition we have for any $i \in \{1, \ldots, n\}$ that $x_i \in S_i$ for some $S_i \in S$. By directedness the sets $S_i, \ldots, S_n$ must have an upper bound $S_E \in S$, and $E \subseteq S_E$ follows by definition.

For the other direction assume that E is compact and consider $\mathrm{P}_{\mathrm{f}}X$, the set of finite subsets of X. It is easy to see that $\mathrm{P}_{\mathrm{f}}X$ is a directed subset of $\mathrm{P}X$ and that $X = \bigsqcup \mathrm{P}_{\mathrm{f}}X$, so $E \subseteq \bigsqcup \mathrm{P}_{\mathrm{f}}X$. By compactness, there must be an $S \in \mathrm{P}_{\mathrm{f}}X$ such that $E \subseteq S$, so E is a subset of a finite set and therefore finite itself. □

Definition 2.2.10 (**Scott domain**)**.** A *Scott domain D* is defined to be a pointed, ω-algebraic cpo, which has also joins for all bounded, finite sets of compact elements.

Lemma 2.2.11 (bounded joins). *Scott domains have joins of all bounded subsets.*

Proof. Given in Stoltenberg-Hansen *et al.* [60, Theorem 1.10]. □

Chapter 3 remodels the kind of domain theory presented in this section with nominal sets, so that it can be used for the denotational semantics of functional metaprogramming languages.

2.3 Nominal sets

In short, nominal sets are sets with a permutation action in which every element is finitely supported. This section gives those definitions and properties of nominal sets that are relevant for the rest of this thesis. A more comprehensive account of nominal sets can be found for example in Gabbay's survey paper [17] or Pitts' book [44].

2.3.1 Atomic names, permutations and finite support

We are interested in the denotational semantics of programs written in languages featuring names that can be tested for equality and locally scoped by binding constructs. Names are assumed to be structureless, so they can be modelled by any countably infinite set. We introduce such atomic names and related notions.

Definition 2.3.1 (atomic names). Fix some countably infinite set $\mathbb{A}$ and call its elements $a, b, c, \ldots \in \mathbb{A}$ *atomic names* (often abbreviated as 'atoms' or 'names'). All we know about atomic names is their identity, so we can check if two atomic names are the same, but we cannot do anything else with them. In particular, atomic names are unordered.

Definition 2.3.2 (permutations). Let $\mathrm{Perm}(\mathbb{A})$ be the the set of *finite permutations* on $\mathbb{A}$, that is, bijective functions $\pi \in \mathbb{A} \to \mathbb{A}$ for which $\pi\, a \neq a$ holds for only finitely many $a \in \mathbb{A}$. We usually call $\pi \in \mathrm{Perm}(\mathbb{A})$ just *permutation* and leave the finiteness implicit. Let also $(a\ b)$ be the *swapping* of $a, b \in \mathbb{A}$, that is, the permutation that swaps the atom a with the atom b, and leaves all other atoms unchanged. Note that $\mathrm{Perm}(\mathbb{A})$ with function composition $\_ \circ \_$ forms a mathematical group with the identity function $\mathrm{id}_{\mathbb{A}}$ as identity and inverse permutations π^{-1} as inverse elements.

What follows are two basic technical lemmas that will be used throughout this thesis.

Lemma 2.3.3 (name equality). *Two atomic names are equal if and only if their permutations are:*

$$(\forall a, b \in \mathbb{A})(\forall \pi \in \mathrm{Perm}(\mathbb{A}))\ a = b \iff \pi\, a = \pi\, b\,.$$

Proof. Follows directly from every π being injective. □

Lemma 2.3.4 (commutativity). *Permutations and swappings commute as follows:*

$$(\forall a, b \in \mathbb{A})(\forall \pi \in \mathrm{Perm}(\mathbb{A}))\ \pi \circ (a\ b) = (\pi\, a\ \pi\, b) \circ \pi\,.$$

Proof. By a simple case distinction on all possible arguments. □

Definition 2.3.5 (permutation action). A *permutation action* on a set X is a binary function $\_ \cdot \_ \in (\mathrm{Perm}(\mathbb{A}) \times X \to X)$ satisfying for all $x \in X$ and $\pi, \pi' \in \mathrm{Perm}(\mathbb{A})$ that $\mathrm{id} \cdot x = x$ and $\pi \cdot (\pi' \cdot x) = (\pi \circ \pi') \cdot x$.

Programs, being finite syntactic objects, only involve finitely many atomic names in their construction; whereas the elements of a set X used to denote program behaviours may well be infinite mathematical objects. In our approach to programming language semantics, we wish to limit our attention to infinite behaviours that depend only upon finitely many atomic names, as doing so yields a rich and well-behaved theory. The notion of nominal set precisely specifies what it means to 'only depend upon finitely many atomic names' entirely in terms of a given permutation action.

Definition 2.3.6 (nominal set). A *nominal set* is a set X with a permutation action as in Definition 2.3.5, where additionally every element has a *finite support*. We say an element $x \in X$ is supported by a set $A \subseteq \mathbb{A}$ of atomic names if every permutation $\pi \in \mathrm{Perm}(\mathbb{A})$ that preserves each name in A also preserves x:

$$((\forall a \in A)\, \pi a = a) \;\Rightarrow\; \pi \cdot x = x\,. \tag{2.4}$$

If an element x is supported by a finite set then it has a unique least support, written $\mathrm{supp}\, x$, and a proof of this property can be found for example in Pitts [44, Proposition 2.3,(2.4)]. An element that has empty support is called *equivariant*.

2.3.2 Constructions on nominal sets

We define some basic nominal sets and give instructions on how more complicated nominal sets can be constructed from simpler ones.

Discrete nominal sets Any set X can be turned into a nominal set, through equipping it with the trivial permutation action:

$$(\forall x \in X)(\forall \pi \in \mathrm{Perm}(\mathbb{A}))\; \pi \cdot x \triangleq x\,. \tag{2.5}$$

In such a nominal set, called *discrete nominal sets*, every element has empty support. For the rest of this thesis, we consider the set of natural numbers $\mathbb{N} \triangleq \{0, 1, 2, \ldots\}$ and the boolean set $\mathbb{B} \triangleq \{\mathrm{true}, \mathrm{false}\}$ to be discrete nominal sets.

Atoms The set of atomic names $\mathbb{A}$ is a nominal set where the permutation action is just function application $\pi \cdot a \triangleq \pi a$ and every atomic name is its own support $\mathrm{supp}\, a = \{a\}$.

Permutations Finite permutations $\mathrm{Perm}(\mathbb{A})$ as in Definition 2.3.2 form a nominal set if we define the permutation action as follows:

$$(\forall \pi, \pi' \in \mathrm{Perm}(\mathbb{A}))\; \pi \cdot \pi' \triangleq \pi \circ \pi' \circ \pi^{-1}\,.$$

The support of a permutation is given by $\mathrm{supp}\, \pi = \{a \in \mathbb{A} \mid \pi a \neq a\}$. Pitts [44, Lemma 2.21] proves these properties and also discusses this choice of permutation action [44, Example 1.6].

Functions The set of all functions $X \to Y$ between two nominal sets X and Y has a permutation action defined by

$$(\forall f \in (X \to Y))(\forall \pi \in \mathrm{Perm}(\mathbb{A}))\ \pi \cdot f \triangleq \lambda x \in X \to \pi \cdot (f(\pi^{-1} \cdot x)). \tag{2.6}$$

However, not every function in $X \to Y$ has finite support. For example, any enumeration of $\mathbb{A}$ (given as surjective function from $\mathbb{N}$ to $\mathbb{A}$) cannot be finitely supported. Another example of functions that cannot have finite support is choice functions (in the sense of the Axiom of Choice), as Pitts [44, Section 2.7] describes.

We will often consider only those functions that have finite support, and define

$$X \to_{\mathrm{fs}} Y \triangleq \{f \in (X \to Y) \mid f \text{ has finite support}\}.$$

It is easy to see that $X \to_{\mathrm{fs}} Y$ is a nominal set. From (2.6) we can derive (as proved in Pitts [44, Lemma 2.17]) that $A \subseteq \mathbb{A}$ supports a function $f \in (X \to Y)$ if and only if it holds for any permutation $\pi \in \mathrm{Perm}(\mathbb{A})$ that

$$((\forall a \in A)\ \pi a = a) \Rightarrow (\forall x \in X)\ f(\pi \cdot x) = \pi \cdot (f\ x).$$

In particular, a function is *equivariant* in the sense of Definition 2.3.6 if and only if $f(\pi \cdot x) = \pi \cdot (f\ x)$ holds for all $\pi \in \mathrm{Perm}(\mathbb{A})$ and $x \in X$.

Products If X_1 and X_2 are nominal sets, then their cartesian product (2.1) is also a nominal set, with the permutation action

$$(\forall x_1 \in X_1)(\forall x_2 \in X_2)(\forall \pi \in \mathrm{Perm}(\mathbb{A}))\ \pi \cdot (x_1, x_2) \triangleq (\pi \cdot x_1, \pi \cdot x_2) \tag{2.7}$$

and $\mathrm{supp}(x_1, x_2) = \mathrm{supp}\, x_1 \cup \mathrm{supp}\, x_2$. Pitts [44, Proposition 2.14] shows that this extends to any finite product of nominal sets.

Lemma 2.3.7 (**equivariance of composition**). *The composition operation* $\_ \circ \_ \in (X \to Y) \times (Y \to Z) \to (X \to Z)$ *for nominal sets* X, Y, Z *is equivariant. Hence we know by Pitts [44, Lemma 2.12(i)] that the composition of two finitely supported functions is finitely supported, and that the same holds for equivariant functions.*

Proof. We calculate using (2.6) that $(\pi \cdot f) \circ (\pi \cdot g) = \lambda x \in X \to (\pi \cdot f)((\pi \cdot g)x) = \lambda x \in X \to \pi \cdot f(\pi^{-1} \cdot \pi \cdot g(\pi^{-1} \cdot x)) = \pi \cdot (f \circ g)$. □

The following lemma is useful for showing the well-definedness of the denotational semantics from Section 4.3.

Lemma 2.3.8 (**equivariance of tupling**). *The function tupling operation* $\langle \_, \_ \rangle \in ((X \to Y) \times (X \to Z)) \to X \to (Y \times Z)$ *defined for any nominal sets* X, Y, Z *by*

$$\langle f, g \rangle\ x \triangleq (f\ x, g\ x) \tag{2.8}$$

is equivariant, and so we have that the tupling of finitely supported functions is finitely supported.

Furthermore, for any nominal sets X and Y, the projection functions $proj_1 \in X \times Y \to X$ and $proj_2 \in X \times Y \to Y$ defined by

$$proj_1(x,y) \triangleq x \tag{2.9}$$
$$proj_2(x,y) \triangleq y \tag{2.10}$$

are equivariant.

Proof. By straightforward calculations using (2.6) and (2.7). □

Subsets For any nominal set X we define the permutation action on $\mathrm{P}X$ by

$$(\forall S \subseteq X)(\forall \pi \in \mathrm{Perm}(\mathbb{A}))\ \pi \cdot S \triangleq \{\pi \cdot x \mid x \in S\}. \tag{2.11}$$

As Pitts [44, Section 1.5] shows, this corresponds to what we get if we consider the characteristic function of a subset as element of $X \to \mathbb{B}$ and apply the permutation action on functions (2.6) to it.

Similarly as for functions, not every subset of a nominal set is finitely supported. For example, any subset of $\mathbb{A}$ is finitely supported if and only if it is finite or cofinite (Pitts [44, Proposition 2.9]). (In general a subset $S \subseteq X$ is defined to be *cofinite* if its complement $X - S$ is finite.) We define the set of finitely supported subsets by

$$\mathrm{P}_{\mathrm{fs}}X \triangleq \{S \subseteq X \mid S \text{ has finite support}\}.$$

and by Pitts [44, Theorem 2.15] this gives a nominal set. Let also $\subseteq_{\mathrm{fs}}$ be the corresponding subset relation. Note that every finite subset of a nominal set is supported by the union of the supports of its elements. If $S \subseteq X$ has empty support (i.e. is equivariant), then it is a nominal set itself (Pitts [44, Lemma 2.22]). This shows that $\mathrm{P}_{\mathrm{f}}X$ is a nominal set, as the permutation of a finite set is finite.

Partial functions A *partial function* is a single-valued subset of the product of its domain and codomain. Hence we define the set of all partial functions between sets X and Y by

$$X \rightharpoonup Y \triangleq \{F \subseteq X \times Y \mid (\forall x \in X)(\forall y, y' \in Y)\,(x,y) \in F \wedge (x,y') \in F \Rightarrow y = y'\}$$

and if it exists we write $F\,x$ for the unique y such that $(x,y) \in F$. We say that $F\,x$ *is defined* if $(x,y) \in F$ for some y. For nominal sets X, Y, partial functions inherit their permutation action by the action for subsets (2.11) and products (2.7). We define the set of those partial functions that are finitely supported by

$$X \rightharpoonup_{\mathrm{fs}} Y \triangleq \{F \in (X \rightharpoonup Y) \mid F \text{ has finite support}\} \tag{2.12}$$

and by Pitts [44, Theorem 2.15] this is a nominal set. Of course every (total) function can be considered to be a partial function that is defined for every argument, and as Pitts [44, Proposition 1.12] shows, the corresponding notions of permutation action and support coincide.

Lemma 2.3.9 (**equivariance of discrete functions**). *Any partial function between two discrete nominal sets is equivariant.*

Proof. Follows directly from the definitions and the simplicity of the permutation action on discrete nominal sets (2.5). □

Example 2.3.10 (**equivariant functions on numbers**). It follows from Lemma 2.3.9 that the (partial) functions $succ : \mathbb{N} \to \mathbb{N}$, $pred : \mathbb{N} \rightharpoonup \mathbb{N}$ and $zero : \mathbb{N} \to \mathbb{B}$ defined by

$$succ\, n \triangleq n + 1 \tag{2.13}$$

$$pred\, n \triangleq \begin{cases} n-1 & \text{if } n > 0 \\ \text{undefined} & \text{otherwise} \end{cases} \tag{2.14}$$

$$zero\, n \triangleq \begin{cases} \text{true} & \text{if } n = 0 \\ \text{false} & \text{otherwise} \end{cases} \tag{2.15}$$

are equivariant.

Category of nominal sets The category **Nom** has nominal sets as its objects and equivariant functions as its morphisms. It is a well-defined category, because the identity function is equivariant and function composition preserves equivariance (Lemma 2.3.7). The discrete nominal set $1 \triangleq \{\star\}$ is a terminal object in **Nom**. Category-theoretic products in **Nom** are given by (2.1), where the mediating morphism is given by (2.8), projections are given by (2.9) and (2.10) and Lemma 2.3.8 ensures that these are well-behaved. **Nom** is also cartesian closed in the sense of Definition 2.1.5 with $X \to_{\text{fs}} Y$ as exponential, as shown by Pitts [44, Theorem 2.19].

2.3.3 Freshness

The notion of an atomic name *not* being in the support of an element, called freshness, is as important as being in the support. When reasoning with nominal sets, we often encounter the situation wherein we have to choose some fresh name, but could have chosen just as well any other fresh name without changing the result. In this section we formalise such some/any properties of fresh names.

Definition 2.3.11 (**freshness relation**). The *freshness relation* $\# \subseteq X \times Y$ between any two nominal sets X, Y is defined by

$$x \# y \triangleq \operatorname{supp} x \cap \operatorname{supp} y = \emptyset. \tag{2.16}$$

Note that since $\operatorname{supp} x$ is a finite set and $\mathbb{A}$ is not, given x we can always find some $a \in \mathbb{A}$ satisfying $a \# x$. For brevity, we use the notation $x_1, \ldots, x_n \# x'_1, \ldots, x'_n$ instead of $(x_1, \ldots, x_n) \# (x'_1, \ldots, x'_n)$ for finite products.

Lemma 2.3.12 (**equivariance of support and freshness**). *The support function* supp *and the freshness relation # are equivariant. So for every finitely supported x and $\pi \in \operatorname{Perm}(\mathbb{A})$ it holds that*

$$\pi \cdot (\operatorname{supp} x) = \operatorname{supp}(\pi \cdot x)$$
$$a \# x \;\Rightarrow\; \pi a \# \pi \cdot x$$

Proof. Equivariance of support is proved in Pitts [44, Proposition 2.11] and equivariance of the freshness relation is a direct consequence of that. □

Notation 2.3.13 (**vectors**). We write vectors of atomic names as $\vec{a} \triangleq a_1, \ldots, a_n$ and define the swapping of two vectors of the same length by

$$(\vec{a}\ \vec{b}) \triangleq (a_1\ b_1) \circ \cdots \circ (a_n\ b_n).$$

Let $\mathbb{A}^{\#n}$ be the set of vectors of length n whose atomic names are distinct from each other. When convenient, we confuse the notation for vectors and sets, and write for example $\vec{a} = \operatorname{supp} x$.

The permutation of a finitely supported element $\pi \cdot x$ can be encoded in terms of multiple swappings (see Definition 2.3.2) between the atomic names in $\operatorname{supp} x$, $\{\pi\, a \mid a \in \operatorname{supp} x\}$ and some fresh atomic names, as Lemma 2.3.14 shows. In PNA this will be used for example in the proofs of Lemmas 5.5.16 and 5.5.16.

Lemma 2.3.14 (permutations through swappings). *Suppose $A \subseteq_{\mathrm{f}} \mathbb{A}$, $\pi \in \mathrm{Perm}$ and $\pi \# A$. For any finitely supported x, define $\vec{a} \triangleq \operatorname{supp} x - A$ with $\vec{a} \in \mathbb{A}^{\#n}$, and let $b_i = \pi(a_i)$ for $i = 1, \ldots, n$. Note that $\vec{b} \in \mathbb{A}^{\#n}$ (because π is a permutation), and $\vec{b} \# A$ (because $\pi \# A$ and $a_i \notin A$). Picking any $\vec{c} \in \mathbb{A}^{\#n}$ with $\vec{c} \# A, \vec{a}, \vec{b}$, we get $\pi \cdot x = ((\vec{b}\ \vec{c}) \circ (\vec{a}\ \vec{c})) \cdot u$.*

Proof. It is easy to check that $(\pi^{-1} \circ (\vec{b}\ \vec{c}) \circ (\vec{a}\ \vec{c}))a = a$ holds for all $a \in \operatorname{supp} x$. Then the defining property (2.4) of $\operatorname{supp} x$ gives us $(\pi^{-1} \circ (\vec{b}\ \vec{c}) \circ (\vec{a}\ \vec{c})) \cdot x = x$, from which the desired property follows by applying π to both sides of the equation. □

The gist of the some/any reasoning for fresh names is encapsulated in the next lemma.

Lemma 2.3.15 (unique fresh element). *For any finitely supported, partial function $F \in (\mathbb{A} \rightharpoonup_{\mathrm{fs}} X)$, where X is a nominal set, we have that if F satisfies*

$$(\exists a \in \mathbb{A})\ a \# F \wedge F\, a \text{ is defined} \wedge a \# (F\, a) \tag{2.17}$$

then there is a unique element $x \in X$ such that

$$(\forall b \in \mathbb{A})\ b \# F \Rightarrow (F\, b = x \wedge b \# x). \tag{2.18}$$

Proof. See (2.12) for a definition of $\mathbb{A} \rightharpoonup_{\mathrm{fs}} X$ and let F and a be given such that they satisfy (2.17). Define $x \triangleq F\, a$ and let any $b \in \mathbb{A}$ be given such that $b \# F$. It is a consequence of Pitts [44, Proposition 2.25] that $b \# x$. Hence $a, b \# F, x$ and by $(a, x) \in F$ and (2.11) we get $(b, x) = (a\ b) \cdot (a, x) \in (a\ b) \cdot F = F$, so (2.18) holds. □

Definition 2.3.16 (freshness quantifier). Given $F \in (\mathbb{A} \rightharpoonup_{\mathrm{fs}} X)$ for which (2.17) holds, we define $(\text{fresh } a \text{ in } F\, a) \in X$ to be the unique element satisfying (2.18) from Lemma 2.3.15.

If F is a total function with a boolean codomain $F \in (\mathbb{A} \to_{\mathrm{fs}} \mathbb{B})$ then (2.17) is always satisfiable, so for any such F the element 'fresh a in $F\,a$' exists. With this we can define the *freshness quantifier* by

$$(\text{И}a)\, F\, a \triangleq \text{fresh}\, a \,\text{in}\, F\, a\,. \tag{2.19}$$

It is called a quantifier because any such F encodes a finitely supported property of atomic names, where $\{a \in \mathbb{A} \mid F\, a = \text{true}\}$ are the atomic names satisfying the property.

The next lemma explains how the freshness quantifier is related to freshness. It justifies the use of the phrase 'for some/any fresh atomic name a the statement $\varphi(a)$ holds' when we mean '$(\text{И}a)\, \varphi(a)$ holds'.

Lemma 2.3.17 (some/any property). *For any $F \in (\mathbb{A} \to_{\mathrm{fs}} \mathbb{B})$ the following statements are equivalent:*

(i) $(\text{И}a)\, F\, a$

(ii) $\{a \in \mathbb{A} \mid F\, a = \text{true}\}$ *is cofinite*

(iii) $(\exists a \in \mathbb{A})\, a \# F \wedge F\, a' = \text{true}$

(iv) $(\forall a \in \mathbb{A})\, a \# F \Rightarrow F\, a' = \text{true}.$

Proof. It is straightforward to show that $\{a \in \mathbb{A} \mid F\, a = \text{true}\}$ is supported by $\operatorname{supp} F$. We can apply Pitts [44, Lemma 3.7] to this to obtain that (ii), (iii) and (iv) are equivalent. It also follows directly from the definition of И in (2.19) and (2.18) that $(i) \Rightarrow (iii)$ and $(iv) \Rightarrow (i)$. □

The freshness quantifier is applicable in a broad range of situations, because many notions that we define turn out to be finitely supported. The *Finite Support Principle* in Pitts [44, Section 2.5] states that "any function or relation that is defined from finitely supported functions and subsets using classical higher-order logic is itself finitely supported, provided we restrict any quantification over functions or subsets to range over ones that are finitely supported".

2.3.4 Name abstraction

Name abstractions are generalised forms of α-equivalence classes that work not just on syntax, but on any nominal set. This ability to express α-equivalence in a general mathematical framework formed the original motivation of why nominal sets were developed for computer science by Gabbay and Pitts [20].

Definition 2.3.18 (name abstraction). Given a nominal set X, we get an equivalence relation $\simeq$ on $\mathbb{A} \times X$ by defining

$$(a, x) \simeq (a', x') \triangleq (\text{И}b)\, (a\ b) \cdot x = (a'\ b) \cdot x' \tag{2.20}$$

The *abstraction set* $[\mathbb{A}]X$ is the set obtained by quotienting $\mathbb{A} \times X$ by this equivalence relation, so $[\mathbb{A}]X \triangleq (\mathbb{A} \times X)/\simeq$. We also write $\langle a \rangle x$ for the $\simeq$-equivalence class of (a, x), and call it a *name abstraction*. Name abstractions are a generalised form of α-equivalence classes, because X itself may not consist of concrete syntactic data (we just need to know how permutations act on its elements).

Proposition 2.3.19 (**name abstractions as a nominal set**). *For any nominal set X, the definition*

$$(\forall \pi \in \mathrm{Perm}(\mathbb{A}))\ \pi \cdot \langle a \rangle x \triangleq \langle \pi\, a \rangle (\pi \cdot x) \tag{2.21}$$

gives a permutation action for $[\mathbb{A}]X$. This turns $[\mathbb{A}]X$ into a nominal set, where the support of a name abstraction satisfies

$$\mathrm{supp} \langle a \rangle x \;=\; (\mathrm{supp}\, x) - \{a\}\,. \tag{2.22}$$

Proof. Pitts [44, Lemma 4.1] shows that the relation $\simeq$ from (2.20) is indeed an equivalence relation, which is equivariant by Pitts [44, (4.9)]. Therefore by Pitts [44, Section 2.9] the quotient $[\mathbb{A}]X$ is a nominal set whose permutation action satisfies (2.21). The proof of (2.22) is given in Pitts [44, Proposition 4.5] □

Remark 2.3.20 (**definitions on name abstractions**). In (2.21) and (2.22) we gave definitions on name abstractions, by directly giving the definition on a concrete representative of the $\simeq$-equivalence class of the name abstraction. Technically we gave a definition on $\mathbb{A} \times X$ instead of $[\mathbb{A}]X$. Please note that in doing so, we run into the risk that our definitions give different results on different representatives of the same $\simeq$-equivalence class, which would mean that our definition is illegal for name abstractions. Despite this risk of ill-definedness, we still prefer to stick to this style of definition, because we think it is easier to read; see for example Definitions 2.3.23 and 3.6.10. Technically, whenever we give a definition on concrete representatives of name abstractions, we need to check well-definedness by proving that our definition is independent from the choice of the representative. Such proofs can be carried out conveniently by using Pitts [44, Theorem 4.15, Corollary 4.17], and they were carried out for all definitions in this thesis.

Lemma 2.3.21 (**equality between name abstractions**). *For any name abstractions, we have that $\langle a \rangle x = \langle a' \rangle x'$ holds if and only if either $a = a'$ and $x = x'$ or $a \# a', x'$ and $x = (a\ a') \cdot x'$.*

Proof. Given in Pitts [44, Lemma 4.3]. □

Syntax as a nominal set The syntax of programming languages with binding can be modelled as a nominal set, where the bound entities are atomic names and binding is modelled by name abstraction. This strategy works for any 'nominal algebraic signature' as in Pitts [44, Definition 8.2].

Example 2.3.22 (λ-**calculus syntax as a nominal set**). By using the notation $X/\sim$ for the quotient of the set X by the equivalence relation $\sim$, the syntax of the λ-calculus can be defined by

$$\Lambda_\alpha \triangleq \{t ::= a \mid t\ t \mid \lambda a.t\}/=_\alpha \quad (\text{where } a \in \mathbb{A}). \tag{2.23}$$

Following standard conventions, $\{t ::= a \mid t\ t \mid \lambda a.t\}$ is defined to be the least set that is closed under the syntactic rules $t ::= a \mid t\ t \mid \lambda a.t$. The *$\alpha$-equivalence relation* $=_\alpha$ is defined to be the least relation closed under the rules

$$\frac{}{a =_\alpha a} \qquad \frac{t_1 =_\alpha t_1' \qquad t_2 =_\alpha t_2'}{t_1\,t_2 =_\alpha t_1'\,t_2'} \qquad \frac{(a_1\,a)\cdot t_1 =_\alpha (a_2\,a)\cdot t_2 \qquad a \notin \mathrm{var}(a_1,t_1,a_2,t_2)}{\lambda a_1.t_1 =_\alpha \lambda a_2.t_2}$$

where $\mathrm{var}(t)$ is the function that lists all atomic names occurring in t. Note that the syntax here is explicitly quotiented by α-equivalence, whereas the languages in Chapters 4 and 5 leave the quotienting implicit. Pitts [44, Section 4.1] shows that Λ_α forms a nominal set where the support of each syntactic term (called *λ-term* and written $[t]_\alpha$) is the set of its free variables (technically these are free names, as λ-calculus variables are modelled as atomic names here), so

$$\mathrm{supp}[t]_\alpha = \mathrm{fvar}\,t$$

where $\mathrm{fvar}\,a \triangleq \{a\}$, $\mathrm{fvar}(t\ t') \triangleq (\mathrm{fvar}\,t) \cup (\mathrm{fvar}\,t')$ and $\mathrm{fvar}(\lambda a.t) \triangleq (\mathrm{fvar}\,t) - \{a\}$. λ-terms are a special case of 'nominal algebraic terms' as in Pitts [44, Definition 8.9]. As such, Λ_α is the initial algebra of the 'nominal algebraic functor' (see Pitts [44, Definition 8.12]) $T : \mathbf{Nom} \to \mathbf{Nom}$ defined by $TX \triangleq X \mapsto \mathbb{A} + ([\mathbb{A}]X) + (X \times X)$. It comes equipped with well-defined equivariant functions $var \in \mathbf{Nom}(\mathbb{A}, \Lambda_\alpha)$, $app \in \mathbf{Nom}(\Lambda_\alpha \times \Lambda_\alpha, \Lambda_\alpha)$ and $lam \in \mathbf{Nom}([\mathbb{A}]\Lambda_\alpha, \Lambda_\alpha)$ defined by

$$\begin{aligned} var\,a &\triangleq [a]_\alpha \\ app([t_1]_\alpha, [t_2]_\alpha) &\triangleq [t_1\,t_2]_\alpha \\ lam\,\langle a\rangle[t]_\alpha &\triangleq [\lambda a.t]_\alpha \end{aligned}$$

as noted in Pitts [44, Example 8.14]. It is a special case of Pitts [44, Theorem 8.15] that for each nominal set X there is an equivariant function $case_X \in \mathbf{Nom}(\Lambda_\alpha \times (\mathbb{A} \to_{\mathrm{fs}} X) \times (\Lambda_\alpha \to_{\mathrm{fs}} \Lambda_\alpha \to_{\mathrm{fs}} X) \times ([\mathbb{A}]\Lambda_\alpha \to_{\mathrm{fs}} X), X)$ defined by

$$case_X(e, f_1, f_2, f_3) \triangleq \begin{cases} f_1\,a & \text{if } e = [a]_\alpha \\ f_2[t_1]_\alpha[t_2]_\alpha & \text{if } e = [t_1\,t_2]_\alpha \\ f_3(\langle a\rangle[t]_\alpha) & \text{if } e = [\lambda a.t]_\alpha\,. \end{cases}$$

Our language PNA from Chapter 4 has a ground type `term` whose denotation is the nominal set Λ_α from above.

2.3.5 Concretion and restriction

Elements of $[\mathbb{A}]X$ can be constructed by forming name abstractions $(a, x) \mapsto \langle a\rangle x$. By (2.21) this operation is equivariant and hence gives a morphism $\langle \_ \rangle \_ \in \mathbf{Nom}(\mathbb{A} \times X, [\mathbb{A}]X)$. The way to deconstruct elements of $[\mathbb{A}]X$ is the concretion operation defined next.

Definition 2.3.23 (concretion). Given $X \in \mathbf{Nom}$, the operation of *concretion* of the name abstraction $\langle a \rangle x \in [\mathbb{A}]X$ at the atomic name $a' \in \mathbb{A}$ is defined by

$$(\langle a \rangle x) @ a' \triangleq \begin{cases} x & \text{if } a = a' \\ (a\ a') \cdot x & \text{if } a \neq a' \wedge a' \# x \\ \text{undefined} & \text{otherwise.} \end{cases}$$

Proposition 2.3.24 (well-definedness and equivariance of concretion). *Concretion is a well-defined partial function* $\_ @ \_ \in ([\mathbb{A}]X \times \mathbb{A} \rightharpoonup X)$*, where for* $y \in [\mathbb{A}]X$ *we have that* $y @ a$ *is defined if and only if* $a \# y$*. It is also equivariant in the sense that* $(\forall \pi \in \mathrm{Perm}(\mathbb{A}))\ \pi \cdot (y @ a) = (\pi \cdot y) @ (\pi a)$*, where* $y @ a$ *is defined if and only if* $(\pi \cdot y) @ (\pi a)$ *is.*

Proof. Well-definedness is a consequence of Lemma 2.3.21. The proof that $\_@\_$ is defined exactly at $\{(y, a) \mid a \# y\}$ follows from Definition 2.3.23 and (2.22). Equivariance follows from Lemmas 2.3.3, 2.3.4, 2.3.12 and (2.21). □

Name abstraction satisfies a form of η-expansion, as the next lemma shows.

Lemma 2.3.25 (abstraction η-law). *For each* $y \in [\mathbb{A}]X$ *and* $a \in \mathbb{A}$ *we have*

$$a \# y \;\Rightarrow\; \langle a \rangle (y @ a) = y\,.$$

The concretion $y @ a$ *is the unique element of* X *that satisfies this property for a given* $a \# y$*.*

Proof. Given in Pitts [44, Proposition 4.9]. Note that using the freshness quantifier the property can be summarised by $(\forall y \in [\mathbb{A}]X)(\text{M}a)\ \langle a \rangle (y @ a) = y$. □

Whenever we want to form the concretion $y @ a$ we need to make sure that a is not in the support of y, otherwise the concretion is not defined. This partiality is a problem, in particular if we want to use concretion as a programming language construct, as we do in Chapter 4. See also Section 6.1.2 for a discussion of work related to this issue. Under certain conditions the situation can be remedied: If the nominal sets involved possess a name restriction operation, then concretion can be turned into a total function.

Definition 2.3.26 (name restriction operation). A *name restriction* operation on a nominal set X is by definition an equivariant function $\_\backslash\_ \in \mathbf{Nom}(\mathbb{A} \times X, X)$ that satisfies the following three properties:

$$(\forall a \in \mathbb{A})(\forall x \in X)\ a \# a\backslash x \tag{2.24}$$

$$(\forall a \in \mathbb{A})(\forall x \in X)\ a \# x \Rightarrow a\backslash x = x \tag{2.25}$$

$$(\forall a_1, a_2 \in \mathbb{A})(\forall x \in X)\ a_1\backslash(a_2\backslash x) = a_2\backslash(a_1\backslash x)\,. \tag{2.26}$$

Since $\_\backslash\_$ is equivariant, property (2.24) is equivalent to asking that $a\backslash\_$ be a binding operation, that is, satisfy $a' \# (a, d) \Rightarrow a\backslash d = a'\backslash((a\ a') \cdot d)$. Properties (2.25) and

(2.26) are basic structural properties that one expects any notion of local scoping to have; see the discussion at the start of Pitts [44, Section 9.1].

A nominal set that is equipped with such a name restriction operation is called *nominal restriction set*. There can be several different name restriction operations for the same nominal set, for example, Lemma 2.3.28 below shows that every equivariant element gives one. By convention, we still refer to the nominal restriction set by its underlying set and refer to the according name restriction operation just by __, as the particular restriction operation used can usually be inferred from the context.

Proposition 2.3.27 (**total concretion**). *For any nominal restriction set X, we can extend the partial concretion function from Definition 2.3.23 to a total and equivariant function $@^t \in \mathbf{Nom}([\mathbb{A}]X \times \mathbb{A}, X)$ by*

$$(\langle a\rangle x)@^t\, a' \triangleq \begin{cases} x & \text{if } a = a' \\ a\backslash(a\ a')\cdot x & \text{otherwise} \end{cases}$$

and this total concretion agrees with the usual partial concretion at fresh names

$$a \# y \;\Rightarrow\; y\,@^t\, a = y\,@\,a\,.$$

Proof. Given in Pitts [44, Corollary 9.19]. □

We continue with some constructions of how we can give name restriction operations to nominal sets.

Lemma 2.3.28 (**restriction with an equivariant element**). *Let X be a nominal set X and let $x \in X$ be an equivariant element, that is* $\operatorname{supp} x = \emptyset$. *Then X possesses a name restriction operation given by*

$$(\forall a \in \mathbb{A})(\forall x' \in X)\, a\backslash x' \triangleq \begin{cases} x' & \text{if } a \# x' \\ x & \text{otherwise.} \end{cases}$$

Proof. This construction is also given in Pitts [44, Example 9.5]. It easy to check that it gives an equivariant function satisfying (2.24), (2.25) and (2.26). □

Lemma 2.3.29 (**restriction for products**). *If X_1 and X_2 are nominal restriction sets, then their cartesian product $X_1 \times X_2$ is one with the restriction operation*

$$(\forall a \in \mathbb{A})(\forall x_1 \in X_1)(\forall x_2 \in X_2)\, a\backslash(x_1, x_2) \triangleq (a\backslash x_1, a\backslash x_2)\,.$$

Proof. We showed in Section 2.3.2 that $X_1 \times X_2$ is a nominal set and Pitts [44, Theorem 9.6] proves the name restriction properties. □

Lemma 2.3.30 (**restriction for functions**). *For any nominal set X_1 and nominal restriction set X_2, we have that $X_1 \to_{\mathrm{fs}} X_2$ is a nominal restriction set once we equip it with the restriction operation*

$$(\forall a \in \mathbb{A})(\forall f \in X_1 \to_{\mathrm{fs}} X_2)\, a\backslash f \triangleq \lambda x \in X_1 \to \text{fresh } a' \text{ in } a'\backslash(((a\ a')\cdot f)\,x)$$

and by Lemma 2.3.15 this is equivalent to defining

$$(\forall a \in \mathbb{A})(\forall f \in X_1 \to_{\mathrm{fs}} X_2)(\forall x \in X_1)\, a \# x \;\Rightarrow\; (a\backslash f)\,x \triangleq a\backslash(f\ x)\,.$$

Proof. This is Pitts [44, Theorem 9.7]. □

Lemma 2.3.31 (**restriction for name abstractions**). *The abstraction set* $[\mathbb{A}]X$ *has a name restriction operation whenever* X *does. It is given by*

$$(\forall a, a' \in \mathbb{A})(\forall x \in X)\, a \neq a' \Rightarrow a\backslash\langle a'\rangle x \triangleq \langle a'\rangle(a\backslash x).$$

Proof. Pitts [44, Theorem 9.18] shows that this determines a name restriction operation on $[\mathbb{A}]X$, alongside some further properties. □

The name restriction operations above will be extended to nominal Scott domains in Section 3.6.2, such that they can be used in Section 4.3.2 as denotation for the Odersky-style local scoping construct $\nu a.e$ in PNA.

2.3.6 Orbit-finiteness

Many branches of mathematics and computer science use structures that, although they are infinite, become finite when quotiented by a suitable notion of symmetry. In the nominal setting, symmetry is expressed by permutations of atomic names and in this section extends the notion of finite subset to subsets that are finite modulo permutation: the orbit-finite subsets.

Definition 2.3.32 (**orbit**). Given a nominal set X, the *orbit* of an element $x \in X$ is the set of elements that can be reached from x through permutations. Formally, the orbit of x is the set $\{x' \in X \mid (\exists \pi \in \mathrm{Perm}(\mathbb{A}))\ \pi \cdot x = x'\}$.

Any nominal set can be partitioned into its set of orbits, where the cardinality of this set can be finite or infinite. For example, the set of atomic names $\mathbb{A}$ has just one orbit; $\mathbb{A} \times \mathbb{A}$ has two, namely $\{(a,a) \mid a \in \mathbb{A}\}$ and $\{(a,a') \mid a,a' \in \mathbb{A} \wedge a \neq a'\}$; and in general $\mathbb{A}^n$ has finitely many orbits, corresponding to equivalence relations on the finite set $\{0, 1, \ldots, n-1\}$. Contrastingly, the nominal set $\mathbb{A}^*$ (which is defined to be the set of finite tuples of atomic names) has infinitely many orbits, since tuples of different length cannot be in the same orbit.

Remark 2.3.33 (**finitely many orbits = finitely presentable**). The category-theoretic generalisation of the order-theoretic notion of directed join is the notion of *filtered colimit*. Furthermore, compactness with respect to directed joins generalises to the notion of an object being *finitely presentable* (fp): an object X in a (locally small) category $\mathbf{C}$ with filtered colimits is fp if the hom-functor $\mathbf{C}(X, \_) : \mathbf{C} \to \mathbf{Set}$ preserves filtered colimits. $\mathbf{C}$ is called *locally finitely presentable* (lfp) if every object is the filtered colimit of fp objects. The category **Nom**, being a Grothendieck topos, is lfp. Although we will not need the characterisation here, it is worth remarking that Petrişan [37, Proposition 2.3.7] shows that *a nominal set is an fp object of* **Nom** *if and only if its set of orbits is finite.* A more detailed argument is given in Pitts [44, Section 5.3].

Definition 2.3.34 (**orbit-finite subset**). A finitely supported subset of a nominal set X is called *orbit-finite* if it is contained in the union of only finitely many orbits of X. We write $\mathrm{P}_{\mathrm{of}}X$ for the collection of orbit-finite subsets of X and $\subseteq_{\mathrm{of}}$ for the corresponding subset relation.

Note that an orbit-finite subset may well have infinitely many different elements. For example, $\mathbb{A}$ is an orbit-finite subset of itself. Therefore, in order to compute with orbit-finite subsets one needs an effective presentation of them and of operations upon them. The following notion turns out to give an alternative characterisation of orbit-finite subsets that is suitable for computation.

Definition 2.3.35 (**hull**). Let X be a nominal set. Given finite subsets $A \subseteq_{\mathrm{f}} \mathbb{A}$ and $F \subseteq_{\mathrm{f}} X$, define

$$\mathrm{hull}_A F \triangleq \{\pi \cdot x \mid \pi \in \mathrm{Perm}(\mathbb{A}) \wedge \pi \# A \wedge x \in F\}$$

and call such a set a *hull*.

What follows are some technical properties of hulls that lead to the main result of this section (Theorem 2.3.38): orbit-finite subsets are exactly those sets that can be expressed as a hull.

Lemma 2.3.36 (**hull properties**). *For every nominal set X, $A \subseteq_{\mathrm{f}} \mathbb{A}$ and $F \subseteq_{\mathrm{f}} X$ it holds that*

$$\mathrm{supp}(\mathrm{hull}_A F) \subseteq A \tag{2.27}$$

$$\mathrm{hull}_A F = \mathrm{hull}_{\mathrm{supp}(\mathrm{hull}_A F)} F \tag{2.28}$$

$$(\forall \pi \in \mathrm{Perm}(\mathbb{A}))\ \pi \cdot \mathrm{hull}_A F = \mathrm{hull}_{\pi \cdot A} \pi \cdot F\,. \tag{2.29}$$

Proof. Turner [62, Lemma 3.4.3.6, Lemma 3.4.3.9, Lemma 3.4.3.8] gives proofs for all three properties. □

Lemma 2.3.37 (**name extension for hulls**). *The finite set of atomic names in a hull can be extended without changing the hull itself. Formally we have*

$$(\forall A \subseteq A' \subseteq_{\mathrm{f}} \mathbb{A})(\forall F \subseteq_{\mathrm{f}} X)(\exists F' \subseteq_{\mathrm{f}} X)\ \mathrm{hull}_A F = \mathrm{hull}_{A'} F'\,.$$

Proof. This result was proved independently by Turner [62, Lemma 3.4.3.5] and Bojańczyk *et al.* [8, Lemma 3]. □

Theorem 2.3.38 (**orbit-finite subset = hull**). *A subset of a nominal set X is orbit-finite if and only if it is a hull. For finite subsets $A \subseteq_{\mathrm{f}} \mathbb{A}$ and $F \subseteq_{\mathrm{f}} X$, we have* $\mathrm{hull}_A F \in \mathrm{P}_{\mathrm{of}} X$. *Conversely, every $S \in \mathrm{P}_{\mathrm{of}} X$ is of the form* $\mathrm{hull}_{\mathrm{supp}\, S} F$ *for some $F \subseteq_{\mathrm{f}} X$.*

Proof. Every $\mathrm{hull}_A F$ is finitely supported (2.27) and is contained in the finite union of the orbits of the elements in F, showing the 'if'-direction. For the 'only if'-direction let S be orbit-finite, so it is finitely supported and contained in the union of the orbits of the elements of some finite set $F \subseteq_{\mathrm{f}} X$. Using hulls, this can be expressed as $S \subseteq \mathrm{hull}_\emptyset F$. By Lemma 2.3.37 there exists $F' \subseteq_{\mathrm{f}} X$ with $\mathrm{hull}_\emptyset F = \mathrm{hull}_{\mathrm{supp}\, S} F'$ and from this it follows (as in Pitts [44, Proposition 5.25]) that $S = \mathrm{hull}_{\mathrm{supp}\, S}(F' \cap S)$. □

Classically, the finite powerset of a countable set is countable. This result carries over to orbit-finite powersets.

Corollary 2.3.39 (**orbit-finite countability**). *For any countable nominal set X, its set of orbit-finite subsets $\mathrm{P}_{\mathrm{of}}X$ is countable.*

Proof. It is well-known that finite powersets and cartesian products of countable sets are countable, so $\mathrm{P}_{\mathrm{f}}A \times \mathrm{P}_{\mathrm{f}}X$ is countable. By Theorem 2.3.38 any $S \subseteq_{\mathrm{of}} X$ can be written as $S = \mathrm{hull}_A F$ for some $A \subseteq_{\mathrm{f}} \mathbb{A}$ and $F \subseteq_{\mathrm{f}} X$. Hence S can be represented by an element of $\mathrm{P}_{\mathrm{f}}A \times \mathrm{P}_{\mathrm{f}}X$, showing that there is an injection from $\mathrm{P}_{\mathrm{of}}X$ to $\mathrm{P}_{\mathrm{f}}A \times \mathrm{P}_{\mathrm{f}}X$. □

2.3.7 Uniform support

We introduce the notion of a finitely supported set where each element is supported by the same finite set of names. It will become relevant in Section 3.2, where we connect it to the notion of directedness in nominal domain theory.

Definition 2.3.40 (**uniform support**). A finitely supported subset S of a nominal set X is said to be *uniformly supported*, if there is a finite set of names $A \subseteq_{\mathrm{f}} \mathbb{A}$ that supports all elements of S in the sense of Definition 2.3.6

$$(\forall x \in S)(\forall \pi \in \mathrm{Perm}(\mathbb{A}))\ \pi \# A \Rightarrow \pi \cdot x = x\,.$$

All elements of a uniformly supported subset are supported by the support of the subset, as the next Lemma shows.

Lemma 2.3.41 (**support of a uniformly supported subset**). *Let S be a uniformly supported subset of a nominal set, then we have*

$$(\forall x \in S)\ \mathrm{supp}\, x \subseteq \mathrm{supp}\, S\,.$$

Proof. Given in Pitts [44, Lemma 5.28]. □

CHAPTER 3

NOMINAL DOMAIN THEORY

In Chapter 2 we introduced the theories of domains and nominal sets. In this chapter we combine these to develop a nominal domain theory, up to the notion of nominal Scott domain. The main advantage of this approach is that the nominal concept of name abstractions (Section 2.3.4) becomes available in the domain theory. As Example 2.3.22 already indicated, name abstractions are very useful to model α-equivalence in syntax. In a metaprogramming language this means that object-level syntax with binding can be defined and manipulated directly, keeping issues concerning α-equivalence 'under the hood' of the metaprogramming language, so that the programmer need not worry about it.

The nominal domain theory of this chapter will be applied in Chapter 4 to give a denotational semantics to the metaprogramming language PNA, a variant of PCF with name abstractions and a ground type of object-level syntax of the λ-calculus.

3.1 Nominal posets

Posets and joins form the basis of classical domain theory. This section rephrases these concepts in terms of nominal sets and gives a characterisation of joins in abstraction sets (Lemma 3.1.7 and Appendix A.1).

Definition 3.1.1 (**nominal poset**)**.** A *nominal poset* is a nominal set D equipped with a partial order $\sqsubseteq$ that is respected by the permutation action:

$$(\forall d, d' \in D)(\forall \pi \in \mathrm{Perm}(\mathbb{A}))\; d \sqsubseteq d' \;\Rightarrow\; \pi \cdot d \sqsubseteq \pi \cdot d'\,. \tag{3.1}$$

Note that (3.1) holds if and only if $\sqsubseteq$ itself is an equivariant subset of $D \times D$.

Remark 3.1.2 (**partial order and support**). The partial order of a nominal poset is not necessarily in any connection with the support relation, as Pitts [44, Remark 11.2] notes. Hence $d \sqsubseteq d'$ does not imply and is not implied by $\mathrm{supp}\, d \subseteq \mathrm{supp}\, d'$ in general.

Example 3.1.3 (**powerset nominal poset**). For any nominal set X, the nominal set $\mathrm{P}_{\mathrm{fs}}X$ of finitely supported subsets of X, ordered by subset inclusion, is a nominal poset.

Proposition 3.1.4 (**joins are equivariant**). *Given any nominal poset D, and a finitely supported subset $S \subseteq_{\mathrm{fs}} D$ whose join $\bigsqcup S$ exists in D, then any permutation $\pi \in \mathrm{Perm}(\mathbb{A})$ satisfies*

$$\pi \cdot \bigsqcup S = \bigsqcup \pi \cdot S\,. \tag{3.2}$$

Proof. We can check by simple calculations that $\pi \cdot \bigsqcup S$ is an upper bound for $\pi \cdot S$ and that it is the least such. Note that this also implies that

$$\operatorname{supp} \bigsqcup S \subseteq \operatorname{supp} S\,. \tag{3.3}$$

□

Lemma 3.1.5 (**bottom is equivariant**). *The least element of a nominal poset (often called bottom as in Definition 2.2.4) is necessarily equivariant.*

Proof. Let $\bot$ be the (unique) least element of a nominal poset D and let any $\pi \in \mathrm{Perm}(\mathbb{A})$ be given. We show that $\pi \cdot \bot = \bot$ by proving that $\pi \cdot \bot$ is a least element. For that let any $d \in D$ be given, then $\bot \sqsubseteq \pi^{-1} \cdot d$ because $\pi^{-1} \cdot d \in D$ and $\bot$ is least. Finally, using (3.1) we get $\pi \cdot \bot \sqsubseteq d$. □

Proposition 3.1.6 (**abstraction poset**). *For any nominal poset D, we can turn its set of name abstractions $[\mathbb{A}]D$ into a nominal poset by defining*

$$\langle a \rangle d \sqsubseteq \langle a' \rangle d' \triangleq (\mathcal{N} b)\,(a\ b) \cdot d \sqsubseteq (a'\ b) \cdot d'\,. \tag{3.4}$$

Additionally, the abstraction function $\langle a \rangle \_$ is an order-embedding in the sense that

$$(\forall d, d' \in D)(\forall a \in \mathbb{A})\; d \sqsubseteq d' \iff \langle a \rangle d \sqsubseteq \langle a \rangle d' \tag{3.5}$$

holds and $[\mathbb{A}]D$ is pointed (Definition 2.2.4) if D is.

Proof. $[\mathbb{A}]D$ is a nominal set by Proposition 2.3.19. Reflexivity, transitivity and antisymmetry of $\sqsubseteq$ follow directly from its definition, as does the order-embedding property of the abstraction function. If D has a least element $\bot$, then it is necessarily equivariant by Lemma 3.1.5 and with that it easy to check that $\langle a \rangle \bot$ (for some/any $a \in \mathbb{A}$) is the least element of $[\mathbb{A}]D$. □

Lemma 3.1.7 (**joins in abstraction posets**). *If a nominal poset D has joins of all its finitely supported subsets, then so does $[\mathbb{A}]D$. The same holds for bounded subsets, directed subsets and uniformly supported subsets.*

Proof. Given in Appendix A.1. □

3.2 Uniform-directedness and uniform-continuity

In classical domain theory (see Section 2.2), a *continuous function* is by definition monotone and preserves joins of directed sets. Proposition 3.1.6 shows that the name abstraction function $d \mapsto \langle a \rangle d$ is monotone. To use it in a nominal domain theory, name abstraction also needs to be continuous in the appropriate sense for nominal posets. This section is concerned with what this 'appropriate sense' is.

The obvious first candidate for the 'right' notion of continuity is to follow classical domain theory and to define a function to be continuous if it preserves all joins of subsets that are finitely supported and directed. However, this turns out not to interact well with name abstraction, as the next example shows.

Example 3.2.1 (**abstraction does not preserve all joins**). The name abstraction function $\langle a \rangle \_ \in D \to_{\mathrm{fs}} [\mathbb{A}]D$ (for a nominal poset D) does not preserve all joins of finitely supported and directed sets. We present a simplified version of the counter example by Turner and Winskel [63, Section 3.1]. Consider the poset $D = \mathrm{P}_{\mathrm{fs}}\mathbb{A}$ (Example 3.1.3) and its subset of finite sets $\mathrm{P}_{\mathrm{f}}\mathbb{A}$. $\mathrm{P}_{\mathrm{f}}\mathbb{A}$ has empty support, is directed and its join $\bigsqcup \mathrm{P}_{\mathrm{f}}\mathbb{A}$ is equal to $\mathbb{A}$. However, fixing upon $a \neq a'$ in $\mathbb{A}$, one has $\langle a \rangle(\bigsqcup \mathrm{P}_{\mathrm{f}}\mathbb{A}) = \langle a \rangle \mathbb{A} = \langle a' \rangle \mathbb{A} \not\sqsubseteq \langle a' \rangle(\mathbb{A} - \{a\})$ and one can check that $\bigsqcup\{\langle a \rangle F \mid F \in \mathrm{P}_{\mathrm{f}}\mathbb{A}\} \sqsubseteq \langle a' \rangle(\mathbb{A} - \{a\})$ (see Pitts [44, Proposition 11.5]). Therefore the join is not preserved: $\langle a \rangle(\bigsqcup \mathrm{P}_{\mathrm{f}}\mathbb{A}) \neq \bigsqcup\{\langle a \rangle F \mid F \in \mathrm{P}_{\mathrm{f}}\mathbb{A}\}$.

Despite this failure of name abstraction to preserve joins of finitely supported and directed subsets, it in fact preserves joins of directed sets that are uniformly supported and directed.

Proposition 3.2.2 (**abstraction preserves all uniform joins**). *For any nominal poset D and $a \in \mathbb{A}$, all joins of uniformly supported subsets (that happen to exist in D) are preserved by the abstraction function $d \mapsto \langle a \rangle d$.*

Proof. Let S be a uniformly supported subset of D, whose join exists in D, and consider the set $S' \triangleq \{\langle a \rangle d \mid d \in S\}$ (for any given $a \in \mathbb{A}$). It is easy to see that S' is uniformly supported and so (A.2) gives us $\bigsqcup S' = \langle a' \rangle(\bigsqcup\{(a\ a') \cdot d \mid d \in S\})$ for some/any $a' \# a, S$. We also have $\langle a \rangle(\bigsqcup S) = \langle a' \rangle(a\ a') \cdot (\bigsqcup S) = \langle a' \rangle(\bigsqcup\{(a\ a') \cdot d \mid d \in S\})$ by Lemma 2.3.21, (3.2) and (2.11), and therefore we get altogether $\bigsqcup S' = \langle a \rangle(\bigsqcup S)$. □

Proposition 3.2.2 directs us towards building our nominal domain theory on functions that preserve all uniformly supported and directed subsets. Another reason to consider this to be the 'right' notion of continuous function in the nominal setting is that Proposition 2.2.7 can be extended from chains and directed sets to finitely supported chains and uniformly supported, directed sets.

Proposition 3.2.3 (**joins of chains = joins of uniform-directed sets**). *A nominal poset has joins of all uniformly supported and directed sets if and only if it has joins of all finitely supported chains. The same correspondence holds for countable chains and directed sets, in the sense that a nominal poset has joins of all uniformly supported, directed and countable sets if and only if it has joins of all finitely supported and countable chains.*

Proof. Turner [62, Lemma 3.4.2.1] and Pitts [44, Proposition 11.9] show that every finitely supported chain is necessarily uniformly supported, which gives the 'only if'-direction. The 'if'-direction with the countability argument can be extracted from the classical proof, which is given for example in Markowsky [29, Corollary 1]. □

So by focusing on directed sets that are uniformly supported we develop in effect a domain theory within the higher-order logic of nominal sets based on chain-completeness.

Definition 3.2.4 (**uniform-directed subset**)**.** We say a subset of a nominal poset is *uniform-directed* if it is uniformly supported (Definition 2.3.40) and directed (Definition 2.2.5). A *uniform-directed complete partial order (udcpo)* is a nominal poset where each uniform-directed subset has a join. The join of a uniform-directed subset is called uniform-directed join.

A function f between two udcpo's D, D' is called *uniform-continuous* if it is monotone, finitely supported and preserves all joins of uniform-directed subsets, in the sense that $f(\bigsqcup S) = \bigsqcup\{f\ d \mid d \in S\}$ holds for all uniform-directed subsets $S \subseteq D$. Note that if f is finitely supported and monotone, then $\{f\ d \mid d \in S\}$ is uniform-directed because S is. We write $D \to_{\mathrm{uc}} D'$ for the set of uniform-continuous functions between the udcpo's D and D'.

Example 3.2.5 (**powerset udcpo**)**.** The nominal poset $\mathrm{P}_{\mathrm{fs}}X$ from Example 3.1.3 possesses joins of all finitely supported subsets, given by union, and hence in particular it is a udcpo.

We continue with some important technical properties of uniform-continuous functions.

Lemma 3.2.6 (**permutation preserves uniform-continuity**)**.** *The permutation action on functions* (2.6) *preserves uniform-continuity. Therefore $D \to_{\mathrm{uc}} D'$ is a nominal set for any udcpo's D, D'.*

Proof. Let $f \in D \to_{\mathrm{uc}} D'$, $\pi \in \mathrm{Perm}(\mathbb{A})$ and a uniform-directed subset $S \subseteq D$ be given, it then follows by (3.2) and uniform-continuity of f that $(\pi \cdot f)(\bigsqcup S) = \pi \cdot (f\ (\pi^{-1} \cdot \bigsqcup S)) = \pi \cdot (f\ (\bigsqcup\{\pi^{-1} \cdot d \mid d \in S\})) = \pi \cdot \bigsqcup\{f\ (\pi^{-1} \cdot d) \mid d \in S\} = \bigsqcup\{\pi \cdot (f\ (\pi^{-1} \cdot d)) \mid d \in S\} = \bigsqcup\{(\pi \cdot f)\, d \mid d \in S\}$. □

Lemma 3.2.7 (**constant functions are uniform-continuous**)**.** *For any udcpo's D, D' and element $d \in D$, the constant function*

$$const_d \triangleq \lambda d' \in D' \to d$$

is a uniform-continuous function from D' to D with $\operatorname{supp} const_d = \operatorname{supp} d$.

Proof. Via straightforward calculations. □

Lemma 3.2.8 (**composition preserves uniform-continuity**)**.** *The composition of two functions is uniform-continuous if the two functions are.*

Proof. For any given uniform-continuous f, g and uniform-directed S, it is straightforward to see that $g \circ f$ is monotone, and by Lemma 2.3.7 it is finitely supported. We also easily check that $g(f(\bigsqcup S)) = g(\bigsqcup\{f\ d \mid d \in S\}) = \bigsqcup\{g(f\ d) \mid d \in S\}$. □

3.3 Uniform-compactness and algebraicity

We model potentially infinite program behaviours in languages with names using denotations that are uniform-directed joins of approximations to the behaviour. Each approximation should be finite in a suitable sense. For classical domain theory this amounts to being compact (Definition 2.2.8, also known as 'finite' or 'isolated') with respect to directed joins. By analogy, we have:

Definition 3.3.1 (**uniform-compact element**)**.** An element $u \in D$ of a udcpo D is *uniform-compact* if for all uniform-directed subsets $S \subseteq D$ it is the case that

$$u \sqsubseteq \bigsqcup S \Rightarrow (\exists s \in S)\, u \sqsubseteq s$$

holds. This definition is analogous to the classical one from Definition 2.2.8 and (2.3). We write $\mathrm{K}D$ for the set of uniform-compact elements of D. We say that D is an *algebraic* udcpo if each of its elements is the join of a uniform-directed subset of $\mathrm{K}D$. D is *ω-algebraic* if in addition $\mathrm{K}D$ is countable.

We continue with some technical properties of uniform-compact elements.

Lemma 3.3.2 (**element characterisation**)**.** *For any element d of an algebraic udcpo D it holds that*

$$d = \bigsqcup \{u \in \mathrm{K}D \mid u \sqsubseteq d\}\,.$$

Proof. First note that $\{u \in \mathrm{K}D \mid u \sqsubseteq d\}$ is not necessarily uniformly supported, so in principle its join might not exist. Nevertheless, we will prove directly that d is the join of this set. The argument that d is an upper bound of $\{u \in \mathrm{K}D \mid u \sqsubseteq d\}$ is immediate. For showing that d it is the least upper bound, let another upper bound e be given. Let also S_d be the uniform-directed subset of $\mathrm{K}D$ satisfying $d = \bigsqcup S_d$, which exists by definition. For any $u' \in S_d$ we have $u' \sqsubseteq \bigsqcup S_d = d$ and hence $u' \in \{u \in \mathrm{K}D \mid u \sqsubseteq d\}$. This shows $S_d \subseteq \{u \in \mathrm{K}D \mid u \sqsubseteq d\}$, so e must be also an upper bound for S_d and with this we get $d = \bigsqcup S_d \sqsubseteq e$. □

Lemma 3.3.3 (**permutation preserves uniform-compactness**)**.** *The permutation of a uniform-compact element is uniform-compact. Hence* $\mathrm{K}D$ *is a nominal set for any udcpo D.*

Proof. Let D, $u \in \mathrm{K}D$ and $\pi \in \mathrm{Perm}(\mathbb{A})$ be given. To show $\pi \cdot u \in \mathrm{K}D$ suppose that $\pi \cdot u \subseteq \bigsqcup S$ for some uniform-directed S. It follows that $u \subseteq \bigsqcup(\pi^{-1} \cdot S)$, which is a uniform-directed join, so there is a $d \in \pi^{-1} \cdot S$ such that $u \sqsubseteq d$. Hence we know $\pi \cdot d \in S$ and $\pi \cdot u \sqsubseteq \pi \cdot d$, showing that $\pi \cdot u$ is uniform-compact. Overall we get that $\mathrm{K}D$ is an equivariant subset of D and therefore it is a nominal set through Pitts [44, Lemma 2.22]. □

Lemma 3.3.4 (**orbit-finite joins of uniform-compact elements**)**.** *Let D be a udcpo. If an orbit-finite subset of uniform-compact elements possesses a join in D, then that join is also uniform-compact.*

Proof. Suppose $U \subseteq_{\mathrm{of}} \mathrm{K}D$ has a join $\bigsqcup U$ in D and that $\bigsqcup U \sqsubseteq \bigsqcup S$ for some uniform-directed subset $S \subseteq D$. $\mathrm{K}D$ is a nominal set by Lemma 3.3.3, so we can apply Theorem 2.3.38 to obtain $U = \mathrm{hull}_A\{u_1, \ldots, u_n\}$ for some $u_1, \ldots, u_n \in \mathrm{K}D$ and $A \subseteq_{\mathrm{f}} \mathbb{A}$. By Lemma 2.3.37 we may assume without loss of generality that $\mathrm{supp}\, S \subseteq A$, and by Lemma 2.3.41 this gives $(\forall d \in S)\ \mathrm{supp}\, d \subseteq A$. For any $i \in \{1, \ldots, n\}$ it holds that $u_i \in \mathrm{hull}_A\{u_1, \ldots, u_n\} = U$, hence we get $u_i \sqsubseteq \bigsqcup U \sqsubseteq \bigsqcup S$, and by uniform-compactness of u_i that $u_i \sqsubseteq d_i$ for some $d_i \in S$. As S is directed $d_1, \ldots, d_n$ have an upper bound $d \in S$ for which then $(\forall i \in \{1, \ldots, n\})\ u_i \sqsubseteq d$ holds. Because $\mathrm{supp}\, d \subseteq A$, if follows that $(\forall u \in \mathrm{hull}_A\{u_1, \ldots, u_n\})\ u \sqsubseteq d$ and therefore $\bigsqcup U = \bigsqcup \mathrm{hull}_A\{u_1, \ldots, u_n\} \sqsubseteq d$. So we indeed have $\bigsqcup U \in \mathrm{K}D$. □

The next theorem gives a concrete characterisation of the uniform-compact elements in abstraction posets.

Theorem 3.3.5 (**uniform-compact abstractions**). *If D is an algebraic udcpo, then so is $[\mathbb{A}]D$ and its uniform-compact elements can be characterised by*

$$\mathrm{K}([\mathbb{A}]D) = \{\langle a \rangle u \mid a \in \mathbb{A} \wedge u \in \mathrm{K}D\}. \tag{3.6}$$

Proof. Suppose D is an algebraic udcpo, by Proposition 3.1.6 and Lemma 3.1.7 we know that $[\mathbb{A}]D$ is a udcpo. To show that it is algebraic note that by its definition (Definition 2.3.18) every element of $[\mathbb{A}]D$ is the form $\langle a \rangle d$ where $a \in \mathbb{A}$ and $d \in D$, and that by algebraicity of D we know $d = \bigsqcup S$ for a uniform-directed subset of uniform-compact elements $S \subseteq \mathrm{K}D$. Proposition 3.2.2 shows that the abstraction function is uniform-continuous $(\forall a \in \mathbb{A})\ \langle a \rangle\_ \in D \to_{\mathrm{uc}} [\mathbb{A}]D$ and so for showing algebraicity of $[\mathbb{A}]D$ it is enough to show $u \in \mathrm{K}D \Rightarrow \langle a \rangle u \in \mathrm{K}([\mathbb{A}]D)$. The proof for this is given in Pitts [44, Proposition 11.26], where he uses (A.2) and Lemma 3.3.3. This also proves the right-to-left inclusion of (3.6). The left-to-right inclusion follows from Lemma 2.3.25 and $e \in \mathrm{K}([\mathbb{A}]D) \wedge a \# e \Rightarrow e @ a \in \mathrm{K}D$, which is also proved in Pitts [44, Proposition 11.26]. □

Recall from Proposition 2.2.9 that a subset of a set is compact with respect to directed joins (unions) of subsets if and only if it is a finite set. Here we are restricting attention to a smaller class of joins, the uniform-directed ones. Therefore, one should expect uniform-compactness to be a more liberal notion of finiteness. Indeed, we show next that it corresponds precisely to the notion of orbit-finite subset from Definition 2.3.34.

Theorem 3.3.6 (**uniform-compact = orbit-finite**). *An element of the udcpo $\mathrm{P}_{\mathrm{fs}}X$ (see Example 3.2.5) is uniform-compact if and only if it is an orbit-finite subset of X. Furthermore $\mathrm{P}_{\mathrm{fs}}X$ is algebraic in the sense of Definition 3.3.1.*

Proof. We start with algebraicity. For every $E \in \mathrm{P}_{\mathrm{fs}}X$, since any $F \subseteq_{\mathrm{f}} E$ satisfies $\mathrm{hull}_{\mathrm{supp}\, E} F \subseteq E$, we have

$$E = \bigcup \{\mathrm{hull}_{\mathrm{supp}\, E} F \mid F \subseteq_{\mathrm{f}} E\} \tag{3.7}$$

and the right-hand side is a directed union of orbit-finite subsets (by Theorem 2.3.38) that have uniform support supp E by (2.27). Therefore to prove the theorem it suffices to prove that any $E \subseteq_{\mathrm{fs}} X$ is uniform-compact if and only if it is orbit-finite.

For the 'if'-direction, suppose $E \in \mathrm{P}_{\mathrm{of}}X$ and $E \subseteq \bigcup S$ with S a uniform-directed subset of $\mathrm{P}_{\mathrm{fs}}X$. By Theorem 2.3.38, $E = \mathrm{hull}_A F$ for some $A \subseteq_{\mathrm{f}} \mathbb{A}$ and $F \subseteq_{\mathrm{f}} X$; and by Lemma 2.3.37 we may assume that A supports each element of S. F is finite, $F \subseteq \mathrm{hull}_A F \subseteq \bigcup S$ and S is directed, so we can apply Proposition 2.2.9 to obtain $F \subseteq E'$ for some $E' \in S$. Since A supports E', $F \subseteq E'$ implies that $E = \mathrm{hull}_A F \subseteq E'$, showing that E is uniform-compact.

Conversely, if $E \subseteq_{\mathrm{fs}} X$ is uniform-compact, then in view of (3.7) we have $E \subseteq \mathrm{hull}_{\mathrm{supp}\, E} F$ for some $F \subseteq_{\mathrm{f}} E$. Hence $E = \mathrm{hull}_{\mathrm{supp}\, E} F$ is orbit-finite by Theorem 2.3.38. □

3.4 Nominal Scott domains

In view of Theorem 3.3.6 there is the following analogy

$$\frac{\text{finite}}{\text{directed}}\text{sets} \sim \frac{\text{orbit-finite}}{\text{uniform-directed}}\text{nominal sets}$$

which we apply to transfer to nominal sets the classical notion of Scott domain that arose in the denotational semantics of functional programming languages, for example in Plotkin [47, Lemma 4.4]. The resulting notion of nominal Scott domain is central to this thesis, because the types of our programming language PNA denote nominal Scott domains (see Section 4.3.1), in the same way as classically PCF-types denote Scott domains.

Definition 3.4.1 (**nominal Scott domain**)**.** In analogy with Definition 2.2.10, a *nominal Scott domain* D is a pointed, ω-algebraic udcpo, which also has joins for all bounded, orbit-finite sets of uniform-compact elements. The category **Nsd** has nominal Scott domains for its objects and for its morphisms it has functions that are both equivariant and uniform-continuous, where the identity and composition are as for normal functions. Lemmas 2.3.7 and 3.2.8 ensure that this gives a well-defined category.

It turns out that, analogous to Lemma 2.2.11, a nominal Scott domain has in fact joins of all bounded (and finitely supported) subsets.

Definition 3.4.2 (**bounded-completeness**)**.** Classically, a poset is *bounded-complete* if it has joins for all bounded subsets. In the nominal setting we are only interested in constructs with finite support, so a nominal poset is called *bounded-complete* if it has joins for all its bounded and finitely supported subsets.

Lemma 3.4.3 (**bounded and finitely supported joins**)**.** *Every nominal Scott domain is bounded-complete.*

Proof. Given in Appendix A.2. □

The next lemma shows that we can improve the characterisation in Lemma 3.3.2 for nominal Scott domains. It uses the partial order $\sqsubseteq_{\mathrm{supp}}$, defined for any nominal poset D and $d, d' \in D$ by

$$d \sqsubseteq_{\mathrm{supp}} d' \triangleq d \sqsubseteq d' \wedge \operatorname{supp} d \subseteq \operatorname{supp} d' .$$

Lemma 3.4.4 (improved element characterisation). *For all $D \in \mathbf{Nsd}$, $u \in \mathrm{K}D$ and $d \in D$ we have*

$$u \sqsubseteq d \Rightarrow (\exists u' \in \mathrm{K}D)\, u \sqsubseteq u' \sqsubseteq_{\mathrm{supp}} d \tag{3.8}$$

and hence

$$d = \bigsqcup \{u \in \mathrm{K}D \mid u \sqsubseteq_{\mathrm{supp}} d\}. \tag{3.9}$$

Proof. If $u \sqsubseteq d$ with $u \in \mathrm{K}D$, then $\mathrm{hull}_{\mathrm{supp}\, d}\{u\}$ is an orbit-finite subset of uniform-compact elements that is supported by $\operatorname{supp} d$, is bounded above by d and contains u. Therefore the join $u' \triangleq \bigsqcup \mathrm{hull}_{\mathrm{supp}\, d}\{u\}$ exists in D, satisfies $u' \in \mathrm{K}D$ (by Lemma 3.3.4) and $u \sqsubseteq u' \sqsubseteq_{\mathrm{supp}} d$ holds. Property (3.9) then follows from (3.8) and Lemma 3.3.2. □

In the following, we give important constructions on nominal Scott domains that will be used in the denotational semantics of the programming language PNA in Section 4.3.

3.4.1 Flat domains

Every countable nominal set can be turned into a nominal Scott domain by the flat domain construction. All ground types of PNA will be denoted by flat domains.

Definition 3.4.5 (flat domain). For nominal set X, the set $X_\perp \triangleq X \cup \{\perp\}$ (where $\perp$ is equivariant and not an element of X) with the order

$$x_1 \sqsubseteq x_2 \Leftrightarrow x_1 = \perp \vee x_1 = x_2 \tag{3.10}$$

is called the *flat domain* on X.

Lemma 3.4.6 (flat nominal Scott domains). *Every flat domain on a countable nominal set X is a nominal Scott domain, called the flat nominal Scott domain, where the uniform-compact elements are $\mathrm{K}X_\perp = X_\perp$*

Proof. It is easy to show that (3.10) is a partial order satisfying (3.1) with $\perp$ as least element, showing that $X_\perp$ is a pointed nominal poset. Note that in a poset with this order, any directed or bounded set that is not a singleton must be of the form $\{\perp, x\}$. With this $\mathrm{K}X_\perp = X_\perp$, ω-algebraicity and bounded-completeness are straightforward to check. We need the countability of X because we require the uniform-compact elements to be countable. □

The next lemma is very useful for proving the uniform-continuity of various functions in this thesis.

Lemma 3.4.7 (**functions from flat domains**). *Any finitely supported function $f \in X_\perp \to_{\mathrm{fs}} D$ from a flat nominal Scott domain $X_\perp$ to an arbitrary nominal Scott domain D is uniform-continuous if and only if it is monotone.*

Proof. By a direct calculation using the simplicity of uniform-directed subsets in flat domains noted in the proof of Lemma 3.4.6. □

3.4.2 Products

The notion of product for nominal Scott domains is the straightforward combination of the products in Scott domains and nominal sets.

Definition 3.4.8 (**product order**). We define the product of two nominal posets D_1, D_2 as the cartesian product of the underlying sets (2.1) with the component-wise permutation action (2.7) and the component-wise partial order

$$(d_1, d_2) \sqsubseteq (d_1', d_2') \triangleq d_1 \sqsubseteq d_1' \wedge d_2 \sqsubseteq d_2',$$

which turns $D_1 \times D_2$ into nominal poset.

We establish some results for products of nominal Scott domains, following the classical results on products of Scott domains.

Proposition 3.4.9 (**product joins**). *If a subset $S \subseteq D_1 \times D_2$ of the product of two nominal posets D_1, D_2 is finitely supported, uniformly-supported, directed or bounded, then so are $S_1 \triangleq \{d_1 \in D_1 \mid (\exists d_2 \in D_2)\ (d_1, d_2) \in S\}$ and $S_2 \triangleq \{d_2 \in D_2 \mid (\exists d_1 \in D_1)\ (d_1, d_2) \in S\}$. Furthermore the join of S, if it exists, can be calculated component-wise $\bigsqcup S = (\bigsqcup S_1, \bigsqcup S_2)$.*

Proof. By straightforward calculations, very similar to the classical proof for example in Abramsky and Jung [4, Proposition 3.2.2]. □

Lemma 3.4.10 (**component-wise uniform-continuity**). *For udcpos D_1, D_2, D_3 we have that a function from a product udcpo $f \in D_1 \times D_2 \to D_3$ (where $D_1 \times D_2$ is a udcpo because of Proposition 3.4.9) is uniform-continuous if and only if it is uniform-continuous in each component, meaning that it is finitely supported, monotone in each component*

$$(\forall d_1 \in D_1)(\forall d_2, d_2' \in D_2)\, d_2 \sqsubseteq d_2' \Rightarrow f(d_1, d_2) \sqsubseteq f(d_1, d_2')$$
$$(\forall d_2 \in D_2)(\forall d_1, d_1' \in D_1)\, d_1 \sqsubseteq d_1' \Rightarrow f(d_1, d_2) \sqsubseteq f(d_1', d_2)$$

and for all uniform-directed subsets $S_1 \subseteq D_1$ and $S_2 \subseteq D_2$ it satisfies

$$(\forall d_2 \in D_2)\, f(\bigsqcup S_1, d_2) = \bigsqcup \{f(d_1, d_2) \mid d_1 \in S_1\}$$
$$(\forall d_1 \in D_1)\, f(d_1, \bigsqcup S_2) = \bigsqcup \{f(d_1, d_2) \mid d_2 \in S_2\}.$$

Proof. By following the structure of the classical proof in Abramsky and Jung [4, Lemma 3.2.6]. □

Proposition 3.4.11 (**categorical product**). *The category* **Nsd** *has a terminal object and binary products. In particular, the product of two nominal Scott domains is a nominal Scott domain with* $\mathrm{K}(D_1 \times D_2) = \mathrm{K}D_1 \times \mathrm{K}D_2$.

Proof. The product of D_1 and D_2 is given in Definition 3.4.8. That it has joins for all uniform-directed subsets, bounded and finitely supported subsets, and has $(\bot, \bot)$ as its least element are all direct consequences of Proposition 3.4.9. ω-algebraicity follows from $\mathrm{K}(D_1 \times D_2) = \mathrm{K}D_1 \times \mathrm{K}D_2$ and this can be proved directly as for classical Scott domains (see Abramsky and Jung [4, Proposition 3.2.4(4)]).

With Proposition 3.4.9 it is straightforward to show that the projections $proj_1$, $proj_2$ (given by (2.9) and (2.10)) and the mediating morphism $\langle f, g \rangle$ (given by (2.8)) are uniform-continuous, for uniform-continuous functions f and g. Altogether we get that $D_1 \times D_2$ is the binary product of **Nsd**. The terminal object of **Nsd** is given by the trivial flat domain $\emptyset_\bot$ and for every nominal Scott domain X the unique morphism $X \to \emptyset_\bot$ is given by $\lambda x \in X \to \bot$. □

Remark 3.4.12 (**finite products**). **Nsd** has finite products by Proposition 3.4.11 and Proposition 2.1.4. Finite tuples $(d_1, \ldots, d_n)$, function tupling $\langle f_1, \ldots, f_n \rangle$ and projections $proj_1, \ldots, proj_n$ are the obvious extensions of the constructs from Proposition 3.4.11.

The next three lemmas help to show that the functions in our denotational semantics for PNA (see Section 4.3.2) are uniform-continuous.

Lemma 3.4.13 (**flat functions**). *Let* $f \in X_1 \times \ldots \times X_n \rightharpoonup Y$ *be a partial function from a finite product of countable nominal sets* $X_1, \ldots, X_n$ *to a countable nominal set* Y. *Then the function between the corresponding flat domains* $f_\bot \in X_{1\bot} \times \ldots \times X_{n\bot} \to Y_\bot$ *defined by*

$$f_\bot(d_1, \ldots, d_n) \triangleq \begin{cases} f(d_1, \ldots, d_n) & \text{if } d_1 \in X_1 \ldots d_n \in X_n \text{ and } f \text{ is defined at } (d_1, \ldots, d_n) \\ \bot & \text{otherwise} \end{cases}$$

is uniform-continuous.

Proof. We show by case distinction that $f_\bot$ is monotone in each argument and the rest follows with Lemmas 3.4.10 and 3.4.7. □

Lemma 3.4.14 (**if-function**). *The function* $\mathrm{if}_D \in \mathbb{B}_\bot \times D \times D \to D$ *defined by*

$$\mathrm{if}_D(x, d, d') \triangleq \begin{cases} d & \text{if } x = \mathrm{true} \\ d' & \text{if } x = \mathrm{false} \\ \bot & \text{otherwise} \end{cases}$$

is equivariant and uniform-continuous for any nominal Scott domain D.

Proof. Both properties follow from simple calculations, where equivariance uses that $\mathbb{B}$ is a discrete nominal set and uniform-continuity uses Lemmas 3.4.7 and 3.4.10. □

Lemma 3.4.15 (swap-function). *For any $D \in \mathbf{Nsd}$ the function $swap_D \in \mathbb{A}_\perp \times \mathbb{A}_\perp \times D \to D$ defined by*

$$swap_D(x_1, x_2, d) \triangleq \begin{cases} (x_1\ x_2) \cdot d & \text{if } x_1, x_2 \in \mathbb{A} \\ \perp & \text{otherwise} \end{cases}$$

is equivariant and uniform-continuous.

Proof. Equivariance follows from Lemma 2.3.4. *swap* is monotone in the first two components, so by Lemma 3.4.7 it is uniform-continuous in them. Uniform-continuity in the third component is a consequence of Proposition 3.1.4, and hence *swap* is uniform-continuous by Lemma 3.4.10. □

3.4.3 Functions

Uniform-continuous functions do not just connect nominal Scott domains; with the point-wise order they also form nominal Scott domains of functions. For the reader familiar with classical domain theory most results in this section should not be surprising, except that many uses of finite subsets are replaced by uses of orbit-finite subsets.

Definition 3.4.16 (function order). For a set D_1 and a poset D_2, define an order relation between functions $f, f' \in D_1 \to D_2$ by

$$f \sqsubseteq f' \triangleq (\forall d \in D_1)\, f\, d \sqsubseteq f'\, d\,. \tag{3.11}$$

We can easily show that this gives a partial order on $D_1 \to D_2$. It is called the *pointwise order*.

Lemma 3.4.17 (monotonicity of composition). *Function composition is a monotone operation for monotone arguments. Given a set D_1, posets D_2, D_3 and functions $f_1, f_2 \in D_1 \to D_2$, $g_1, g_2 \in D_2 \to D_3$, where g_2 is a monotone function (as in Definition 2.2.2), then by using* (3.11) *we have that*

$$f_1 \sqsubseteq f_2 \wedge g_1 \sqsubseteq g_2 \Rightarrow (g_1 \circ f_1) \sqsubseteq (g_2 \circ f_2)\,.$$

Proof. For any argument $d \in D_1$ we know by $f_1 \sqsubseteq f_2$ that $f_1\, d \sqsubseteq f_2\, d$ and by monotonicity of g_2 this implies $g_2(f_1\, d) \sqsubseteq g_2(f_2\, d)$. By $g_1 \sqsubseteq g_2$ we also know $g_1(f_1\, d) \sqsubseteq g_2(f_1\, d)$ and with transitivity of $\sqsubseteq$ we altogether get $g_1(f_1\, d) \sqsubseteq g_2(f_2\, d)$. □

Proposition 3.4.18 (joins of functions). *If D_1 and D_2 are udcpo's, pointed udcpo's or bounded-complete nominal posets, then so is the set of uniform-continuous functions between them $D_1 \to_{\mathrm{uc}} D_2$, where we use the permutation action*

$$(\forall f \in (D_1 \to_{\mathrm{uc}} D_2))(\forall \pi \in \mathrm{Perm}(\mathbb{A}))\ \pi \cdot f \triangleq \lambda d \in D_1 \to \pi \cdot (f(\pi^{-1} \cdot d))$$

from (2.6) *and the pointwise order*

$$f \sqsubseteq f' \triangleq (\forall d \in D_1)\, f\, d \sqsubseteq f'\, d$$

from (3.11) *above.*

Proof. Let udcpo's D_1 and D_2 be given. By Lemma 3.2.6 we have that $D_1 \to_{\mathrm{uc}} D_2$ is a nominal set. It is easy to see that (3.11) gives a partial order satisfying (3.1), and that $\lambda d \in D_1 \to \bot$ is the least element of $D_1 \to_{\mathrm{uc}} D_2$ if $\bot$ is the least element of D_2. If $S \subseteq (D_1 \to_{\mathrm{uc}} D_2)$ is uniform-directed then by standard calculations we can show for any $d \in D_1$ that $\{f\ d \mid f \in S\}$ is directed and uniformly supported by $\operatorname{supp} S \cup \operatorname{supp} d$, and that the join of S can be described by

$$\bigsqcup S = \lambda d \in D_1 \to \bigsqcup \{f\ d \mid f \in S\}, \tag{3.12}$$

showing that $D_1 \to_{\mathrm{uc}} D_2$ is a udcpo, as this join always exists. Similarly, if $S \subseteq (D_1 \to_{\mathrm{uc}} D_2)$ is bounded and finitely supported, then so is $\{f\ d \mid f \in S\}$ for any $d \in D_1$. As the join of S is still given by (3.12) we have that $D_1 \to_{\mathrm{uc}} D_2$ is bounded-complete if D_2 is. □

Lemma 3.4.19 (**uniform-compact arguments**). *Two uniform-continuous functions between nominal Scott domains $f, g \in (D \to_{\mathrm{uc}} D')$ are related by the partial order if and only if they are so on all uniform-compact arguments:*

$$f \sqsubseteq g \iff (\forall u \in \mathrm{K}D)\ f\ u \sqsubseteq g\ u\,.$$

Proof. The left-to-right direction is immediate. For the right-to-left direction let any $d \in D$ be given. By Lemma 3.4.4 we have $d = \bigsqcup \{u \mid u \in \mathrm{K}D \wedge u \sqsubseteq_{\mathrm{supp}} d\}$ and hence $f\ d = f(\bigsqcup \{u \mid u \in \mathrm{K}D \wedge u \sqsubseteq_{\mathrm{supp}} d\}) = \bigsqcup \{f\ u \mid u \in \mathrm{K}D \wedge u \sqsubseteq_{\mathrm{supp}} d\} \sqsubseteq \bigsqcup \{g\ u \mid u \in \mathrm{K}D \wedge u \sqsubseteq_{\mathrm{supp}} d\} = g(\bigsqcup \{u \mid u \in \mathrm{K}D \wedge u \sqsubseteq_{\mathrm{supp}} d\}) = g\ d$. As d is arbitrary $f \sqsubseteq g$ follows. □

The following lemma is less standard, because it uses retracts to characterise the function order. It will be used in the proof of Lemma 5.5.13 in Appendix A.6.

Lemma 3.4.20 (**retracted arguments**). *Let D_1, D_2, D' be nominal Scott domains and let D_1 be a retract of D', in the sense that there are uniform-continuous functions $f_i \in D_1 \to_{\mathrm{uc}} D'$ and $f_r \in D' \to_{\mathrm{uc}} D_1$ satisfying $f_r \circ f_i = \mathrm{id}_{D_1}$. Then any two uniform-continuous functions $f, g \in D_1 \to_{\mathrm{uc}} D_2$ satisfy*

$$f \sqsubseteq g \iff (\forall u' \in \mathrm{K}D')\ f(f_r\ u') \sqsubseteq g(f_r\ u').$$

Proof. The left-to-right direction follows directly from (3.11). For the right-to-left direction, we know by Lemma 3.2.8 that $f \circ f_r$ and $g \circ f_r$ are uniform-continuous. Therefore we can apply Lemma 3.4.19 to get $f \circ f_r \sqsubseteq g \circ f_r$. By Lemma 3.4.17 we obtain $f \circ f_r \circ f_i \sqsubseteq g \circ f_r \circ f_i$, which means with $f_r \circ f_i = \mathrm{id}_{D_1}$ that $f \sqsubseteq g$. □

To get a nominal Scott domain of functions, we need to consider algebraicity and uniform-compact elements. It turns out that uniform-compact functions are given by the joins of orbit-finite sets of step functions (Proposition 3.4.27).

Definition 3.4.21 (step functions). Given $D_1, D_2 \in \mathbf{Nsd}$ and uniform-compact elements $u_1 \in \mathrm{K}D_1$ and $u_2 \in \mathrm{K}D_2$, a *step function* is a function $(u_1 \searrow u_2) \in D_1 \to D_2$ of the following form

$$(u_1 \searrow u_2) \triangleq \lambda d \in D_1 . \begin{cases} u_2 & \text{if } u_1 \sqsubseteq d \\ \bot & \text{otherwise.} \end{cases} \tag{3.13}$$

The set of all such step functions for a particular domain D_1 and codomain D_2 is given by

$$D_1 \to_{\text{step}} D_2 \triangleq \{(u_1 \searrow u_2) \mid u_1 \in \mathrm{K}D_1 \wedge u_2 \in \mathrm{K}D_2\}.$$

A finitely supported set of step functions $S \subseteq_{\text{fs}} (D_1 \to_{\text{step}} D_2)$ is called *consistent* if for every finitely supported subset $S' \subseteq_{\text{fs}} S$ it holds that

$$\{u_1 \mid (u_1 \searrow u_2) \in S'\} \text{ is bounded in } D_1 \Rightarrow \{u_2 \mid (u_1 \searrow u_2) \in S'\} \text{ is bounded in } D_2. \tag{3.14}$$

This notion of consistency is already present in the classical theory of Scott domains [47, Lemma 4.4] [61, Page 100]. Consistency will be crucial for showing the existence of certain joins of step functions, see Lemma 3.4.24.

What follows are some technical properties of step functions, leading to the characterisation of uniform-compact elements of function domains in terms of step functions in Proposition 3.4.27.

Lemma 3.4.22 (step function permutation and order). *For any $D_1, D_2 \in \mathbf{Nsd}$, $u_1 \in \mathrm{K}D_1$ and $u_2 \in \mathrm{K}D_2$ the step function $(u_1 \searrow u_2)$ satisfies for any $\pi \in \mathrm{Perm}(\mathbb{A})$ that*

$$\pi \cdot (u_1 \searrow u_2) = (\pi \cdot u_1 \searrow \pi \cdot u_2) \tag{3.15}$$

and for any $f \in D_1 \to_{\text{uc}} D_2$ that

$$(u_1 \searrow u_2) \sqsubseteq f \iff u_2 \sqsubseteq f\, u_1. \tag{3.16}$$

Proof. (3.15) follows directly from (2.6) and (3.1). (3.16) is also easy to prove using (3.11) and monotonicity of f. □

Lemma 3.4.23 (step functions are uniform-compact). *For each $u_1 \in \mathrm{K}D_1$ and $u_2 \in \mathrm{K}D_2$ we have $(u_1 \searrow u_2) \in \mathrm{K}(D_1 \to_{\text{uc}} D_2)$.*

Proof. Let $u_1 \in \mathrm{K}D_1$ and $u_2 \in \mathrm{K}D_2$ be given, by (3.15) we know that $(u_1 \searrow u_2)$ is finitely supported by $\operatorname{supp} u_1 \cup \operatorname{supp} u_2$. It is easy to see that $(u_1 \searrow u_2)$ is monotone, and it preserves uniform-directed joins because u_1 is uniform-compact. Therefore $(u_1 \searrow u_2)$ is an element of $D_1 \to_{\text{uc}} D_2$. For showing that it is a uniform-compact one, let a uniform-directed $S \subseteq (D_1 \to_{\text{uc}} D_2)$ be given. With (3.16), (3.12) and uniform-compactness of u_2 we get $(u_1 \searrow u_2) \sqsubseteq \bigsqcup S \Rightarrow u_2 \sqsubseteq (\bigsqcup S)\, u_1 = \bigsqcup\{f\, u_1 \mid f \in S\} \Rightarrow u_2 \sqsubseteq f\, u_1$ for some $f \in S \Rightarrow (u_1 \searrow u_2) \sqsubseteq f$ for some $f \in S$. □

Lemma 3.4.24 (joins of step functions). *A finitely supported set of step functions $S \subseteq_{\text{fs}} (D_1 \to_{\text{step}} D_2)$ between nominal Scott domains D_1, D_2 has a join if and only if S is consistent in the sense of* (3.14)*. The join is then given by*

$$\bigsqcup S = \lambda d \in D_1 . \bigsqcup\{u_2 \mid u_1 \sqsubseteq d \wedge (u_1 \searrow u_2) \in S\}. \tag{3.17}$$

Proof. For the 'only if'-direction, assume $\bigsqcup S$ exists and let any $S' \subseteq_{\text{fs}} S$ be given such that $\{u_1 \mid (u_1 \searrow u_2) \in S'\}$ is bounded in D_1, say by d. It follows that each element of $\{u_2 \mid (u_1 \searrow u_2) \in S'\})$ satisfies $u_2 = (u_1 \searrow u_2)\, d \sqsubseteq (\bigsqcup S)\, d$, so $(\bigsqcup S)\, d$ is an upper bound for $\{u_2 \mid (u_1 \searrow u_2) \in S'\})$. For the 'if'-direction, let S be consistent and define for any $d \in D_1$ the finitely supported subset $S_d \triangleq \{(u_1 \searrow u_2) \mid u_1 \sqsubseteq d \wedge (u_1 \searrow u_2) \in S\}$. Observe that the set $\{u_2 \mid (u_1 \searrow u_2) \in S_d\}$ has an upper bound by consistency applied to $S_d \subseteq_{\text{fs}} S$, and it has a join by Lemma 3.4.3. Hence the function $\lambda d \in D_1 \to \bigsqcup\{u_2 \mid (u_1 \searrow u_2) \in S_d\}$ is well-defined, and it is easy to show that this is the join of S. (3.17) follows from $\{u_2 \mid (u_1 \searrow u_2) \in S_d\} = \{u_2 \mid u_1 \sqsubseteq d \wedge (u_1 \searrow u_2) \in S\}$. □

Notation 3.4.25 (**upper bounds**). For a nominal Scott domain D and $d, d' \in D$ we introduce notation for having an upper bound $d \uparrow d' \triangleq (\exists d'' \in D)\, d \sqsubseteq d'' \wedge d' \sqsubseteq d''$ and not having one $d \not\uparrow d' \triangleq \neg(d \uparrow d')$.

Lemma 3.4.26 (**upper bound of step functions**). *For any $D_1, D_2 \in$ **Nsd** and consistent sets of step functions $S, S' \subseteq_{\text{fs}} (D_1 \to_{\text{step}} D_2)$ it holds that $\bigsqcup S \not\uparrow \bigsqcup S'$ implies that there are $(u_1 \searrow u_2) \in S$ and $(u_1' \searrow u_2') \in S'$ such that $u_1 \uparrow u_1'$ and $u_2 \not\uparrow u_2'$.*

Proof. The proof works by contraposition, so we show that $((\forall (u_1 \searrow u_2) \in S)((u_1' \searrow u_2') \in S')\ u_1 \uparrow u_1' \Rightarrow u_2 \uparrow u_2') \Rightarrow \bigsqcup S \uparrow \bigsqcup S'$. If $(\forall (u_1 \searrow u_2) \in S)((u_1' \searrow u_2') \in S')\ u_1 \uparrow u_1' \Rightarrow u_2 \uparrow u_2'$ holds, then $S \cup S'$ is consistent and therefore by Lemma 3.4.24 $\bigsqcup(S \cup S')$ exists, which is an upper bound for $\bigsqcup S$ and $\bigsqcup S'$. □

Proposition 3.4.27 (**uniform-compact functions**). *If D_1 and D_2 are nominal Scott domains, then $D_1 \to_{\text{uc}} D_2$ is a nominal Scott domain with*

$$\mathrm{K}(D_1 \to_{\text{uc}} D_2) = \{\textstyle\bigsqcup S \mid S \subseteq_{\text{of}} (D_1 \to_{\text{step}} D_2) \wedge S \text{ is consistent}\}. \tag{3.18}$$

Proof. Given in Appendix A.3 □

Knowing that uniform-continuous functions form a nominal Scott domain allows us to prove the next theorem. It is crucial for using **Nsd** in the denotational semantics of programming languages with higher-order functions.

Theorem 3.4.28 (**cartesian closedness of Nsd**). **Nsd** *is a cartesian closed category (Definition 2.1.5) with $D_1 \to_{\text{uc}} D_2$ being the exponential for $D_1, D_2 \in$ **Nsd**.*

Proof. **Nsd** has finite products by Proposition 3.4.11. Proposition 3.4.27 shows that $D_1 \to_{\text{uc}} D_2 \in$ **Nsd** for $D_1, D_2 \in$ **Nsd**. It is easy to show that the evaluation function

$$ev \triangleq \lambda((f, d) \in (D_1 \to_{\text{uc}} D_2) \times D_1) \to f\, d \tag{3.19}$$

is uniform-continuous and equivariant, so $ev \in$ **Nsd**$((D_1 \to_{\text{uc}} D_2) \times D_1, D_2)$, and that for any $f \in$ **Nsd**$(D_1 \times D_2, D_3)$ currying

$$cur(f) \triangleq \lambda d_1 \in D_1 \to \lambda d_2 \in D_2 \to f\,(d_1, d_2) \tag{3.20}$$

gives a morphism $cur(f) \in$ **Nsd**$(D_1, D_2 \to_{\text{uc}} D_3)$. It is now straightforward to show that $\_ \to_{\text{uc}} \_$, ev and $cur(\_)$ satisfy the universal property of cartesian closedness for **Nsd** from Definition 2.1.5. □

3.4.4 Least fixed points

In domain theory, the fact that every continuous endofunction has a least fixed point is crucial. It allows us to model recursively defined terms and historically motivated the development of domain theory for computer science. Nominal Scott domains and uniform-continuous functions support least fixed points in the usual way.

Definition 3.4.29 (least pre-fixed point). In any poset D, a *pre-fixed point* of a function $f \in D \to D$ is an element $d \in D$ that satisfies $f\, d \sqsubseteq d$. Such an element d is the *least pre-fixed point* if it is additionally the least such: $(\forall d' \in D)\, f\, d' \sqsubseteq d' \Rightarrow d \sqsubseteq d'$. The least pre-fixed point of a function is necessarily unique and furthermore it is a fixed point, that is it satisfies $f\, d = d$.

Proposition 3.4.30 (existence of least pre-fixed points). *In a pointed udcpo D, every uniform-continuous function $f \in (D \to_{\mathrm{uc}} D)$ has a least pre-fixed point given by*

$$\mathrm{fix}\, f \triangleq \bigsqcup \{ f^n \bot \mid n \in \mathbb{N} \}. \tag{3.21}$$

where f^n is recursively defined for any $n \in \mathbb{N}$ by $f^0\, d \triangleq d$ and $f^{n+1}\, d \triangleq f\,(f^n\, d)$. For a nominal Scott domain $D \in \mathbf{Nsd}$ this gives us a morphism $\mathrm{fix} \in \mathbf{Nsd}(D \to_{\mathrm{uc}} D, D)$.

Proof. The set $\{ f^n \bot \mid n \in \mathbb{N} \}$ is directed and uniformly supported by $\mathrm{supp}\, f$, hence its join must exist in D. We can show by induction that $(\forall n \in \mathbb{N})\, (\pi \cdot f)^n \bot = \pi \cdot (f^n \bot)$ holds, and this implies that $\pi \cdot (\mathrm{fix}\, f) = \mathrm{fix}\,(\pi \cdot f)$. The rest of the proof the follows the classical Tarskian argument as in Abramsky and Jung [4, Theorem 2.1.19]. □

3.5 Examples

We present some examples of uniform-compact functions, which will be relevant for the full abstraction results in Chapter 5. The examples show that although the flat domain (Definition 3.4.5) of atomic names $\mathbb{A}_\bot$ has a countably infinite underlying set, it has very different uniform-compactness properties from the flat domain of natural numbers $\mathbb{N}_\bot$.

It seems not too surprising that the equality test for atomic names below is a uniform-compact function, and we can prove it by using the characterisation in (3.18). However, its equivalent formulation for natural numbers is already not uniform-compact, as Remark 3.5.4 shows.

Example 3.5.1 (name equality test). For each atomic name $a \in \mathbb{A}$, consider the function eq_a mapping between $\mathbb{A}_\bot$ and $\mathbb{B}_\bot$ given by

$$eq_a \triangleq \lambda d \in \mathbb{A}_\bot \to \begin{cases} \mathrm{true} & \text{if } d = a \\ \mathrm{false} & \text{if } d \in \mathbb{A} - \{a\} \\ \bot & \text{if } d = \bot. \end{cases} \tag{3.22}$$

It can be expressed in terms of hulls (Definition 2.3.35) and step functions (Definition 3.4.21) by

$$eq_a = \bigsqcup \mathrm{hull}_{\{a\}} \{ (a \searrow \mathrm{true}), (a' \searrow \mathrm{false}) \}$$

where a' is any atomic name not equal to a. Note that $\mathrm{hull}_{\{a\}}\{(a \searrow \mathrm{true}), (a' \searrow \mathrm{false})\}$ is a consistent set of step functions and by Theorem 2.3.38 it is orbit-finite, thus by (3.18) its join is uniform-compact: $eq_a \in \mathrm{K}(\mathbb{A}_\perp \to_{\mathrm{uc}} \mathbb{B}_\perp)$.

Similarly we can show that the name equality test for two names eq defined by

$$eq \triangleq \lambda(d_1, d_2) \in \mathbb{A}_\perp \times \mathbb{A}_\perp \to \begin{cases} \mathrm{true} & \text{if } d_1 = a_1,\ d_2 = a_2 \text{ and } a_1 = a_2 \\ \mathrm{false} & \text{if } d_1 = a_1,\ d_2 = a_2 \text{ and } a_1 \neq a_2 \\ \perp & \text{otherwise} \end{cases} \tag{3.23}$$

is uniform-compact $eq \in \mathrm{K}(\mathbb{A}_\perp \times \mathbb{A}_\perp \to_{\mathrm{uc}} \mathbb{B}_\perp)$ through observing that

$$eq = \bigsqcup \mathrm{hull}_\emptyset\{((a,a) \searrow \mathrm{true}), ((a,a') \searrow \mathrm{false})\},$$

with $a, a' \in \mathbb{A}$ and $a \neq a'$.

What is more surprising is that the functional for existential quantification over names below is uniform-compact. Furthermore, it can be given an operational semantics so that it can be added to PNA, as it is done in Section 5.2.

Example 3.5.2 (existential quantification over names). For each $f \in \mathbb{A}_\perp \to_{\mathrm{uc}} \mathbb{B}_\perp$ define

$$exists_{\mathbb{A}} f \triangleq \begin{cases} \mathrm{true} & \text{if } (\exists a \in \mathbb{A})\ f\, a = \mathrm{true} \\ \mathrm{false} & \text{if } (\forall a \in \mathbb{A})\ f\, a = \mathrm{false} \\ \perp & \text{otherwise.} \end{cases} \tag{3.24}$$

$exists_{\mathbb{A}}$ is not only uniform-continuous, it is in fact a uniform-compact element of the nominal Scott domain $(\mathbb{A}_\perp \to_{\mathrm{uc}} \mathbb{B}_\perp) \to_{\mathrm{uc}} \mathbb{B}_\perp$. To see this, first note that picking any $a \in \mathbb{A}$, the set $\mathrm{hull}_\emptyset\{(a \searrow \mathrm{false})\}$ is orbit-finite and consistent, hence by Lemma 3.4.24 its join exists and by (3.18) it is an element of $\mathrm{K}(\mathbb{A}_\perp \to_{\mathrm{uc}} \mathbb{B}_\perp)$. Observe that this join is exactly the function

$$k_{\mathrm{false}} \triangleq \lambda d \in \mathbb{A}_\perp \to \begin{cases} \mathrm{false} & \text{if } d \neq \perp \\ \perp & \text{otherwise} \end{cases} \tag{3.25}$$

and satisfies by (3.11) and (3.10) that

$$(\forall f \in \mathbb{A}_\perp \to_{\mathrm{uc}} \mathbb{B}_\perp)\ k_{\mathrm{false}} \sqsubseteq f \iff (\forall a \in \mathbb{A})\ f\, a = \mathrm{false}\,. \tag{3.26}$$

Furthermore, for any $g \in (\mathbb{A}_\perp \to_{\mathrm{uc}} \mathbb{B}_\perp) \to_{\mathrm{uc}} \mathbb{B}_\perp$ we have

$$\begin{aligned} & exists_{\mathbb{A}} \sqsubseteq g \\ \iff & (\forall f \in \mathbb{A}_\perp \to_{\mathrm{uc}} \mathbb{B}_\perp)\ exists_{\mathbb{A}} f \sqsubseteq g\, f \\ \iff & (\forall f \in \mathbb{A}_\perp \to_{\mathrm{uc}} \mathbb{B}_\perp)\ (((\exists a \in \mathbb{A})\ f\, a = \mathrm{true}) \Rightarrow g\, f = \mathrm{true}) \\ & \qquad\qquad \wedge (((\forall a' \in \mathbb{A})\ f\, a' = \mathrm{false}) \Rightarrow g\, f = \mathrm{false}) \\ \iff & (\forall f \in \mathbb{A}_\perp \to_{\mathrm{uc}} \mathbb{B}_\perp)(\forall a \in \mathbb{A})\ (f\, a = \mathrm{true} \Rightarrow g\, f = \mathrm{true}) \\ & \qquad\qquad \wedge ((\forall a' \in \mathbb{A})\ f\, a' = \mathrm{false}) \Rightarrow g\, f = \mathrm{false} \end{aligned}$$

which by (3.16) and (3.26) holds if and only if

$$(\forall f \in \mathbb{A}_\perp \to_{\mathrm{uc}} \mathbb{B}_\perp)(\forall a \in \mathbb{A})\,((a \searrow \mathrm{true}) \sqsubseteq f \Rightarrow g\,f = \mathrm{true}) \wedge (k_{\mathrm{false}} \sqsubseteq f \Rightarrow g\,f = \mathrm{false})$$

and by (3.11) and (3.13) this holds if and only if

$$(\forall a \in \mathbb{A})\,((a \searrow \mathrm{true}) \searrow \mathrm{true}) \sqsubseteq g \;\wedge\; (k_{\mathrm{false}} \searrow \mathrm{false}) \sqsubseteq g\,.$$

Thus picking some $a \in \mathbb{A}$, we have

$$\mathit{exists}_{\mathbb{A}} = \bigsqcup \mathrm{hull}_\emptyset\{((a \searrow \mathrm{true}) \searrow \mathrm{true}), (k_{\mathrm{false}} \searrow \mathrm{false})\}$$

and hence $\mathit{exists}_{\mathbb{A}} \in \mathrm{K}((\mathbb{A}_\perp \to_{\mathrm{uc}} \mathbb{B}_\perp) \to_{\mathrm{uc}} \mathbb{B}_\perp)$ holds by (3.18).

The construct of 'definite description' has a long history in logic, but it is not common in programming languages. We can show that it is uniform-compact for atomic names, using the same techniques as for existential quantification. Later in Section 5.2 we will give an operational semantics for definite description and add it to PNA.

Example 3.5.3 (**definite description over names**)**.** The name equality tests $eq_a \in \mathbb{A}_\perp \to_{\mathrm{uc}} \mathbb{B}_\perp$ from (3.22) satisfy

$$\begin{gathered}(\forall \pi \in \mathrm{Perm}(\mathbb{A}))\;\pi \cdot eq_a = eq_{\pi a}\\ eq_a = eq_{a'} \;\Rightarrow\; a = a'\\ eq_a \sqsubseteq f \;\Rightarrow\; eq_a = f\,.\end{gathered}$$

From these properties and uniform-compactness of eq_a it follows that the function $\mathit{the}_{\mathbb{A}}$ defined by

$$\mathit{the}_{\mathbb{A}} \triangleq \lambda(f \in \mathbb{A}_\perp \to_{\mathrm{uc}} \mathbb{B}_\perp) \to \begin{cases} a & \text{if } f = eq_a \text{ for some } a \in \mathbb{A} \\ \perp & \text{otherwise} \end{cases} \tag{3.27}$$

satisfies for all $g \in (\mathbb{A}_\perp \to_{\mathrm{uc}} \mathbb{B}_\perp) \to_{\mathrm{uc}} \mathbb{A}_\perp$ that $\mathit{the}_{\mathbb{A}} \sqsubseteq g$ holds if and only if $(\forall a \in \mathbb{A})\,(eq_a \searrow a) \sqsubseteq g$. Therefore picking any $a \in \mathbb{A}$, we have $\mathit{the}_{\mathbb{A}} = \bigsqcup \mathrm{hull}_\emptyset\{(eq_a \searrow a)\}$ and hence by (3.18) that $\mathit{the}_{\mathbb{A}} \in \mathrm{K}((\mathbb{A}_\perp \to_{\mathrm{uc}} \mathbb{B}_\perp) \to_{\mathrm{uc}} \mathbb{A}_\perp)$.

Remark 3.5.4 (**functionals for natural numbers**)*.* The analogue of the equality test function of Example 3.5.1 for natural numbers, $eq_n \in \mathbb{N}_\perp \to \mathbb{B}_\perp$, is not uniform-compact. To see this, note that it can be expressed as the join of a finitely supported chain of functions $eq_n = \bigsqcup\{eq_{n,m} \mid m \in \mathbb{N}\}$, where $eq_{n,m}$ is defined by

$$eq_{n,m}\,d \triangleq \begin{cases} eq_n\,d & \text{if } d = m' \wedge m' \leq m \\ \perp & \text{otherwise} \end{cases}$$

If eq_n was uniform-compact, then $eq_n \sqsubseteq eq_{n,m}$ would hold for some $m \in \mathbb{N}$, which is not the case. We cannot use the same kind of argument for eq_a, because there is no finitely supported total ordering of the elements of $\mathbb{A}$.

The functionals for existential quantification and definite description for natural numbers $\mathit{exists}_{\mathbb{N}}$ and $\mathit{the}_{\mathbb{N}}$ not only fail to be uniform-compact, they are also not uniform-continuous. For definite description, $eq_n = \bigsqcup\{eq_{n,m} \mid m \in \mathbb{N}\}$ from above gives a counter-example to uniform-continuity, because $\mathit{the}_{\mathbb{N}}(eq_n) = n$ as well as $(\forall m \in \mathbb{N})\ \mathit{the}_{\mathbb{N}}(eq_{n,m}) = \bot$ hold. For existential quantification, observe that the function from (3.25) (for numbers instead of names, so $k_{\mathrm{false}} \in \mathbb{N}_\bot \to_{\mathrm{uc}} \mathbb{B}_\bot$) can be characterised by $k_{\mathrm{false}} = \bigsqcup\{k_{\mathrm{false},m} \mid m \in \mathbb{N}\}$, where

$$k_{\mathrm{false},m}\, d \triangleq \begin{cases} \mathrm{false} & \text{if } d = m' \wedge m' \leq m \\ \bot & \text{otherwise.} \end{cases}$$

This contradicts uniform-continuity of $\mathit{exists}_{\mathbb{N}}$, as $\mathit{exists}_{\mathbb{N}}(k_{\mathrm{false}}) = \mathrm{false}$ and $(\forall m \in \mathbb{N})\ \mathit{exists}_{\mathbb{N}}(k_{\mathrm{false},m}) = \bot$.

3.6 Abstraction, concretion and restriction

Name abstractions can elegantly express α-equivalence in syntax and therefore it is desirable to have name abstractions in a metaprogramming language. We show next that our nominal domain theory is a good match for name abstractions, because the constructs for abstraction, concretion and restriction give well-behaved uniform-continuous functions.

3.6.1 Abstraction

The results in Proposition 3.1.6, Lemma 3.1.7, Proposition 3.2.2 and Theorem 3.3.5 already proved properties of name abstractions in the domain-theoretic setting, and this section analyses its properties further. We start with showing that name abstractions form a nominal Scott domain.

Theorem 3.6.1 (**nominal Scott domain of abstractions**). *If D is a nominal Scott domain, then so is the nominal poset $[\mathbb{A}]D$ from Proposition 3.1.6. The operation of name abstraction extends to a morphism $\langle \_ \rangle \_ \in \mathbf{Nsd}(\mathbb{A}_\bot \times D, [\mathbb{A}]D)$ once we define $\langle \bot \rangle d \triangleq \bot$.*

Proof. Theorem 3.3.5 shows that $[\mathbb{A}]D$ is an ω-algebraic udcpo and by Lemma 3.1.7 we know that it has joins for all bounded and finitely supported subsets, so it is indeed a nominal Scott domain. Equivariance of name abstraction follows from (2.21). Uniform-continuity in the first component is a special case of Lemma 3.4.7 and uniform-continuity in the second component is Proposition 3.2.2, so overall by Lemma 3.4.10 $\langle \_ \rangle \_$ is uniform-continuous. □

The function case_X (that is defined for any nominal set X) from Example 2.3.22 can be extended to the nominal Scott domain setting.

Lemma 3.6.2 (**case-function**). *For any nominal Scott domain D, the function*

$$case_D(e, f_1, f_2, f_3) \triangleq \begin{cases} f_1\, a & \text{if } e = [a]_\alpha \\ f_2\, [t_1]_\alpha\, [t_2]_\alpha & \text{if } e = [t_1\, t_2]_\alpha \\ f_3\, (\langle a \rangle [t]_\alpha) & \text{if } e = [\lambda a.t]_\alpha \\ \bot & \text{otherwise} \end{cases}$$

is equivariant and uniform-continuous, and hence is a morphism $case_D \in \mathbf{Nsd}((\Lambda_\alpha)_\bot \times (\mathbb{A}_\bot \to_{\mathrm{uc}} D) \times ((\Lambda_\alpha)_\bot \to_{\mathrm{uc}} (\Lambda_\alpha)_\bot \to_{\mathrm{uc}} X) \times ([\mathbb{A}](\Lambda_\alpha)_\bot \to_{\mathrm{uc}} D), D)$.

Proof. First note that $case_D$ takes uniform-continuous functions as arguments. Equivariance follows from equivariance of $case_X$ in Example 2.3.22. We can prove uniform-continuity in each component separately, using Lemma 3.4.7 for the first component, and then the overall uniform-continuity follows by Lemma 3.4.10. □

The next lemma shows that the abstraction of a flat domain is again a flat domain.

Lemma 3.6.3 (**flat domain of abstractions**). *For any nominal set X it holds that* $[\mathbb{A}](X_\bot)$ *and* $([\mathbb{A}]X)_\bot$ *are isomorphic.*

Proof. As in Proposition 3.1.6, we have $\langle a \rangle \bot = \bot$. If $\langle a \rangle d \sqsubseteq \langle a' \rangle d'$ holds in $[\mathbb{A}](X_\bot)$, then for a fresh b either $(a\ b) \cdot d = \bot$ or $(a\ b) \cdot d = (a'\ b) \cdot d'$ must hold, so either $\langle a \rangle d = \bot$ or $\langle a \rangle d = \langle a' \rangle d'$. With this it follows that $[\mathbb{A}](X_\bot) \cong ([\mathbb{A}]X)_\bot$. □

As already pointed out in Section 2.3.4, it is desirable to turn the partial operation of concretion from Definition 2.3.23 into a total one. In our domain-theoretic setting, one apparent simple solution is to let the undefined cases denote bottom by defining

$$e \mathbin{@^f} a \triangleq \begin{cases} e \mathbin{@} a & \text{if } a \# e \\ \bot & \text{otherwise.} \end{cases}$$

Unfortunately, this approach does *not* work, because it gives us a function that is not monotone in general. See Pitts [44, Example 11.7] for a counter-example. Therefore $\_ \mathbin{@^f} \_$ is ill-defined. The remedy is to use name restriction operations (Definition 2.3.26) that are uniform-continuous.

3.6.2 Uniform-continuous name restriction

We extend the definition of a name restriction operation on nominal sets from Definition 2.3.26 to nominal Scott domains.

Definition 3.6.4 (**uniform-continuous name restriction**). For any nominal Scott domain D, a *uniform-continuous name restriction operation* on D is an equivariant and uniform-continuous function $\_\backslash\_ \in \mathbf{Nsd}(\mathbb{A}_\bot \times D, D)$ that satisfies

$$(\forall d \in D)\ \bot \backslash d\ =\ \bot \tag{3.28}$$

as well as (2.24), (2.25) and (2.26) with X replaced by D.

Many nominal Scott domains have such a restriction operation. The next four lemmas show that restriction extends to various domain constructions. They will form the basis of giving a denotational semantics to PNA's Odersky-style local names, see Theorem 4.3.1.

Lemma 3.6.5 (**flat restriction**). *Every flat nominal Scott domain $X_\perp$ has a uniform-continuous name restriction operation defined by*

$$d\backslash d' \triangleq \begin{cases} x & \text{if } d = a \wedge d' = x \wedge a \# x \\ \perp & \text{otherwise.} \end{cases} \tag{3.29}$$

Proof. Lemma 3.1.5 shows that $\perp$ is equivariant, so we can apply Lemma 2.3.28 to see that equivariance, (2.24), (2.25) and (2.26) are satisfied. (3.28) is immediate, and uniform-continuity follows from Lemma 3.4.7. □

Lemma 3.6.6 (**product restriction**). *If $D_1, D_2 \in \mathbf{Nsd}$ have uniform-continuous name restriction operations, then their product $D_1 \times D_2$ has one given by*

$$(\forall d \in \mathbb{A}_\perp)(\forall d_1 \in D_1)(\forall d_2 \in D_2)\, d\backslash(d_1, d_2) \triangleq (d\backslash d_1, d\backslash d_2).$$

Proof. Uniform-continuity and (3.28) are consequences of Lemma 3.4.9. The rest of the properties follow as in Lemma 2.3.29. □

Lemma 3.6.7 (**function restriction**). *Assuming $D_1, D_2 \in \mathbf{Nsd}$ and D_2 has a uniform-continuous name restriction operation, then whether or not D_1 has one as well, the exponential $D_1 \to_{\mathrm{uc}} D_2$ has such an operation, satisfying*

$$(\forall a \in \mathbb{A})(\forall f \in D_1 \to_{\mathrm{uc}} D_2)(\forall d_1 \in D_1)\, a \# d_1 \Rightarrow (a\backslash f)\, d_1 = a\backslash(f\, d_1). \tag{3.30}$$

This fully specifies the restriction operation, since by Lemma 2.3.15 and (3.28) *it implies that for all $d \in \mathbb{A}_\perp$ and $f \in D_1 \to_{\mathrm{uc}} D_2$ we have*

$$d\backslash f = \lambda d_1 \in D_1 \to \begin{cases} \text{fresh } a' \text{ in } a'\backslash(((a\ a') \cdot f)\, d_1) & \text{if } d = a \\ \perp & \text{otherwise.} \end{cases} \tag{3.31}$$

Proof. Equivariance of $(d, f) \mapsto a\backslash f$, finite support of $d\backslash f$, (2.24), (2.25) and (2.26) follow from Lemma 2.3.30. (3.28) is immediate by definition. Lemma 2.3.41, (3.3) and uniform-continuity of $d' \mapsto a\backslash d'$ imply that every $a\backslash f$ is uniform-continuous. Uniform-continuity of $d' \mapsto a\backslash d'$ together with (3.12) also implies that $f \mapsto a\backslash f$ is uniform-continuous, and hence $(a, f) \mapsto a\backslash f$ is so by Lemma 3.4.10. □

Lemma 3.6.8 (**abstraction restriction**). *For any nominal Scott domain D possessing a uniform-continuous name restriction operation, it holds that $[\mathbb{A}]D$ has one too. It is fully specified by* (3.28) *and*

$$(\forall a, a' \in \mathbb{A})(\forall d \in D)\, a \neq a' \Rightarrow a\backslash\langle a'\rangle d = \langle a'\rangle(a\backslash d). \tag{3.32}$$

Proof. Lemma 2.3.31 shows well-definedness, equivariance, (2.24), (2.25) and (2.26). What remains to be shown is uniform-continuity of $e \mapsto a\backslash e$ and this follows from (A.2), uniform-continuity of $d \mapsto a\backslash d$, Proposition 3.2.2 and Lemma 2.3.25. □

In Definition 2.3.26 we discussed that a nominal set can have several name restriction operations. The next remark shows that on flat domains we have the uniqueness property that our uniform-continuous name restriction operation (3.29) is the least such.

Remark 3.6.9 (**least restriction**). By (2.25) we are forced to return x in the first case of (3.29). This means that for any flat nominal Scott domain $X_\perp$ the restriction operation in Lemma 3.6.5 is the least such operation, in the sense that for every other uniform-continuous name restriction operation $\_\backslash_2\_$ we have

$$(\forall d \in \mathbb{A}_\perp)(\forall d' \in X_\perp)\,(d\backslash d') \sqsubseteq (d\backslash_2 d').$$

3.6.3 Total concretion

The total concretion function from Proposition 2.3.27 can be extended to nominal Scott domains, given that the restriction operation is uniform-continuous.

Theorem 3.6.10 (total concretion in Nsd). *Assuming D is a nominal Scott domain possessing a uniform-continuous name restriction operation, we can extend the partial operation of concretion from Definition 2.3.23 to a total function by defining for all $\langle a\rangle d \in [\mathbb{A}]D$ and $d' \in \mathbb{A}_\perp$*

$$(\langle a\rangle d)@^t\, d' \triangleq \begin{cases} d & \text{if } d' = a' \wedge a = a' \\ a\backslash(a\,a')\cdot d & \text{if } d' = a' \wedge a \neq a' \\ \perp & \text{otherwise.} \end{cases} \tag{3.33}$$

This total concretion function is well-defined, equivariant and uniform-continuous, hence is a morphism $\_@^t\_ \in \mathbf{Nsd}([\mathbb{A}]D \times \mathbb{A}_\perp, D)$.

Proof. Equivariance follows from Proposition 2.3.27. We prove uniform-continuity component-wise using Lemma 3.4.10. For the first component we use the description of uniform-directed joins in abstraction domains (A.2), and for the second component we apply Lemma 3.4.7. □

Lemma 3.6.11 (total and partial concretion coincide). *For fresh names total concretion and partial concretion coincide in* **Nsd**. *For any $e \in [\mathbb{A}]D$, where D is a nominal Scott domain with uniform-continuous name restriction operation, we have*

$$a \# e \;\Rightarrow\; e@a = e@^t\, a\,.$$

Proof. By Lemma 2.3.25 we have $e = \langle a\rangle(e@a)$, and then the property follows by (2.25). □

We will use the total concretion morphism to interpret name concretion expressions in the denotational semantics of PNA in the next chapter.

CHAPTER 4

PNA: PCF WITH NAMES

In Chapter 3 we derived nominal Scott domains (Definition 3.4.1) and discussed constructs on them. This chapter applies nominal Scott domains for giving a denotational semantics to the pure (in the sense of state-free or referentially transparent) functional programming language PNA (Programming with Name Abstractions), an extension of Plotkin's PCF [47, Section 2] with nominal constructs for metaprogramming.

Sections 4.1 and 4.2 introduce PNA and its type system. Section 4.3 defines the nominal-Scott-domain-based denotational semantics for PNA and Section 4.4 gives two different flavours of operational semantics to it.

4.1 Syntax

Like PCF, PNA has arithmetic constructs, call-by-name higher-order functions and fixed-point recursion. What distinguishes the two languages is that PNA treats names as first-class citizens, having a data type of names and constructs for locally scoping them, for swapping and testing them for equality, and for name abstraction and concretion. The main practical advantage of using PNA over PCF is that name abstraction allows us to express computation over object-level programming languages with binding, without having to worry about α-equivalence. To exercise this use of name abstraction, PNA features a representative datatype for programming language syntax, namely a type for α-equivalence classes of λ-calculus syntax as in Example 2.3.22, called `term`.[1] This datatype comes with three constructors (`V` for variables, `A` for applications and `L` for λ-abstractions) and a pattern matching construct

$$\texttt{case}\, e\, \texttt{of}\, (\texttt{V}\, x_1 \to e_1 \mid \texttt{A}\, x_2\, x_2' \to e_2 \mid \texttt{L}\, x_3 \to e_3).$$

[1] It would be straightforward to extend PNA to deal with many-sorted atomic names and object level syntax over any 'nominal algebraic signature' as explained in Pitts [44, Section 4.7 and Definition 8.2].

Using these new constructs, computation over α-equivalence classes of syntax can be expressed directly in PNA. For example, when *subst* is the PNA expression

$$\begin{aligned} \mathit{subst} \triangleq\ & \lambda y' : \mathtt{term} \to \lambda x : \mathtt{name} \to \mathtt{fix}(\lambda(f : \mathtt{term} \to \mathtt{term}) \to \lambda y : \mathtt{term} \to \\ & \mathtt{case}\ y\ \mathtt{of} \\ & \quad \mathtt{V}\, x_1 \to \mathtt{if}\, x_1 = x\, \mathtt{then}\, y'\, \mathtt{else}\, y \\ & \quad \mid \mathtt{A}\, x_2\, x_2' \to \mathtt{A}(f\ x_2)(f\ x_2') \\ & \quad \mid \mathtt{L}\, x_3 \to \mathtt{L}(\alpha a.\, f(x_3\ @\ a))) \end{aligned} \tag{4.1}$$

then $\mathit{subst}\, e_1\, a\, e_2$ computes the λ-term obtained by capture-avoiding substitution of the λ-term represented by e_1 for all free occurrences of the variable named a in the λ-term represented by e_2.

4.1.1 Expressions

Figure 4.1 gives the grammar of expressions for PNA, the part below the dotted line is what is being added to PCF (we will keep this convention throughout the chapter). These additional name-related constructs below the dotted line are directly derived from the intended model in nominal Scott domains, as we will see in more detail in Section 4.3, where we define denotational semantics.

Identifiers and binding There are two kinds of identifier in the language: *variables* $x, y, z, f, \ldots \in \mathbb{V}$ and *atomic names* $a, b, c, \ldots \in \mathbb{A}$; the sets $\mathbb{V}$ of variables and $\mathbb{A}$ of atomic names are disjoint and countably infinite. We use the set of atomic names $\mathbb{A}$ from Definition 2.3.1 in the syntax, allowing a slight overlap between syntax and semantics. Both kinds of identifier may be bound: the language's binding forms are $\lambda x : \tau \to \_$, $\nu a.\_$, $\mathtt{case}\, e\, \mathtt{of}\, (\mathtt{V}\, x_1 \to \_ \mid \mathtt{A}\, x_2\, x_2' \to \_ \mid \mathtt{L}\, x_3 \to \_)$ and $\alpha a.\_$. We identify expressions up to α-equivalence of bound identifiers.

Notation 4.1.1 (**free identifiers**). For any expression e, we write $\mathrm{fn}\, e$ for its set of free atomic names, and $\mathrm{fv}\, e$ for its set of free variables.

The reason for making a syntactic distinction between variables $x \in \mathbb{V}$ and atomic names $a \in \mathbb{A}$ is that they behave differently: an occurrence of x in an expression stands for an unknown expression; whereas an occurrence of a denotes an entity whose identity is definitely different from the other atomic names that occur in the expression. Thus if x and y are two different variables, the meaning of the boolean expression $x = y$ is indeterminate, whereas if a and b are two different atomic names, then the meaning of $a = b$ is the boolean value false. For this reason various properties of PNA, such as its typing judgement, are preserved by the operation of substitution of expressions for variables, but are only preserved by permutations of atomic names rather than more general forms of substitution for names.

Permutation action Figure 4.2 defines the permutation action $\pi \cdot e$ for all permutations $\pi \in \mathrm{Perm}(\mathbb{A})$ (introduced in Definition 2.3.2) and expressions $e \in \mathrm{Exp}^{\mathsf{PNA}}$, by structural recursion on e.

$e \in \mathrm{Exp}^{\mathsf{PNA}} ::=$	*expressions*
x	variable ($x \in \mathbb{V}$)
$\mathtt{T}$	truth
$\mathtt{F}$	falsity
$\mathtt{if}\, e\, \mathtt{then}\, e\, \mathtt{else}\, e$	conditional
$\mathtt{O}$	number zero
$\mathtt{S}\, e$	successor
$\mathtt{pred}\, e$	predecessor
$\mathtt{zero}\, e$	zero test
$(e\,, e)$	pair
$\mathtt{fst}\, e$	first projection
$\mathtt{snd}\, e$	second projection
$\lambda x : \tau \to e$	function abstraction
$e\, e$	function application
$\mathtt{fix}\, e$	fixed-point recursion
a	atomic name ($a \in \mathbb{A}$)
$\nu a.\, e$	local name scoping
$(e \rightleftharpoons e)\, e$	name swapping
$e = e$	name equality test
$\mathtt{V}\, e$	variable term
$\mathtt{A}\, e\, e$	application term
$\mathtt{L}\, e$	lambda term
$\mathtt{case}\, e\, \mathtt{of}\, (\mathtt{V}\, x \to e \mid \mathtt{A}\, x\, x \to e \mid \mathtt{L}\, x \to e)$	term case
$\alpha a.\, e$	name abstraction
$e \mathbin{@} e$	name concretion

Figure 4.1: Expressions of PNA

$$\pi \cdot x = x \qquad \pi \cdot \mathtt{T} = \mathtt{T} \qquad \pi \cdot \mathtt{F} = \mathtt{F}$$

$$\pi \cdot \mathtt{if}\, e_1\, \mathtt{then}\, e_2\, \mathtt{else}\, e_3 = \mathtt{if}\, \pi \cdot e_1\, \mathtt{then}\, \pi \cdot e_2\, \mathtt{else}\, \pi \cdot e_3 \qquad \pi \cdot \mathtt{0} = \mathtt{0}$$

$$\pi \cdot \mathtt{S}\, e = \mathtt{S}(\pi \cdot e) \qquad \pi \cdot \mathtt{pred}\, e = \mathtt{pred}(\pi \cdot e) \qquad \pi \cdot \mathtt{zero}\, e = \mathtt{zero}(\pi \cdot e)$$

$$\pi \cdot (e_1, e_2) = (\pi \cdot e_1, \pi \cdot e_2) \qquad \pi \cdot \mathtt{fst}\, e = \mathtt{fst}\, (\pi \cdot e) \qquad \pi \cdot \mathtt{snd}\, e = \mathtt{snd}\, (\pi \cdot e)$$

$$\pi \cdot \lambda x : \tau \to e = \lambda x : \tau \to \pi \cdot e \qquad \pi \cdot (e_1\, e_2) = (\pi \cdot e_1)(\pi \cdot e_2) \qquad \pi \cdot \mathtt{fix}\, e = \mathtt{fix}(\pi \cdot e)$$

$$\pi \cdot a = \pi\, a \qquad \pi \cdot \nu a.\, e = \nu(\pi\, a).\, \pi \cdot e \qquad \pi \cdot (e_1 \rightleftharpoons e_2)\, e_3 = (\pi \cdot e_1 \rightleftharpoons \pi \cdot e_2)\, \pi \cdot e_3$$

$$\pi \cdot (e_1 = e_2) = (\pi \cdot e_1) = (\pi \cdot e_2) \qquad \pi \cdot \mathtt{V}\, e = \mathtt{V}(\pi \cdot e) \qquad \pi \cdot \mathtt{A}\, e_1\, e_2 = \mathtt{A}(\pi \cdot e_1)(\pi \cdot e_2)$$

$$\pi \cdot \mathtt{L}\, e = \mathtt{L}(\pi \cdot e) \qquad \pi \cdot \mathtt{case}\, e\, \mathtt{of}\, (\mathtt{V}\, x_1 \to e_1 \mid \mathtt{A}\, x_2\, x_2' \to e_2 \mid \mathtt{L}\, x_3 \to e_3) =$$

$$\mathtt{case}\, \pi \cdot e\, \mathtt{of}\, (\mathtt{V}\, x_1 \to \pi \cdot e_1 \mid \mathtt{A}\, x_2\, x_2' \to \pi \cdot e_2 \mid \mathtt{L}\, x_3 \to \pi \cdot e_3) \qquad \pi \cdot \alpha a.\, e = \alpha(\pi\, a).\, \pi \cdot e$$

$$\pi \cdot (e_1 \mathbin{@} e_2) = (\pi \cdot e_1) \mathbin{@} (\pi \cdot e_2)$$

Figure 4.2: Permutation action for PNA

Lemma 4.1.2 (**nominal set of expressions**). *Figure 4.2 defines a permutation action (Definition 2.3.5) that turns* $\mathrm{Exp}^{\mathsf{PNA}}$ *into a nominal set (Definition 2.3.6). As expressions are identified up to* α*-equivalence, the support of an expression is its set of free atomic names, so*

$$\operatorname{supp} e \;=\; \operatorname{fn} e \tag{4.2}$$

for all $e \in \mathrm{Exp}^{\mathsf{PNA}}$.

Proof. By structural induction on e. The fact that α-equivalence is an equivariant equivalence relation is crucial. See also Pitts [44, Section 4.1 and Proposition 8.10]. For any expression e, we have that $\operatorname{fn} e$ strongly supports e in the sense that $(\forall \pi \in \mathrm{Perm}(\mathbb{A}))((\forall a \in \operatorname{fn} e)\, \pi\, a = a) \Leftrightarrow \pi \cdot e = e$ and so by Pitts [44, Theorem 2.7] property (4.2) follows. □

Capture-avoiding substitution The substitution operation avoids capture of both free variables and free atomic names by the language's binding constructs. We write $e[e'/x]$ for the substitution of an expression $e' \in \mathrm{Exp}^{\mathsf{PNA}}$ for a variable $x \in \mathbb{V}$ in an expression $e \in \mathrm{Exp}^{\mathsf{PNA}}$. It is defined by structural recursion on e in Figure 4.3.

Lemma 4.1.3 (**equivariance of substitution**). *For all* $\pi \in \mathrm{Perm}(\mathbb{A})$, $x \in \mathbb{V}$ *and* $e, e' \in \mathrm{Exp}^{\mathsf{PNA}}$ *it holds that*

$$\pi \cdot (e[e'/x]) \;=\; (\pi \cdot e)[(\pi \cdot e')/x].$$

Proof. By structural induction on e. □

$$x'[e'/x] = \begin{cases} e' & \text{if } x' = x \\ x' & \text{otherwise} \end{cases} \qquad \mathtt{T}[e'/x] = \mathtt{T} \qquad \mathtt{F}[e'/x] = \mathtt{F}$$

$$(\mathtt{if}\, e_1\, \mathtt{then}\, e_2\, \mathtt{then}\, e_3)[e'/x] = \mathtt{if}\, e_1[e'/x]\, \mathtt{then}\, e_2[e'/x]\, \mathtt{else}\, [e'/x]$$

$$\mathtt{O}[e'/x] = \mathtt{O} \qquad (\mathtt{S}\, e)[e'/x] = \mathtt{S}(e[e'/x]) \qquad (\mathtt{pred}\, e)[e'/x] = \mathtt{pred}(e[e'/x])$$

$$(\mathtt{zero}\, e)[e'/x] = \mathtt{zero}(e[e'/x]) \qquad (e_1, e_2)[e'/x] = (e_1[e'/x], e_2[e'/x])$$

$$(\mathtt{fst}\, e)[e'/x] = \mathtt{fst}\, e[e'/x] \qquad (\mathtt{snd}\, e)[e'/x] = \mathtt{snd}\, e[e'/x]$$

$$(\lambda x' : \tau \to e)[e'/x] = \lambda x' : \tau \to e[e'/x] \quad \text{if } x' \notin \mathrm{fv}\, e' \cup \{x\}$$

$$(e_1\, e_2)[e'/x] = e_1[e'/x] e_2[e'/x] \qquad (\mathtt{fix}\, e)[e'/x] = \mathtt{fix}\, e[e'/x] \qquad a[e'/x] = a$$

$$(\nu a.\, e)[e'/x] = \nu a.\, (e[e'/x]) \quad \text{if } a \notin \mathrm{fn}\, e'$$

$$((e_1 \rightleftharpoons e_2)\, e_3)[e'/x] = (e_1[e'/x] \rightleftharpoons e_2[e'/x])\, e_3[e'/x]$$

$$(e_1 = e_2)[e'/x] = e_1[e'/x] = e_2[e'/x] \qquad (\mathtt{V}\, e)[e'/x] = \mathtt{V}(e[e'/x])$$

$$(\mathtt{A}\, e_1\, e_2)[e'/x] = \mathtt{A}(e_1[e'/x])(e_2[e'/x]) \qquad (\mathtt{L}\, e)[e'/x] = \mathtt{L}(e[e'/x])$$

$$\mathtt{case}\, e\, \mathtt{of}\, (\mathtt{V}\, x_1 \to e_1 \mid \mathtt{A}\, x_2\, x_2' \to e_2 \mid \mathtt{L}\, x_3 \to e_3)[e'/x] = \mathtt{case}\, e[e'/x]\, \mathtt{of}\, (\mathtt{V}\, x_1 \to e_1[e'/x] \mid$$
$$\mathtt{A}\, x_2\, x_2' \to e_2[e'/x] \mid \mathtt{L}\, x_3 \to e_3[e'/x]) \quad \text{if } x_1, x_2, x_2', x_3 \notin \mathrm{fv}\, e' \cup \{x\}$$

$$(\alpha a.\, e)[e'/x] = \alpha a.\, e[e'/x] \quad \text{if } a \notin \mathrm{fn}\, e' \qquad (e_1 \mathbin{@} e_2)[e'/x] = e_1[e'/x] \mathbin{@} e_2[e'/x]$$

Figure 4.3: Capture-avoiding substitution for PNA

$$
\begin{array}{ll}
c \in \mathrm{Can}^{\mathsf{PNA}} ::= & \textit{canonical forms} \\
\mathtt{T} \mid \mathtt{F} \mid \mathtt{0} \mid \mathtt{S}\, c \mid (e\,, e) \mid \lambda x : \tau \to e \mid a \mid \mathtt{V}\, c \mid \mathtt{A}\, c\, c \mid \mathtt{L}\, c \mid \alpha a.c &
\end{array}
$$

Figure 4.4: Canonical forms of PNA

Substitution is not commutative in general. However, the composition of two substitutions satisfies the following property.

Lemma 4.1.4 (substitution composition). *For all* $x, y \in \mathbb{V}$ *and* $e, e', e'' \in \mathrm{Exp}^{\mathsf{PNA}}$ *it holds that*

$$x \neq y \wedge x \notin \mathrm{fv}\, e'' \Rightarrow (e[e'/x])[e''/y] = (e[e''/y])[e'[e''/y]/x].$$

Proof. By structural induction on e. □

Remark 4.1.5 (**α-structural recursion and induction**). In the previous definitions and proofs we frequently used structural recursion and induction on expressions. Strictly speaking, this was not structural recursion and induction over plain syntax-trees, because we identify expressions up to α-equivalence of bound identifiers. In fact, we deal with syntax trees with 'pointers' for binding and use the framework of α-structural recursion and induction (see Pitts [41] and Pitts [44, Section 8.5 and 8.6]). We implicitly use the α-structural framework whenever we apply recursion or induction on the structure of syntax with binding. It justifies the use of side-conditions for bound identifiers, such as $a \notin \mathrm{fn}\, e'$ in the case for name abstractions in Figure 4.3.

Simultaneous substitution It is straightforward to extend the above definition and lemmas for substitution of a single expression to multiple ones. The *simultaneous substitution* of the expressions $e_1, \ldots, e_n \in \mathrm{Exp}^{\mathsf{PNA}}$ for distinct variables $x_1, \ldots, x_n \in \mathbb{V}$ in an expression $e \in \mathrm{Exp}^{\mathsf{PNA}}$ is written as $e[e_1/x_1, \ldots, e_n/x_n]$. For notational convenience we leave the detailed definition of simultaneous substitution to the reader. If the substituted expressions are variable-closed ($\forall i \in \{1, \ldots, n\}$) $\mathrm{fv}\, e_i = \emptyset$ then we can express simultaneous substitution as a sequence of substitutions $e[e_1/x_1, \ldots, e_n/x_n] = e[e_1/x_1] \ldots [e_n/x_n]$.

Using vector notation, we sometimes write simultaneous substitution as $e[\vec{e}/\vec{x}]$, where $\vec{e} = e_1, \ldots, e_n$ and $\vec{x} = x_1, \ldots, x_n$ are as before.

4.1.2 Canonical forms

Canonical forms are fully evaluated expressions. As such, they play a crucial role in the operational semantics of Section 4.4. Figure 4.4 gives the syntax of canonical forms in PNA. As usual, α-equivalent canonical forms are implicitly identified.

We can easily show by structural induction that every canonical form is an expression, that is $\mathrm{Can}^{\mathsf{PNA}} \subseteq \mathrm{Exp}^{\mathsf{PNA}}$. Henceforth canonical forms inherit the permutation action and substitution operation from expressions. Canonical forms are closed under the permutation action, the equivalent of Lemma 4.1.2 holds and hence $\mathrm{Can}^{\mathsf{PNA}}$ is a nominal set. The next lemma will be used in Appendix A.5.

$C \in \mathrm{Cont}^{\mathsf{PNA}} ::=$ *contexts*
$[-] \mid \mathtt{if}\, C\, \mathtt{then}\, e\, \mathtt{else}\, e \mid \mathtt{if}\, e\, \mathtt{then}\, C\, \mathtt{else}\, e \mid \mathtt{if}\, e\, \mathtt{then}\, e\, \mathtt{else}\, C \mid \mathtt{S}\, C \mid \mathtt{pred}\, C$
$\mid \mathtt{zero}\, C \mid (C, e) \mid (e, C) \mid \mathtt{fst}\, C \mid \mathtt{snd}\, C \mid \lambda x : \tau \to C \mid C\, e \mid e\, C \mid \mathtt{fix}\, C \mid \nu a.\, C$
$\mid (C \rightleftharpoons e)\, e \mid (e \rightleftharpoons C)\, e \mid (e \rightleftharpoons e)\, C \mid C = e \mid e = C \mid \mathtt{V}\, C \mid \mathtt{A}\, C\, e \mid \mathtt{A}\, e\, C \mid \mathtt{L}\, C$
$\mid \mathtt{case}\, C\, \mathtt{of}\, (\mathtt{V}\, x \to e \mid \mathtt{A}\, x\, x \to e \mid \mathtt{L}\, x \to e) \mid \mathtt{case}\, e\, \mathtt{of}\, (\mathtt{V}\, x \to C \mid \mathtt{A}\, x\, x \to e \mid \mathtt{L}\, x \to e)$
$\mid \mathtt{case}\, e\, \mathtt{of}\, (\mathtt{V}\, x \to e \mid \mathtt{A}\, x\, x \to C \mid \mathtt{L}\, x \to e) \mid \mathtt{case}\, e\, \mathtt{of}\, (\mathtt{V}\, x \to e \mid \mathtt{A}\, x\, x \to e \mid \mathtt{L}\, x \to C)$
$\mid \alpha a.\, C \mid C\, @\, e \mid e\, @\, C$

Figure 4.5: Contexts of PNA

Lemma 4.1.6 (**substituting canonical forms**). *For all* $e \in \mathrm{Exp}^{\mathsf{PNA}}$ *and* $c \in \mathrm{Can}^{\mathsf{PNA}}$ *it holds that*

$$e[c/x] \in \mathrm{Can}^{\mathsf{PNA}} \Rightarrow (\forall c' \in \mathrm{Can}^{\mathsf{PNA}})\, e[c'/x] \in \mathrm{Can}^{\mathsf{PNA}}.$$

Proof. By induction on the structure of e. □

4.1.3 Contexts

A PNA *context* is an expression with a single sub-expression replaced by a hole, written '—'. Figure 4.5 defines contexts formally. We often write $C[-]$ for a context C to emphasize the hole.

Contexts are *not* quotiented by α-equivalence of bound identifiers, so for example $\lambda x : \mathtt{bool} \to -$ and $\lambda y : \mathtt{bool} \to -$ are different contexts. However, we still identify contexts that have α-equivalent sub-expressions, thus $(-, \lambda x : \tau \to x)$ is the same context as $(-, \lambda y : \tau \to y)$.

For any $C \in \mathrm{Cont}^{\mathsf{PNA}}$, define $C[e]$ to be the expression that results from replacing the hole $-$ by an expression $e \in \mathrm{Exp}^{\mathsf{PNA}}$, possibly capturing free variables and atomic names of e. We omit the formal definition which is a straightforward structural induction on C. In the same way we define context composition $C[C']$ to be the capturing replacement of the hole in C with the context C'.

Lemma 4.1.7 (**context composition**). *Context composition is well-defined. If* C *and* C' *are contexts then* $C[C']$ *is one, and it holds that* $(C[C'])[e] = C[C'[e]]$

Proof. By structural induction on C. □

4.1.4 Frame-stacks

Figure 4.6 defines the syntax of PNA frames and frame-stacks. They are closely related to contexts and will be used in the operational semantics of Section 4.4.3. Like contexts, frames are expressions with a hole (written as $\cdot$ for frames). However, their structure is more restricted, in particular the hole always appears in the uppermost level of frames and henceforth the definition of frames is not recursive. Recursiveness is added through frame-stacks which are just finite lists of frames, whose length can be defined by $|\mathrm{Id}| \triangleq 0$ and $|F \circ E| \triangleq |F| + 1$.

$E \in \mathrm{Frame}^{\mathsf{PNA}} ::=$ *frames*

$\mathtt{if}\cdot\,\mathtt{then}\,e\,\mathtt{else}\,e \mid \mathtt{S}\cdot \mid \mathtt{pred}\cdot \mid \mathtt{zero}\cdot \mid \mathtt{fst}\cdot \mid \mathtt{snd}\cdot \mid \cdot\, e \mid \nu a.\cdot \mid (\cdot \rightleftharpoons e)\, e \mid (a \rightleftharpoons \cdot)\, e$
$\mid (a \rightleftharpoons a)\cdot \mid \cdot = e \mid a = \cdot \mid \mathtt{V}\cdot \mid \mathtt{A}\cdot e \mid \mathtt{A}\, c\cdot \mid \mathtt{L}\cdot \mid \mathtt{case}\cdot\,\mathtt{of}\,(\mathtt{V}\, x \to e \mid \mathtt{A}\, x\, x \to e \mid \mathtt{L}\, x \to e)$
$\mid \alpha a.\cdot \mid \cdot\,@\,e \mid c\,@\,\cdot$

$F \in \mathrm{Stack}^{\mathsf{PNA}} ::= \mathrm{Id} \mid F \circ E$ *frame-stacks*

Figure 4.6: Frame-stacks of PNA

The permutation action on frames $\pi \cdot E$ is defined in accordance with the permutation action on expressions in Figure 4.2, by applying the permutation to all subexpressions, including all atomic names. A simple recursive definition $\pi \cdot \mathrm{Id} \triangleq \mathrm{Id}$ and $\pi \cdot (F \circ E) \triangleq (\pi \cdot F) \circ (\pi \cdot E)$ gives the permutation action on frame stacks.

There is a translation $(\_)^\circ : \mathrm{Stack}^{\mathsf{PNA}} \to \mathrm{Cont}^{\mathsf{PNA}}$ defined by $\mathrm{Id}^\circ \triangleq [-]$ and $(F \circ E)^\circ \triangleq F^\circ[E[-/\cdot]]$. By this every frame-stack can be considered to be a context. Yet there are many more contexts than frame-stacks, because the grammar of frames is more restrictive. For instance, $\lambda x : \mathtt{bool} \to -$ is a context but there is *no* frame-stack $\mathrm{Id} \circ (\lambda x : \mathtt{bool} \to \cdot)$. The image of $(\_)^\circ$ gives us exactly the 'evaluation contexts' of Felleisen and Hieb [14].

As it is the case for contexts, frames are *not* quotiented by α-equivalence of bound identifiers, yet α-equivalent sub-expressions are identified. Thus for example $\nu a.\cdot$ and $\nu a'.\cdot$ are different frames, but $\cdot = \nu a.a$ is the same frame as $\cdot = \nu a'.a'$. In the $\mathtt{case}$-frame all of the bracket is treated as sub-expression, so x_1 is bound in $\mathtt{case}\cdot\mathtt{of}(\mathtt{V}\, x_1 \to x_1 \mid \cdots)$. We define the free name function for frames accordingly, leading to $\mathrm{fn}\, \nu a.\cdot = \{a\}$ and $\mathrm{fn}\,(a \rightleftharpoons \cdot)\, e = \{a\} \cup \mathrm{fn}\, e$, for instance. This extends to frame-stacks in the obvious way: $\mathrm{fn}\,\mathrm{Id} \triangleq \emptyset$ and $\mathrm{fn}\,(F \circ E) \triangleq (\mathrm{fn}\, F) \cup (\mathrm{fn}\, E)$. The free variable function is defined in the same way. It happens to be slightly simpler, because frames are defined in a way that no variable-binder has a hole in its scope (for example there is no frame like $\lambda x : \mathtt{bool} \to \cdot$).

$F[e]$ is the operation of replacing the hole in a frame-stack $F \in \mathrm{Stack}^{\mathsf{PNA}}$ with an expression $e \in \mathrm{Exp}^{\mathsf{PNA}}$. It is defined by

$$\begin{aligned} \mathrm{Id}[e] &\triangleq e \\ (F \circ E)[e] &\triangleq F[E[e/\cdot]], \end{aligned}$$

where $E[e/\cdot]$ is the (possibly name-capturing) replacement of the hole $\cdot$ in the frame E with e. The hole-replacement operation also coincides with the according operation for contexts: $F[e] = F^\circ[e]$.

It is useful to define the concatenation of two frame-stacks $F \bullet F'$. This is done by recursion on F' through $F \bullet \mathrm{Id} \triangleq F$ and $F \bullet (F' \circ E') \triangleq (F \bullet F') \circ E'$.

Lemma 4.1.8 (frame-stack hole replacement). *For any frame-stacks $F, F' \in \mathrm{Stack}^{\mathsf{PNA}}$ it holds that*

$$(F \bullet F')[e] = F[F'[e]].$$

$\tau \in \mathrm{Typ}^{\mathsf{PNA}} ::=$	PNA-*types*
$\mathtt{bool}$	booleans
$\mathtt{nat}$	natural numbers
$\tau \times \tau$	products
$\tau \to \tau$	functions
$\mathtt{name}$	atomic names
$\mathtt{term}$	λ-terms
$\delta\,\tau$	name abstractions
$\gamma \in \mathrm{Gnd}^{\mathsf{PNA}} ::= \mathtt{bool} \mid \mathtt{nat} \mid \mathtt{name} \mid \mathtt{term}$	*ground types*
$\gamma' \in \mathrm{Grnd}^{\mathsf{PNA}} ::= \mathtt{bool} \mid \mathtt{nat} \mid \mathtt{name} \mid \mathtt{term} \mid \delta\,\gamma'$	*extended ground types*

Figure 4.7: Types of PNA

Proof. By induction on F'. □

Definition 4.1.9 (substitution for frame-stacks). In the failure of full abstraction proofs in Section 5.4 we use a substitution operation on frame-stacks, which applies the substitution to each frame

$$\mathrm{Id}[e'/x] \triangleq \mathrm{Id} \qquad (F \circ E)[e'/x] \triangleq F[e'/x] \circ E[e'/x]$$

and for frames it is just capture-avoiding substitution into all sub-expressions, while ignoring the hole. For instance, we have $(\mathtt{if}\cdot\,\mathtt{then}\ e_1\ \mathtt{else}\ e_2)[e'/x] = \mathtt{if}\cdot\,\mathtt{then}\ e_1[e'/x]\ \mathtt{else}\ e_2[e'/x]$, $(\nu a.\cdot)[e'/x] = \nu a.\cdot$ (without side conditions), and $(\mathtt{case}\cdot\,\mathtt{of}\ (\mathtt{V}\,x_1 \to e_1 \mid \mathtt{A}\,x_2\,x_2' \to e_2 \mid \mathtt{L}\,x_3 \to e_3))[e'/x] = \mathtt{case}\cdot\,\mathtt{of}\ (\mathtt{V}\,x_1 \to e_1[e'/x] \mid \mathtt{A}\,x_2\,x_2' \to e_2[e'/x] \mid \mathtt{L}\,x_3 \to e_3[e'/x])$ under the side condition that $x_1, x_2, x_2', x_3 \notin \mathrm{fn}\,e' \cup \{x\}$.

4.2 Typing

PNA is a simply-typed language, meaning that it has a static type system where the types have a simple structure not involving advanced features such as type quantifiers or sub-typing. In this section we define the syntax of types, define the type system for expressions and also derive a type system for contexts.

4.2.1 Syntax of types

The grammar of PNA-types is given in Figure 4.7. The same figure gives the grammar of ground types and, for technical convenience, the grammar of extended ground types.

The types for booleans, natural numbers, products and functions are the same as for PCF. Regarding the types below the dotted line (the types not being in PCF), there is a ground type `name` of atomic names and a ground type `term` for the object-level syntax of the λ-calculus. Name abstraction types correspond to the semantic

$$\frac{(x:\tau)\in\Gamma}{\Gamma\vdash x:\tau} \qquad \frac{c\in\{\mathtt{T},\mathtt{F}\}}{\Gamma\vdash c:\mathtt{bool}} \qquad \frac{\Gamma\vdash e_1:\mathtt{bool} \qquad \Gamma\vdash e_2:\tau \qquad \Gamma\vdash e_3:\tau}{\Gamma\vdash \mathtt{if}\, e_1\, \mathtt{then}\, e_2\, \mathtt{else}\, e_3:\tau}$$

$$\frac{}{\Gamma\vdash \mathtt{0}:\mathtt{nat}} \qquad \frac{\Gamma\vdash e:\mathtt{nat}}{\Gamma\vdash \mathtt{S}e:\mathtt{nat}} \qquad \frac{\Gamma\vdash e:\mathtt{nat}}{\Gamma\vdash \mathtt{pred}\, e:\mathtt{nat}} \qquad \frac{\Gamma\vdash e:\mathtt{nat}}{\Gamma\vdash \mathtt{zero}\, e:\mathtt{bool}}$$

$$\frac{\Gamma\vdash e_1:\tau_1 \qquad \Gamma\vdash e_2:\tau_2}{\Gamma\vdash (e_1,e_2):\tau_1\times\tau_2} \qquad \frac{\Gamma\vdash e:\tau_1\times\tau_2}{\Gamma\vdash \mathtt{fst}\, e:\tau_1} \qquad \frac{\Gamma\vdash e:\tau_1\times\tau_2}{\Gamma\vdash \mathtt{snd}\, e:\tau_2}$$

$$\frac{\Gamma,x:\tau\vdash e:\tau'}{\Gamma\vdash \lambda x:\tau\to e:\tau\to\tau'} \qquad \frac{\Gamma\vdash e_1:\tau\to\tau' \qquad \Gamma\vdash e_2:\tau}{\Gamma\vdash e_1\, e_2:\tau'} \qquad \frac{\Gamma\vdash e:\tau\to\tau}{\Gamma\vdash \mathtt{fix}\, e:\tau}$$

. . . . . .

$$\frac{a\in\mathbb{A}}{\Gamma\vdash a:\mathtt{name}} \qquad \frac{a\in\mathbb{A} \qquad \Gamma\vdash e:\tau}{\Gamma\vdash \nu a.e:\tau} \qquad \frac{\Gamma\vdash e_1:\mathtt{name} \qquad \Gamma\vdash e_2:\mathtt{name} \qquad \Gamma\vdash e_3:\tau}{\Gamma\vdash (e_1\rightleftharpoons e_2)\, e_3:\tau}$$

$$\frac{\Gamma\vdash e_1:\mathtt{name} \qquad \Gamma\vdash e_2:\mathtt{name}}{\Gamma\vdash e_1=e_2:\mathtt{bool}} \qquad \frac{\Gamma\vdash e:\mathtt{name}}{\Gamma\vdash \mathtt{V}e:\mathtt{term}} \qquad \frac{\Gamma\vdash e_1:\mathtt{term} \qquad \Gamma\vdash e_2:\mathtt{term}}{\Gamma\vdash \mathtt{A}\, e_1\, e_2:\mathtt{term}}$$

$$\frac{\Gamma\vdash e:\delta\,\mathtt{term}}{\Gamma\vdash \mathtt{L}e:\mathtt{term}} \qquad \frac{\begin{array}{c}\Gamma\vdash e:\mathtt{term} \qquad \Gamma,x_1:\mathtt{name}\vdash e_1:\tau \\ \Gamma,x_2:\mathtt{term},x_2':\mathtt{term}\vdash e_2:\tau \qquad \Gamma,x_3:\delta\,\mathtt{term}\vdash e_3:\tau\end{array}}{\Gamma\vdash \mathtt{case}\, e\, \mathtt{of}\, (\mathtt{V}\, x_1\to e_1 \mid \mathtt{A}\, x_2\, x_2'\to e_2 \mid \mathtt{L}\, x_3\to e_3):\tau}$$

$$\frac{a\in\mathbb{A} \qquad \Gamma\vdash e:\tau}{\Gamma\vdash \alpha a.e:\delta\,\tau} \qquad \frac{\Gamma\vdash e_1:\delta\,\tau \qquad \Gamma\vdash e_2:\mathtt{name}}{\Gamma\vdash e_1\,@\,e_2:\tau}$$

Figure 4.8: Type system of PNA

construction of abstraction sets from Definition 2.3.18. The sets $\mathrm{Typ}^{\mathsf{PNA}}$, $\mathrm{Gnd}^{\mathsf{PNA}}$, and $\mathrm{Grnd}^{\mathsf{PNA}}$ form discrete nominal sets (as in Section 2.3.2) that are ordered by subset inclusion as follows: $\mathrm{Gnd}^{\mathsf{PNA}} \subseteq \mathrm{Grnd}^{\mathsf{PNA}} \subseteq \mathrm{Typ}^{\mathsf{PNA}}$.

4.2.2 Type system

Figure 4.8 defines the rules of the type system, including the usual rules for PCF and, below the dotted line, rules concerning names. It is given in terms of a typing judgement

$$\Gamma\vdash e:\tau$$

(read as 'in the environment $\Gamma\in\mathrm{Env}^{\mathsf{PNA}}$ the expression $e\in\mathrm{Exp}^{\mathsf{PNA}}$ has type $\tau\in\mathrm{Typ}^{\mathsf{PNA}}$'). Occurrences of free variables in e are tracked by the typing environment

Γ.

Definition 4.2.1 (**typing environment**). A (PNA-) *typing environment* Γ is a finite partial function from $\mathbb{V}$ to $\mathrm{Typ}^{\mathsf{PNA}}$, written as $\Gamma = \{x_1 : \tau_1, \ldots, x_n : \tau_n\}$. Let $\mathrm{Env}^{\mathsf{PNA}}$ be the set of all such typing environments. Furthermore we use the notation $\mathrm{dom}\,\Gamma$ for the domain of Γ and $\Gamma\, x = \tau$ for $(x : \tau) \in \Gamma$. The size $|\Gamma|$ of a typing environment is defined to be the size of its domain $|\mathrm{dom}\,\Gamma|$. We also define $\Gamma, x : \tau$ to be the typing environment that maps x to τ and behaves like Γ otherwise. Similarly, Γ, Γ' is just the typing environment that behaves like Γ' on $\mathrm{dom}\,\Gamma'$ and behaves like Γ otherwise. In the forthcoming developments, when using Γ, Γ' we implicitly assume the typing environments to be disjoint, in the sense that $(\mathrm{dom}\,\Gamma) \cap (\mathrm{dom}\,\Gamma') = \emptyset$.

Note that no freshness assumptions or similar type system alterations are needed to type-check the metaprogramming constructs of PNA (in particular name concretion). In this sense, the type system of PNA is as simple as the one for PCF. The type systems of several other languages for nominal metaprogramming are significantly more complicated, see the discussion in Section 6.1.2.

What follows are proofs of some properties of the type system that will be used throughout the rest of this thesis.

Lemma 4.2.2 (**equivariance of the type system**). *For any* $\pi \in \mathrm{Perm}(\mathbb{A})$ *we have*

$$\Gamma \vdash e : \tau \;\Rightarrow\; \Gamma \vdash \pi \cdot e : \tau\,.$$

Proof. It is easy to show that if (H, c) is a rule of the type system (where H is the set of hypotheses and c is the conclusion), then $(\pi \cdot H, \pi \cdot c)$ is also a valid rule of the type system. This means that the set of rules is equivariant and with Pitts [44, Theorem 7.3] this concludes the proof. □

Lemma 4.2.3 (**uniqueness of typing**). *Types are unique, in the sense that it holds that*

$$\Gamma \vdash e : \tau \;\wedge\; \Gamma \vdash e : \tau' \;\Rightarrow\; \tau = \tau'\,.$$

Proof. By rule induction on $\Gamma \vdash e : \tau$. □

The next lemma proves that the typing environment always contains the free variables of the expression, as already stated above.

Lemma 4.2.4 (**free variables**). *It holds that*

$$\Gamma \vdash e : \tau \;\Rightarrow\; \mathrm{fv}\, e \subseteq \mathrm{dom}\,\Gamma\,.$$

Proof. By rule induction on $\Gamma \vdash e : \tau$. □

The typing judgement is independent from variables that are not free in the expression.

Lemma 4.2.5 (type weakening). *For all* $\Gamma, \Gamma' \in \mathrm{Env}^{\mathsf{PNA}}$ *we have*

$$\Gamma \vdash e : \tau \;\wedge\; ((\forall x \in \mathrm{fv}\, e)\, \Gamma\, x = \Gamma'\, x) \;\Rightarrow\; \Gamma' \vdash e : \tau\,.$$

This implies that the typing judgement can be weakened by arbitrary typing environments:

$$\Gamma \vdash e : \tau \;\Rightarrow\; (\forall \Gamma' \in \mathrm{Env}^{\mathsf{PNA}})\, \Gamma', \Gamma \vdash e : \tau\,.$$

Proof. By rule induction on $\Gamma \vdash e : \tau$. □

Lemma 4.2.6 (PNA Substitution Lemma). *Substitution preserves typing in the sense that it holds that*

$$\Gamma, x : \tau' \vdash e : \tau \;\wedge\; \Gamma \vdash e' : \tau' \;\Rightarrow\; \Gamma \vdash e[e'/x] : \tau\,.$$

Proof. By rule induction on $\Gamma, x : \tau' \vdash e : \tau$. □

Definition 4.2.7 (sets of typed expressions). The set of variable-closed expressions and canonical forms of a certain type are defined as follows:

$$\mathrm{Exp}^{\mathsf{PNA}}(\tau) \triangleq \{e \in \mathrm{Exp}^{\mathsf{PNA}} \mid \emptyset \vdash e : \tau\}$$
$$\mathrm{Can}^{\mathsf{PNA}}(\tau) \triangleq \{c \in \mathrm{Can}^{\mathsf{PNA}} \mid \emptyset \vdash c : \tau\}\,.$$

It follows directly from Lemma 4.2.2 that $\mathrm{Exp}^{\mathsf{PNA}}(\tau)$ and $\mathrm{Can}^{\mathsf{PNA}}(\tau)$ are nominal sets.

The next lemma shows that the structure of a canonical form is determined by its type.

Lemma 4.2.8 (structure of canonical forms). *The following holds for all canonical forms* $c \in \mathrm{Can}^{\mathsf{PNA}}$*:*

- *If* $\Gamma \vdash c : \mathtt{bool}$ *then* $c = \mathtt{T}$ *or* $c = \mathtt{F}$.
- *If* $\Gamma \vdash c : \mathtt{nat}$ *then* $c = \mathtt{S}^n\,\mathtt{0}$ *for some* $n \in \mathbb{N}$*, where* $\mathtt{S}^0\,\mathtt{0} = \mathtt{0}$ *and* $\mathtt{S}^{n+1}\,\mathtt{0} = \mathtt{S}\,\mathtt{S}^n\,\mathtt{0}$.
- *If* $\Gamma \vdash c : \tau_1 \times \tau_2$ *then* $c = (e_1, e_2)$ *where* $\Gamma \vdash e_1 : \tau_1$ *and* $\Gamma \vdash e_2 : \tau_2$.
- *If* $\Gamma \vdash c : \tau_1 \to \tau_2$ *then* $c = \lambda x : \tau_1 \to e$ *where* $\Gamma, x : \tau_1 \vdash e : \tau_2$.
- *If* $\Gamma \vdash c : \mathtt{name}$ *then* $c = a$ *for some* $a \in \mathbb{A}$.
- *If* $\Gamma \vdash c : \mathtt{term}$ *then* $c = \mathtt{V}\,a$ *with* $a \in \mathbb{A}$*, or* $c = \mathtt{A}\,c_1\,c_2$ *with* $\Gamma \vdash c_1 : \mathtt{term}$ *and* $\Gamma \vdash c_2 : \mathtt{term}$*, or* $c = \mathtt{L}\,c'$ *with* $\Gamma \vdash c' : \delta\,\mathtt{term}$.
- *If* $\Gamma \vdash c : \delta\,\tau$ *then* $c = \alpha a.\, c'$ *with* $\Gamma \vdash c' : \tau$ *and* $a \in \mathbb{A}$.

Proof. By the typing rules and induction on the structure of c. □

In this thesis, we focus our attention on expressions that have a type, and consider all other expressions to be erroneous. In particular, the denotational semantics in Section 4.3.2 is only defined for typeable expressions.

$$\frac{}{[-] : (\Gamma \triangleright \tau) \rightsquigarrow (\Gamma \triangleright \tau)} \qquad \frac{C : (\Gamma \triangleright \tau) \rightsquigarrow (\Gamma' \triangleright \texttt{bool}) \qquad \Gamma' \vdash e_2 : \tau' \qquad \Gamma' \vdash e_3 : \tau'}{\texttt{if}\, C\, \texttt{then}\, e_2\, \texttt{else}\, e_3 : (\Gamma \triangleright \tau) \rightsquigarrow (\Gamma' \triangleright \tau')}$$

$$\frac{\Gamma' \vdash e_1 : \texttt{bool} \qquad C : (\Gamma \triangleright \tau) \rightsquigarrow (\Gamma' \triangleright \tau') \qquad \Gamma' \vdash e_3 : \tau'}{\texttt{if}\, e_1\, \texttt{then}\, C\, \texttt{else}\, e_3 : (\Gamma \triangleright \tau) \rightsquigarrow (\Gamma' \triangleright \tau')} \qquad \frac{C : (\Gamma \triangleright \tau) \rightsquigarrow (\Gamma' \triangleright \texttt{nat})}{\texttt{S}\, C : (\Gamma \triangleright \tau) \rightsquigarrow (\Gamma' \triangleright \texttt{nat})}$$

$$\frac{C : (\Gamma \triangleright \tau) \rightsquigarrow (\Gamma', x : \tau_1' \triangleright \tau_2')}{\lambda x : \tau_1' \to C : (\Gamma \triangleright \tau) \rightsquigarrow (\Gamma' \triangleright \tau_1' \to \tau_2')} \qquad \frac{C : (\Gamma \triangleright \tau) \rightsquigarrow (\Gamma' \triangleright \tau_1' \to \tau') \qquad \Gamma' \vdash e_2 : \tau_1'}{C\, e_2 : (\Gamma \triangleright \tau) \rightsquigarrow (\Gamma' \triangleright \tau')}$$

$$\frac{\Gamma' \vdash e_1 : \tau_1' \to \tau' \qquad C : (\Gamma \triangleright \tau) \rightsquigarrow (\Gamma' \triangleright \tau_1')}{e_1\, C : (\Gamma \triangleright \tau) \rightsquigarrow (\Gamma' \triangleright \tau')} \qquad \frac{C : (\Gamma \triangleright \tau) \rightsquigarrow (\Gamma' \triangleright \tau' \to \tau')}{\texttt{fix}\, C : (\Gamma \triangleright \tau) \rightsquigarrow (\Gamma' \triangleright \tau')}$$

$$\frac{a \in \mathbb{A} \qquad C : (\Gamma \triangleright \tau) \rightsquigarrow (\Gamma' \triangleright \tau')}{\alpha a.\, C : (\Gamma \triangleright \tau) \rightsquigarrow (\Gamma' \triangleright \delta\ \tau')} \qquad \frac{C : (\Gamma \triangleright \tau) \rightsquigarrow (\Gamma' \triangleright \delta\ \tau') \qquad \Gamma' \vdash e_2 : \texttt{name}}{C\, @\, e_2 : (\Gamma \triangleright \tau) \rightsquigarrow (\Gamma' \triangleright \tau')}$$

$$\frac{\Gamma' \vdash e_1 : \delta\ \tau' \qquad C : (\Gamma \triangleright \tau) \rightsquigarrow (\Gamma' \triangleright \texttt{name})}{e_1\, @\, C : (\Gamma \triangleright \tau) \rightsquigarrow (\Gamma' \triangleright \tau')}$$

Figure 4.9: Selected typing rules for PNA contexts

4.2.3 Context typing

As explained in Section 4.1.3, contexts can be seen as expressions with a hole that can be replaced with other expressions. For later purposes, we want that the replacement of the hole with a well-typed expression e leads to a well-typed result $C[e]$, which leads us to the notion of well-typed contexts. We define the typing judgement for contexts

$$C : (\Gamma \triangleright \tau) \rightsquigarrow (\Gamma' \triangleright \tau')$$

by rules including those in Figure 4.9. For brevity, the figure does not give all rules of the definition, but the missing ones can be easily derived from the typing rules for expressions in Figure 4.8. We adapt the notation of Harper [21, Section 47.1].

The intuitive meaning of $C : (\Gamma \triangleright \tau) \rightsquigarrow (\Gamma' \triangleright \tau')$ is that whenever we are given $\Gamma \vdash e : \tau$ then the hole replacement satisfies $\Gamma' \vdash C[e] : \tau'$, and this intuition is confirmed by Lemma 4.2.9.

Lemma 4.2.9 (typed hole replacement). *It holds that* $C : (\Gamma \triangleright \tau) \rightsquigarrow (\Gamma' \triangleright \tau')$ *and* $\Gamma \vdash e : \tau$ *imply* $\Gamma' \vdash C[e] : \tau'$.

Proof. By rule induction on $C : (\Gamma \triangleright \tau) \rightsquigarrow (\Gamma' \triangleright \tau')$. □

Lemma 4.2.10 (typed context composition). *Context composition is well-typed. If* $C : (\Gamma' \triangleright \tau') \rightsquigarrow (\Gamma'' \triangleright \tau'')$ *and* $C' : (\Gamma \triangleright \tau) \rightsquigarrow (\Gamma' \triangleright \tau')$, *then* $C[C'] : (\Gamma \triangleright \tau) \rightsquigarrow (\Gamma'' \triangleright \tau'')$.

$$[\![\mathtt{bool}]\!] \triangleq \mathbb{B}_\perp \qquad [\![\mathtt{nat}]\!] \triangleq \mathbb{N}_\perp \qquad [\![\tau \times \tau']\!] \triangleq [\![\tau]\!] \times [\![\tau']\!] \qquad [\![\tau \to \tau']\!] \triangleq [\![\tau]\!] \to_{\mathrm{uc}} [\![\tau']\!]$$

$$[\![\mathtt{name}]\!] \triangleq \mathbb{A}_\perp \qquad [\![\mathtt{term}]\!] \triangleq (\Lambda_\alpha)_\perp \qquad [\![\delta\ \tau]\!] \triangleq [\mathbb{A}][\![\tau]\!]$$

$$[\![\{x_1 : \tau_1, \ldots, x_n : \tau_n\}]\!] \triangleq [\![\tau_1]\!] \times \cdots \times [\![\tau_n]\!]$$

Figure 4.10: Denotations of PNA types

Proof. By rule induction on $C : (\Gamma \triangleright \tau) \rightsquigarrow (\Gamma' \triangleright \tau')$. □

Lemma 4.2.11 (**weakening for context typing**). *If* $C : (\Gamma \triangleright \tau) \rightsquigarrow (\Gamma' \triangleright \tau')$ *then for any* $\Gamma'' \in \mathrm{Env}^{\mathsf{PNA}}$ *we have* $C : (\Gamma'', \Gamma \triangleright \tau) \rightsquigarrow (\Gamma'', \Gamma' \triangleright \tau')$.

Proof. By rule induction on $C : (\Gamma \triangleright \tau) \rightsquigarrow (\Gamma' \triangleright \tau')$. □

Remark 4.2.12 (**rule-free context typing**). It might be tempting to the reader to define typed contexts by $C : (\Gamma \triangleright \tau) \rightsquigarrow (\Gamma' \triangleright \tau') \triangleq (\forall e \in \mathrm{Exp}^{\mathsf{PNA}})\ \Gamma \vdash e : \tau \Rightarrow \Gamma' \vdash C[e] : \tau'$, in order to avoid the additional rules in Figure 4.9. However, this means that we loose the ability to perform proofs by rule induction on $C : (\Gamma \triangleright \tau) \rightsquigarrow (\Gamma' \triangleright \tau')$ which leads to significant problems in forthcoming proofs. For example, Lemma 4.3.7 would be extremely tedious to prove, whereas by using rule induction on $(\Gamma \triangleright \tau) \rightsquigarrow (\Gamma' \triangleright \tau')$ the proof is simple and elegant.

4.3 Denotational semantics

This section applies the nominal domain theory from Chapter 3, in order to give a denotational semantics to PNA. In our model, types are denoted by nominal Scott domains and expressions are denoted by uniform-continuous functions. We present the denotational semantics before the operational semantics, because we consider it more basic. In particular, the operational rules for name abstraction, concretion and restriction in Section 4.4 are directly derived from the denotational semantics.

4.3.1 Denotations for types

For each PNA type $\tau \in \mathrm{Typ}^{\mathsf{PNA}}$, we define a nominal Scott domain $[\![\tau]\!]$ in Figure 4.10 by recursion on the structure of τ.

The types for booleans, natural numbers, atomic names and λ-calculus syntax are denoted by flat domains (as in Lemma 3.4.6). For booleans and natural numbers, the discrete nominal sets of $\mathbb{B} = \{\mathrm{true}, \mathrm{false}\}$ and $\mathbb{N} = \{0, 1, 2, \ldots\}$ (see Section 2.3.2) are underlying the flat domain constructions. For atomic names, the underlying set is $\mathbb{A}$ and for λ-terms it is the nominal set of λ-terms Λ_α as it is given in Example 2.3.22. Product types are denoted by the product of nominal Scott domains (Proposition 3.4.11) and function types are denoted by the nominal Scott domain

of uniform-continuous functions (Proposition 3.4.27). Finally, abstraction types are denoted by the nominal Scott domain of name abstractions as in Theorem 3.6.1.

The denotations of typing environments are given by finite cartesian products. Finite tuples $\rho \in [\![\Gamma]\!]$ can be interpreted as partial functions from variables to domains such that $\mathrm{dom}\,\rho = \mathrm{dom}\,\Gamma$ and $\rho(x) \in [\![\Gamma(x)]\!]$ for all $x \in \mathrm{dom}\,\Gamma$. We call such partial functions Γ-*valuation*. If $\rho \in [\![\Gamma]\!]$, $x \notin \mathrm{dom}(\Gamma)$ and $d \in [\![\tau]\!]$, then we write $\rho[x \mapsto d]$ for the $(\Gamma, x : \tau)$-valuation that maps x to d and otherwise acts like ρ. Similarly, given $\rho \in [\![\Gamma]\!]$ and $\rho' \in [\![\Gamma']\!]$ we write $\rho'\rho$ for the (Γ', Γ)-valuation that behaves like ρ on $\mathrm{dom}\,\Gamma$ and behaves like ρ' otherwise.

By using the results in Section 3.6.2, we can give a uniform-continuous name restriction operation to every PNA-type, as the next theorem shows.

Theorem 4.3.1 (**uniform-continuous restriction for** PNA)**.** *Every nominal Scott domain $[\![\tau]\!]$ denoting a* PNA *type $\tau \in \mathrm{Typ}^{\mathsf{PNA}}$ possesses a uniform-continuous name restriction operation as in Definition 3.6.4. Furthermore, the nominal Scott domain of uniform-continuous functions between typing environment denotations and types $[\![\Gamma]\!] \to_{\mathrm{uc}} [\![\tau]\!]$ for $\Gamma \in \mathrm{Env}^{\mathsf{PNA}}$ and $\tau \in \mathrm{Typ}^{\mathsf{PNA}}$ possesses such an operation too.*

Proof. We proceed by induction on τ. The induction bases for ground types $\tau \in \mathrm{Gnd}^{\mathsf{PNA}}$ are all special cases of Lemma 3.6.5. The inductive step for product types is Lemma 3.6.6, for function types it is Lemma 3.6.7 and for abstraction types it is Lemma 3.6.8. This shows that every $[\![\tau]\!]$ has a uniform-continuous restriction operation and for $[\![\Gamma]\!] \to_{\mathrm{uc}} [\![\tau]\!]$ we apply Lemma 3.6.7 once more. □

4.3.2 Denotations for expressions

For each well-typed expression $\Gamma \vdash e : \tau$ and Γ-valuation $\rho \in [\![\Gamma]\!]$ we define an element $[\![e]\!]\rho \in [\![\tau]\!]$ satisfying the clauses in Figure 4.11, by recursion over the typing relation. So formally we define uniform-continuous functions

$$[\![\Gamma \vdash e : \tau]\!] \in [\![\Gamma]\!] \to_{\mathrm{uc}} [\![\tau]\!]\,. \tag{4.3}$$

However, we will often leave the types implicit for better readability and write $[\![e]\!]$ instead of $[\![\Gamma \vdash e : \tau]\!]$. This convention is already used in Figure 4.11.

The clauses for constructs from PCF (above the dotted line) are analogous to the standard denotational semantics of PCF with classical Scott domains. The functions $proj_1$ and $proj_2$ in the clauses for $\mathtt{fst}\,e$ and $\mathtt{snd}\,e$ are the first and second projection functions for pairs from (2.9) and (2.10). fix in the clause for $\mathtt{fix}\,e$ is the least fixed point function from (3.21).

The clauses below the dotted line in Figure 4.11 are for the new syntactic constructs of PNA. Atomic names are their own denotation. Explicit swapping expressions $(e_1 \rightleftharpoons e_2)\,e_3$ use the permutation action (Definition 2.3.5) that each nominal Scott domain possesses by definition. We follow Example 2.3.22 in that α-equivalence classes of λ-terms are written as $[t]_\alpha$. The name abstraction clause $\alpha a.\,e$ corresponds to semantic name abstraction in Theorem 3.6.1. The clause for $\nu a.\,e$ makes use of the

$$[\![x]\!]\rho \triangleq \rho\, x \qquad [\![\mathtt{T}]\!]\rho \triangleq \mathrm{true} \qquad [\![\mathtt{F}]\!]\rho \triangleq \mathrm{false} \qquad [\![\mathtt{0}]\!]\rho \triangleq \mathtt{0}$$

$$[\![\begin{array}{l}\mathtt{if}\ e_1\ \mathtt{then}\\ e_2\ \mathtt{else}\ e_3\end{array}]\!]\rho \triangleq \begin{cases} [\![e_2]\!]\rho & \text{if } [\![e_1]\!]\rho = \mathrm{true} \\ [\![e_3]\!]\rho & \text{if } [\![e_1]\!]\rho = \mathrm{false} \\ \bot & \text{otherwise} \end{cases} \qquad [\![\mathtt{zero}\, e]\!]\rho \triangleq \begin{cases} \mathrm{true} & \text{if } [\![e]\!]\rho = \mathtt{0} \\ \mathrm{false} & \text{if } [\![e]\!]\rho = n+1 \\ \bot & \text{otherwise} \end{cases}$$

$$[\![\mathtt{S}\, e]\!]\rho \triangleq \begin{cases} n+1 & \text{if } [\![e]\!]\rho = n \\ \bot & \text{otherwise} \end{cases} \qquad [\![\mathtt{pred}\, e]\!]\rho \triangleq \begin{cases} n & \text{if } [\![e]\!]\rho = n+1 \\ \bot & \text{otherwise} \end{cases}$$

$$[\![(e_1, e_2)]\!]\rho \triangleq ([\![e_1]\!]\rho, [\![e_2]\!]\rho) \qquad [\![\mathtt{fst}\, e]\!]\rho \triangleq proj_1([\![e]\!]\rho) \qquad [\![\mathtt{snd}\, e]\!]\rho = proj_2([\![e]\!]\rho)$$

$$[\![\lambda x : \tau \to e]\!]\rho \triangleq \lambda d \in [\![\tau]\!] \to [\![e]\!]\rho[x \mapsto d] \qquad [\![e_1\, e_2]\!]\rho \triangleq [\![e_1]\!]\rho([\![e_2]\!]\rho)$$

$$[\![\mathtt{fix}\, e]\!]\rho \triangleq \mathrm{fix}([\![e]\!]\rho)$$

$$[\![a]\!]\rho \triangleq a \qquad [\![\nu a.\, e]\!]\rho \triangleq a\backslash([\![e]\!]\rho) \quad \text{if } a \# \rho$$

$$[\![(e_1 \rightleftharpoons e_2)\, e_3]\!]\rho \triangleq \begin{cases} (a_1\, a_2) \cdot ([\![e_3]\!]\rho) & \text{if } [\![e_1]\!]\rho = a_1 \text{ and } [\![e_2]\!]\rho = a_2 \\ \bot & \text{otherwise} \end{cases}$$

$$[\![e_1 = e_2]\!]\rho \triangleq \begin{cases} \mathrm{true} & \text{if } [\![e_i]\!]\rho = a_i \text{ and } a_1 = a_2 \\ \mathrm{false} & \text{if } [\![e_i]\!]\rho = a_i \text{ and } a_1 \neq a_2 \\ \bot & \text{otherwise} \end{cases} \qquad [\![\mathtt{V}\, e]\!]\rho \triangleq \begin{cases} [a]_\alpha & \text{if } [\![e]\!]\rho = a \\ \bot & \text{otherwise} \end{cases}$$

$$[\![\mathtt{A}\, e_1\, e_2]\!]\rho \triangleq \begin{cases} [t_1\, t_2]_\alpha & \text{if } [\![e_i]\!]\rho = [t_i]_\alpha \\ \bot & \text{otherwise} \end{cases} \qquad [\![\mathtt{L}\, e]\!]\rho \triangleq \begin{cases} [\lambda a.\, t]_\alpha & \text{if } [\![e]\!]\rho = \langle a \rangle [t]_\alpha \\ \bot & \text{otherwise} \end{cases}$$

$$[\![\begin{array}{l}\mathtt{case}\, e\ \mathtt{of}\ (\mathtt{V}\, x_1 \to e_1\\ \mid \mathtt{A}\, x_2\, x_2' \to e_2 \mid \mathtt{L}\, x_3 \to e_3)\end{array}]\!]\rho \triangleq \begin{cases} [\![e_1]\!]\rho[x_1 \mapsto a] & \text{if } [\![e]\!]\rho = [a]_\alpha \\ [\![e_2]\!]\rho[x_2 \mapsto [t]_\alpha, x_2' \mapsto [t']_\alpha] & \text{if } [\![e]\!]\rho = [t\ t']_\alpha \\ [\![e_3]\!]\rho[x_3 \mapsto \langle a \rangle [t]_\alpha] & \text{if } [\![e]\!]\rho = [\lambda a.\, t]_\alpha \\ \bot & \text{otherwise} \end{cases}$$

$$[\![\alpha a.\, e]\!]\rho \triangleq \langle a \rangle([\![e]\!]\rho) \quad \text{if } a \# \rho \qquad [\![e_1\, @\, e_2]\!]\rho \triangleq ([\![e_1]\!]\rho) @^t ([\![e_2]\!]\rho)$$

Figure 4.11: Denotations of PNA expressions

uniform-continuous name restriction operation that each $[\![\tau]\!]$ has by virtue of Theorem 4.3.1. The clause for concretion $e_1 @ e_2$ also uses this restriction operation through the total concretion operation from Theorem 3.6.10.

Notation 4.3.2 (**empty typing environment**). Consider the empty typing environment $\emptyset$. Its denotation $[\![\emptyset]\!] = \{\bot\}$ contains a unique $\emptyset$-valuation, $\rho_0 = \{\}$. Given a variable-closed expression $\emptyset \vdash e : \tau$, we simply write $[\![e]\!]$ for $[\![e]\!]\rho_0$.

We can characterise the denotational semantics of PNA more abstractly, as the next Lemma shows. In the abstract formulation it is much easier to prove basic well-definedness properties, such as Proposition 4.3.4, Lemma 4.3.5 and Lemma 4.3.7.

Lemma 4.3.3 (**Characterisation Lemma for** PNA)**.** *The denotational semantics can be characterised more abstractly.*

- $[\![x]\!] = proj_i$, *where* $proj_i \in \mathbf{Nsd}([\![\Gamma]\!], [\![\tau]\!])$ *is the i-th projection function from Remark 3.4.12 and* $x : \tau$ *is in the i-th position of* $\Gamma = \{x_1 : \tau_1, \ldots, x_n : \tau_n\}$.
- $[\![\mathtt{T}]\!] = const_{\text{true}}$, *where the constant function const is defined in Lemma 3.2.7.*
- $[\![\mathtt{F}]\!] = const_{\text{false}}$.
- $[\![\mathtt{if}\, e_1\, \mathtt{then}\, e_2\, \mathtt{else}\, e_3]\!] = \text{if}_{[\![\tau]\!]} \circ \langle [\![e_1]\!], [\![e_2]\!], [\![e_3]\!] \rangle$, *where* if_D *is defined in Lemma 3.4.14, the tupling of functions is defined in* (2.8) *(see also Remark 3.4.12) and* $\circ$ *is the usual function composition.*
- $[\![\mathtt{0}]\!] = const_0$.
- $[\![\mathtt{S}\, e]\!] = succ_\bot \circ [\![e]\!]$, *where succ is defined in* (2.13) *and the construction* $f_\bot$ *is introduced in Lemma 3.4.13.*
- $[\![\mathtt{pred}\, e]\!] = pred_\bot \circ [\![e]\!]$, *where pred is defined in* (2.14).
- $[\![\mathtt{zero}\, e]\!] = zero_\bot \circ [\![e]\!]$, *where zero is defined in* (2.15).
- $[\![(e_1 , e_2)]\!] = \langle [\![e_1]\!], [\![e_2]\!] \rangle$.
- $[\![\mathtt{fst}\, e]\!] = proj_1 \circ [\![e]\!]$, *where* $proj_1$ *is the first projection function from* (2.9) *and Proposition 3.4.11.*
- $[\![\mathtt{snd}\, e]\!] = proj_2 \circ [\![e]\!]$, *where* $proj_2$ *is the second projection function from* (2.10) *and Proposition 3.4.11.*
- $[\![\lambda x : \tau \to e]\!] = cur([\![e]\!])$, *where the currying function cur is defined in* (3.20).
- $[\![e_1\, e_2]\!] = ev \circ \langle [\![e_1]\!], [\![e_2]\!] \rangle$, *where the evaluation function ev is defined in* (3.19).
- $[\![\mathtt{fix}\, e]\!] = \text{fix} \circ [\![e]\!]$, *where* fix *is defined in Proposition 3.4.30.*
- $[\![a]\!] = const_a$.
- $[\![\nu a . e]\!] = a \backslash [\![e]\!]$, *where the restriction operation is defined in Theorem 4.3.1.*

- $[\![(e_1 \rightleftharpoons e_2)\, e_3]\!] = \mathit{swap}_{[\![\tau]\!]} \circ \langle [\![e_1]\!], [\![e_2]\!], [\![e_3]\!] \rangle$, *where swap is defined in Lemma 3.4.15.*
- $[\![e_1 = e_2]\!] = \mathit{eq} \circ \langle [\![e_1]\!], [\![e_2]\!] \rangle$, *where the equality test function is defined in* (3.23).
- $[\![\mathtt{V}\, e]\!] = \mathit{var}_\perp \circ [\![e]\!]$, *where var is defined in Example 2.3.22.*
- $[\![\mathtt{A}\, e_1\, e_2]\!] = \mathit{app}_\perp \circ \langle [\![e_1]\!], [\![e_2]\!] \rangle$, *where app is defined in Example 2.3.22.*
- $[\![\mathtt{L}\, e]\!] = \mathit{lam}_\perp \circ [\![e]\!]$, *where lam is defined in Example 2.3.22. We can use the construction $f_\perp$ from Lemma 3.4.13 because Lemma 3.6.3 shows that $[\![\delta\ \mathtt{term}]\!]$ is a flat domain.*
- $[\![\mathtt{case}\, e\, \mathtt{of}\, (\mathtt{V}\, x_1 \to e_1 \mid \mathtt{A}\, x_2\, x_2' \to e_2 \mid \mathtt{L}\, x_3 \to e_3)]\!] = \mathit{case}_{[\![\tau]\!]} \circ \langle [\![e]\!], \mathit{cur}([\![e_1]\!]), \mathit{cur}(\mathit{cur}([\![e_2]\!])), \mathit{cur}([\![e_3]\!]) \rangle$, *where case_D is defined in Lemma 3.6.2.*
- $[\![\alpha a.\, e]\!] = a\backslash(\langle \_ \rangle\_ \circ \langle \mathit{const}_a, [\![e]\!] \rangle)$, *where the restriction operation is defined in Theorem 4.3.1 and name abstraction on nominal Scott domains is defined in Theorem 3.6.1.*
- $[\![e_1\ @\ e_2]\!] = @^t \circ \langle [\![e_1]\!], [\![e_2]\!] \rangle$, *where $@^t$ is defined in Theorem 3.6.10.*

Proof. For most of the constructs, the characterisation follows directly from the definitions of the respective functions. The side-condition-free denotational semantics of $\nu a.\, e$ and $\alpha a.\, e$ appeal to the slightly subtle properties of the name restriction operation for exponential domains (Lemma 3.6.7). Note that the side conditions are always satisfiable as we identify expressions up to α-equivalence. These characterisations of $[\![\nu a.\, e]\!]$ and $[\![\alpha a.\, e]\!]$ can already be found in Pitts [43, Figure 4]. □

Proposition 4.3.4 (denotational well-definedness). *The denotation of each well-typed expression $\Gamma \vdash e : \tau$ is an element of the exponential domain $[\![\Gamma]\!] \to_{\mathrm{uc}} [\![\tau]\!]$; in other words, $[\![e]\!]$ is a uniform-continuous function mapping Γ-valuations $\rho \in [\![\Gamma]\!]$ to elements $[\![e]\!]\rho$ of the nominal Scott domain $[\![\tau]\!]$.*

Proof. By rule induction on $\Gamma \vdash e : \tau$, using the more abstract presentation of denotational semantics in the Characterisation Lemma 4.3.3. It relies on the fact that the composition and the tupling of uniform-continuous functions is again uniform-continuous (Lemma 3.2.8 and Proposition 3.4.11). □

The next lemma shows that the denotational semantics is equivariant.

Lemma 4.3.5 (Equivariance Lemma). *For all $\Gamma \vdash e : \tau$ and $\pi \in \mathrm{Perm}(\mathbb{A})$ it holds that*

$$\pi \cdot [\![e]\!] = [\![\pi \cdot e]\!].$$

Proof. First note that every function occurring in the Characterisation Lemma 4.3.3 is equivariant, except const_a and $a\backslash d$ but these satisfy $\pi \cdot \mathit{const}_a = \mathit{const}_{\pi a}$ and $\pi \cdot (a\backslash d) = (\pi a)\backslash \pi \cdot d$. Then the property follows easily with equivariance of composition (Lemma 2.3.7) and function tupling (Lemma 2.3.8) by rule induction on $\Gamma \vdash e : \tau$. □

Corollary 4.3.6 (**support inclusion**). *The Equivariance Lemma implies that for every* $\Gamma \vdash e : \tau$ *we have* $\operatorname{supp}[\![e]\!] \subseteq \operatorname{supp} e$.

Proof. Let $\pi \# e$ be given, then by Lemma 4.3.5 we get $\pi \cdot [\![e]\!] = [\![\pi \cdot e]\!] = [\![e]\!]$. □

We are now in the position to prove one of the fundamental features of the denotational way to give semantics: It is compositional, in the sense that the semantics of every expression only depends on the semantics of its subexpressions.

Lemma 4.3.7 (**compositionality**). *For all well-typed expression* $\Gamma \vdash e : \tau, \Gamma \vdash e' : \tau$ *and contexts* $C : (\Gamma \triangleright \tau) \rightsquigarrow (\Gamma' \triangleright \tau')$ *we know that*

$$[\![e]\!] \sqsubseteq [\![e']\!] \Rightarrow [\![C[e]]\!] \sqsubseteq [\![C[e']]\!].$$

Proof. By rule induction over $C : (\Gamma \triangleright \tau) \rightsquigarrow (\Gamma' \triangleright \tau')$, using the Characterisation Lemma 4.3.3 in which every function involved is monotone. □

What follows are some more technical properties of the denotational semantics, which will be used in Chapter 5. We start by showing that the equivalent of Lemma 4.2.5 holds for denotations: they remain invariant under changes to the valuation that do not involve the free variables of the expression.

Lemma 4.3.8 (**denotational weakening**). *Being explicit about typing in the denotation of expressions as in* (4.3)*, we have that for each* $\Gamma \vdash e : \tau$, $\rho \in [\![\Gamma]\!]$ *and* $\rho' \in [\![\Gamma']\!]$ *it holds that*

$$((\forall x \in \operatorname{fv} e)\, \Gamma x = \Gamma' x \wedge \rho x = \rho' x) \Rightarrow [\![\Gamma \vdash e : \tau]\!]\rho = [\![\Gamma' \vdash e : \tau]\!]\rho'$$

and this has the consequence that valuations can be weakened without changing the denotation

$$[\![\Gamma \vdash e : \tau]\!]\rho = [\![\Gamma', \Gamma \vdash e : \tau]\!](\rho'\, \rho).$$

Proof. By rule induction on $\Gamma \vdash e : \tau$. □

Lemma 4.3.9 (**Denotational Substitution Lemma**). *If* $\Gamma \vdash e : \tau$ *and* $\Gamma, x : \tau \vdash e' : \tau'$*, then for all* $\rho \in [\![\Gamma]\!]$ *we have*

$$[\![e'[e/x]]\!]\rho = [\![e']\!]\rho[x \mapsto [\![e]\!]\rho].$$

Proof. By Lemma 4.2.6 we know that $\Gamma \vdash e[e'/x] : \tau$ holds, so $[\![e'[e/x]]\!]\rho$ is well-defined. The property then follows by rule induction on $\Gamma, x : \tau \vdash e' : \tau'$. In several cases Lemma 4.3.8 is used. □

Lemma 4.3.10 (**non-bottom denotation**). *The denotation of every canonical form* $c \in \mathrm{Can}^{\mathsf{PNA}}(\gamma')$ *of extended ground type* $\gamma' \in \mathrm{Grnd}^{\mathsf{PNA}}$ *is non-bottom:* $[\![c]\!] \neq \bot$.

Proof. By structural induction on c. □

Lemma 4.3.11 (**support of ground canonical forms**). *For every canonical form* $c \in \mathrm{Can}^{\mathsf{PNA}}(\gamma')$ *of extended ground type* $\gamma' \in \mathrm{Grnd}^{\mathsf{PNA}}$ *it holds that* $\operatorname{fn} c = \operatorname{supp}[\![c]\!]$.

Proof. The proof follows directly by Lemma 4.2.8 for `bool`, `nat` and `name`. For `term` and $\delta\, \gamma'$ it follows by induction on the structure of c. □

$$\frac{c \in \{\mathtt{T}, \mathtt{F}, \mathtt{O}\}}{a \backslash\backslash c := c} \qquad \frac{a \backslash\backslash c := c'}{a \backslash\backslash \mathtt{S}\, c := \mathtt{S}\, c'} \qquad \frac{}{a \backslash\backslash (e_1, e_2) := (\nu a.\, e_1, \nu a.\, e_2)}$$

$$\frac{}{a \backslash\backslash \lambda x : \tau \to e := \lambda x : \tau \to \nu a.\, e} \qquad \frac{a \neq a'}{a \backslash\backslash a' := a'} \qquad \frac{a \backslash\backslash c := c'}{a \backslash\backslash \mathtt{V}\, c := \mathtt{V}\, c'}$$

$$\frac{a \backslash\backslash c_1 := c_1' \qquad a \backslash\backslash c_2 := c_2'}{a \backslash\backslash \mathtt{A}\, c_1\, c_2 := \mathtt{A}\, c_1'\, c_2'} \qquad \frac{a \backslash\backslash c := c'}{a \backslash\backslash \mathtt{L}\, c := \mathtt{L}\, c'} \qquad \frac{a \backslash\backslash c := c' \qquad a \neq a'}{a \backslash\backslash \alpha a'.\, c := \alpha a'.\, c'}$$

Figure 4.12: PNA operational name restriction on canonical forms

4.4 Operational semantics

This section introduces two different styles of operational semantics for PNA. Intuitively, they both describe how an abstract computer would execute a PNA program. This process of executing a program to get an end result is called *evaluation*. The first style, the *big-step operational semantics*, describes evaluation by a direct relation $e \Downarrow c$ between expressions and canonical forms. It sometimes also called 'natural semantics' and is arguably better at giving an intuitive understanding of evaluation. The second style, the *frame-stack operational semantics*, is more fine-grained and describes evaluation by a sequence of transitions steps $\langle F\,, e\rangle \to \langle F'\,, e'\rangle \to^* \langle \mathrm{Id}\,, c\rangle$. This fine-grainedness gives technical advantages for proving certain properties, in particular the failure of full abstraction results in Section 5.4. Despite the technical differences, Theorem 4.4.26 shows that the two operational semantics essentially describe the same evaluation behaviour.

Both operational semantics rely on an operational version of the name restriction operation from Theorem 4.3.1 that acts on canonical forms.

4.4.1 Operational name restriction

In Sections 4.4.2 and 4.4.3 the evaluation rules for local names make use of an auxiliary definition, which implements the characteristic feature of Odersky's functional theory of local names [36]: scopes intrude in a type-directed fashion. This *operational name restriction* is defined by the rules in Figure 4.12 as a relation $a \backslash\backslash c := c'$ between a name $a \in \mathbb{A}$ and canonical forms $c, c' \in \mathrm{Can}^{\mathsf{PNA}}$. The relation is functional, as Lemma 4.4.2 shows. However, it is not a total relation: there is no c such that $a \backslash\backslash a := c$. Thus, unlike Pitts [43], we choose to follow Odersky [36] and make $\nu a.\, a$ a stuck expression that does not evaluate to any canonical form (and whose denotation is $\bot$). This has the advantage that there are no 'exotic' canonical forms of type `term` in PNA: the only canonical forms of that type correspond to α-equivalence classes of λ-terms.

Remark 4.4.1 (**generative names**). The use of Odersky-style local names means that the operational semantics of PNA is stateless, unlike the operational semantics of the

more usual, *generative* version of $\nu a.\_$ used in the ν-calculus [46], FreshML [57] and most practical languages. Although being unusual, Odersky-style local names are known to be as expressive as generative ones, at least in the simply typed setting. This follows from the existence of an adequate continuation-passing style translation from the ν-calculus to the $\lambda\nu$-calculus [25]. Indeed here we do not escape the subtle properties of generative names modulo contextual equivalence, but encounter them higher up the type hierarchy – see for instance (5.11) in Example 5.1.4 below.

What follows are some technical lemmas about name restriction that will be useful in the following sections. We start by showing that operational name restriction is functional and equivariant.

Lemma 4.4.2 (**functionality of restriction**). *If* $a \backslash\backslash c := c_1$ *and* $a \backslash\backslash c := c_2$ *then* $c_1 = c_2$.

Proof. By rule-induction over $a \backslash\backslash c := c'$. □

Lemma 4.4.3 (**equivariance of restriction**). *For all canonical forms* $c, c' \in \mathrm{Can}^{\mathsf{PNA}}$ *and permutations* $\pi \in \mathrm{Perm}(\mathbb{A})$ *we have*

$$a \backslash\backslash c := c' \;\Rightarrow\; (\pi\, a) \backslash\backslash \pi \cdot c := \pi \cdot c' .$$

Proof. Follows by Pitts [44, Theorem 7.3] through the fact that the set of rules defining $a \backslash\backslash c := c'$ is equivariant. □

The next lemma shows that the restricted name is removed from the free names of the canonical form.

Lemma 4.4.4 (**name removal**). *For all canonical forms* $c, c' \in \mathrm{Can}^{\mathsf{PNA}}$ *and names* $a \in \mathbb{A}$ *it holds that*

$$a \backslash\backslash c := c' \;\Rightarrow\; \mathrm{fn}\, c' = \mathrm{fn}\, c - \{a\} .$$

Proof. By rule induction on $a \backslash\backslash c := c'$. □

Operational name restriction semantically acts like a binder, in the sense that its result is independent from the choice of restricted name.

Lemma 4.4.5 (**restriction binder**). *For all types* τ, *canonical forms* $c, c' \in \mathrm{Can}^{\mathsf{PNA}}$ *and names* $a, a' \in \mathbb{A}$ *it holds that*

$$(a' \notin \mathrm{fn}\, c \;\wedge\; a \backslash\backslash c := c') \;\Rightarrow\; a' \backslash\backslash (a\; a') \cdot c := c'$$

Proof. By rule induction on $a \backslash\backslash c := c'$. The cases for products and functions use that $\nu a.\_$ is a binding form and therefore $a' \notin (\mathrm{fn}\, e - \{a\}) \Rightarrow \nu a.\, e = \nu a'.\, (a\; a') \cdot e$. □

The following type preservation property holds for operational name restriction.

Lemma 4.4.6 (**type preservation for restriction**). *If* $a \backslash\backslash c := c'$ *and* $c \in \mathrm{Can}^{\mathsf{PNA}}(\tau)$ *then* $c' \in \mathrm{Can}^{\mathsf{PNA}}(\tau)$ *follows.*

Proof. By rule-induction over $a \backslash\backslash c := c'$. □

At (extended) ground types, operational name restriction is particularly simple. It is defined only on fresh names, and when it is defined it is the identity.

Lemma 4.4.7 (**ground restriction**). *For every canonical form of extended ground type* $c \in \mathrm{Can}^{\mathsf{PNA}}(\gamma')$ *with* $\gamma' \in \mathrm{Grnd}^{\mathsf{PNA}}$*, the following three statements are equivalent:*

1. $a \notin \mathrm{fn}\, c$
2. $(\exists c')\, a \backslash\backslash c := c'$
3. $a \backslash\backslash c := c$.

Proof. For `name`, `bool` and `nat` the statement follows directly from the definition of $a \backslash\backslash c := c'$. For `term` and $\delta\, \gamma'$ it can be proved by structural induction on c. □

The next lemma connects our operational name restriction with the uniform-continuous name restriction operation that the denotation of every PNA-type possesses by virtue of Theorem 4.3.1. This is crucial for correctness of the denotational semantics of our local scoping construct $\nu a.e$.

Lemma 4.4.8 (**Restriction Lemma**). *If* $c, c' \in \mathrm{Can}^{\mathsf{PNA}}(\tau)$ *are variable-closed canonical forms of type* τ*, then*

$$a \backslash\backslash c := c' \;\Rightarrow\; a \backslash [\![c]\!] = [\![c']\!]$$

holds, where $\backslash$ *is the uniform-continuous name restriction operation from Theorem 4.3.1.*

Proof. Follows by rule induction over $a \backslash\backslash c := c'$, where for λ-abstractions we need the fact that (3.30) uniquely determines the restriction operation on functions defined by (3.31). We also use that (3.32) uniquely determines the restriction operation for name abstractions. □

The operational name restriction operation preserves substitution of fresh functions when dealing with non-variable expressions. This less standard result will be used in Appendix A.5.

Lemma 4.4.9 (**fresh function substitution**). *It holds that*

$$\begin{aligned} &e \notin \mathbb{V} \wedge e', e'' \in \mathrm{Exp}^{\mathsf{PNA}}(\tau_1 \to \tau_2) \wedge a \,\#\, e', e'' \wedge e[e'/x] \in \mathrm{Can}^{\mathsf{PNA}}(\tau) \wedge a \backslash\backslash e[e'/x] := c \\ &\Rightarrow (\exists e_1 \in \mathrm{Exp}^{\mathsf{PNA}})\, c = e_1[e'/x] \wedge \mathrm{fn}\, e_1 = \mathrm{fn}\, e - \{a\} \wedge a \backslash\backslash e[e''/x] := e_1[e''/x]. \end{aligned}$$

Proof. By induction on τ and an analysis of $e[e'/x] \in \mathrm{Can}^{\mathsf{PNA}}(\tau)$ with Lemma 4.2.8. We apply Lemma 4.4.7 at ground types. □

$$\frac{c \in \{\mathtt{T}, \mathtt{F}, \mathtt{O}, (e_1, e_2), \lambda x : \tau \to e\}}{c \Downarrow c} \qquad \frac{e_1 \Downarrow \mathtt{T} \qquad e_2 \Downarrow c}{\mathtt{if}\, e_1\, \mathtt{then}\, e_2\, \mathtt{else}\, e_3 \Downarrow c}$$

$$\frac{e_1 \Downarrow \mathtt{F} \qquad e_3 \Downarrow c}{\mathtt{if}\, e_1\, \mathtt{then}\, e_2\, \mathtt{else}\, e_3 \Downarrow c} \qquad \frac{e \Downarrow c}{\mathtt{S}\, e \Downarrow \mathtt{S}\, c} \qquad \frac{e \Downarrow \mathtt{S}\, c}{\mathtt{pred}\, e \Downarrow c} \qquad \frac{e \Downarrow \mathtt{O}}{\mathtt{zero}\, e \Downarrow \mathtt{T}}$$

$$\frac{e \Downarrow \mathtt{S}\, c}{\mathtt{zero}\, e \Downarrow \mathtt{F}} \qquad \frac{e \Downarrow (e_1, e_2) \qquad e_1 \Downarrow c}{\mathtt{fst}\, e \Downarrow c} \qquad \frac{e \Downarrow (e_1, e_2) \qquad e_2 \Downarrow c}{\mathtt{snd}\, e \Downarrow c}$$

$$\frac{e_1 \Downarrow \lambda x : \tau \to e \qquad e[e_2/x] \Downarrow c}{e_1\, e_2 \Downarrow c} \qquad \frac{e\,(\mathtt{fix}\, e) \Downarrow c}{\mathtt{fix}\, e \Downarrow c}$$

$$\frac{a \in \mathbb{A}}{a \Downarrow a} \qquad \frac{e \Downarrow c \qquad a \backslash\backslash c := c'}{\nu a.\, e \Downarrow c'} \qquad \frac{e_1 \Downarrow a_1 \qquad e_2 \Downarrow a_2 \qquad e_3 \Downarrow c}{(e_1 \rightleftharpoons e_2)\, e_3 \Downarrow (a_1\, a_2) \cdot c}$$

$$\frac{e_1 \Downarrow a \qquad e_2 \Downarrow a}{e_1 = e_2 \Downarrow \mathtt{T}} \qquad \frac{e_1 \Downarrow a \qquad e_2 \Downarrow a' \qquad a \neq a'}{e_1 = e_2 \Downarrow \mathtt{F}} \qquad \frac{e \Downarrow c}{\mathtt{V}\, e \Downarrow \mathtt{V}\, c}$$

$$\frac{e_1 \Downarrow c_1 \qquad e_2 \Downarrow c_2}{\mathtt{A}\, e_1\, e_2 \Downarrow \mathtt{A}\, c_1\, c_2} \qquad \frac{e \Downarrow c}{\mathtt{L}\, e \Downarrow \mathtt{L}\, c} \qquad \frac{e \Downarrow \mathtt{V}\, c \qquad e_1[c/x_1] \Downarrow c'}{\mathtt{case}\, e\, \mathtt{of}\, (\mathtt{V}\, x_1 \to e_1 \mid \cdots) \Downarrow c'}$$

$$\frac{e \Downarrow \mathtt{A}\, c\, c' \qquad e_2[c/x_2, c'/x_2'] \Downarrow c''}{\mathtt{case}\, e\, \mathtt{of}\, (\cdots \mid \mathtt{A}\, x_2\, x_2' \to e_2 \mid \cdots) \Downarrow c''} \qquad \frac{e \Downarrow \mathtt{L}\, c \qquad e_3[c/x_3] \Downarrow c'}{\mathtt{case}\, e\, \mathtt{of}\, (\cdots \mid \mathtt{L}\, x_3 \to e_3) \Downarrow c'}$$

$$\frac{e \Downarrow c}{\alpha a.\, e \Downarrow \alpha a.\, c} \qquad \frac{e_1 \Downarrow \alpha a.\, c \qquad e_2 \Downarrow a' \qquad a \neq a' \qquad a \backslash\backslash (a\, a') \cdot c := c'}{e_1\, @\, e_2 \Downarrow c'}$$

Figure 4.13: PNA big-step evaluation rules

4.4.2 Big-step evaluation

The PNA big-step *evaluation relation* is of the form

$$e \Downarrow c$$

and it describes when an expression $e \in \mathrm{Exp}^{\mathsf{PNA}}$ evaluates to a canonical form $c \in \mathrm{Can}^{\mathsf{PNA}}$. In Figure 4.13 we extend PCF's usual evaluation rules with the rules below the dotted line that concern atomic names. As for PCF, we only evaluate expressions that are variable-closed in the sense that $\mathrm{fv}\, e = \emptyset$. However, expressions for evaluation may contain free atomic names; this is because, unlike variables, atomic names are canonical forms.

To deconstruct name abstractions, PNA features an operational version of the total concretion operation discussed in Theorem 3.6.10. The evaluation rules for local names and for concretion make use of the operational name restriction from Figure 4.12.

We also choose to evaluate under name abstractions, so that $\alpha a.\, e$ is in canonical form if and only if e is. This permits a representation of λ-terms (see Example 2.3.22) in PNA that is as simple as PCF's representation of numbers: they are in bijection with variable-closed canonical forms of type $\mathtt{term}$.[2] In this sense, the representation of λ-terms in PNA is 'junk-free', as defined in Sheard [53, Section 13].

In the following, we establish some basic properties of the evaluation relation, leading to the fact that evaluation is sound with respect to the denotational semantics, Proposition 4.4.16.

Lemma 4.4.10 (equivariance of evaluation). *For all $\pi \in \mathrm{Perm}(\mathbb{A})$, $e \in \mathrm{Exp}^{\mathsf{PNA}}$ and $c \in \mathrm{Can}^{\mathsf{PNA}}$ it holds that*

$$e \Downarrow c \;\Rightarrow\; \pi \cdot e \Downarrow \pi \cdot c\,.$$

Proof. Follows with Pitts [44, Theorem 7.3] from the fact that the set of rules defining the evaluation relation $e \Downarrow c$ is equivariant. The rules involving substitutions require Lemma 4.1.3, and the rules involving operational name restriction require Lemma 4.4.3. Several rules also use Lemma 2.3.3 and Lemma 2.3.4. □

Lemma 4.4.11 (no name creation). *Evaluation does not create free names:*

$$e \Downarrow c \;\Rightarrow\; \mathrm{fn}\, c \subseteq \mathrm{fn}\, e\,.$$

Proof. By rule induction over $e \Downarrow c$. The local scoping and name abstraction cases use Lemma 4.4.4. □

Lemma 4.4.12 (reflexivity at canonical forms). *Every canonical form $c \in \mathrm{Can}^{\mathsf{PNA}}$ evaluates to itself: $c \Downarrow c$.*

Proof. By induction on the structure of c. □

[2] It is certainly possible to give a different operational semantics in which one does not evaluate under name abstractions. The corresponding denotational semantics would make more use of lifting than does the one in Section 4.3.2.

The big-step evaluation relation is functional, in the sense that if an expression evaluates to a canonical form, then this canonical form is unique.

Lemma 4.4.13 (**uniqueness of evaluation**). *For all* $e \in \mathrm{Exp}^{\mathsf{PNA}}$ *and* $c_1, c_2 \in \mathrm{Can}^{\mathsf{PNA}}$ *we have*

$$e \Downarrow c_1 \wedge e \Downarrow c_2 \Rightarrow c_1 = c_2 .$$

Proof. By rule induction on $e \Downarrow c_1$. The local scoping and name abstraction cases work through Lemma 4.4.2. □

The next lemma will be used in Section 4.4.3 to show that the big-step and the frame-stack evaluation relations (defined there) coincide. It says that when evaluating an instantiated frame-stack, the expression in the hole has to be evaluated too.

Lemma 4.4.14 (**evaluation under a frame-stack**). *For all* $F \in \mathrm{Stack}^{\mathsf{PNA}}$, $e \in \mathrm{Exp}^{\mathsf{PNA}}$ *and* $c \in \mathrm{Can}^{\mathsf{PNA}}$ *it holds that*

$$F[e] \Downarrow c \Leftrightarrow e \Downarrow c' \wedge F[c'] \Downarrow c .$$

Proof. We prove both directions separately by induction on the structure of F. To simplify the argument for the inductive steps we use that every non-empty frame-stack can be written as $(\mathrm{Id} \circ E) \bullet F'$ and Lemma 4.1.8 gives us $((\mathrm{Id} \circ E) \bullet F')[e] = E[F'[e]/\cdot]$. Furthermore Lemma 4.4.12 and Lemma 4.4.13 are applied in the base-cases and in several inductive steps. □

Evaluating an expression preserves its type.

Lemma 4.4.15 (**type preservation**). *Each* $\tau \in \mathrm{Typ}^{\mathsf{PNA}}$, $e \in \mathrm{Exp}^{\mathsf{PNA}}(\tau)$ *and* $c \in \mathrm{Can}^{\mathsf{PNA}}$ *satisfies*

$$e \Downarrow c \Rightarrow c \in \mathrm{Can}^{\mathsf{PNA}}(\tau).$$

Proof. By rule induction over $e \Downarrow c$. For the rules for local and name abstraction we need Lemma 4.4.6. □

The next proposition gives the main result of this section: big-step evaluation is sound with respect to the denotational semantics.

Proposition 4.4.16 (PNA **soundness**). *For all* $e \in \mathrm{Exp}^{\mathsf{PNA}}(\tau)$ *and* $c \in \mathrm{Can}^{\mathsf{PNA}}$ *we have*

$$e \Downarrow c \Rightarrow [\![e]\!] = [\![c]\!] .$$

Proof. By rule induction over $e \Downarrow c$. The cases for function application and λ-term use the Denotational Substitution Lemma 4.3.9, the cases for local scoping and name concretion use the Restriction Lemma 4.4.8, and the cases for name swapping and name concretion use the Equivariance Lemma 4.3.5. □

4.4.3 Frame-stack evaluation

Additionally to the big-step evaluation relation in Section 4.4.2, we define a more fine-grained description of the operational semantics: the *frame-stack evaluation relation*. We follow the style of Felleisen and Hieb [14], but with their 'evaluation contexts' formulated as stacks of evaluation frames. Pitts [39] discusses the usefulness of this semantic style for proving contextual equivalences.

Binding in frame-stack configurations The frame-stack evaluation relation is defined in terms of a transition system between *configurations*. A configuration $\langle F, e\rangle$ is built from a frame-stack $F \in \mathrm{Stack}^{\mathsf{PNA}}$ and an expression $e \in \mathrm{Exp}^{\mathsf{PNA}}$. However, a configuration is not just a pair of a frame-stack and an expression, because we identify configurations by a form of α-equivalence in which a binding form in the frame-stack can bind atomic names in later frames of the frame-stack or in the expression of the configuration. So for example $\langle \mathrm{Id} \circ (\nu a.\cdot) \circ (\cdot = a), a\rangle$ is α-equivalent to $\langle \mathrm{Id} \circ (\nu a'.\cdot) \circ (\cdot = a'), a'\rangle$, but it is not α-equivalent to $\langle \mathrm{Id}, \nu a.\, a = a\rangle$. Definition 4.4.17 gives the formal definition of this kind of α-equivalence.

Definition 4.4.17 (**α-equivalence for configurations**)**.** In this definition we work with an explicit notation of α-equivalence for expressions $e =_\alpha e'$ and frames $E =_\alpha E'$ (as in Example 2.3.22 for the λ-calculus). With that we can formally (and explicitly) define the α-equivalence relation on configurations $\langle F, e\rangle =_\alpha \langle F', e'\rangle$ by the following rules

$$\frac{e =_\alpha e'}{\langle \mathrm{Id}, e\rangle =_\alpha \langle \mathrm{Id}, e'\rangle}$$

$$\frac{\langle \mathrm{Id} \circ E_2 \circ \ldots \circ E_n, e\rangle =_\alpha \langle \mathrm{Id} \circ E_2' \circ \ldots \circ E_n', e'\rangle \qquad E_1 =_\alpha E_1' \qquad E_1 \notin \{\nu a.\cdot, \alpha a.\cdot\}}{\langle \mathrm{Id} \circ E_1 \circ E_2 \circ \ldots \circ E_n, e\rangle =_\alpha \langle \mathrm{Id} \circ E_1' \circ E_2' \circ \ldots \circ E_n', e'\rangle}$$

$$\frac{\begin{array}{c}(a\ b)\cdot\langle \mathrm{Id} \circ E_2 \circ \ldots \circ E_n, e\rangle =_\alpha (a'\ b)\cdot\langle \mathrm{Id} \circ E_2' \circ \ldots \circ E_n', e'\rangle \\ b \notin \mathit{name}(E_2, \ldots, E_n, e, E_2', \ldots, E_n', e')\end{array}}{\langle \mathrm{Id} \circ (\nu a.\cdot) \circ E_2 \circ \ldots \circ E_n, e\rangle =_\alpha \langle \mathrm{Id} \circ (\nu a'.\cdot) \circ E_2' \circ \ldots \circ E_n', e'\rangle}$$

$$\frac{\begin{array}{c}(a\ b)\cdot\langle \mathrm{Id} \circ E_2 \circ \ldots \circ E_n, e\rangle =_\alpha (a'\ b)\cdot\langle \mathrm{Id} \circ E_2' \circ \ldots \circ E_n', e'\rangle \\ b \notin \mathit{name}(E_2, \ldots, E_n, e, E_2', \ldots, E_n', e')\end{array}}{\langle \mathrm{Id} \circ (\alpha a.\cdot) \circ E_2 \circ \ldots \circ E_n, e\rangle =_\alpha \langle \mathrm{Id} \circ (\alpha a'.\cdot) \circ E_2' \circ \ldots \circ E_n', e'\rangle}$$

where the function *name* just returns all names occurring in its arguments (no matter if they are free, not free or binders). The α-equivalence relation for frames $E =_\alpha E'$ that is used in the second rule is defined in the obvious way by recursion into all subexpressions. We give some exemplary rules:

$$\frac{}{\mathtt{S}\cdot =_\alpha \mathtt{S}\cdot} \qquad \frac{e_2 =_\alpha e_2' \qquad e_3 =_\alpha e_3'}{\mathtt{if}\cdot\mathtt{then}\ e_2\ \mathtt{else}\ e_3 =_\alpha \mathtt{if}\cdot\mathtt{then}\ e_2'\ \mathtt{else}\ e_3'} \qquad \frac{a_1 = a_1' \qquad e_3 =_\alpha e_3'}{(a_1 \rightleftharpoons \cdot)\, e_3 =_\alpha (a_1' \rightleftharpoons \cdot)\, e_3'}.$$

There is a characterisation of α-equivalence for configurations via α-equivalence for expressions.

Lemma 4.4.18 (**α-equivalence characterisation**). *Recall the definitions of $|F|$ and $F[e]$ from Section 4.1.4. We have that the following property holds:*

$$\langle F\,, e\rangle =_\alpha \langle F'\,, e'\rangle \Leftrightarrow |F| = |F'| \wedge F[e] =_\alpha F'[e'].$$

Proof. The left-to-right direction is an induction over the rules defining $\langle F\,, e\rangle =_\alpha \langle F'\,, e'\rangle$ and the right-to-left direction is an induction on the structure of F, using the definition of $=_\alpha$ for expressions and the definition of hole filling $F[e]$. □

The α-equivalence relation for configurations in Definition 4.4.17 is given explicitly. From now on we revoke this explicit notation and start to implicitly identify configurations by α-equivalence, as it is done for expressions. Let $\mathrm{Config}^{\mathsf{PNA}}$ be the set of all such configurations for PNA. The notions of permutation action, free names, free variables and capture-avoiding substitution extend to configurations in the obvious way, taking α-equivalence into account.

Remark 4.4.19 (**configuration identification**). In most frame-stack operational semantics the identification of configurations (and frame-stacks) by α-equivalence as in Definition 4.4.17 is unnecessary, because usually no frame involves a binding form that scopes over a hole. In PNA however, we have frames like $\nu a.\cdot$ and so we need to take binding over a hole into account. If we did not identify configurations by α-equivalence, then for example the $\langle F, \nu a.\, e\rangle$-rule would introduce non-deterministic transitions (and thereby Lemma 4.4.22 would fail), which is unwanted here.

Frame-stack transition system Figure 4.14 defines the frame-stack evaluation relation

$$\langle F\,, e\rangle \to \langle F'\,, e'\rangle\,.$$

as a transition system between configurations. It is only defined for variable-closed configurations $\langle F\,, e\rangle, \langle F'\,, e'\rangle \in \mathrm{Config}^{\mathsf{PNA}}$. In the big-step semantics, evaluation of a variable-closed expression $e \in \mathrm{Exp}^{\mathsf{PNA}}$ to a canonical form $c \in \mathrm{Can}^{\mathsf{PNA}}$ is defined in one (big) step $e \Downarrow c$, whereas in the frame stack semantics evaluation takes many transition steps $\langle \mathrm{Id}\,, e\rangle \to \langle F'\,, e'\rangle \to \cdots \to \langle \mathrm{Id}\,, c\rangle$. This fine-grainedness is useful for many proofs (in our case the failure of full abstraction proofs in Section 5.4), which is why we introduce the frame-stack evaluation in the first place.

Definition 4.4.20 (**many steps relations**). For each $n \in \mathbb{N}$, let the relation $\to^n$ be defined by

$$\begin{aligned}\langle F\,, e\rangle \to^0 \langle F'\,, e'\rangle &\triangleq \langle F\,, e\rangle = \langle F'\,, e'\rangle\\ \langle F\,, e\rangle \to^{n+1} \langle F'\,, e'\rangle &\triangleq \langle F\,, e\rangle \to \langle F''\,, e''\rangle \wedge \langle F''\,, e''\rangle \to^n \langle F'\,, e'\rangle\end{aligned}$$

and $\langle F\,, e\rangle \to^* \langle F'\,, e'\rangle$ is defined to hold if and only if $(\exists n \in \mathbb{N})\ \langle F\,, e\rangle \to^n \langle F'\,, e'\rangle$.

$$\langle F\,,\mathtt{if}\,e_1\,\mathsf{then}\,e_2\,\mathsf{else}\,e_3\rangle \to \langle F\circ \mathtt{if}\cdot\mathsf{then}\,e_2\,\mathsf{else}\,e_3\,,e_1\rangle$$

$$\langle F\circ \mathtt{if}\cdot\mathsf{then}\,e_2\,\mathsf{else}\,e_3\,,\mathtt{T}\rangle \to \langle F\,,e_2\rangle \qquad \langle F\circ \mathtt{if}\cdot\mathsf{then}\,e_2\,\mathsf{else}\,e_3\,,\mathtt{F}\rangle \to \langle F\,,e_3\rangle$$

$$\langle F\,,\mathtt{S}\,e\rangle \to \langle F\circ\mathtt{S}\cdot\,,e\rangle \qquad \langle F\circ\mathtt{S}\cdot\,,c\rangle \to \langle F\,,\mathtt{S}\,c\rangle \qquad \langle F\,,\mathtt{pred}\,e\rangle \to \langle F\circ\mathtt{pred}\cdot\,,e\rangle$$

$$\langle F\circ\mathtt{pred}\cdot,\mathtt{S}\,c\rangle \to \langle F\,,c\rangle \qquad \langle F\,,\mathtt{zero}\,e\rangle \to \langle F\circ\mathtt{zero}\cdot,e\rangle \qquad \langle F\circ\mathtt{zero}\cdot,\mathtt{0}\rangle \to \langle F\,,\mathtt{T}\rangle$$

$$\langle F\circ\mathtt{zero}\cdot,\mathtt{S}\,c\rangle \to \langle F,\mathtt{F}\rangle \qquad \langle F,\mathtt{fst}\,e\rangle \to \langle F\circ\mathtt{fst}\cdot,e\rangle \qquad \langle F\circ\mathtt{fst}\cdot,(e_1,e_2)\rangle \to \langle F,e_1\rangle$$

$$\langle F,\mathtt{snd}\,e\rangle \to \langle F\circ\mathtt{snd}\cdot,e\rangle \qquad \langle F\circ\mathtt{snd}\cdot,(e_1,e_2)\rangle \to \langle F,e_2\rangle \qquad \langle F,e_1\,e_2\rangle \to \langle F\circ\cdot\,e_2,e_1\rangle$$

$$\langle F\circ\cdot\,e_2\,,\lambda x:\tau\to e\rangle \to \langle F\,,e[e_2/x]\rangle \qquad \langle F\,,\mathtt{fix}\,e\rangle \to \langle F\,,e\,(\mathtt{fix}\,e)\rangle$$

....

$$\langle F\,,\nu a.e\rangle \to \langle F\circ\nu a.\cdot\,,e\rangle \qquad \langle F\circ\nu a.\cdot\,,c\rangle \to \langle F\,,c'\rangle \quad \text{if } a\backslash\!\!\backslash c := c'$$

$$\langle F\,,(e_1 \rightleftharpoons e_2)\,e_3\rangle \to \langle F\circ(\cdot\rightleftharpoons e_2)\,e_3\,,e_1\rangle \qquad \langle F\circ(\cdot\rightleftharpoons e_2)\,e_3\,,a_1\rangle \to \langle F\circ(a_1\rightleftharpoons\cdot)\,e_3\,,e_2\rangle$$

$$\langle F\circ(a_1\rightleftharpoons\cdot)\,e_3\,,a_2\rangle \to \langle F\circ(a_1\rightleftharpoons a_2)\cdot\,,e_3\rangle \qquad \langle F\circ(a_1\rightleftharpoons a_2)\cdot\,,c\rangle \to \langle F\,,(a_1\,a_2)\cdot c\rangle$$

$$\langle F\,,e_1 = e_2\rangle \to \langle F\circ\cdot = e_2\,,e_1\rangle \qquad \langle F\circ\cdot = e_2\,,a_1\rangle \to \langle F\circ a_1 = \cdot\,,e_2\rangle$$

$$\langle F\circ a_1 = \cdot\,,a_2\rangle \to \langle F\,,\mathtt{T}\rangle \quad \text{if } a_1 = a_2 \qquad \langle F\circ a_1 = \cdot\,,a_2\rangle \to \langle F\,,\mathtt{F}\rangle \quad \text{if } a_1 \neq a_2$$

$$\langle F\,,\mathtt{V}\,e\rangle \to \langle F\circ\mathtt{V}\cdot\,,e\rangle \qquad \langle F\circ\mathtt{V}\cdot\,,c\rangle \to \langle F\,,\mathtt{V}\,c\rangle \qquad \langle F\,,\mathtt{A}\,e_1\,e_2\rangle \to \langle F\circ\mathtt{A}\cdot e_2\,,e_1\rangle$$

$$\langle F\circ\mathtt{A}\cdot e_2,c_1\rangle \to \langle F\circ\mathtt{A}\,c_1\cdot,e_2\rangle \qquad \langle F\circ\mathtt{A}\,c_1\cdot,c_2\rangle \to \langle F,\mathtt{A}\,c_1\,c_2\rangle \qquad \langle F,\mathtt{L}\,e\rangle \to \langle F\circ\mathtt{L}\cdot,e\rangle$$

$$\langle F\circ\mathtt{L}\cdot\,,c\rangle \to \langle F\,,\mathtt{L}\,c\rangle \qquad \langle F\,,\mathsf{case}\,e\,\mathsf{of}\,(\cdots)\rangle \to \langle F\circ\mathsf{case}\cdot\mathsf{of}\,(\cdots)\,,e\rangle$$

$$\langle F\circ\mathsf{case}\cdot\mathsf{of}\,(\mathtt{V}\,x_1\to e_1\mid\cdots)\,,\mathtt{V}\,c\rangle \to \langle F\,,e_1[c/x_1]\rangle$$

$$\langle F\circ\mathsf{case}\cdot\mathsf{of}\,(\cdots\mid\mathtt{A}\,x_2\,x_2'\to e_2\mid\cdots)\,,\mathtt{A}\,c\,c'\rangle \to \langle F\,,e_2[c/x_2,c'/x_2']\rangle$$

$$\langle F\circ\mathsf{case}\cdot\mathsf{of}\,(\cdots\mid\mathtt{L}\,x_3\to e_3)\,,\mathtt{L}\,c\rangle \to \langle F\,,e_3[c/x_1]\rangle \qquad \langle F\,,\alpha a.e\rangle \to \langle F\circ\alpha a.\cdot\,,e\rangle$$

$$\langle F\circ\alpha a.\cdot,c\rangle \to \langle F,\alpha a.c\rangle \qquad \langle F,e_1\,\mathtt{@}\,e_2\rangle \to \langle F\circ\cdot\mathtt{@}\,e_2,e_1\rangle \qquad \langle F\circ\cdot\mathtt{@}\,e_2,c_1\rangle \to \langle F\circ c_1\mathtt{@}\cdot,e_2\rangle$$

$$\langle F\circ\alpha a.c\,\mathtt{@}\cdot\,,a'\rangle \to \langle F\,,c'\rangle \quad \text{if } a\neq a' \text{ and } a\backslash\!\!\backslash(a\,a')\cdot c := c'$$

Figure 4.14: PNA frame-stack evaluation rules

What follows are some technical properties of the frame-stack evaluation. They lead to Theorem 4.4.26, which says that big-step evaluation and frame-stack evaluation are essentially descriptions of the same operational semantics. The first property we prove is that frame-stack transitions remain valid under concatenating another frame-stack.

Lemma 4.4.21 (**frame-stack weakening**). *It holds that*

$$\langle F\,,e\rangle \to \langle F'\,,e'\rangle \;\Rightarrow\; (\forall F'' \in \mathrm{Stack}^{\mathsf{PNA}})\,\langle F''\bullet F\,,e\rangle \to \langle F''\bullet F'\,,e'\rangle\,.$$

Proof. By a simple case distinction on $\langle F\,,e\rangle \to \langle F'\,,e'\rangle$. □

The next lemma shows that the transition system of the frame-stack evaluation relation is deterministic.

Lemma 4.4.22 (**frame-stack determinacy**). *The following property holds*

$$\langle F\,,e\rangle \to \langle F_1\,,e_1\rangle \wedge \langle F\,,e\rangle \to \langle F_2\,,e_2\rangle \;\Rightarrow\; \langle F_1\,,e_1\rangle = \langle F_2\,,e_2\rangle\,.$$

Proof. By case distinction on $\langle F\,,e\rangle \to \langle F_1\,,e_1\rangle$. The cases for $\langle F\,,\nu a.\,e\rangle$ and $\langle F\,,\alpha a.\,e\rangle$ crucially depend on the identification of configurations by α-equivalence as described in Definition 4.4.17. Lemma 4.4.2 is needed for the cases for $\langle F\circ \nu a.\,\cdot\,,c\rangle$ and $\langle F\circ \alpha a.\,c\;@\,\cdot\,,a'\rangle$. □

Frame-stack evaluation always needs to consider the expression first.

Lemma 4.4.23 (**frame-stack evaluation order**). *For all $n\in\mathbb{N}$ we have that $\langle F\,,e\rangle \to^n \langle \mathrm{Id}\,,c\rangle$ implies that there exist $c'\in\mathrm{Can}^{\mathsf{PNA}}$ and $m\in\mathbb{N}$ with $m\le n$, such that $\langle \mathrm{Id}\,,e\rangle \to^m \langle \mathrm{Id}\,,c'\rangle$ and $\langle F\,,c'\rangle \to^{n-m} \langle \mathrm{Id}\,,c\rangle$ hold.*

Proof. The proof works by induction on n and a case distinction on the first transition in the inductive step. For many cases in the we use that $\langle \mathrm{Id}\,,e\rangle \to^m \langle \mathrm{Id}\,,c\rangle \Rightarrow \langle F\,,e\rangle \to^m \langle F\,,c\rangle$ for any $m\in\mathbb{N}$ and $F\in\mathrm{Stack}^{\mathsf{PNA}}$, which is a direct consequence of Lemma 4.4.21. □

We continue with two technical lemmas that connect the frame-stack and the big-step evaluation relation. They lead up to Theorem 4.4.26.

Lemma 4.4.24 (**big-step implies frame-stack**). *The property*

$$e \Downarrow c \;\Rightarrow\; (\forall F\in\mathrm{Stack}^{\mathsf{PNA}})\,\langle F\,,e\rangle \to^* \langle F\,,c\rangle$$

holds for all $e\in\mathrm{Exp}^{\mathsf{PNA}}$ and $c\in\mathrm{Can}^{\mathsf{PNA}}$.

Proof. By rule induction on $e\Downarrow c$. □

Lemma 4.4.25 (**frame-stack preserves big-step**). *It holds that*

$$\langle F\,,e\rangle \to \langle F'\,,e'\rangle \wedge F'[e']\Downarrow c \;\Rightarrow\; F[e]\Downarrow c\,.$$

Proof. By case analysis on $\langle F\,,\,e\rangle \to \langle F'\,,\,e'\rangle$. Lemma 4.4.14 is crucial for several cases. □

The main result of this section is that frame-stack and big-step operational semantics coincide.

Theorem 4.4.26 (evaluation relations coincide). *For all* $e \in \mathrm{Exp}^{\mathsf{PNA}}$ *and* $c \in \mathrm{Can}^{\mathsf{PNA}}$ *we have*

$$e \Downarrow c \iff \langle \mathrm{Id}\,,\,e\rangle \to^* \langle \mathrm{Id}\,,\,c\rangle\,.$$

Proof. The left-to-right direction is just Lemma 4.4.24. For the right-to-left direction we prove the more general statement $(\forall n \in \mathbb{N})\langle F\,,\,e\rangle \to^n \langle \mathrm{Id}\,,\,c\rangle \Rightarrow F[e] \Downarrow c$ by induction on n, where we use Lemma 4.4.12 for the base case and Lemma 4.4.25 for the inductive step. □

Theorem 4.4.26 ensures that frame-stack and big-step operational semantics can be used interchangeably. For example, Corollary 5.4.3 is concerned with a property of the big-step evaluation relation, but we use the frame-stack evaluation relation in its proof.

4.5 Programming with PNA

After having investigated the syntax and semantics of PNA, we will now see PNA in use. In Section 4.5.1 we introduce some syntactic sugar that makes PNA programs easier to write. Section 4.5.2 gives a few example programs and their evaluation behaviour. These examples showcase the use of PNA for metaprogramming.

4.5.1 Syntactic sugar

PNA is designed to have a minimal syntax, such that proofs involving the syntax remain reasonably short. Writing programs in PNA can therefore be relatively tedious. We improve this situation by defining some syntactic sugar for common programming phrases.

Non-termination Sometimes it is convenient to have an expression at hand whose evaluation will never terminate. For any type $\tau \in \mathrm{Typ}^{\mathsf{PNA}}$, such an expression $\mathtt{bot}_\tau \in \mathrm{Exp}^{\mathsf{PNA}}(\tau)$ can be defined for example by

$$\mathtt{bot}_\tau \triangleq \mathtt{fix}(\lambda x : \tau \to x). \tag{4.4}$$

Evaluation of this expression will run forever

$$\langle \mathrm{Id}\,,\mathtt{bot}_\tau\rangle \to^3 \langle \mathrm{Id}\,,\mathtt{bot}_\tau\rangle \to^3 \ldots$$

and its denotation is bottom $[\![\mathtt{bot}_\tau]\!] = \bot$ at each $\tau \in \mathrm{Typ}^{\mathsf{PNA}}$.

Vectors We use vectors as notation for finite tuples as follows:

$$\vec{x} \triangleq (x_1, \ldots, x_n).$$

The exact size of the vector $n \in \mathbb{N}$ is often left implicit. For a set X we write X^n for the set of n-sized vectors of X. Sometimes we confuse vectors with finite sets and write for example $\vec{a} = \mathrm{fn}\, e$ for $e \in \mathrm{Exp}^{\mathsf{PNA}}$, and in these cases any ordering of the vector may be chosen. The swapping of two vectors (of the same length) of expressions $\vec{e}, \vec{e}\,' \in (\mathrm{Exp}^{\mathsf{PNA}})^n$ is defined by

$$(\vec{e} \rightleftharpoons \vec{e}\,') \triangleq (e_1 \rightleftharpoons e_1') \cdots (e_n \rightleftharpoons e_n').$$

Boolean operations The boolean operations of negation, conjunction and disjunction are easily defined by

$$\begin{aligned} \mathtt{not}\ e &\triangleq \mathtt{if}\ e\ \mathtt{then}\ \mathtt{F}\ \mathtt{else}\ \mathtt{T} \\ e\ \mathtt{and}\ e' &\triangleq \mathtt{if}\ e\ \mathtt{then}\ e'\ \mathtt{else}\ \mathtt{F} \\ e\ \mathtt{or}\ e' &\triangleq \mathtt{if}\ e\ \mathtt{then}\ \mathtt{T}\ \mathtt{else}\ e' \end{aligned} \tag{4.5}$$

and we can also define a case distinction for boolean tuples by

$$\begin{aligned} &\mathtt{case}_{\mathtt{bool}}(e_1, \ldots, e_n)\ \mathtt{of}\ ((b_{11}, \ldots, b_{n1}) \to e_1' \mid \ldots \mid (b_{1m}, \ldots, b_{nm}) \to e_m' \mid \_ \to e'') \\ &\quad \triangleq \mathtt{if}(e_1 = b_{11}\ \mathtt{and} \ldots \mathtt{and}\ e_n = b_{n1})\ \mathtt{then}\ e_1'\ \mathtt{else} \\ &\qquad \vdots \\ &\qquad \mathtt{if}(e_1 = b_{1m}\ \mathtt{and} \ldots \mathtt{and}\ e_n = b_{nm})\ \mathtt{then}\ e_m'\ \mathtt{else}\ e'' \end{aligned} \tag{4.6}$$

where $n, m \in \mathbb{N}$ are arbitrary and $b_{ij} \in \{\mathtt{T}, \mathtt{F}\}$.

Freshness We introduce notation for checking if two atomic names are *not* equal:

$$e \neq e' \triangleq \mathtt{not}(e = e').$$

The extension of this constructs to vectors is a construct that checks if a vector $\vec{e} \in (\mathrm{Exp}^{\mathsf{PNA}})^n$ consists of distinct atomic names

$$\mathtt{distinct}\, \vec{e} \triangleq e_1 \neq e_2\ \mathtt{and}\ e_1 \neq e_3\ \mathtt{and} \ldots \mathtt{and}\ e_{n-1} \neq e_n.$$

We define a construct that checks the freshness of two vectors $\vec{e} \in (\mathrm{Exp}^{\mathsf{PNA}})^n$ and $\vec{e}\,' \in (\mathrm{Exp}^{\mathsf{PNA}})^m$ for arbitrary $n, m \in \mathbb{N}$ by

$$\vec{e}\ \mathtt{freshfor}\ \vec{e}\,' \triangleq e_1 \neq e_1'\ \mathtt{and}\ e_1 \neq e_2'\ \mathtt{and} \ldots \mathtt{and}\ e_n \neq e_{m-1}'\ \mathtt{and}\ e_n \neq e_m'.$$

Natural numbers The encoding of any natural number $n \in \mathbb{N}$ in PNA is defined by

$$\mathtt{S}^n\,\mathtt{0} \triangleq \begin{cases} \mathtt{0} & \text{if } n = 0 \\ \mathtt{S}(\mathtt{S}^{n-1}\,\mathtt{0}) & \text{otherwise}. \end{cases}$$

Furthermore, we define the equality test for natural numbers by

$$\begin{aligned}(e_1 =_{\mathtt{nat}} e_2) \triangleq\ & (\mathtt{fix}\,\lambda(f : \mathtt{nat} \to \mathtt{nat} \to \mathtt{bool}) \to \lambda x_1 : \mathtt{nat} \to \lambda x_2 : \mathtt{nat} \to \mathtt{if}\,\mathtt{zero}\,x_1 \\ & \mathtt{then}\,\mathtt{zero}\,x_2\,\mathtt{else}\,(\mathtt{if}\,\mathtt{zero}\,x_2\,\mathtt{then}\,\mathtt{F}\,\mathtt{else}\,f\,(\mathtt{pred}\,x_1)(\mathtt{pred}\,x_2)))\,e_1\,e_2\end{aligned}$$

and the addition of two natural numbers by

$$\begin{aligned}(e_1 +_{\mathtt{nat}} e_2) \triangleq\ & (\mathtt{fix}\,\lambda(f : \mathtt{nat} \to \mathtt{nat} \to \mathtt{nat}) \to \lambda x_1 : \mathtt{nat} \to \lambda x_2 : \mathtt{nat} \to \mathtt{if}\,\mathtt{zero}\,x_1 \\ & \mathtt{then}\,x_2\,\mathtt{else}\,f\,(\mathtt{pred}\,x_1)(\mathtt{S}\,x_2))\,e_1\,e_2\,.\end{aligned}$$

Non-binding name abstraction Many nominal languages use a non-binding name abstraction construct $\langle\!\langle a \rangle\!\rangle e$ (where a is free), as opposed to our $\alpha a.\,e$ (where a is bound in e). Pitts [43, Definition 3.5] shows that in the presence of explicit swapping in the syntax, we can define non-binding name abstraction as syntactic sugar. We present the slightly generalised version $\langle\!\langle e \rangle\!\rangle e'$ that was used by Pitts [44, Note 10.2] in the context of generative local names (rather than Odersky-style here):

$$\langle\!\langle e \rangle\!\rangle e' \triangleq \alpha a.\,(e \rightleftharpoons a)\,e' \qquad \text{where } a \notin \mathrm{fn}(e, e'). \tag{4.7}$$

Syntactically, the free names of $\langle\!\langle e \rangle\!\rangle e'$ are given by $\mathrm{fn}\,e \cup \mathrm{fn}\,e'$. Its denotational semantics satisfies $[\![\langle\!\langle e \rangle\!\rangle e']\!]\rho = \langle [\![e]\!]\rho \rangle ([\![e']\!]\rho)$, independent from the question if $[\![e]\!]\rho \# \rho$ or not (compare this to $[\![\alpha a.\,e]\!]$ in Figure 4.11). It is a consequence of the clause for $\alpha a.\,e$ in the Characterisation Lemma 4.3.3 that in the presence of locally scoped names, binding name abstraction can be expressed in terms of non-binding name abstraction as well:

$$[\![\alpha a.\,e]\!] = [\![\nu a.\,\langle\!\langle a \rangle\!\rangle e]\!]\,.$$

Pattern matching for name abstractions For a user unfamiliar with nominal sets, programming directly with name abstractions and concretions might be unintuitive. A pattern matching construct for deconstructing name abstractions is likely to be easier to use. Pitts [43, Remark 3.4] introduces such a pattern matching construct that can be defined as syntactic sugar in PNA by

$$\mathtt{let}\,\langle\!\langle x \rangle\!\rangle y = e\,\mathtt{in}\,e' \triangleq \nu a.\,(e'[a/x, (e\,@\,a)/y]) \qquad \text{where } a \notin \mathrm{fn}(e, e') \tag{4.8}$$

where occurrences of x and y are bound in e'. We choose to match with two variables x and y, whereas Pitts [43, Remark 3.4] uses a name and a variable.

Pattern matching for λ-terms We argued that (4.1) directly expresses capture-avoiding substitution for λ-terms and we promote PNA as a programming language that conveniently and correctly expresses this kind of computation over object-level syntax with binding. However, the concretion operation in the $\mathtt{L}$-clause of (4.1) might be confusing to the programmer not familiar with nominal sets. Therefore we define user-friendly syntactic sugar for pattern matching of λ-terms in the flavour of FreshML [57]. The sugared pattern matching uses the pattern matching for name abstractions from (4.8) as follows

$$\begin{aligned}&\mathtt{case}_{\mathit{sugar}}\, e\, \mathtt{of}\, (\mathtt{V}\, x_1 \to e_1 \mid \mathtt{A}\, x_2\, x_2' \to e \mid \mathtt{L}\, \langle\!\langle x_3' \rangle\!\rangle x_3'' \to e_3) \\ &\qquad \triangleq \mathtt{case}\, e\, \mathtt{of}\, (\mathtt{V}\, x_1 \to e_1 \mid \mathtt{A}\, x_2\, x_2' \to e \mid \mathtt{L}\, x_3 \to \mathtt{let}\, \langle\!\langle x_3' \rangle\!\rangle x_3'' = x_3\, \mathtt{in}\, e_3)\end{aligned} \tag{4.9}$$

where $x_3 \notin \mathrm{fv}\, e_3$.

4.5.2 Metaprogramming examples

We give small case studies that illustrate the use of PNA for metaprogramming. In particular, we give PNA programs that

- count the (free and bound) identifiers of a λ-term,
- test equality for expressions of type $\delta\,(\mathtt{name} \times \mathtt{name})$,
- test if a identifier is free in a λ-term, and
- compute the capture-avoiding substitution for λ-terms.

All our examples use the syntactic sugar from Section 4.5.1.

In order to check the evaluation behaviour of our programs, we work with an exemplary PNA expression $L \in \mathrm{Can}^{\mathsf{PNA}}(\mathtt{term})$ defined by

$$L \triangleq \mathtt{A}(\mathtt{L}\, \alpha a_1.\mathtt{A}(\mathtt{V}\, a_2)(\mathtt{V}\, a_1))(\mathtt{V}\, a_3). \tag{4.10}$$

It stands for the λ-term $[(\lambda a_1.a_2\, a_1)a_3]_\alpha$ (see Example 2.3.22), where a_2 and a_3 are free identifiers and a_1 is bound.

Counting identifiers The following program $\mathit{count} \in \mathrm{Exp}^{\mathsf{PNA}}(\mathtt{term} \to \mathtt{nat})$ counts the number of identifier occurrences, both free and bound, in the λ-term represented by the input.

$$\begin{aligned}\mathit{count} \triangleq\ & \mathtt{fix}(\lambda(f : \mathtt{term} \to \mathtt{nat}) \to \lambda y : \mathtt{term} \to \mathtt{case}_{\mathit{sugar}}\, y\, \mathtt{of} \\ & \quad \mathtt{V}\, x_1 \to \mathtt{S}\,\mathtt{O} \\ & \quad \mid \mathtt{A}\, x_2\, x_2' \to (f\ x_2) +_{\mathtt{nat}} (f\ x_2') \\ & \quad \mid \mathtt{L}\, \langle\!\langle x_3 \rangle\!\rangle x_3' \to f\ x_3')\end{aligned}$$

It is easy to check that our example expression from (4.10) satisfies $L \Downarrow \mathtt{S}^3\mathtt{O}$.

Equality test Another good example of how metaprogramming works in PNA are equality tests of abstraction types. The test for two expressions of type $\delta\,(\mathtt{name} \times \mathtt{name})$ is defined by

$$e_1 =_{\delta\,(\mathtt{name}\times\mathtt{name})} e_2 \triangleq \lambda(x_1 : \delta\,(\mathtt{name} \times \mathtt{name})) \to \lambda(x_2 : \delta\,(\mathtt{name} \times \mathtt{name})) \to \quad (4.11)$$
$$\nu a.((\mathtt{fst}\,(x_1\,@\,a) = \mathtt{fst}\,(x_2\,@\,a))\ \mathtt{and}\ (\mathtt{snd}\,(x_1\,@\,a) = \mathtt{snd}\,(x_2\,@\,a))).$$

The expressions $ex_1 \triangleq \alpha a.(a\,,\,b)$ and $ex_2 \triangleq \alpha c.(c\,,\,b)$ are syntactically equal (as we implicitly identify expressions by α-equivalence) and indeed they evaluate as follows: $ex_1 =_{\delta\,(\mathtt{name}\times\mathtt{name})} ex_2 \Downarrow \mathtt{T}$. The 'trick' of why this works is that $\nu a.\_$ is a binder and substitution is name-capture avoiding (see Figure 4.3). In other words, we use the binding mechanism of the meta-language (PNA). Practically this means that whenever we have an expression $\nu a.x$, we know that any substitution for x has to be fresh for a, or we have to α-rename $\nu a.x$ to perform the substitution. For example, $(\nu a.x)[a/x] = (\nu a'.x)[a/x] = \nu a'.a$. In our case, this mechanism ensures that the concretions $x_1\,@\,a$ and $x_2\,@\,a$ will always evaluate to a concretion with a fresh name.

We can adjust the above program to an equality test $e_1 =_{\mathtt{term}} e_2$ for λ-terms, by using again the fact that $\nu a.\_$ is a binder. The details are straight-forward and we leave them to the reader.

Free identifier test We give a function $\mathit{free} \in \mathrm{Exp}^{\mathsf{PNA}}(\mathtt{name} \to \mathtt{term} \to \mathtt{bool})$ that computes if a given identifier appears free in a given λ-term:

$$\begin{array}{l}\mathit{free} \triangleq \lambda x : \mathtt{name} \to \mathtt{fix}(\lambda(f : \mathtt{term} \to \mathtt{bool}) \to \lambda y : \mathtt{term} \to \mathtt{case}_{\mathit{sugar}}\, y\ \mathtt{of}\\ \quad \mathtt{V}\,x_1 \to \mathtt{not}\,(x_1 = x)\\ \quad \mid \mathtt{A}\,x_2\,x_2' \to (f\,x_2)\ \mathtt{and}\ (f\,x_2')\\ \quad \mid \mathtt{L}\,\langle\!\langle x_3 \rangle\!\rangle\, x_3' \to f\,x_3')\end{array}$$

The above definition behaves correctly, because it implicitly uses the binding 'trick' from (4.11) to ensure that when we pattern match $\mathtt{L}\,\langle\!\langle x_3 \rangle\!\rangle\, x_3'$ the atomic name substituted for x_3 will always be fresh for the atomic name that x stands for. We implicitly use local scoping $\nu a.e$ in this pattern matching, through the syntactic sugar in (4.9) and (4.8). Our example λ-term from (4.10) evaluates as $\mathit{free}\,a_2\,L \Downarrow \mathtt{T}$ and $\mathit{free}\,a_1\,L \Downarrow \mathtt{F}$.

Capture-avoiding substitution (again) Following up on our definition of capture-avoiding substitution in PNA without syntactic sugar from (4.1), we now give the sugared version of this program:

$$\begin{array}{l}\mathit{subst}_{\mathit{sugar}} \triangleq \lambda y' : \mathtt{term} \to \lambda x : \mathtt{name} \to \mathtt{fix}(\lambda(f : \mathtt{term} \to \mathtt{term}) \to \lambda y : \mathtt{term} \to \quad (4.12)\\ \quad \mathtt{case}_{\mathit{sugar}}\, y\ \mathtt{of}\\ \quad\quad \mathtt{V}\,x_1 \to \mathtt{if}\,x_1 = x\ \mathtt{then}\,y'\ \mathtt{else}\,y\\ \quad\quad \mid \mathtt{A}\,x_2\,x_2' \to \mathtt{A}(f\,x_2)(f\,x_2')\\ \quad\quad \mid \mathtt{L}\,\langle\!\langle x_3 \rangle\!\rangle\, x_3' \to \mathtt{L}(\langle\!\langle x_3 \rangle\!\rangle (f\,x_3'))).\end{array}$$

In the context of generative local names, a very similar variant of this program appears in Pitts [44, Example 10.4].

CHAPTER 5

PROGRAM EQUIVALENCE IN PNA

Having an operational semantics allows us to formulate a context-based notion of program equivalence: two expressions are defined to be contextually equivalent if they give the same *observable results* when put in any context that forms a *complete program*.

Assuming the programming language in question has a denotational semantics, there is another obvious notion of program equivalence: two programs are considered equivalent if their denotations are equal. *Full abstraction* is by definition satisfied if the two notions of program equivalence above coincide: two expressions are contextually equivalent if and only if they have equal denotations. This gives a very strong connection between the operational and denotational semantics of a programming language. A weaker (but easier to achieve) connection is *computational adequacy*, which is defined to be the 'if'-direction of full abstraction: a programming language is computationally adequate if denotational equality implies contextual equivalence.

Traditionally, Scott domains (Definition 2.2.10) give a computationally adequate, but not fully abstract, denotational semantics for PCF. Once 'parallel-or' is added, full abstraction is achieved. We mirror this traditional story in the nominal setting with PNA and nominal Scott domains.

The notion of contextual equivalence is discussed in Section 5.1. In Section 5.2 we extend PNA to PNA^+ with two new constructs and add some further syntactic sugar. We prove computational adequacy for PNA^+ in Section 5.3 and in Section 5.4 we show that full abstraction fails for PNA if only one of the two new constructs is added. Finally we prove in Section 5.5 that full abstraction holds for PNA^+.

5.1 Contextual equivalence

In Section 5.1.1 we formally define the contextual equivalence relation for PNA via contexts and give examples of expressions that are related by it. Section 5.1.2 gives an alternative characterisation of contextual equivalence as the largest type-respecting binary relation that is adequate and compatible.

5.1.1 Definitions and examples

Recall the generic definition of contextual equivalence for any programming language: giving the same observable results in any context that forms a complete program. For PNA, a complete program is taken to be a variable-closed expression of type `bool` and the observable result is taken to be evaluation to `T`.

Definition 5.1.1 (**contextual preorder**). Contexts $C \in \mathrm{Cont}^{\mathsf{PNA}}$ are formally defined in Section 4.1.3 and their typing relation is given in Section 4.2.3. For two expressions $e, e' \in \mathrm{Exp}^{\mathsf{PNA}}$, we define the *contextual preorder* $\Gamma \vdash e \lesssim_{\mathsf{PNA}} e' : \tau$ to hold if the expressions are of the same type $\Gamma \vdash e : \tau$ and $\Gamma \vdash e' : \tau$ and for all closed contexts of boolean type $C : (\Gamma \triangleright \tau) \rightsquigarrow (\emptyset \triangleright \mathtt{bool})$ it is the case that

$$C[e] \Downarrow \mathtt{T} \Rightarrow C[e'] \Downarrow \mathtt{T}.$$

The equivalence relation generated by $\lesssim_{\mathsf{PNA}}$ is written $\cong_{\mathsf{PNA}}$ and is called *contextual equivalence*.

We continue with some examples of valid instances of the PNA contextual preorder.

Example 5.1.2 (**Odersky-style vs. generative local names**). Although PNA contains the expressions of the ν-calculus [46] as a sublanguage, the two languages have different semantics for local names: Odersky-style for PNA versus generative for the ν-calculus (see also Remark 4.4.1). This affects properties of contextual equivalence in the two languages. For example, if $\Gamma, x : \tau \vdash e : \tau'$, then the contextual equivalence

$$\Gamma \vdash \nu a.\, \lambda x : \tau \to e \;\cong_{\mathsf{PNA}}\; \lambda x : \tau \to \nu a.\, e : \tau \to \tau' \tag{5.1}$$

is valid in PNA, but not valid in the ν-calculus, see Pitts and Stark [46, Example 2].

Example 5.1.3 (**motivation of computational adequacy**). Some ν-calculus equivalences are also true for PNA, once one takes into account the fact that, like PCF, PNA is call-by-name, whereas the ν-calculus is call-by-value. For example, here are call-by-name analogues of two equivalences in Pitts and Stark [46, Example 4] that are valid in PNA:

$$\emptyset \vdash \nu a.\, \lambda x : \mathtt{name} \to (x = a) \;\cong_{\mathsf{PNA}}\; \lambda x : \mathtt{name} \to \mathtt{if}\ x = x\ \mathtt{then}\ \mathtt{F}\ \mathtt{else}\ \mathtt{F} : \mathtt{name} \to \mathtt{bool} \tag{5.2}$$

$$\begin{aligned} &\emptyset \vdash \nu a.\, \nu a'.\, \lambda(f : \mathtt{name} \to \mathtt{bool}) \to \mathtt{eq}(f\, a)(f\, a') \;\cong_{\mathsf{PNA}} \\ &\qquad \lambda(f : \mathtt{name} \to \mathtt{bool}) \to \nu a.\, \mathtt{if}\ f\, a\ \mathtt{then}\ \mathtt{T}\ \mathtt{else}\ \mathtt{T} : (\mathtt{name} \to \mathtt{bool}) \to \mathtt{bool}. \end{aligned} \tag{5.3}$$

Here $\mathtt{eq} : \mathtt{bool} \to \mathtt{bool} \to \mathtt{bool}$ is syntactic sugar for a boolean-equality test that can be defined using conditionals. In contrast to the ν-calculus, where it takes significant effort to prove equivalences like (5.2) and (5.3), for PNA these properties are easily seen to hold by checking denotational equality, thanks to our computational adequacy result in Theorem 5.3.19. See Example 5.3.20 for proofs.

The next example introduces two instances of the contextual preorder, (5.11) and (5.12), that are highly relevant for the rest of this thesis. They show that PNA in its current version fails to be fully abstract. Our proofs of these contextual preorders (given in Sections 5.4.1 and 5.4.1) are labour-intensive, because we cannot utilise the denotational semantics in these arguments.

Example 5.1.4 (failure of full abstraction). Note that not all valid PNA contextual equivalences can be verified using the denotational semantics. There are instances of the PNA contextual preorder that cannot be established by calculating denotations in our model. Here are two examples of this phenomenon, suggested by Tzevelekos [private communication]. Consider the following variable-closed PNA expressions:

$$\mathtt{eqBot}_a \triangleq \lambda x : \mathtt{name} \to \mathtt{if}\ x = a\ \mathtt{then}\ \mathtt{T}\ \mathtt{else}\ \mathtt{bot}_{\mathtt{bool}} \tag{5.4}$$
$$\mathtt{kBot} \triangleq \lambda x : \mathtt{name} \to \mathtt{bot}_{\mathtt{bool}} \tag{5.5}$$
$$\mathtt{F}_1 \triangleq \lambda(f : (\mathtt{name} \to \mathtt{bool}) \to \mathtt{bool}) \to \nu a.\, f\ \mathtt{eqBot}_a \tag{5.6}$$
$$\mathtt{F}_2 \triangleq \lambda(f : (\mathtt{name} \to \mathtt{bool}) \to \mathtt{bool}) \to f\ \mathtt{kBot} \tag{5.7}$$
$$\mathtt{eq}_a \triangleq \lambda x : \mathtt{name} \to (x = a) \tag{5.8}$$
$$\mathtt{G}_1 \triangleq \lambda(g : (\mathtt{name} \to \mathtt{bool}) \to \mathtt{name}) \to \nu a.\, (g\ \mathtt{eq}_a = a) \tag{5.9}$$
$$\mathtt{G}_2 \triangleq \lambda(g : (\mathtt{name} \to \mathtt{bool}) \to \mathtt{name}) \to \mathtt{F}\,. \tag{5.10}$$

The constructs $\mathtt{eqBot}_a$ and $\mathtt{kBot}$ are variable-closed expressions of type $\mathtt{name}\to\mathtt{bool}$; $\mathtt{F}_1$ and $\mathtt{F}_2$ are of type $((\mathtt{name}\to\mathtt{bool})\to\mathtt{bool})\to\mathtt{bool}$; and $\mathtt{G}_1$ and $\mathtt{G}_2$ are of type $((\mathtt{name}\to\mathtt{bool})\to\mathtt{name})\to\mathtt{bool}$. In Section 5.4 we will prove that

$$\emptyset \vdash \mathtt{F}_1 \cong_{\mathsf{PNA}} \mathtt{F}_2 : ((\mathtt{name} \to \mathtt{bool}) \to \mathtt{bool}) \to \mathtt{bool} \tag{5.11}$$
$$\emptyset \vdash \mathtt{G}_1 \lesssim_{\mathsf{PNA}} \mathtt{G}_2 : ((\mathtt{name} \to \mathtt{bool}) \to \mathtt{name}) \to \mathtt{bool}\,. \tag{5.12}$$

The intuitive justification of (5.11) is that whatever argument of type $(\mathtt{name}\to\mathtt{bool})\to\mathtt{bool}$ is supplied for f by a context, it cannot have a free occurrence of a (because substitution for f in $\nu a.(f\ \mathtt{eqBot}_a)$ is capture-avoiding) and hence cannot distinguish $\mathtt{eqBot}_a$ from $\mathtt{kBot}$. Similarly for (5.12), whatever argument of type $(\mathtt{name}\to\mathtt{bool})\to\mathtt{name}$ is supplied for g by a context will not contain a free and hence cannot produce this name when applied to $\mathtt{eq}_a$. (Since that application may diverge, in (5.12) we only have $\lesssim_{\mathsf{PNA}}$ rather than $\cong_{\mathsf{PNA}}$.)

In the nominal Scott domain model of Section 4.3 however, we have $[\![\mathtt{F}_1]\!] \neq [\![\mathtt{F}_2]\!]$ and $[\![\mathtt{G}_1]\!] \not\sqsubseteq [\![\mathtt{G}_2]\!]$. These inequalities can be derived by using the functions $\mathit{exists}_{\mathbb{A}}$ (3.24), $\mathit{the}_{\mathbb{A}}$ (3.27) and eq_a (3.22) in the following observations:

$$[\![\mathtt{F}_1]\!](\mathit{exists}_{\mathbb{A}}) = a\backslash(\mathit{exists}_{\mathbb{A}}\,[\![\mathtt{eqBot}_a]\!]) = a\backslash\mathrm{true} = \mathrm{true} \tag{5.13}$$
$$[\![\mathtt{F}_2]\!](\mathit{exists}_{\mathbb{A}}) = \mathit{exists}_{\mathbb{A}}\,[\![\mathtt{kBot}]\!] = \bot \tag{5.14}$$

as well as

$$[\![\mathtt{G}_1]\!](\mathit{the}_{\mathbb{A}}) = a\backslash(\mathit{eq}_a(\mathit{the}_{\mathbb{A}}\,\mathit{eq}_a)) = a\backslash(\mathit{eq}_a\,a) = a\backslash\mathrm{true} = \mathrm{true} \tag{5.15}$$
$$[\![\mathtt{G}_2]\!](\mathit{the}_{\mathbb{A}}) = \mathrm{false}\,. \tag{5.16}$$

Once we extend PNA with operational versions of $\mathit{exists}_{\mathbb{A}}$ and $\mathit{the}_{\mathbb{A}}$, we gain the property that two expressions are contextually equivalent if and only if their denotations are the same. That is, we gain full abstraction and this is proved in Section 5.5. Section 5.2 gives the details of these extensions to PNA.

5.1.2 The relational approach

In this section we give an alternative characterisation of the contextual preorder: it is exactly the largest compatible and preadequate relation. We give the necessary definitions and prove some important properties of the contextual preorder, which we need for our full abstraction results in later sections.

Definition 5.1.5 (**type-respecting binary relation**). A *type-respecting binary relation* for PNA is a set of quadruples $\mathscr{R} \subseteq \mathrm{Env}^{\mathsf{PNA}} \times \mathrm{Exp}^{\mathsf{PNA}} \times \mathrm{Exp}^{\mathsf{PNA}} \times \mathrm{Typ}^{\mathsf{PNA}}$ that satisfies

$$\Gamma \vdash e_1 \mathscr{R}\ e_2 : \tau \;\Rightarrow\; \Gamma \vdash e_1 : \tau \wedge \Gamma \vdash e_2 : \tau\,,$$

where we write $\Gamma \vdash e_1 \mathscr{R}\ e_2 : \tau$ instead of $(\Gamma, e_1, e_2, \tau) \in \mathscr{R}$. Such relation is called *preadequate* if all $\emptyset \vdash e_1 \mathscr{R}\ e_2 : \mathtt{bool}$ satisfy

$$e_1 \Downarrow \mathtt{T} \;\Rightarrow\; e_2 \Downarrow \mathtt{T}$$

and it is called *compatible* if we have for all $C : (\Gamma \triangleright \tau) \rightsquigarrow (\Gamma' \triangleright \tau')$ that

$$\Gamma \vdash e_1 \mathscr{R}\ e_2 : \tau \;\Rightarrow\; \Gamma' \vdash C[e_1] \mathscr{R}\ C[e_2] : \tau'\,. \tag{5.17}$$

A *type-respecting preorder* is a type-respecting binary relation that is reflexive $\Gamma \vdash e : \tau \Rightarrow \Gamma \vdash e \mathscr{R}\ e : \tau$ and transitive $\Gamma \vdash e_1 \mathscr{R}\ e_2 : \tau \wedge \Gamma \vdash e_2 \mathscr{R}\ e_3 : \tau \Rightarrow \Gamma \vdash e_1 \mathscr{R}\ e_3 : \tau$. A type-respecting preorder that is symmetric $\Gamma \vdash e_1 \mathscr{R}\ e_2 : \tau \Rightarrow \Gamma \vdash e_2 \mathscr{R}\ e_1 : \tau$ is called *type-respecting equivalence*.

Lemma 5.1.6 (**type-respecting relations**). *The contextual preorder is a type-respecting preorder and contextual equivalence is a type-respecting equivalence.*

Proof. All properties follow directly from the definition. □

The next proposition enables a new proof technique: to show that two expressions are in the contextual preorder, we show that they are in a relation that is type-respecting, preadequate and compatible.

Proposition 5.1.7 (**largest relation**). *The contextual preorder is characterised as the largest type-respecting relation that is preadequate and compatible, in the sense that any such relation is a subset of it.*

Proof. We start with showing that the $\lesssim_{\mathsf{PNA}}$ is preadequate and compatible. Preadequacy also follows directly from its definition, instantiating it with the empty context $[-]$. Compatibility follows from the well-definedness of context composition as in Lemmas 4.1.7 and 4.2.10.

To show that $\lesssim_{\mathsf{PNA}}$ is the largest type-respecting, preadequate and compatible relation, let any other such relation $\mathscr{R}$ be given, assume $\Gamma \vdash e_1 \mathscr{R}\ e_2 : \tau$ and let any $C : (\Gamma \triangleright \tau) \rightsquigarrow (\emptyset \triangleright \mathtt{bool})$ be given. By compatibility we get $\emptyset \vdash C[e_1] \mathscr{R}\, C[e_2] : \mathtt{bool}$ and by preadequacy we get $C[e_1] \Downarrow \mathtt{T} \Rightarrow C[e_2] \Downarrow \mathtt{T}$, which proves $\Gamma \vdash e_1 \lesssim_{\mathsf{PNA}} e_2 : \tau$. □

It is convenient for forthcoming proofs to introduce notation for simultaneous substitutions that replace all variables of a typing environment with well-typed and variable-closed expressions.

Definition 5.1.8 (**Γ-substitutions**). For a typing environment $\Gamma \in \mathrm{Env}^{\mathsf{PNA}}$, a function $s : \mathrm{dom}\,\Gamma \to \mathrm{Exp}^{\mathsf{PNA}}$ is defined to be a Γ-*substitution* if it maps each argument $x \in \mathrm{dom}\,\Gamma$ to a variable-closed expression of the right type: $s\,x \in \mathrm{Exp}^{\mathsf{PNA}}(\Gamma\,x)$. Let $\mathrm{Subst}^{\mathsf{PNA}}(\Gamma)$ be the set of all such Γ-substitutions. Any Γ-substitution s can be considered to be the simultaneous substitution operation $[(s\,x_1)/x_1, \ldots, (s\,x_n)/x_n]$ (where $\Gamma = \{x_1 : \tau_1, \ldots, x_n : \tau_n\}$), and in accordance to that we define the application of s to an expression by $e\,s \triangleq e[(s\,x_1)/x_1, \ldots, (s\,x_n)/x_n]$. It is a consequence of the Substitution Lemma 4.2.6 that for every $\Gamma \vdash e : \tau$ and $s \in \mathrm{Subst}^{\mathsf{PNA}}(\Gamma)$ it holds that $\emptyset \vdash e\,s : \tau$.

The next lemma shows that Γ-substitutions preserve the contextual preorder.

Lemma 5.1.9 (**order preservation by substitution**). *If* $\Gamma \vdash e_1 \lesssim_{\mathsf{PNA}} e_2 : \tau$ *and* $s \in \mathrm{Subst}^{\mathsf{PNA}}(\Gamma)$ *then* $\emptyset \vdash e_1\,s \lesssim_{\mathsf{PNA}} e_2\,s : \tau$.

Proof. Assume without loss of generality that $\Gamma = \{x_1 : \tau_1, \ldots, x_n : \tau_n\}$. For proving $\emptyset \vdash e_1\,s \lesssim_{\mathsf{PNA}} e_2\,s : \tau$, let $C : (\emptyset \triangleright \tau) \rightsquigarrow (\emptyset \triangleright \mathtt{bool})$ be given and note that, as C has no free variables, it holds that $C[e_1\,s] = (C[e_1])\,s$ and $C[e_2\,s] = (C[e_2])\,s$. Define now the context corresponding to C and s by

$$C' \triangleq (\lambda x_1 : \tau_1 \to \ldots \lambda x_n : \tau_n \to C[-])(s\,x_1)\ldots(s\,x_n).$$

Lemma 4.2.11 gives us $C : (\Gamma \triangleright \tau) \rightsquigarrow (\Gamma \triangleright \mathtt{bool})$ and with this and Lemma 4.2.10 we get $C' : (\Gamma \triangleright \tau) \rightsquigarrow (\emptyset \triangleright \mathtt{bool})$. By construction (and the definition of substitution) it holds that $C'[e_1] \Downarrow \mathtt{T} \Leftrightarrow (C[e_1])\,s \Downarrow \mathtt{T}$ and $C'[e_2] \Downarrow \mathtt{T} \Leftrightarrow (C[e_2])\,s \Downarrow \mathtt{T}$. $\Gamma \vdash e_1 \lesssim_{\mathsf{PNA}} e_2 : \tau$ gives $C'[e_1] \Downarrow \mathtt{T} \Rightarrow C'[e_2]$ which leads to $(C[e_1])\,s \Downarrow \mathtt{T} \Rightarrow (C[e_2])\,s \Downarrow \mathtt{T}$ and this was to show for $\emptyset \vdash e_1\,s \lesssim_{\mathsf{PNA}} e_2\,s : \tau$. □

5.2 Extending PNA

As we will show in Section 5.3, the language presented so far, PNA, is computationally adequate with respect to the nominal Scott domain model (Theorem 5.3.19). However, as Example 5.1.4 indicates, PNA is not fully abstract. This section is concerned with the constructs that we need to add to PNA in order to make it fully abstract.

We add two constructs, definite description over names ('the unique atomic name such that…') and existential quantification over names ('there exists some atomic name such that…'). Together with PNA they form the language PNA^+. In Section 5.5 we prove full abstraction for PNA^+, and in Section 5.4 we show that both added constructs are indeed necessary: full abstraction fails if we leave one of them out.

The next lemma shows that if we want to check evaluation to a canonical form for all fresh names, then checking it for just one fresh name is enough. This gives the basis for the correctness of the operational semantics of the new constructs.

Syntax:

$$\texttt{the}\, x.e \quad \text{with } x \text{ bound in } e$$

Typing:

$$\frac{\Gamma, x : \texttt{name} \vdash e : \texttt{bool}}{\Gamma \vdash \texttt{the}\, x.e : \texttt{name}}$$

Denotational semantics:

$$[\![\texttt{the}\, x.e]\!]\rho \triangleq \begin{cases} a & \text{if } (\forall a' \in \mathbb{A})\, [\![e]\!]\rho[x \mapsto a'] = \begin{cases} \text{true} & \text{if } a' = a \\ \text{false} & \text{otherwise} \end{cases} \\ \bot & \text{otherwise} \end{cases}$$

Big-step evaluation:

$$\frac{e[a/x] \Downarrow \texttt{T} \qquad b \# (e,a) \qquad (\forall a' \in (\mathrm{fn}\, e - \{a\}) \cup \{b\})\, e[a'/x] \Downarrow \texttt{F}}{\texttt{the}\, x.e \Downarrow a}$$

Frame-stack evaluation:

$$\begin{array}{l} \langle F\,, \texttt{the}\, x.e \rangle \to \langle F\,, \nu b.\, \texttt{case}_{\texttt{bool}}(e[a_1/x], e[a_2/x], \ldots, e[a_n/x], e[b/x])\ \texttt{of} \\ \quad ((\texttt{T},\texttt{F},\ldots,\texttt{F},\texttt{F}) \to a_1 \mid (\texttt{F},\texttt{T},\ldots,\texttt{F},\texttt{F}) \to a_2 \mid \ldots \mid (\texttt{F},\texttt{F},\ldots,\texttt{T},\texttt{F}) \to a_n \mid \_ \to \texttt{bot}_{\texttt{name}}) \rangle \\ \quad \text{where } \{a_1,\ldots,a_n\} = \mathrm{fn}\, e \text{ and } b \# e \end{array}$$

Figure 5.1: Syntax and semantics of definite description over names

Lemma 5.2.1 (substitution of fresh names). *For all* $x : \texttt{name} \vdash e : \tau$ *it holds that*

$$((\exists a \in \mathbb{A})\, a \# e, c \wedge e[a/x] \Downarrow c) \Leftrightarrow ((\forall b \in \mathbb{A})\, b \# e, c \Rightarrow e[b/x] \Downarrow c).$$

Proof. The right-to-left direction is obvious. The left-to-right direction a direct consequence of the equivariance of substitution (Lemma 4.1.3) and big-step evaluation (Lemma 4.4.10). □

5.2.1 Definite description over names

Figure 5.1 defines the syntax and semantics of definite description over names. Its frame-stack rules use the syntactic sugar from (4.4) and (4.6) for better readability.

We can characterise the denotational semantics of definite description in the style of Lemma 4.3.3.

Lemma 5.2.2 (Characterisation Lemma for $\texttt{the}\, x.e$**).** *The following equality holds*

$$[\![\texttt{the}\, x.e]\!] = \mathit{the}_{\mathbb{A}} \circ \mathit{cur}([\![e]\!])$$

with the currying function cur being defined in (3.20) *and* $\mathit{the}_{\mathbb{A}}$ *being defined in* (3.27).

Proof. Follows directly from unfolding the definitions. □

The frame-stack transitions for definite description in Figure 5.1 is validated by the big-step evaluation.

Lemma 5.2.3 (**frame-stack soundness for** $\mathtt{the}\, x.e$)**.** *For all* $x : \mathtt{name} \vdash e : \mathtt{bool}$ *and* $a \in \mathbb{A}$ *we have*

$$\mathtt{the}\, x.e \Downarrow a \iff \nu b.\, \mathtt{case}_{\mathtt{bool}}(e[a_1/x], e[a_2/x], \ldots, e[a_n/x], e[b/x])\, \mathtt{of}\, (\ldots) \Downarrow a$$

where $\{a_1, \ldots, a_n\} = \mathrm{fn}\, e$ *and* $b \# e$.

Proof. For the left-to-right direction assume $\mathtt{the}\, x.e \Downarrow a$. Then we know for sure that $a \in \mathrm{fn}\, e$, because if we had $a \# e$ then by Lemma 5.2.1 $e[b/x] \Downarrow \mathtt{T}$ would also hold, which is a contradiction. The right-to-left direction is a straightforward calculation to show that by definition of $\mathtt{case}_{\mathtt{bool}}$ the hypotheses of the $\mathtt{the}\, x.e \Downarrow a$ rule must hold. □

Note that Lemma 5.2.3 does not imply that $\mathtt{the}\, x.e$ can be defined as syntactic sugar in PNA. The reason why $\mathtt{the}\, x.e$ is not 'just' syntactic sugar in PNA is that Lemma 5.2.3 holds only when $\mathtt{the}\, x.e$ is variable-closed, yet it is crucial for expressivity that we can form definite description over any open expression. This explains why $\mathtt{the}\, x.e$ is a proper extension of PNA, even though for variable-closed expressions we could define it as syntactic sugar.

5.2.2 Existential quantification over names

The syntax and semantics of existential quantification over names are given in Figure 5.2, where we use the syntactic sugar from (4.4) and (4.6). A weaker form of existential quantification for natural numbers (rather than, as here, for names) already occurs in Plotkin's original PCF paper [47, Section 5].

The evaluation rules for existential quantification over names in Figure 5.2 introduce non-determinism in the frame-stack transition system, violating the determinism results of Lemma 4.4.22. The same failure of frame-stack determinism would occur if we added the classical parallel-or construct (as it appeared in Plotkin's work on full abstraction for PCF [47, Table 1]) to PNA. In fact, in Section 5.2.4 we will show how parallel-or can be expressed in terms of existential quantification over names.

Lemma 5.2.4 (**Characterisation Lemma for** $\mathtt{ex}\, x.e$)**.** *The denotational semantics of existential quantification can be characterised by*

$$[\![\mathtt{ex}\, x.e]\!] = \mathit{exists}_{\mathbb{A}} \circ \mathit{cur}([\![e]\!])$$

where the currying function cur is defined in (3.20) *and* $\mathit{exists}_{\mathbb{A}}$ *is defined in* (3.24)*.*

Proof. Follows directly from unfolding the definitions. □

Just as for definite description, the frame-stack transitions for existential quantification in Figure 5.2 are validated by the big-step evaluation.

Syntax:

$$\mathtt{ex}\, x.e \quad \text{with } x \text{ bound in } e$$

Typing:

$$\frac{\Gamma, x : \mathtt{name} \vdash e : \mathtt{bool}}{\Gamma \vdash \mathtt{ex}\, x.e : \mathtt{bool}}$$

Denotational semantics:

$$[\![\mathtt{ex}\, x.e]\!]\rho \triangleq \begin{cases} \text{true} & \text{if } (\exists a \in \mathbb{A})\, [\![e]\!]\rho[x \mapsto a] = \text{true} \\ \text{false} & \text{if } (\forall a \in \mathbb{A})\, [\![e]\!]\rho[x \mapsto a] = \text{false} \\ \bot & \text{otherwise} \end{cases}$$

Big-step evaluation:

$$\frac{a \in \mathbb{A} \qquad e[a/x] \Downarrow \mathtt{T}}{\mathtt{ex}\, x.e \Downarrow \mathtt{T}}$$

$$\frac{b \# e \qquad \{a_1, \ldots, a_n\} = \mathrm{fn}\, e \cup \{b\} \qquad e[a_1/x] \Downarrow \mathtt{F} \quad \ldots \quad e[a_n/x] \Downarrow \mathtt{F}}{\mathtt{ex}\, x.e \Downarrow \mathtt{F}}$$

Frame-stack evaluation:

$$\langle F\, , \mathtt{ex}\, x.e \rangle \to \langle F\, , \mathtt{if}\, e[a/x]\, \mathtt{then}\, \mathtt{T}\, \mathtt{else}\, \mathtt{bot}_{\mathtt{bool}} \rangle \qquad \text{for all } a \in \mathbb{A}$$

$$\langle F\, , \mathtt{ex}\, x.e \rangle \to \langle F\, , \mathtt{case}_{\mathtt{bool}}(e[a_1/x], \ldots, e[a_n/x])\, \mathtt{of}\, ((\mathtt{F}, \ldots, \mathtt{F}) \to \mathtt{F} \mid \_ \to \mathtt{bot}_{\mathtt{bool}}) \rangle$$
where $\{a_1, \ldots, a_n\} = \mathrm{fn}\, e \cup \{b\}$ and $b \# e$

Figure 5.2: Syntax and semantics of existential quantification over names

Lemma 5.2.5 (**frame-stack soundness for** $\mathtt{ex}\,x.e$)**.** *For all* $x : \mathtt{name} \vdash e : \mathtt{bool}$ *and* $c \in \{\mathtt{T},\mathtt{F}\}$ *we have*

$$\begin{aligned}\mathtt{ex}\,x.e \Downarrow c \iff &((\exists a \in \mathbb{A})\ \mathtt{if}\, e[a/x]\, \mathtt{then}\, \mathtt{T}\, \mathtt{else}\, \mathtt{bot}_{\mathtt{bool}} \Downarrow c) \vee \\ &\mathtt{case}_{\mathtt{bool}}(e[a_1/x],\ldots,e[a_n/x])\, \mathtt{of}\, ((\mathtt{F},\ldots,\mathtt{F}) \to \mathtt{F} \mid \_ \to \mathtt{bot}_{\mathtt{bool}}) \Downarrow c\end{aligned}$$

where $\{a_1,\ldots,a_n\} = \mathrm{fn}\, e \cup \{b\}$ *and* $b \# e$.

Proof. Both directions are straightforward when we use Lemma 5.2.1. □

As described above, the frame-stack transition system of PNA with $\mathtt{ex}\,x.e$ is non-deterministic. However, in the case where no existential quantification over names is involved, the transitions remain deterministic.

Lemma 5.2.6 (**pseudo-determinacy**)**.** *If we consider the frame-stack transition rules of* PNA *(Figure 4.14) together with the rules for* $\mathtt{the}\,x.e$ *and* $\mathtt{ex}\,x.e$*, then the frame-stack transition system is deterministic for expressions that are not an existential quantification. Formally we have*

$$e \neq \mathtt{ex}\,x.e' \wedge \langle F\,, e\rangle \to \langle F_1\,, e_1\rangle \wedge \langle F\,, e\rangle \to \langle F_2\,, e_2\rangle \;\Rightarrow\; \langle F_1\,, e_1\rangle = \langle F_2\,, e_2\rangle\,.$$

Proof. By the proof Lemma 4.4.22 plus one case for definite description, which uses the name restriction in $\nu b.\mathtt{case}_{\mathtt{bool}}\ldots$ to show that for any choice of fresh name $b \# e$ we take the same transition. □

Remark 5.2.7 (**finite modulo symmetry**). Note that the analogue for natural numbers of our definite description and existential quantification functionals are not computable, as indicated already in Remark 3.5.4. The computability of definite description and existential quantification *over names* provide an example of the phenomenon of 'finite modulo symmetry' mentioned in the introduction. For example, to prove $\mathtt{ex}\,x.e \Downarrow \mathtt{F}$, we just have to show $e[a/x] \Downarrow \mathtt{F}$ for each of the finitely many atomic names a that occur free in e and then pick any one of the infinitely many atomic names b that do not occur free in e and show $e[b/x] \Downarrow \mathtt{F}$; Lemma 5.2.1 shows that if $e[b/x] \Downarrow \mathtt{F}$, then $e[b'/x] \Downarrow \mathtt{F}$ holds for any other b' not occurring free in e.

5.2.3 New languages

We define three different extensions of PNA: PNA+the where definite description over names is added, PNA+ex where existential quantification over names is added, and PNA^+ where both are added. Proposition 5.2.9 then shows that all semantic results presented so far, with one exception, carry over to each of the three extensions.

Definition 5.2.8 (**extensions of** PNA)**.** Define the language PNA^+ to be extension of PNA with $\mathtt{the}\,x.e$ and $\mathtt{ex}\,x.e$. The syntax, type system, denotational semantics and operational semantics of PNA^+ are those of PNA extended with the rules in Figures 5.1 and 5.2. The permutation action and substitution operation are extended by

$$\begin{aligned}\pi \cdot \mathtt{the}\,x.e &\triangleq \mathtt{the}\,x.\pi \cdot e \qquad (\mathtt{the}\,x'.e)[e'/x] \triangleq \mathtt{the}\,x'.e[e'/x] \quad \text{if } x' \notin \mathrm{fv}\,e' \cup \{x\} \\ \pi \cdot \mathtt{ex}\,x.e &\triangleq \mathtt{ex}\,x.\pi \cdot e \qquad (\mathtt{ex}\,x'.e)[e'/x] \triangleq \mathtt{ex}\,x'.e[e'/x] \quad \text{if } x' \notin \mathrm{fv}\,e' \cup \{x\},\end{aligned}$$

canonical forms stay the same, contexts are extended by

$$C \in \mathrm{Cont}^{\mathsf{PNA}^+} \triangleq \ldots(\text{as for } \mathsf{PNA})\ldots \mid \mathtt{the}\, x.\, C \mid \mathtt{ex}\, x.\, C\,,$$

frame-stacks stay the same and types stay the same.

Similarly we define PNA+the to be PNA with only the definite description as in Figure 5.1 added, and define PNA+ex to be PNA with only the existential quantification as in Figure 5.2 added. The contextual preorders $\lesssim_{\mathsf{PNA}^+}$, $\lesssim_{\mathsf{PNA+the}}$ and $\lesssim_{\mathsf{PNA+ex}}$ are the obvious extensions of Definition 5.1.1 for the according languages.

Unless specified otherwise, all results from now on will be for PNA^+, and those results will specialise to PNA, PNA+the and PNA+ex. We start with reviewing the results presented so far.

Proposition 5.2.9 (previous results extend). *The results from Chapter 4 and from this chapter so far extend from* PNA *to* PNA^+ *(as well as to* PNA+the *and* PNA+ex*), when the definitions are changed in accordance to Definition 5.2.8. The only exception is Lemma 4.4.22, which needs to be replaced by Lemma 5.2.6.*

Proof. We have to repeat every proof with the extended definitions. Most of them are straightforward updates of the old proofs. With the help of Lemmas 5.2.2 and 5.2.4 we can show that $[\![\mathtt{the}\, x.\, e]\!]$ and $[\![\mathtt{ex}\, x.\, e]\!]$ are well-defined and uniform-continuous functions, so this shows Proposition 4.3.4 and Lemma 4.3.5. Equivariance of evaluation (Lemma 4.4.10) needs (4.2) and Lemma 2.3.12. Evaluation still does not create fresh names (so Lemma 4.4.11 still holds) because $\mathtt{the}\, x.\, e \Downarrow a$ implies $a \in \mathrm{fn}\, e$ as we saw in the proof of Lemma 5.2.3.

The proofs that required Lemma 4.4.22 do now work with Lemma 5.2.6. For Lemmas 4.4.24 and 4.4.25 we additionally need Lemma 5.2.3 and 5.2.5. □

5.2.4 Further syntactic sugar

The parallel nature of existential quantification over names (Section 5.2.2) allows us to express several other parallel constructs that are already present in classical work on PCF.

The parallel-or construct $e\ \mathtt{por}\ e'$ is central for PCF (see Plotkin [47, Table 1]), because PCF fails to be fully abstract without parallel-or, but becomes full abstract with it. Parallel-or is a boolean construct that can evaluate both expressions in parallel and is true when one of the expressions evaluates to true, no matter if the other one diverges. Its denotational semantics satisfies

$$[\![e\ \mathtt{por}\ e']\!]\rho = \begin{cases} \mathrm{true} & \text{if } [\![e]\!]\rho = \mathrm{true} \text{ or } [\![e']\!]\rho = \mathrm{true} \\ \mathrm{false} & \text{if } [\![e]\!]\rho = \mathrm{false} \text{ and } [\![e']\!]\rho = \mathrm{true} \\ \bot & \text{otherwise}\,. \end{cases}$$

Contrast this with the or-construct $e\ \mathtt{or}\ e'$ in (4.5), which needs to evaluate e first and hence always diverges if e does so. In the presence of $\mathtt{ex}\, x.\, e$ we can define parallel-or as syntactic sugar by

$$e\ \mathtt{por}\ e' \triangleq \nu a.\nu a'.\mathtt{ex}\, x.\ \mathtt{if}\ x = a\ \mathtt{then}\ e\ \mathtt{else}\ \mathtt{if}\ x = a'\ \mathtt{then}\ e'\ \mathtt{else}\ \mathtt{F}$$

where $a, a' \# e, e'$ are arbitrary fresh names. The purpose of the name restrictions is to ensure that $\mathrm{fn}(e \,\mathtt{por}\, e') = \mathrm{fn}\, e \cup \mathrm{fn}\, e'$.

A parallel-and construct can now be defined easily by

$$e \,\mathtt{pand}\, e' \triangleq \mathtt{not}\,((\mathtt{not}\; e)\,\mathtt{por}\,(\mathtt{not}\; e')).$$

We can also define a parallel-if for natural numbers (as in PCF) satisfying

$$[\![\mathtt{pif}_{\mathtt{nat}}\, e_1 \,\mathtt{then}\, e_2 \,\mathtt{else}\, e_3]\!]\rho \triangleq \begin{cases} [\![e_2]\!]\rho & \text{if } [\![e_1]\!]\rho = \text{true or } [\![e_2]\!]\rho = n = [\![e_3]\!]\rho \\ [\![e_3]\!]\rho & \text{if } [\![e_1]\!]\rho = \text{false} \\ \bot & \text{otherwise} \end{cases} \tag{5.18}$$

by using parallel-or as follows (see also Streicher [61, Lemma 13.13])

$$\begin{aligned} &\mathtt{pif}_{\mathtt{nat}}\, e_1 \,\mathtt{then}\, e_2 \,\mathtt{else}\, e_3 \triangleq \\ &(\mathtt{fix}\,\lambda(s : (\mathtt{nat} \to (\mathtt{nat} \to \mathtt{nat}) \to \mathtt{nat}) \to \mathtt{nat} \to (\mathtt{nat} \to \mathtt{nat}) \to \mathtt{nat}) \to \\ &\lambda x : \mathtt{nat} \to \lambda(f : \mathtt{nat} \to \mathtt{nat}) \to \mathtt{if}\,\mathtt{zero}\,(f\; x)\,\mathtt{then}\; x\,\mathtt{else}\,(s\,(\mathtt{S}\,x)\,f))\,\mathtt{0}\,(\lambda y : \mathtt{nat} \to \\ &((y =_{\mathtt{nat}} e_2)\,\mathtt{pand}\,(y =_{\mathtt{nat}} e_3))\,\mathtt{por}\,((y =_{\mathtt{nat}} e_2)\,\mathtt{pand}\, e_1)\,\mathtt{por}\,((y =_{\mathtt{nat}} e_3)\,\mathtt{pand}\,(\mathtt{not}\; e_1))). \end{aligned}$$

Thus $\mathtt{pif}_{\mathtt{nat}}\, e_1 \,\mathtt{then}\, e_2 \,\mathtt{else}\, e_3$ can evaluate to a natural number n even if e_1 diverges, as long as e_2 and e_3 both evaluate to n.

5.3 Computational adequacy

This section is about proving that two expressions are contextually equivalent if they are denotationally equal. This property is called computational adequacy and the formal definition is the statement of Theorem 5.3.19. With that as a basis, we can furthermore prove extensionality results for the contextual preorder in Section 5.3.4.

The results of this section are formulated for PNA^+, but they all specialise to PNA, PNA+the and PNA+ex.

5.3.1 Logical relation

As in the classical proof for PCF [47, Theorem 3.1], we prove computational adequacy by devising a suitable logical relation between the syntax and denotational semantics of PNA^+. See also Streicher [61, Chapter 4] for a good exposition of this method for PCF.

Definition 5.3.1 (logical relation). We define the *logical relation*

$$\lhd_\tau \subseteq [\![\tau]\!] \times \mathrm{Exp}^{\mathsf{PNA}^+}(\tau)$$

for PNA^+ by recursion on the structure of $\tau \in \mathrm{Typ}^{\mathsf{PNA}^+}$:

$$\begin{aligned}
d \lhd_\gamma e &\triangleq (d = \bot) \vee (\exists c \in \mathrm{Can}^{\mathsf{PNA}^+})\, e \Downarrow c \wedge [\![c]\!] = d \qquad \text{for } \gamma \in \mathrm{Gnd}^{\mathsf{PNA}^+} \\
d \lhd_{\tau_1 \times \tau_2} e &\triangleq \mathit{proj}_1\, d \lhd_{\tau_1} \mathtt{fst}\, e \wedge \mathit{proj}_2\, d \lhd_{\tau_2} \mathtt{snd}\, e \\
d \lhd_{\tau_1 \to \tau_2} e &\triangleq (\forall d_1 \in [\![\tau_1]\!], e_1 \in \mathrm{Exp}^{\mathsf{PNA}^+}(\tau_1))\, d_1 \lhd_{\tau_1} e_1 \Rightarrow d\, d_1 \lhd_{\tau_2} e\, e_1 \\
d \lhd_{\delta\, \tau} e &\triangleq (\mathit{N}a)\, d @ a \lhd_\tau e @ a\,.
\end{aligned}$$

This definition is standard except for the last clause, which is for name abstraction types, in which we use the freshness quantifier $(\mathit{N}a)$ from Definition 2.3.16. Thus by Lemma 2.3.17 $d \lhd_{\delta\,\tau} e$ holds if and only if $d @ a \lhd_\tau e @ a$ holds for some $a \# d, e$, or equivalently, for any $a \# d, e$.

Lemma 5.3.2 (**equivariance of the logical relation**). *The logical relation satisfies the following equivariance property:*

$$(\forall \pi \in \mathrm{Perm}(\mathbb{A}))\, d \lhd_\tau e \;\Rightarrow\; \pi \cdot d \lhd_\tau \pi \cdot e\,.$$

Proof. We proceed by induction on $\tau \in \mathrm{Typ}^{\mathsf{PNA}^+}$. For the base case $\tau \in \mathrm{Gnd}^{\mathsf{PNA}^+}$ we use Lemma 3.1.5 for $d = \bot$, and for $d \neq \bot$ we use Lemma 4.4.10 and the Equivariance Lemma 4.3.5. The inductive steps follow from the definition with equivariance of the functions $\mathit{proj}_1, \mathit{proj}_2, @$ and ev, see the Characterisation Lemma 4.3.3. □

The next lemma is needed to prove the fundamental property of the logical relation for the fixed point recursion construct $\mathtt{fix}\, e$.

Lemma 5.3.3 (**Scott-admissibility**). *The logical relation is Scott-admissible, in the sense that for each type $\tau \in \mathrm{Typ}$ and expression $e \in \mathrm{Exp}(\tau)$ the following holds:*

- $\bot \lhd_\tau e$.
- $\{d \mid d \lhd_\tau e\}$ *is closed under uniform-directed joins, meaning that* $S \subseteq \{d \mid d \lhd_\tau e\} \Rightarrow \bigsqcup S \in \{d \mid d \lhd_\tau e\}$ *holds for all uniform-directed* $S \subseteq [\![\tau]\!]$.

Proof. By structural induction over τ. The proof uses the definitions of bottom elements and joins in $[\![\tau]\!]$, where for $\tau = \delta\, \tau'$ bottom is given by $\langle a \rangle \bot$ as in Proposition 3.1.6 and joins are given as in (A.1). □

We extend the logical relation to typing environments and open expressions.

Definition 5.3.4 (**extended logical relation**). For every typing environment $\Gamma \in \mathrm{Env}^{\mathsf{PNA}^+}$, define the relation $\lhd_\Gamma \subseteq [\![\Gamma]\!] \times \mathrm{Subst}^{\mathsf{PNA}^+}(\Gamma)$ between Γ-valuations (see Section 4.3.1) and Γ-substitutions (see Definition 5.1.8) by

$$\rho \lhd_\Gamma s \;\triangleq\; (\forall x \in \mathrm{dom}\, \Gamma)\, \rho\, x \lhd_{\Gamma x} s\, x\,.$$

This allows us to extend the logical relation to open expressions

$$\lhd_{\Gamma|\tau} \subseteq ([\![\Gamma]\!] \to_{\mathrm{uc}} [\![\tau]\!]) \times \{e \in \mathrm{Exp}^{\mathsf{PNA}^+} \mid \Gamma \vdash e : \tau\}$$

as follows

$$d \lhd_{\Gamma|\tau} e \;\triangleq\; (\forall \rho \in [\![\Gamma]\!])(\forall s \in \mathrm{Subst}^{\mathsf{PNA}^+}(\Gamma))\, \rho \lhd_\Gamma s \;\Rightarrow\; d\, \rho \lhd_\tau e\, s\,.$$

The above definitions allow us to formulate our main tool for proving computational adequacy: the *fundamental property of the logical relation*. Its definition is the statement of Proposition 5.3.5 and it will be proved in Section 5.3.3.

Proposition 5.3.5 (**fundamental property**). *Every* $\Gamma \vdash e : \tau$ *satisfies* $[\![e]\!] \lhd_{\Gamma|\tau} e$.

5.3.2 Kleene preorders

Usually proofs of the fundamental property (Proposition 5.3.5) work by using the so-called 'Kleene preorder'. It is often easy to check if two expressions are in Kleene preorder and this makes it a good tool for forthcoming proofs.

Definition 5.3.6 (**Kleene preorder**). For any two variable-closed expressions $e, e' \in \mathrm{Exp}^{\mathsf{PNA}^+}(\tau)$ of some type $\tau \in \mathrm{Typ}^{\mathsf{PNA}^+}$, the *Kleene preorder* is defined by

$$e \mathrel{\overset{k}{\leqq}} e' \triangleq (\forall c \in \mathrm{Can}^{\mathsf{PNA}^+})\, e \Downarrow c \Rightarrow e' \Downarrow c\,.$$

The corresponding equivalence relation is called *Kleene equivalence* and written as $e \mathrel{\overset{k}{=}} e'$.

The Kleene preorder is strictly stronger than the contextual preorder. It being stronger $e \mathrel{\overset{k}{\leqq}} e' \Rightarrow e \lesssim_{\mathsf{PNA}^+} e'$ is a consequence of Lemma 5.3.11 and Propositions 5.3.5, 5.3.14 and 5.3.24. It is strictly stronger, because for example the canonical forms $\lambda x : \mathtt{nat} \to \mathtt{pred}(\mathtt{S}\, x)$ and $\lambda x : \mathtt{nat} \to x$ are contextually equivalent but not Kleene equivalent. Yet at ground types the two preorders coincide as the next lemma shows.

Lemma 5.3.7 (**ground Kleene**). *At ground types, contextual preorder implies Kleene preorder. For all* $\gamma \in \mathrm{Gnd}^{\mathsf{PNA}^+}$ *we have*

$$\emptyset \vdash e_1 \lesssim_{\mathsf{PNA}^+} e_2 : \gamma \;\Rightarrow\; e_1 \mathrel{\overset{k}{\leqq}} e_2\,.$$

Proof. Consider the property that for every $c \in \mathrm{Can}^{\mathsf{PNA}^+}(\gamma)$ there is a context $C_c[-] : (\emptyset \triangleright \gamma) \rightsquigarrow (\emptyset \triangleright \mathtt{bool})$ such that

$$(\forall e \in \mathrm{Exp}^{\mathsf{PNA}^+}(\gamma))\, e \Downarrow c \;\Leftrightarrow\; C_c[e] \Downarrow \mathtt{T}\,. \tag{5.19}$$

If this holds then by $\emptyset \vdash e_1 \lesssim_{\mathsf{PNA}^+} e_2 : \gamma$ we know for any $c \in \mathrm{Can}^{\mathsf{PNA}^+}(\gamma)$ that $C_c[e_1] \Downarrow \mathtt{T} \Rightarrow C_c[e_2] \Downarrow \mathtt{T}$, which then by (5.19) leads to $e_1 \Downarrow c \Rightarrow e_2 \Downarrow c$. We can define $C_c[-]$ by induction on the structure of c as follows, where we use Lemma 4.2.8 implicitly in the definition:

$$\begin{aligned}
C_{\mathtt{T}}[-] &\triangleq [-] \\
C_{\mathtt{F}}[-] &\triangleq \mathtt{not}\ [-] \\
C_{\mathtt{S}^n\mathtt{0}}[-] &\triangleq [-] =_{\mathtt{nat}} \mathtt{S}^n\mathtt{0} \\
C_a[-] &\triangleq [-] = a \\
C_{\mathtt{V}a}[-] &\triangleq \mathtt{case}\,[-]\,\mathtt{of}\,(\mathtt{V}\,x_1 \to C_a[x_1] \mid \mathtt{A}\,x_2\,x_2' \to \mathtt{F} \mid \mathtt{L}\,x_3 \to \mathtt{F}) \\
C_{\mathtt{A}c_1c_2}[-] &\triangleq \mathtt{case}\,[-]\,\mathtt{of}\,(\mathtt{V}\,x_1 \to \mathtt{F} \mid \mathtt{A}\,x_2\,x_2' \to C_{c_1}[x_2]\ \mathtt{and}\ C_{c_2}[x_2'] \mid \mathtt{L}\,x_3 \to \mathtt{F}) \\
C_{\mathtt{L}\alpha a.c}[-] &\triangleq \mathtt{case}\,[-]\,\mathtt{of}\,(\mathtt{V}\,x_1 \to \mathtt{F} \mid \mathtt{A}\,x_2\,x_2' \to \mathtt{F} \mid \mathtt{L}\,x_3 \to \nu a.\, C_c[x_3\ @\ a])\,.
\end{aligned}$$

The definition includes some of the syntactic sugar of Section 4.5.1. The proof of (5.19) is done by structural induction on c, it uses Lemma 4.4.7 for $C_{\mathrm{L}\alpha a.c}[-]$. □

In our nominal setting, the Kleene preorder is too strong for proving the fundamental property. For example, we would need $\nu a.\nu b.e \leq^{k} \nu b.\nu a.e$ to hold. However, this property fails already for $e = (a\,, a)$, because $\nu a.\nu b.a$ and $\nu b.\nu a.a$ are not α-equivalent and hence are syntactically different. To overcome these issues, we develop a notion of weak Kleene preorder, which will be suited just fine for our proof of the fundamental property.

Definition 5.3.8 (**weak Kleene preorder**). For every type $\tau \in \mathrm{Typ}^{\mathsf{PNA}^+}$, define relations

$$\begin{aligned} glswk-leq-pnap_{\tau} &\subseteq \mathrm{Exp}^{\mathsf{PNA}^+}(\tau) \times \mathrm{Exp}^{\mathsf{PNA}^+}(\tau) \\ \leq^{\mathrm{wk}}{}_{\tau} &\subseteq \mathrm{Can}^{\mathsf{PNA}^+}(\tau) \times \mathrm{Can}^{\mathsf{PNA}^+}(\tau) \end{aligned}$$

by simultaneous structural recursion over τ:

$$\begin{aligned} e \leq^{\mathrm{wk}}_{\tau} e' &\triangleq (\forall c \in \mathrm{Can}^{\mathsf{PNA}^+})\, e \Downarrow c \;\Rightarrow\; (\exists c' \in \mathrm{Can}^{\mathsf{PNA}^+})\, e' \Downarrow c' \;\wedge\; c \leq^{\mathrm{wk}}_{\tau} c' \\ c \leq^{\mathrm{wk}}_{\gamma} c' &\triangleq c = c' \qquad \text{for } \gamma \in \mathrm{Gnd}^{\mathsf{PNA}^+} \\ c \leq^{\mathrm{wk}}_{\tau_1 \times \tau_2} c' &\triangleq (\exists e_1, e_2, e_1', e_2' \in \mathrm{Exp}^{\mathsf{PNA}^+})\, c = (e_1\,, e_2) \;\wedge\; c' = (e_1'\,, e_2') \\ &\quad \wedge\; e_1 \leq^{\mathrm{wk}}_{\tau_1} e_1' \;\wedge\; e_2 \leq^{\mathrm{wk}}_{\tau_2} e_2' \\ c \leq^{\mathrm{wk}}_{\tau_1 \to \tau_2} c' &\triangleq (\exists x \in \mathbb{V})(\exists e, e' \in \mathrm{Exp}^{\mathsf{PNA}^+})\, c = \lambda x : \tau_1 \to e \;\wedge\; c' = \lambda x : \tau_1 \to e' \\ &\quad \wedge\; (\forall e_1 \in \mathrm{Exp}^{\mathsf{PNA}^+}(\tau_1))\, e[e_1/x] \leq^{\mathrm{wk}}_{\tau_2} e'[e_1/x] \\ c \leq^{\mathrm{wk}}_{\delta\,\tau} c' &\triangleq (\mathit{Ma})(\exists c_1, c_1' \in \mathrm{Can}^{\mathsf{PNA}^+})\, c = \alpha a.c_1 \;\wedge\; c' = \alpha a.c_1' \;\wedge\; c_1 \leq^{\mathrm{wk}}_{\tau} c_1'. \end{aligned}$$

We call $\leq^{\mathrm{wk}}_{\tau}$ the *weak Kleene preorder* and call the corresponding equivalence relation $=^{\mathrm{wk}}{}_{\tau}$ the *weak Kleene equivalence*.

The following three lemmas establish some basic properties of the weak Kleene preorder.

Lemma 5.3.9 (**weak Kleene is preorder**). *The relations $\leq^{\mathrm{wk}}_{\tau}$ and $\leq^{\mathrm{wk}}_{\tau}$ are indeed preorders.*

Proof. Reflexivity and transitivity of $\leq^{\mathrm{wk}}_{\tau}$ imply those properties for $\leq^{\mathrm{wk}}_{\tau}$, at every type τ. Using that, we can prove these properties for $\leq^{\mathrm{wk}}_{\tau}$ by induction on τ, where we need Lemma 4.2.8 for the reflexivity part. □

Lemma 5.3.10 (**equivariance of weak Kleene**). *The preorders $\leq^{\mathrm{wk}}_{\tau}$ and $\leq^{\mathrm{wk}}_{\tau}$ are equivariant. For every $\pi \in \mathrm{Perm}(\mathbb{A})$ we have*

$$\begin{aligned} e \leq^{\mathrm{wk}}_{\tau} e' &\;\Rightarrow\; \pi \cdot e \leq^{\mathrm{wk}}_{\tau} \pi \cdot e' \\ c \leq^{\mathrm{wk}}_{\tau} c' &\;\Rightarrow\; \pi \cdot c \leq^{\mathrm{wk}}_{\tau} \pi \cdot c'. \end{aligned}$$

Proof. By simultaneous structural induction on τ, using Lemmas 4.4.10 and 4.1.3. □

Lemma 5.3.11 (**Kleene stronger than weak Kleene**). *For all* $e, e' \in \mathrm{Exp}^{\mathsf{PNA}^+}(\tau)$ *it holds that*

$$e \leqq^{k} e' \;\Rightarrow\; e \leqq^{\mathrm{wk}}_{\tau} e' .$$

Proof. Follows directly from the definitions, with reflexivity of $\leqq^{\mathrm{wk}}_{\tau}$ and $\leq^{\mathrm{wk}}_{\tau}$ from Lemma 5.3.9. □

What follows are two technical lemmas that relate the weak Kleene preorder with various PNA constructs. They lead to Proposition 5.3.14, which says that the logical relation is closed under composition with the weak Kleene preorder.

Lemma 5.3.12. *The projections and function application preserve the weak Kleene preorder:*

$$e \leqq^{\mathrm{wk}}_{\tau_1 \times \tau_2} e' \;\Rightarrow\; \mathtt{fst}\, e \leqq^{\mathrm{wk}}_{\tau_1} \mathtt{fst}\, e' \;\wedge\; \mathtt{snd}\, e \leqq^{\mathrm{wk}}_{\tau_2} \mathtt{snd}\, e' \tag{5.20}$$

$$e \leqq^{\mathrm{wk}}_{\tau_1 \to \tau_2} e' \;\Rightarrow\; (\forall e_1 \in \mathrm{Exp}(\tau_1))\, e\, e_1 \leqq^{\mathrm{wk}}_{\tau_2} e\, e_1 \tag{5.21}$$

Proof. Both properties follow directly from the definition of $\leqq^{\mathrm{wk}}$. □

Lemma 5.3.13. *The weak Kleene preorder is preserved by local scoping and concretion at all names:*

$$c \leq^{\mathrm{wk}}_{\tau} c' \;\wedge\; a \backslash\backslash c := c_1 \;\Rightarrow\; (\exists c_1' \in \mathrm{Can}^{\mathsf{PNA}^+})\, a \backslash\backslash c' := c_1' \;\wedge\; c_1 \leq^{\mathrm{wk}}_{\tau} c_1' \tag{5.22}$$

$$e \leqq^{\mathrm{wk}}_{\tau} e' \;\Rightarrow\; (\forall a \in \mathbb{A})\, \nu a.\, e \leqq^{\mathrm{wk}}_{\tau} \nu a.\, e' \tag{5.23}$$

$$e \leqq^{\mathrm{wk}}_{\delta\, \tau} e' \;\Rightarrow\; (\forall a \in \mathbb{A})\, e\, @a \leqq^{\mathrm{wk}}_{\tau} e'\, @a\, . \tag{5.24}$$

Proof. At every $\tau \in \mathrm{Typ}^{\mathsf{PNA}^+}$, (5.23) is implied by (5.22) with the definition of $\leqq^{\mathrm{wk}}$. We use that to prove (5.22) by induction on the structure of τ. Property (5.24) also follows from (5.22), by using also Lemma 5.3.10. □

Proposition 5.3.14 (**closure under weak Kleene**). *The logical relation is closed under composition with the weak Kleene preorder.*

$$d \lhd_{\tau} e \;\wedge\; e \leqq^{\mathrm{wk}}_{\tau} e' \;\Rightarrow\; d \lhd_{\tau} e'$$

Proof. We proceed by structural induction on τ, where we need (5.20) for product types, (5.21) for function types and (5.24) for abstraction types. □

In the next two lemmas we further connect the weak Kleene preorder with name restriction and concretion. They allow us to prove the main result of this section (Proposition 5.3.17): Name restriction, abstraction and concretion preserve the logical relation for all atomic names.

Lemma 5.3.15. *The following structural properties for name restriction hold:*

$$a'\backslash\backslash c := c_1 \wedge a\backslash\backslash c_1 := c_2 \Rightarrow (\exists c_1', c_2' \in \mathrm{Can}^{\mathsf{PNA}^+})\, a\backslash\backslash c := c_1' \wedge a'\backslash\backslash c_1' := c_2' \wedge c_2 \leq_\tau^{\mathrm{wk}} c_2' \tag{5.25}$$

$$\nu a.\nu a'.e \leqq_\tau^{\mathrm{wk}} \nu a'.\nu a.e \tag{5.26}$$

$$a \# c \Rightarrow (\exists c' \in \mathrm{Can}^{\mathsf{PNA}^+})\, a\backslash\backslash c := c' \wedge c \leq_\tau^{\mathrm{wk}} c' \wedge c' \leq_\tau^{\mathrm{wk}} c \tag{5.27}$$

$$a \# e \Rightarrow \nu a.e \leqq_\tau^{\mathrm{wk}} e \wedge e \leqq_\tau^{\mathrm{wk}} \nu a.e \tag{5.28}$$

$$a \neq a' \Rightarrow \nu a.(e\, @a') \leqq_\tau^{\mathrm{wk}} (\nu a.e)\, @a' \tag{5.29}$$

$$\nu a.\mathtt{fst}\, e \leqq_\tau^{\mathrm{wk}} \mathtt{fst}\, (\nu a.e) \tag{5.30}$$

$$\nu a.\mathtt{snd}\, e \leqq_\tau^{\mathrm{wk}} \mathtt{snd}\, (\nu a.e) \tag{5.31}$$

$$a \# e_1 \Rightarrow \nu a.(e\, e_1) \leqq_\tau^{\mathrm{wk}} (\nu a.e)\, e_1\,. \tag{5.32}$$

Proof. At every $\tau \in \mathrm{Typ}^{\mathsf{PNA}^+}$, (5.26) and (5.28) are implied by (5.25) and (5.27) respectively, which follows directly from the definition of $\leqq_\tau^{\mathrm{wk}}$. We use these implications to prove (5.25) and (5.27) by induction on the structure of τ, where we apply Lemma 4.4.7 at the base cases when $\tau \in \mathrm{Gnd}^{\mathsf{PNA}^+}$ as well as Lemma 4.4.7 for the other cases. Property (5.29) follows from (5.25) and Lemma 5.3.10. Properties (5.30), (5.31) and (5.32) are immediate consequences of Lemma 5.3.11. □

Lemma 5.3.16. *The following properties of name concretions hold:*

$$e \leqq_\tau^{\mathrm{wk}} (\alpha a.e)\, @a \tag{5.33}$$

$$a' \# (a, e) \Rightarrow \nu a'.(a\, a') \cdot (e\, @a') \leqq_\tau^{\mathrm{wk}} e\, @a\,. \tag{5.34}$$

Proof. Given in Appendix A.4. □

Proposition 5.3.17 (**closure properties**). *The logical relation is closed under name restriction, abstraction and concretion.*

$$d \lhd_\tau e \Rightarrow (\forall a \in \mathbb{A})\, a\backslash d \lhd_\tau \nu a.e \tag{5.35}$$

$$d \lhd_\tau e \Rightarrow (\forall a \in \mathbb{A})\, \langle a\rangle d \lhd_{\delta\,\tau} \alpha a.e \tag{5.36}$$

$$d \lhd_{\delta\,\tau} e \Rightarrow (\forall a \in \mathbb{A})\, d @^t a \lhd_\tau e\, @a \tag{5.37}$$

Proof. For (5.35) we apply Proposition 5.3.14 in a structural induction on τ, using Lemma 4.3.11 and Lemma 4.4.7 for ground types, (5.29) for name-abstraction types, (5.30) and (5.31) for product types, as well as (5.32) and Lemma 5.3.2 for function types.

Property (5.36) follows from Proposition Lemma 5.3.2, 5.3.14 and (5.33).

Finally, for (5.37), suppose $d \lhd_{\delta\,\tau} e$. Given any $a \in \mathbb{A}$, pick $a' \# (a, d, e)$. Then putting $d' \triangleq d\, @a'$, we have $d = \langle a'\rangle d'$ and $d @^t a = a'\backslash(a\, a') \cdot d'$. By definition of $\lhd_{\delta\,\tau}$ we have $d' \lhd_\tau e\, @a'$ and hence by Lemma 5.3.2 have $(a\, a') \cdot d' \lhd_\tau (a\, a') \cdot (e\, @a')$. So by (5.35) it holds that

$$d @^t a = a'\backslash(a\, a') \cdot d' \lhd_\tau \nu a'.(a\, a') \cdot (e\, @a')\,.$$

Applying (5.34) and Proposition 5.3.14 to this gives $d @^t a \lhd_\tau e\, @a$. □

5.3.3 Proving computational adequacy

The development in the Section 5.3.2, in particular Proposition 5.3.17, allows us to prove the fundamental property of the logical relation (Proposition 5.3.5), which will directly lead to a proof of computational adequacy.

Proof of the fundamental property, Proposition 5.3.5. We apply rule induction on the typing judgement $\Gamma \vdash e : \tau$, where we use Lemma 5.3.11 together with Proposition 5.3.14. The $(\nu a.\, e)$-case additionally uses (5.35), the $(\alpha a.\, e)$-case uses (5.36) and the $(e_1 \mathbin{@} e_2)$-case uses (5.37). The case for $(e_1 \rightleftharpoons e_2)\, e_3$ works because of Lemma 5.3.10. The second bullet point of Lemma 5.3.3 is needed for fixed points $\mathtt{fix}\, e$ and the first bullet point is needed for several other cases. The cases for $\mathtt{the}\, x.\, e$ and $\mathtt{ex}\, x.\, e$ do not cause additional complications. □

Corollary 5.3.18 (**adequacy at** true)**.** *For all* $e \in \mathrm{Exp}^{\mathsf{PNA}^+}(\mathtt{bool})$ *it holds that*

$$[\![e]\!] = \mathrm{true} \;\Rightarrow\; e \Downarrow \mathtt{T}\,.$$

Proof. Proposition 5.3.5 gives us $[\![e]\!] \lhd_{\mathtt{bool}} e$ and then the property follows directly from the definition of $\lhd_{\mathtt{bool}}$ (Definition 5.3.1). □

We are now in the position of being able to prove one of the major results of this thesis: computational adequacy holds for PNA^+ (and hence also for PNA, PNA+the and PNA+ex as the results specialise).

Theorem 5.3.19 (**computational adequacy**)**.** *Given* $\Gamma \vdash e : \tau$ *and* $\Gamma \vdash e' : \tau$*, then*

$$[\![e]\!] \sqsubseteq [\![e']\!] \in [\![\Gamma]\!] \to_{\mathrm{uc}} [\![\tau]\!] \;\Rightarrow\; \Gamma \vdash e \lesssim_{\mathsf{PNA}^+} e' : \tau\,.$$

Consequently, if $[\![e]\!] = [\![e']\!]$*, then* $\Gamma \vdash e \cong_{\mathsf{PNA}^+} e' : \tau$*.*

Proof. To prove $\Gamma \vdash e \lesssim_{\mathsf{PNA}^+} e' : \tau$ (see Definitions 5.1.1 and 5.2.8), let any context $C : (\Gamma \triangleright \tau) \rightsquigarrow (\emptyset \triangleright \mathtt{bool})$ be given, then

$$\begin{aligned}
C[e] \Downarrow \mathtt{T} &\Rightarrow [\![C[e]]\!] = \mathrm{true} && \text{(by soundness, Proposition 4.4.16)}\\
&\Rightarrow [\![C[e']]\!] = \mathrm{true} && \text{(by compositionality, Lemma 4.3.7)}\\
&\Rightarrow C[e'] \Downarrow \mathtt{T} && \text{(by adequacy at true, Corollary 5.3.18)}
\end{aligned}$$

holds. □

With computational adequacy we can prove many contextual equivalences in PNA in a straightforward manner via the denotational semantics.

Example 5.3.20 (**proofs by computational adequacy**)**.** The contextual equivalences of (5.1), (5.2) and (5.3) are easy to prove through computational adequacy. We prove

(5.2) in detail. Since we identify expressions up to α-equivalence, for any given $a' \in \mathbb{A}$ we can pick a representative expression $\nu a.\, \lambda x : \mathtt{name} \to (x = a)$ such that $a \neq a'$, then

$$
\begin{aligned}
&[\![\nu a.\, \lambda x : \mathtt{name} \to (x = a)]\!]\, a' \\
&= (a \backslash [\![\lambda x : \mathtt{name} \to (x = a)]\!])\, a' && \text{using the definition in Figure 4.11} \\
&= a \backslash ([\![\lambda x : \mathtt{name} \to (x = a)]\!]\, a') && \text{by (3.30), since } a \neq a' \\
&= a \backslash \mathrm{false} && \text{as } a \neq a' \\
&= \mathrm{false} && \text{by (3.29)} \\
&= [\![\lambda x : \mathtt{name} \to \mathtt{if}\ x = x\ \mathtt{then}\ \mathtt{F}\ \mathtt{else}\ \mathtt{F}]\!]\, a'.
\end{aligned}
$$

Similarly $[\![\nu a.\, \lambda x : \mathtt{name} \to (x = a)]\!]\, \bot = \bot = [\![\lambda x : \mathtt{name} \to \mathtt{if}\ x = x\ \mathtt{then}\ \mathtt{F}\ \mathtt{else}\ \mathtt{F}]\!]\, \bot$. Hence $[\![\nu a.\, \lambda x : \mathtt{name} \to (x = a)]\!] = [\![\lambda x : \mathtt{name} \to \mathtt{if}\ x = x\ \mathtt{then}\ \mathtt{F}\ \mathtt{else}\ \mathtt{F}]\!]$ and so (5.2) holds by Theorem 5.3.19.

To prove example (5.3) one can combine the definition of the denotational semantics (Figure 4.11) with the fact that if $a, a' \# f \in (\mathbb{A}_\bot \to_{\mathrm{uc}} \mathbb{B}_\bot) \to_{\mathrm{uc}} \mathbb{B}_\bot$, then $f\, a = f\, ((a\, a') \cdot a') = (a\, a') \cdot (f\, a') = f\, a'$ (since $f\, a' \in \mathbb{B}_\bot$).

The next lemma is useful for Appendix A.5.

Lemma 5.3.21 (bottom substitution). *For any $\Gamma, x : \tau \vdash e : \tau'$, $\Gamma \vdash e_1 : \tau$ and $\Gamma \vdash e_2 : \tau$ it holds that*

$$[\![e_1]\!] = \bot \;\Rightarrow\; \Gamma \vdash e[e_1/x] \lesssim_{\mathsf{PNA}^+} e[e_2/x] : \tau'.$$

Proof. We know that $[\![e_1]\!] \sqsubseteq [\![e_2]\!]$, because $[\![e_1]\!] = \bot$. Then for any Γ-valuation $\rho \in [\![\Gamma]\!]$ we have $\rho[x \mapsto [\![e_1]\!]] \sqsubseteq \rho[x \mapsto [\![e_2]\!]]$ and hence $[\![e[e_1/x]]\!] = \lambda\rho \in [\![\Gamma]\!] \to [\![e]\!]\rho[x \mapsto [\![e_1]\!]] \sqsubseteq \lambda\rho \in [\![\Gamma]\!] \to [\![e]\!]\rho[x \mapsto [\![e_2]\!]] = [\![e[e_2/x]]\!]$. The computational adequacy property (Theorem 5.3.19) then gives us $\Gamma \vdash e[e_1/x] \lesssim_{\mathsf{PNA}^+} e[e_2/x] : \tau'$ as desired. □

5.3.4 Extensionality

Another consequence of the fundamental property of the logical relation (Proposition 5.3.5) is that the contextual preorder satisfies certain extensionality properties that are listed in Theorem 5.3.25. Before we can prove that, we need to establish some results that closely connect the logical relation with the contextual preorder.

Lemma 5.3.22 (compatible logical relation). *The logical relation is compatible in the sense of* (5.17)*, so for any $\Gamma \vdash e_1 : \tau$, $\Gamma \vdash e_2 : \tau$ and $C : (\Gamma \triangleright \tau) \rightsquigarrow (\Gamma' \triangleright \tau')$ it holds that*

$$[\![e_1]\!] \lhd_{\Gamma|\tau} e_2 \;\Rightarrow\; [\![C[e_1]]\!] \lhd_{\Gamma'|\tau'} C[e_2].$$

Proof. By rule induction over $C : (\Gamma \triangleright \tau) \rightsquigarrow (\Gamma' \triangleright \tau')$, where we frequently use the fundamental property (Proposition 5.3.5) for subexpressions of contexts. The rest of the proof is then very similar to the proof of the fundamental property as it is given at the start of Section 5.3.3. All propositions and lemmas that are given in the proof of the fundamental property are also needed for this proof. □

Lemma 5.3.23 (**closure under contextual preorder**). *The logical relation is closed under composition with the contextual preorder. It holds that*

$$d \lhd_{\Gamma|\tau} e \ \wedge\ \Gamma \vdash e \lesssim_{\mathsf{PNA}^+} e' : \tau \ \Rightarrow\ d \lhd_{\Gamma|\tau} e' .$$

Proof. We first show that the statement holds for closed expressions

$$d \lhd_{\tau} e \ \wedge\ \emptyset \vdash e \lesssim_{\mathsf{PNA}^+} e' : \tau \ \Rightarrow\ d \lhd_{\tau} e' \tag{5.38}$$

by induction on τ. For ground types we need Lemma 5.3.7 and the inductive steps use that $\lesssim_{\mathsf{PNA}^+}$ is compatible (5.17) as proved in Proposition 5.1.7.

For proving the statement for open expressions let $\rho \lhd_{\Gamma} s$ be given, this leads to $d\,\rho \lhd_{\tau} e\,s$. By Lemma 5.1.9 we know $\emptyset \vdash e\,s \lesssim_{\mathsf{PNA}^+} e'\,s : \tau$, and hence we can apply (5.38) to get $d\,\rho \lhd_{\tau} e'\,s$, which was to show for $d \lhd_{\Gamma|\tau} e'$. □

Proposition 5.3.24 (**contextual preorder through the logical relation**). *The contextual preorder coincides exactly with the logical relation as follows:*

$$\Gamma \vdash e_1 \lesssim_{\mathsf{PNA}^+} e_2 : \tau \ \Leftrightarrow\ [\![e_1]\!] \lhd_{\Gamma|\tau} e_2 .$$

Proof. By the fundamental property (Proposition 5.3.5) we know $[\![e_1]\!] \lhd_{\Gamma|\tau} e_1$ and the left-to-right direction follows then directly by Lemma 5.3.23. The logical relation interpreted as type-respecting binary relation is compatible by Lemma 5.3.22 and preadequate by Proposition 4.4.16 and the definition of $\lhd_{\mathtt{bool}}$. Therefore the right-to-left direction is a consequence of Proposition 5.1.7. □

The main contribution of this section are several extensionality properties. They can be applied to prove instances of the contextual preorder relation of variable-closed expressions, without relying on the denotational semantics.

Theorem 5.3.25 (**extensionality**). *The contextual preorder satisfies the following extensionality properties for variable-closed expressions.*

- *For ground types $\gamma \in \mathrm{Gnd}^{\mathsf{PNA}^+}$ it holds that*

$$\emptyset \vdash e_1 \lesssim_{\mathsf{PNA}^+} e_2 : \gamma \ \Leftrightarrow\ e_1 \mathrel{\overset{k}{\leqq}} e_2 . \tag{5.39}$$

- *For product types it holds that*

$$\emptyset \vdash e_1 \lesssim_{\mathsf{PNA}^+} e_2 : \tau_1 \times \tau_2 \ \Leftrightarrow\ \emptyset \vdash \mathtt{fst}\,e_1 \lesssim_{\mathsf{PNA}^+} \mathtt{fst}\,e_2 : \tau_1 \ \wedge\ \emptyset \vdash \mathtt{snd}\,e_1 \lesssim_{\mathsf{PNA}^+} \mathtt{snd}\,e_2 : \tau_2 . \tag{5.40}$$

- *For function types it holds that*

$$\emptyset \vdash e_1 \lesssim_{\mathsf{PNA}^+} e_2 : \tau_1 \to \tau_2 \ \Leftrightarrow\ (\forall e \in \mathrm{Exp}^{\mathsf{PNA}^+}(\tau_1))\ \emptyset \vdash e_1\,e \lesssim_{\mathsf{PNA}^+} e_2\,e : \tau_2 . \tag{5.41}$$

- *For abstraction types it holds that*

$$\emptyset \vdash e_1 \lesssim_{\mathsf{PNA}^+} e_2 : \delta\,\tau \ \Leftrightarrow\ (\mathit{\Lambda} a)\,\emptyset \vdash e_1\,@a \lesssim_{\mathsf{PNA}^+} e_2\,@a : \tau . \tag{5.42}$$

Proof. The left-to-right direction for ground types is Lemma 5.3.7. For the other types the left-to-right directions follow from the compatibility (5.17) of $\lesssim_{\mathsf{PNA}^+}$, see Proposition 5.1.7.

For the right-to-left directions, Proposition 5.3.24 tells us that it is enough to prove the according properties for $[\![\_]\!] \lhd_\tau \_$ instead of $\emptyset \vdash \_ \lesssim_{\mathsf{PNA}^+} \_ : \tau$. At ground types, if $[\![e_1]\!] = \bot$ then $[\![e_1]\!] \lhd_\gamma e_2$ is immediate. Otherwise we know by the fundamental property (Proposition 5.3.5) that $[\![e_1]\!] \lhd_\gamma e_1$, hence $e_1 \Downarrow c$ with $[\![c]\!] = [\![e_1]\!]$ by definition, and so $e_2 \Downarrow c$ by $e_1 \leqq^k e_2$, which was needed for $[\![e_1]\!] \lhd_\gamma e_2$. The right-to-left direction for product types follows directly from the definition of $\lhd_{\tau_1 \times \tau_2}$, and for abstraction types it follows from the definition of $\lhd_{\delta\,\tau}$ together with Corollary 4.3.6 and Lemma 3.6.11. For function types we want to show $[\![e_1]\!] \lhd_{\tau_1 \to \tau_2} e_2$, so let $d \lhd_{\tau_1} e$ be given. By the fundamental property $[\![e_1]\!]\, d \lhd_{\tau_2} e_1\, e$ and by assumption $\emptyset \vdash e_1\, e \lesssim_{\mathsf{PNA}^+} e_2\, e : \tau_2$, hence $[\![e_1]\!]\, d \lhd_{\tau_2} e_1\, e$ holds with Lemma 5.3.23. □

5.4 Failures of full abstraction

In this section we show that both additions to PNA discussed in this thesis, definite description $\mathtt{the}\, x.\, e$ and existential quantification $\mathtt{ex}\, x.\, e$, are needed for full abstraction. We achieve this by proving that full abstraction fails if we leave one of the constructs out. In other words, we prove that the languages PNA+the and PNA+ex are both not fully abstract.

Capture-avoiding substitution for configurations The proofs of failure of full abstraction rely on using the frame-stack operational semantics for showing that certain contextual preorder relations hold. The main technical Lemmas 5.4.2 and 5.4.7 centrally feature a substitution operation on frame-stack configurations, which we analyse further in this subsection.

In Section 4.4.3 we defined binding and α-equivalence for configurations $\langle F\,, e\rangle$, where name binders in a frame of the frame-stack F can bind free names in subsequent frames and the expression e. For example, the configurations $\langle \mathrm{Id} \circ (\nu a.\, \cdot)\,, x = a\rangle$ and $\langle \mathrm{Id} \circ (\nu a'.\, \cdot)\,, x = a'\rangle$ are α-equivalent (and hence are equal, as we implicitly identify by α-equivalence). Substitution of configurations is defined to be capture-avoiding, so for instance the substitution $\langle \mathrm{Id} \circ (\nu a.\, \cdot)\,, x = a\rangle[a/x]$ results in $\langle \mathrm{Id} \circ (\nu a'.\, \cdot)\,, a = a'\rangle$.

This means that substitution of configurations cannot be defined directly in terms of separate substitutions for frames (as in Definition 4.1.9) and expressions (as in Figure 4.3). Continuing the example above, separate substitutions give $\langle (\mathrm{Id} \circ \nu a.\, \cdot)[a/x]\,, (x = a)[a/x]\rangle = \langle \mathrm{Id} \circ \nu a.\, \cdot\,, a = a\rangle$, but this is *not* the right result of the configuration substitution $\langle \mathrm{Id} \circ \nu a.\, \cdot\,, x = a\rangle[a/x]$. However, if we choose the binding names in the frame-stack well, separate substitution for configuration works, as the next lemma shows.

Lemma 5.4.1 (choice of bound names). *Given a frame-stack $F \in \mathrm{Stack}^{\mathsf{PNA}^+}$ and expressions $e, e' \in \mathrm{Exp}^{\mathsf{PNA}^+}$, if it holds that all the frames in F of the form $\nu a.\, \cdot$ and $\alpha a'.\, \cdot$*

satisfy that their binding names are fresh for e', so $a, a' \notin \operatorname{fn} e'$, then

$$\langle F, e\rangle[e'/x] = \langle F[e'/x], e[e'/x]\rangle \qquad (\textit{binders in } F \textit{ are fresh for } e').$$

Proof. By induction on F and the definition of the respective substitutions. □

5.4.1 Counter-example for PNA+the

Recall the definitions from Example 5.1.4, in particular the expressions $\mathtt{F}_1$ from (5.6) and $\mathtt{F}_2$ from (5.7). The crucial property that leads to a counter-example to full abstraction (Theorem 5.5.20) for PNA+the is that for any $e \in \mathrm{Exp}^{\mathsf{PNA+the}}((\mathtt{name} \to \mathtt{bool}) \to \mathtt{bool})$ and $a \in \mathbb{A}$ we have

$$a \# e \;\wedge\; e\,\mathtt{eqBot}_a \Downarrow c \;\Rightarrow\; e\,\mathtt{kBot} \Downarrow c \tag{5.43}$$

where $\mathtt{eqBot}_a$ is defined in (5.4) and $\mathtt{kBot}$ is defined in (5.5). We can prove (5.43) as a corollary of a more general induction on the steps of the frame-stack semantics, given in the next lemma.

Lemma 5.4.2 (fresh $\mathtt{eqBot}_a$ substitution). *Define for $n \in \mathbb{N}$ and $a \in \mathbb{A}$*

$$\begin{aligned}
\mathtt{eqBot}_{a,n} &\triangleq \lambda x : \mathtt{name} \to \nu b_1. \ldots \nu b_n.\ \mathtt{if}(x = a)\ \mathtt{then}\ \mathtt{T}\ \mathtt{else}\ \mathtt{bot}_{\mathtt{bool}} \\
\mathit{EqBot}_a &\triangleq \{\mathtt{eqBot}_{a,n} \mid n \in \mathbb{N}\} \\
\mathtt{kBot}_n &\triangleq \lambda x : \mathtt{name} \to \nu b_1. \ldots \nu b_n.\ \mathtt{bot}_{\mathtt{bool}} \\
\mathit{KBot} &\triangleq \{\mathtt{kBot}_n \mid n \in \mathbb{N}\}
\end{aligned}$$

where $b_1, \ldots, b_n$ are distinct and fresh for a. Note that the definitions of $\mathtt{eqBot}_{a,n}$ and $\mathtt{kBot}_n$ are independent of the choice of $b_1, \ldots, b_n$ due to α-equivalence. It then holds that

$$\begin{aligned}
&(\forall i \in \mathbb{N})(\forall a \in \mathbb{A})(\forall j \in \mathbb{N})(\forall \vec{x} \in \mathbb{V}^j)(\forall \vec{e} \in (\mathit{EqBot}_a)^j)(\forall \gamma \in \mathrm{Gnd}^{\mathsf{PNA+the}}) \\
&(\forall \langle F, e\rangle \in \mathrm{Config}^{\mathsf{PNA+the}})(\forall c \in \mathrm{Can}^{\mathsf{PNA+the}}) \\
&x_1 : \mathtt{name} \to \mathtt{bool}, \ldots, x_j : \mathtt{name} \to \mathtt{bool} \vdash F[e] : \gamma \;\wedge\; a \# \langle F, e\rangle \;\wedge \\
&\langle F, e\rangle[\vec{e}/\vec{x}] \to^i \langle \mathrm{Id}, c\rangle \;\Rightarrow\; (\forall \vec{e'} \in \mathit{KBot}^j)\, \langle F, e\rangle[\vec{e'}/\vec{x}] \to^* \langle \mathrm{Id}, c\rangle .
\end{aligned}$$

Proof. Given in Appendix A.5. □

Corollary 5.4.3 (proof of (5.43)). *Property (5.43) holds for all $e \in \mathrm{Exp}^{\mathsf{PNA+the}}((\mathtt{name} \to \mathtt{bool}) \to \mathtt{bool})$ and $a \in \mathbb{A}$.*

Proof. Observe that $e\,\mathtt{eqBot}_a \Downarrow c$ can only hold if $e \Downarrow \lambda(x : \mathtt{name} \to \mathtt{bool}) \to e'$ and $e'[\mathtt{eqBot}_a/x] \Downarrow c$. Therefore by Theorem 4.4.26 there is an $i \in \mathbb{N}$ such that $\langle \mathrm{Id}, e'[\mathtt{eqBot}_a/x]\rangle \to^i \langle \mathrm{Id}, c\rangle$. It is an instance of Lemma 5.4.2 that this implies $\langle \mathrm{Id}, e'[\mathtt{kBot}/x]\rangle \to^i \langle \mathrm{Id}, c\rangle$ and again with Theorem 4.4.26 we get $e'[\mathtt{kBot}/x] \Downarrow c$, which implies $e\,\mathtt{kBot} \Downarrow c$. □

Lemma 5.4.4 ($\mathtt{F}_1$-$\mathtt{F}_2$ **preorder for** PNA+the)**.** *With* $\mathtt{F}_1$ *and* $\mathtt{F}_2$ *given as in* (5.6) *and* (5.7) *it holds that*

$$\emptyset \vdash \mathtt{F}_1 \lesssim_{\mathsf{PNA+the}} \mathtt{F}_2 : ((\mathtt{name} \to \mathtt{bool}) \to \mathtt{bool}) \to \mathtt{bool}\,. \tag{5.44}$$

Proof. By the extensionality properties for function types (5.41) and ground types (5.39) we know that (5.44) holds if and only if $(\forall e \in \mathrm{Exp}^{\mathsf{PNA+the}}((\mathtt{name} \to \mathtt{bool}) \to \mathtt{bool}))\, \mathtt{F}_1\, e \mathrel{\leqq^k} \mathtt{F}_2\, e$. To prove this, let e be given and assume $\mathtt{F}_1\, e \Downarrow c$. By α-equivalence we may also assume that the restricted name a in (5.6) satisfies $a \# e$. By the definition of $\mathtt{F}_1$ and the operational semantics this implies $e\, \mathtt{eqBot}_a \Downarrow c$. This gives $e\, \mathtt{kBot} \Downarrow c$ by (5.43) and hence $\mathtt{F}_2\, e \Downarrow c$ holds. □

Already in Section 5.1.1 we argued that PNA fails to be fully abstract. We can now give a formal proof of this result, with the extended language PNA+the.

Theorem 5.4.5 (**failure of** PNA+the)**.** PNA+the *is not fully abstract with respect to the denotational semantics described in Section 4.3.*

Proof. In (5.13) and (5.14) we show $[\![\mathtt{F}_1]\!] \not\sqsubseteq [\![\mathtt{F}_2]\!]$ and together with (5.44) this gives a counter-example to full abstraction (Theorem 5.5.20) for PNA+the. □

The symmetric property of (5.44) can be proved more easily, which shows that the contextual equivalence (5.11) is valid as these proofs of the contextual preorder specialise to PNA.

Lemma 5.4.6 ($\mathtt{F}_2$-$\mathtt{F}_1$ **preorder for** PNA+the)**.** *The following contextual preorder holds*

$$\emptyset \vdash \mathtt{F}_2 \lesssim_{\mathsf{PNA+the}} \mathtt{F}_1 : ((\mathtt{name} \to \mathtt{bool}) \to \mathtt{bool}) \to \mathtt{bool}\,.$$

Proof. By definition we know $[\![\mathtt{kBot}]\!] = \bot$, and from this we can show with the monotonicity of function evaluation (Theorem 3.4.28) and (3.29) that $[\![\mathtt{F}_2]\!] \sqsubseteq [\![\mathtt{F}_1]\!]$, from which the property follows by computational adequacy, Theorem 5.3.19. □

5.4.2 Counter-example for PNA+ex

The property that leads to the failure of full abstraction for PNA+ex is that every $e \in \mathrm{Exp}^{\mathsf{PNA+ex}}((\mathtt{name} \to \mathtt{bool}) \to \mathtt{name})$ and $a \in \mathbb{A}$ satisfy

$$a \# e \;\wedge\; e\, \mathtt{eq}_a \Downarrow a' \;\Rightarrow\; a \neq a'\,, \tag{5.45}$$

where $\mathtt{eq}_a$ is given in (5.8). Its proof works in a similar way as Lemma 5.4.2.

Lemma 5.4.7 (**fresh** $\mathtt{eq}_a$ **substitution**)**.** *Define for any* $n \in \mathbb{N}$

$$\begin{aligned} \mathtt{eq}_{a,n} &\triangleq \lambda x : \mathtt{name} \to \nu b_1. \ldots \nu b_n. (x = a) \\ Eq_a &\triangleq \{\mathtt{eq}_{a,n} \mid n \in \mathbb{N}\} \end{aligned}$$

where $b_1, \ldots, b_n$ are distinct and not equal to a. Note that the definition of $\mathtt{eq}_{a,n}$ is independent of the choice of $b_1, \ldots, b_n$ due to α-equivalence. It follows that:

$$(\forall i \in \mathbb{N})(\forall a \in \mathbb{A})(\forall j \in \mathbb{N})(\forall \vec{x} \in var^j)(\forall \vec{e} \in (Eq_a)^j)(\forall \langle F, e\rangle \in \mathrm{Config}^{\mathsf{PNA+ex}})(\forall a' \in \mathbb{A})$$
$$x_1 : \mathtt{name} \to \mathtt{bool}, \ldots, x_j : \mathtt{name} \to \mathtt{bool} \vdash F[e] : \mathtt{name} \wedge a \# \langle F, e\rangle \wedge$$
$$\langle F, e\rangle[\vec{e}/\vec{x}] \to^i \langle \mathrm{Id}, a'\rangle \Rightarrow a \neq a'.$$

Proof. The argument is very similar to the one in the proof of Lemma 5.4.2. We use the same lemmas and properties, and the argument simplifies in many cases, due to the simpler induction hypothesis. Therefore we omit most details here.

The only major difference is that we do not have a case for $\mathtt{the}\, y.e$ (as it is not part of PNA+ex), but we have to consider a case for $\mathtt{ex}\, y.e$. For this we may assume $\langle F[\vec{e}/\vec{x}], \mathtt{ex}\, y.e[\vec{e}/\vec{x}]\rangle \to^{i+1} \langle \mathrm{Id}, a'\rangle$ and $a \# F, \mathtt{ex}\, y.e$. By Lemma 4.4.23 we know that $\langle \mathrm{Id}, \mathtt{ex}\, y.e[\vec{e}/\vec{x}]\rangle \to^m \langle \mathrm{Id}, c\rangle$ and $\langle F[\vec{e}/\vec{x}], c\rangle \to^{i+1-m} \langle \mathrm{Id}, a'\rangle$. By typing (Lemma 4.2.8) we know $c = \mathtt{T}$ or $c = \mathtt{F}$ and by the frame-stack rules for $\mathtt{ex}\, y.e$ we know that it must be that $m > 0$. Hence we can use induction on $\langle F[\vec{e}/\vec{x}], c\rangle \to^{i+1-m} \langle \mathrm{Id}, a'\rangle$ (as $a \# c$ and $c = c[\vec{e}/\vec{x}]$) to obtain $a \neq a'$, which was to show. □

Corollary 5.4.8 (**proof of** (5.45))**.** *Property* (5.45) *holds for all* $e \in \mathrm{Exp}^{\mathsf{PNA+ex}}((\mathtt{name} \to \mathtt{bool}) \to \mathtt{name})$ *and* $a \in \mathbb{A}$.

Proof. We know that $e\, \mathtt{eq}_a \Downarrow c$ can only hold if $e \Downarrow \lambda(x : \mathtt{name} \to \mathtt{bool}) \to e'$ and $e'[\mathtt{eq}_a/x] \Downarrow c$. As $a \# e$ we know $a \# e'$ by Lemma 4.4.11, and Theorem 4.4.26 shows that there is an $i \in \mathbb{N}$ such that $\langle \mathrm{Id}, e'[\mathtt{eq}_a/x]\rangle \to^i \langle \mathrm{Id}, a'\rangle$. Therefore we can apply Lemma 5.4.7 to obtain $a \neq a'$. □

Lemma 5.4.9 ($\mathtt{G}_1$-$\mathtt{G}_2$ **preorder for** PNA+ex)**.** *Recall the definitions of* $\mathtt{G}_1$ *and* $\mathtt{G}_2$ *from* (5.9) *and* (5.10)*. These expressions satisfy*

$$\emptyset \vdash \mathtt{G}_1 \lesssim_{\mathsf{PNA+ex}} \mathtt{G}_2 : ((\mathtt{name} \to \mathtt{bool}) \to \mathtt{name}) \to \mathtt{bool}. \tag{5.46}$$

Proof. By the extensionality properties for function types (5.41) and ground types (5.39) we know that (5.46) holds if and only if $(\forall e \in \mathrm{Exp}^{\mathsf{PNA+ex}}((\mathtt{name} \to \mathtt{bool}) \to \mathtt{name}))\ \mathtt{G}_1\, e \leq^k \mathtt{G}_2\, e$. To prove this, let e be given and assume $\mathtt{G}_1\, e \Downarrow c$. By α-equivalence we may also assume that the restricted name a in $\mathtt{G}_1$ satisfies $a \# e$. By the definition of $\mathtt{G}_1$ and the operational semantics this implies $(e\, \mathtt{eq}_a) = a \Downarrow c$ and $e\, \mathtt{eq}_a \Downarrow a'$. By (5.45) it must be that $a \neq a'$ and thus $c = \mathtt{F}$. It also holds that $\mathtt{G}_2\, e \Downarrow \mathtt{F}$, so $\mathtt{G}_1\, e \Downarrow c$. □

Similar to the previous section, full abstraction fails for PNA+ex.

Theorem 5.4.10 (**failure of** PNA+ex)**.** PNA+ex *is not fully abstract with respect to the denotational semantics described in Section 4.3.*

Proof. In (5.15) and (5.16) we show $[\![\mathtt{G}_1]\!] \not\sqsubseteq [\![\mathtt{G}_2]\!]$ and together with (5.46) this gives a counter-example to full abstraction (Theorem 5.5.20) for PNA+ex. □

5.5 Full abstraction for PNA^+

In this section we show that the nominal Scott domain model is fully abstract for PNA^+. Formally the property of full abstraction holds for PNA^+ if all well-typed expressions $\Gamma \vdash e : \tau$ and $\Gamma \vdash e' : \tau$ satisfy

$$[\![e]\!] \sqsubseteq [\![e']\!] \iff \Gamma \vdash e \lesssim_{\mathsf{PNA}^+} e' : \tau\,. \tag{5.47}$$

Computational adequacy (Theorem 5.3.19) already gives the left-to-right direction of full abstraction. The rest of this section will be concerned with proving the right-to-left direction. It useful to consider this direction for each type and typing environment separately, thus we define for each $\Gamma \in \mathrm{Env}^{\mathsf{PNA}^+}$ and $\tau \in \mathrm{Typ}^{\mathsf{PNA}^+}$ the property $(\mathrm{FA}_{\Gamma|\tau})$ by

$$(\forall e, e' \in \mathrm{Exp}^{\mathsf{PNA}^+})\, \Gamma \vdash e \lesssim_{\mathsf{PNA}^+} e' : \tau \;\Rightarrow\; [\![e]\!] \sqsubseteq [\![e']\!]\,. \tag{$\mathrm{FA}_{\Gamma|\tau}$}$$

5.5.1 Simple types and definable retracts

Most proofs of full abstraction (including ours) work by showing that certain elements of the domains in consideration are definable in the language.

Definition 5.5.1 (definability). An element in the nominal Scott domain model $d \in [\![\tau]\!]$ is PNA^+-*definable* if there is a variable-closed expression $e \in \mathrm{Exp}^{\mathsf{PNA}^+}(\tau)$ that denotes it: $[\![e]\!] = d$. We say *uniform-compact definability* holds at a type $\tau \in \mathrm{Typ}^{\mathsf{PNA}^+}$ if all uniform-compact elements of $[\![\tau]\!]$ are definable:

$$(\forall u \in \mathrm{K}[\![\tau]\!])(\exists e \in \mathrm{Exp}^{\mathsf{PNA}^+}(\tau))\; [\![e]\!] = u\,. \tag{DEF_τ}$$

If we knew that (DEF_τ) holds for all types $\tau \in \mathrm{Typ}^{\mathsf{PNA}^+}$, then we could prove $(\mathrm{FA}_{\Gamma|\tau})$ by following the traditional argument as surveyed by Curien [12, Criterion 2.2]. However, our proof of uniform-compact definability only works with types that avoid the use of function types $\tau_1 \to \tau_2$ in which the nominal Scott domain $[\![\tau_2]\!]$ might contain elements with non-empty support. So $\tau_2 = \mathtt{nat}$ is OK, but $\tau_2 = \mathtt{name}$ is not, for example. This leads us to making the following definition:

Definition 5.5.2 (simple types). Let the set of *simple types* be defined by the following grammar:

$$\sigma \in \mathrm{Styp}^{\mathsf{PNA}^+} ::= \mathtt{nat} \mid \mathtt{name} \mid \sigma \times \sigma \mid \sigma \to \mathtt{nat}\,.$$

Every simple type is also a 'normal' type as in Figure 4.7, so $\mathrm{Styp}^{\mathsf{PNA}^+} \subseteq \mathrm{Typ}^{\mathsf{PNA}^+}$. The other subset inclusion direction does not hold, because, for example, $\sigma \to \mathtt{name}$ is not a simple type. The key to the usefulness of simple types is Proposition 5.5.12, which says that all types are definable retracts of simple types.

Definition 5.5.3 (definable retract). A type $\tau_1 \in \mathrm{Typ}^{\mathsf{PNA}^+}$ is a *definable retract* of another type $\tau_2 \in \mathrm{Typ}^{\mathsf{PNA}^+}$, written $\tau_1 \preceq \tau_2$, if there are closed expressions $i \in \mathrm{Exp}^{\mathsf{PNA}^+}(\tau_1 \to \tau_2)$ and $r \in \mathrm{Exp}^{\mathsf{PNA}^+}(\tau_2 \to \tau_1)$ that satisfy

$$[\![r]\!] \circ [\![i]\!] \;=\; \mathrm{id}_{[\![\tau_1]\!]}\,. \tag{5.48}$$

We continue with several technical properties of definable retracts and simple types, building up to Proposition 5.5.12.

Lemma 5.5.4 (**retract preorder**). *The definable retract relation $\preceq$ is reflexive and transitive.*

Proof. Reflexivity $\tau \preceq \tau$ follows by taking $i, r \triangleq \lambda x : \tau \to x$. For transitivity assume that we are given $\tau_1 \preceq \tau_2 \preceq \tau_3$ with expressions $i_1 \in \mathrm{Exp}^{\mathsf{PNA}^+}(\tau_1 \to \tau_2)$, $i_2 \in \mathrm{Exp}^{\mathsf{PNA}^+}(\tau_2 \to \tau_3)$, $r_1 \in \mathrm{Exp}^{\mathsf{PNA}^+}(\tau_2 \to \tau_1)$ and $r_2 \in \mathrm{Exp}^{\mathsf{PNA}^+}(\tau_3 \to \tau_2)$. Through the definitions $i \triangleq \lambda x : \tau_1 \to i_2(i_1\, x)$ and $r \triangleq \lambda x : \tau_3 \to r_1(r_2\, x)$ we can prove that $\tau_1 \preceq \tau_3$. □

Lemma 5.5.5 (**retract currying**). *For any types $\tau_1, \tau_2, \tau_3 \in \mathrm{Typ}^{\mathsf{PNA}^+}$ it holds that $\tau_1 \to \tau_2 \to \tau_3 \preceq \tau_1 \times \tau_2 \to \tau_3$ and $\tau_1 \times \tau_2 \to \tau_3 \preceq \tau_1 \to \tau_2 \to \tau_3$. Furthermore $\tau_1 \to (\tau_2 \times \tau_3) \preceq (\tau_1 \to \tau_2) \times (\tau_1 \to \tau_3)$ and $(\tau_1 \to \tau_2) \times (\tau_1 \to \tau_3) \preceq \tau_1 \to (\tau_2 \times \tau_3)$.*

Proof. Through currying and uncurrying $\tau_1 \times \tau_2 \to \tau_3$ and $\tau_1 \to \tau_2 \to \tau_3$ are isomorphic types (see Theorem 3.4.28) and therefore they are definable retracts of each others, and similarly for $\tau_1 \to (\tau_2 \times \tau_3)$ and $(\tau_1 \to \tau_2) \times (\tau_1 \to \tau_3)$. □

The next two lemmas show that definable retracts are covariant for products and functions.

Lemma 5.5.6 (**product covariance**). *Definable retracts are covariant for products. If $\tau_1 \preceq \tau_1'$ and $\tau_2 \preceq \tau_2'$, then $\tau_1 \times \tau_2 \preceq \tau_1' \times \tau_2'$.*

Proof. Let i_1, i_2, r_1, r_2 be given such that $[\![r_1]\!] \circ [\![i_1]\!] = \mathrm{id}_{[\![\tau_1]\!]}$ and $[\![r_2]\!] \circ [\![i_2]\!] = \mathrm{id}_{[\![\tau_2]\!]}$. The expressions

$$\begin{aligned} i_3 &\triangleq \lambda x : \tau_1 \times \tau_2 \to (i_1(\mathtt{fst}\ x), i_2(\mathtt{snd}\ x)) \\ r_3 &\triangleq \lambda x : \tau_1' \times \tau_2' \to (r_1(\mathtt{fst}\ x), r_2(\mathtt{snd}\ x)) \end{aligned}$$

then satisfy $[\![r_3]\!] \circ [\![i_3]\!] = \mathrm{id}_{[\![\tau_1 \times \tau_2]\!]}$ and hence witness $\tau_1 \times \tau_2 \preceq \tau_1' \times \tau_2'$. □

Lemma 5.5.7 (**function covariance**). *Definable retracts are covariant for functions. If $\tau_1 \preceq \tau_1'$ and $\tau_2 \preceq \tau_2'$, then $\tau_1 \to \tau_2 \preceq \tau_1' \to \tau_2'$.*

Proof. Let i_1, i_2, r_1, r_2 be given such that $[\![r_1]\!] \circ [\![i_1]\!] = \mathrm{id}_{[\![\tau_1]\!]}$ and $[\![r_2]\!] \circ [\![i_2]\!] = \mathrm{id}_{[\![\tau_2]\!]}$. Defining

$$\begin{aligned} i_3 &\triangleq \lambda(f : \tau_1 \to \tau_2) \to \lambda x : \tau_1' \to i_2(f\,(r_1\, x)) \\ r_3 &\triangleq \lambda(f : \tau_1' \to \tau_2') \to \lambda x : \tau_1 \to r_2(f\,(i_1\, x)). \end{aligned}$$

gives us $[\![r_3]\!] \circ [\![i_3]\!] = \mathrm{id}_{[\![\tau_1 \to \tau_2]\!]}$ and therefore $\tau_1 \to \tau_2 \preceq \tau_1' \to \tau_2'$ holds. □

The following three lemmas prove some concrete instances of the definable retract relation. Note that the proof of Lemma 5.5.8 is the only place in our proof of full abstraction where the existence of the definite description construct $\mathtt{the}\, x.e$ in PNA^+ is used.

Lemma 5.5.8 (name **retract**). name *is a definable retract of* name → nat.

Proof. With the definitions

$$i \triangleq \lambda x : \mathtt{name} \to \lambda y : \mathtt{name} \to \mathtt{if}\ x = y\ \mathtt{then}\ \mathtt{0}\ \mathtt{else}\ \mathtt{S}\,\mathtt{0}$$
$$r \triangleq \lambda(f : \mathtt{name} \to \mathtt{nat}) \to \mathtt{the}\, x.\ \mathtt{zero}\,(f\ x)$$

we obtain $\mathtt{name} \preceq (\mathtt{name} \to \mathtt{nat})$. □

Lemma 5.5.9 (**abstraction retract**). *For each type* $\tau \in \mathrm{Typ}^{\mathsf{PNA}^+}$ *it holds that* $\delta\,\tau$ *is a definable retract of* $\mathtt{name} \to \tau$.

Proof. Define the expressions

$$i \triangleq \lambda(x : \delta\,\tau) \to \lambda y : \mathtt{name} \to x\ @\ y \tag{5.49}$$
$$r \triangleq \lambda(f : \mathtt{name} \to \tau) \to \alpha a.\, f\ a \tag{5.50}$$

and let any $d \in [\![\delta\,\tau]\!]$ be given. By α-equivalence we may assume that a in r satisfies $a \# d$. We then obtain with Lemmas 2.3.25 and 3.6.11 that $[\![r]\!]([\![i]\!]\,d) = \langle a\rangle(d @^t a) = \langle a\rangle(d @ a) = d$. As d was chosen arbitrarily this shows $\delta\,\tau \preceq (\mathtt{name} \to \tau)$. A similar proof is given in Pitts [43, Theorem 2.13 and Appendix B]. □

Lemma 5.5.10 (term **retract**). term *is a definable retract of* $(\mathtt{nat} \times (\mathtt{name} \to \mathtt{nat})) \times \mathtt{nat}$.

Proof. We use a suitable Gödel-numbering of λ-terms, facilitating an environment consisting of a finite list of distinct atomic names (which encodes the free variables of the λ-term). The second component nat is used to code the term's syntax tree and the first component $\mathtt{nat} \times (\mathtt{name} \to \mathtt{nat})$ is used to code environments as a pair consisting of list-length and a function (taking value 0 at all but finitely many arguments) giving positions in the list. □

After one more helper lemma, we can prove that every PNA^+-type is a definable retract of some simple type, Proposition 5.5.12.

Lemma 5.5.11 (**simple function retract**). *For all simple types* $\sigma_1, \sigma_2 \in \mathrm{Styp}^{\mathsf{PNA}^+}$, *there exists a simple type* $\sigma' \in \mathrm{Styp}^{\mathsf{PNA}^+}$ *so that* $\sigma_1 \to \sigma_2 \preceq \sigma'$.

Proof. Let σ_1 be given and proceed by structural induction on σ_2:

- *Case* $\sigma_2 = \mathtt{nat}$*:* In this case $\sigma_1 \to \sigma_2$ is already a simple type.
- *Case* $\sigma_2 = \sigma_3 \times \sigma_4$*:* By induction it follows that there are simple types $\sigma_3', \sigma_4' \in \mathrm{Styp}^{\mathsf{PNA}^+}$ satisfying $\sigma_1 \to \sigma_3 \preceq \sigma_3'$ and $\sigma_1 \to \sigma_4 \preceq \sigma_4'$. By Lemmas 5.5.5 and 5.5.6 we know $\sigma_1 \to \sigma_3 \times \sigma_4 \preceq (\sigma_1 \to \sigma_3) \times (\sigma_1 \to \sigma_4) \preceq \sigma_3' \times \sigma_4'$ and hence conclude with transitivity of $\preceq$.
- *Case* $\sigma_2 = \sigma_3 \to \mathtt{nat}$*:* With Lemma 5.5.5 it follows that $\sigma_1 \to \sigma_3 \to \mathtt{nat} \preceq \sigma_1 \times \sigma_3 \to \mathtt{nat}$, which is a simple type.

- *Case* $\sigma_2 = \mathtt{name}$*:* Through Lemma 5.5.8 we know that $\mathtt{name} \preceq \mathtt{name} \to \mathtt{nat}$ and then $\sigma_1 \to \mathtt{name} \preceq \sigma_1 \times \mathtt{name} \to \mathtt{nat}$ follows by Lemmas 5.5.4, 5.5.5 and 5.5.7.

□

Proposition 5.5.12 (all types are retracts). *Every type is a definable retract of some simple type:*

$$(\forall \tau \in \mathrm{Typ}^{\mathsf{PNA}^+})(\exists \sigma \in \mathrm{Styp}^{\mathsf{PNA}^+})\ \tau \preceq \sigma\,.$$

Proof. We proceed by structural induction over τ.

- *Case* $\tau \in \{\mathtt{nat}, \mathtt{name}\}$*:* Follows directly from reflexivity of $\preceq$ (Lemma 5.5.4).
- *Case* $\tau = \mathtt{bool}$*:* We get $\mathtt{bool} \preceq \mathtt{nat}$ with $i \triangleq \lambda x : \mathtt{bool} \to \mathtt{if}\ x\ \mathtt{then}\ \mathtt{0}\ \mathtt{else}\ \mathtt{S0}$ and $r := \lambda x : \mathtt{nat} \to \mathtt{zero}\ x$.
- *Case* $\tau = \tau_1 \times \tau_2$*:* Follows by induction and Lemma 5.5.6.
- *Case* $\tau = \tau_1 \to \tau_2$*:* By induction and Lemma 5.5.7 we know that there are $\sigma_1, \sigma_2 \in \mathrm{Styp}^{\mathsf{PNA}^+}$ such that $\tau_1 \to \tau_2 \preceq \sigma_1 \to \sigma_2$ and by Lemma 5.5.11 that there is a $\sigma' \in \mathrm{Styp}^{\mathsf{PNA}^+}$ such that $\sigma_1 \to \sigma_2 \preceq \sigma'$. Transitivity of $\preceq$ (Lemma 5.5.4) concludes this case.
- *Case* $\tau = \mathtt{term}$*:* We know $\mathtt{term} \preceq (\mathtt{nat} \times (\mathtt{name} \to \mathtt{nat})) \times \mathtt{nat}$ by Lemma 5.5.10.
- *Case* $\tau = \delta\,\tau_1$*:* Lemma 5.5.9 gives $\delta\,\tau_1 \preceq \mathtt{name} \to \tau_1$, by induction there is a $\sigma_1 \in \mathrm{Styp}^{\mathsf{PNA}^+}$ such that $\tau_1 \preceq \sigma_1$ and by Lemma 5.5.11 there is a $\sigma' \in \mathrm{Styp}^{\mathsf{PNA}^+}$ such that $\mathtt{name} \to \sigma_1 \preceq \sigma'$. Overall we obtain $\delta\,\tau_1 \preceq \sigma'$ by Lemma 5.5.7 and transitivity of $\preceq$.

□

Proposition 5.5.12 allows us, after two more lemmas, to reduce the task of proving full abstraction at all types to the task of proving uniform-compact definability at only simple types (Theorem 5.5.15). This approach seems novel in the literature, as we avoid reasoning about definability at all types. See also Remark 5.5.21 and Open Problem 6.2.1.

Lemma 5.5.13 (simple definability implies simple full abstraction). *Using Proposition 5.5.12, uniform-compact definability at simple types implies the right-to-left direction of full abstraction for them, that is*

$$((\forall \sigma' \in \mathrm{Styp}^{\mathsf{PNA}^+})\,(\mathrm{DEF}_{\sigma'})) \Rightarrow (\forall \sigma \in \mathrm{Styp}^{\mathsf{PNA}^+})\,(\mathrm{FA}_{\Gamma|\sigma})\,.$$

Proof. Given in Appendix A.6. □

Lemma 5.5.14 (simple full abstraction implies full abstraction). *If* τ *is a* PNA^+*-definable retract of* σ*, then the right-to-left direction of full abstraction at* σ *implies the same at* τ*. Formally this is* $(\mathrm{FA}_{\Gamma|\sigma}) \Rightarrow (\mathrm{FA}_{\Gamma|\tau})$.

Proof. Suppose $(\mathrm{FA}_{\Gamma|\sigma})$ holds and that $\Gamma \vdash e \lesssim_{\mathsf{PNA}^+} e' : \tau$. Let i, r be the expressions corresponding to $\tau \preceq \sigma$, then we have $\Gamma \vdash i\, e \lesssim_{\mathsf{PNA}^+} i\, e' : \sigma$ because $\lesssim_{\mathsf{PNA}^+}$ is compatible (Proposition 5.1.7). Thus by $(\mathrm{FA}_{\Gamma|\sigma})$, for any $\rho \in [\![\Gamma]\!]$ it holds that $[\![i]\!]([\![e]\!]\rho) = [\![i\, e]\!]\rho \sqsubseteq [\![i\, e']\!]\rho = [\![i]\!]([\![e']\!]\rho)$. We know that $[\![i]\!]$ has a monotone left inverse $[\![r]\!]$, so $[\![e]\!]\rho \sqsubseteq [\![e']\!]\rho$. As ρ was chosen arbitrarily we get $[\![e]\!] \sqsubseteq [\![e']\!] \in [\![\Gamma]\!] \to_{\mathrm{uc}} [\![\tau]\!]$. □

Theorem 5.5.15 (**simple definability implies full abstraction**). *Definability at simple types implies the right-to-left direction of full abstraction at all types:*

$$(\forall \sigma \in \mathrm{Styp}^{\mathsf{PNA}^+})\,(\mathrm{DEF}_\sigma) \Rightarrow (\forall \tau \in \mathrm{Typ}^{\mathsf{PNA}^+})\,(\mathrm{FA}_{\Gamma|\tau}).$$

Proof. Combine Lemma 5.5.14 with Proposition 5.5.12 and Lemma 5.5.13. □

5.5.2 Definability at simple types

As Theorem 5.5.15 indicates, what is left to show for full abstraction is definability at simple types. Our proof of definability in principle follows the structure of the traditional argument by Plotkin [47, Lemma 4.5]; a modern account can be found in Streicher [61, Theorem 13.9]. However, in our nominal setting many uses of finite subsets in the traditional proof are replaced by uses of orbit-finite subsets and their representation as hulls (see Theorem 2.3.38).

The definition of $\mathrm{hull}_A F$ (Definition 2.3.35) involves an existential quantification over $\mathrm{Perm}(\mathbb{A})$, and for the definability proof (Theorem 5.5.18) we need to reduce this to existential quantifications over $\mathbb{A}$. This is where the presence of $\mathsf{ex}\, x.\, e$ expressions in PNA^+gets applied: in order to prove Lemmas 5.5.16 and 5.5.17. Neither is trivial to prove; the arguments can be found in Appendices A.7 and A.8. In particular Lemma 5.5.17 works by a subtle case distinction over all the different combinations in which the atomic names in the supports of the uniform-compact elements involved in the lemma can overlap.

Lemma 5.5.16 (**one element hull definability**). *Recall the definition of step functions in* (3.13). *For each* $\tau \in \mathrm{Typ}^{\mathsf{PNA}^+}$, $u \in \mathrm{K}[\![\tau]\!]$ *and* $A \subseteq_{\mathrm{f}} \mathbb{A}$ *it holds that*

$$(u \searrow \mathrm{true})\ \textit{is}\ \mathsf{PNA}^+\textit{-definable} \Rightarrow \bigsqcup \mathrm{hull}_A\{(u \searrow \mathrm{true})\}\ \textit{is}\ \mathsf{PNA}^+\textit{-definable}.$$

Note that the set $\mathrm{hull}_A\{(u \searrow \mathrm{true})\}$ *is orbit-finite and consistent, so its join exists by Lemma 3.4.24 and is uniform-compact by Lemma 3.3.4.*

Proof. Given in Appendix A.7. □

Lemma 5.5.17 (**two elements hull definability**). *Suppose that* $\tau \in \mathrm{Typ}^{\mathsf{PNA}^+}$, $u, u' \in \mathrm{K}[\![\tau]\!]$ *and* $A \subseteq_{\mathrm{f}} \mathbb{A}$ *satisfy:*

- *for all uniform-compact elements* $v, v' \in \mathrm{K}[\![\tau]\!]$ *we have that* $v \not\asymp v'$ *implies that the join* $(v \searrow \mathrm{true}) \sqcup (v' \searrow \mathrm{false})$ *(exists and) is* PNA^+*-definable;*
- *for all finite permutations* $\pi \in \mathrm{Perm}(\mathbb{A})$ *satisfying* $\pi \# A$, *it holds that* $u \not\asymp \pi \cdot u'$.

Then the join $\bigsqcup \mathrm{hull}_A\{(u \searrow \mathrm{true}), (u' \searrow \mathrm{false})\}$ *(exists and) is* PNA^+*-definable.*

Proof. Given in Appendix A.8. □

Having established Lemmas 5.5.16 and 5.5.17, we are now in the position to prove uniform-compact definability at all simple types. By Theorem 5.5.15 this then directly leads to the main result of this thesis: the nominal Scott domain model is fully abstract for PNA^+ (Theorem 5.5.20).

Theorem 5.5.18 (**definability at simple types**). *All uniform-compact elements of simple type are definable in* PNA^+. *In other words* $(\forall \sigma \in \mathrm{Styp}^{\mathsf{PNA}^+})\,(\mathrm{DEF}_\sigma)$ *holds.*

Proof. Given in Appendix A.9. □

Corollary 5.5.19. $(\mathrm{FA}_{\Gamma|\tau})$ *holds for all* $\Gamma \in \mathrm{Env}^{\mathsf{PNA}^+}$ *and* $\tau \in \mathrm{Typ}^{\mathsf{PNA}^+}$.

Proof. By combining Theorems 5.5.15 and 5.5.18. □

Theorem 5.5.20 (**full abstraction for** PNA^+). PNA^+ *is fully abstract in the sense that* (5.47) *is satisfied. Consequently it also holds that* $[\![e]\!] = [\![e']\!] \Leftrightarrow \Gamma \vdash e \cong_{\mathsf{PNA}^+} e' : \tau$.

Proof. Theorem 5.3.19 gives the left-to-right direction and Corollary 5.5.19 gives the right-to-left direction. □

Remark 5.5.21 (**definability of all uniform-compact elements**). Are all elements of $\mathrm{K}[\![\tau]\!]$ definable in PNA^+, for *any* type $\tau \in \mathrm{Typ}^{\mathsf{PNA}^+}$? We introduced simple types because we did not find a way to prove such a definability result at all types. Instead, we bypass the problem by proving definability only at simple types and connecting simple types with all types via definable retracts. If all the definable retracts used in our proof of Proposition 5.5.12 were actually definable *embedding-projection pairs* (in the sense that also $[\![i]\!] \circ [\![r]\!] \sqsubseteq \mathrm{id}_{[\![\sigma]\!]}$ holds), then uniform-compact definability at simple types would immediately imply uniform-compact definability at any type.[1]

Unfortunately, (5.49) and (5.50) do *not* form an embedding-projection pair for name abstraction types. For example, when $\tau = \mathtt{bool}$ we can calculate that $[\![r]\!] \in (\mathbb{A}_\perp \to_{\mathrm{uc}} \mathbb{B}_\perp) \to_{\mathrm{uc}} [\mathbb{A}]\mathbb{B}_\perp$ maps the element eq_a from (3.22) to $\langle a \rangle \mathrm{false}$; and that $[\![i]\!]$ maps $\langle a \rangle \mathrm{false}$ to k_{false} from (3.25). Since $eq_a \not\sqsubseteq k_{\mathrm{false}}$, we have that $[\![i]\!] \circ [\![r]\!] \not\sqsubseteq \mathrm{id}_{[\![\mathtt{name} \to \mathtt{bool}]\!]}$. So it seems that PNA^+-types are not definably embeddable into simple types.

If we left name abstractions out of PNA, then embedding-projection pairs as above could be used to show uniform-compact definability at every type of the remaining simpler language. For all of PNA^+, however, definability of all uniform-compact elements remains an open problem (that does not impede full abstraction), see Open Problem 6.2.1.

[1] We thank an anonymous referee of Lösch and Pitts [26] for pointing this out.

CHAPTER 6

CONCLUSION

This chapter concludes this thesis with an overview of related work (Section 6.1), a list of open problems (Section 6.2) and an overall summary (Section 6.3).

6.1 Related work

In addition to the pointers in the previous chapters, we give an overview of the work we built on to achieve our results.

6.1.1 Representation of object-level binding

Modelling syntax with binding and the associated notion of α-equivalence (see Example 2.3.22 for an explicit definition of the α-equivalence relation for the λ-calculus) is a pressing issue in metaprogramming, in particular in the field of mechanised theorem proving. In the words of Aydemir *et al.* [7]:

> "Representing binders has been recognized as crucial by the theorem proving community, and many different solutions to this problem have been proposed. In our (still limited) experience, none emerge as clear winners."

We give a short overview of the many approaches for representing syntax with binding, from the perspective of this thesis. The literature usually distinguishes between *first-order* and *higher-order* representation of syntax. In the former, variables and binders are represented as concrete nodes in first-order algebraic structures. In the latter, these aspects are 'lifted to the meta-language' by representing the bodies of binding constructs as meta-language functions.

Named representation What is likely the most obvious approach, often called *named representation*, is to use the first-order representation of binding as (bound

name, body)-pairs and to explicitly define functions for α-conversion and capture-avoiding substitution in the meta-language. However, this approach is typically dismissed because it is error-prone and involves writing much 'boilerplate' code that only deals with tedious checks for binding.

De Bruijn indices Another classical first-order approach is *de Bruijn indices* [13]. The idea is to use natural numbers instead of names for variables and to chose the numbers in a unique way, such that every α-equivalence class of object-level syntax is represented by exactly one meta-language expression. De Bruijn indices are good for mathematical reasoning, as they give immediate structural recursion and induction principles that respect α-equivalence. However, they are also hard to read for humans and do not correspond to informal reasoning practice, which leads to errors in the formulation of algorithms and proofs. As McBride and McKinna [30] put it:

> "However, we do recommend that anyone planning to use de Bruijn syntax for systematic constructions like the above should think again. Performing constructions in either of these systems requires a lot of arithmetic. This obscures the idea being implemented, results in unreadable, unreliable, unmaintainable code, and is besides hard work."

Locally nameless representation There are hybrid first-order approaches [6, 30], known as *locally nameless representation*, that use explicit names for free names and de Bruijn indices for bound names. These approaches avoid some of the problems of named representations and de Bruijn indices, but in principle still suffer from the same shortcomings. For a more detailed discussion, see Aydemir *et al.* [6, Sections 2, 3 and 4].

Higher-order abstract syntax The higher-order approach of representing object-language binders, called *higher-order abstract syntax* [32, 38], is to use function expressions in the meta-language to represent binding operations. It is very effective in hiding binding issues from the user, for example, object-level capture-avoiding substitution is usually provided directly by the meta-language. However, the notion of 'function' is not absolute in the way that the notion of 'pair' is. As a result it requires some ingenuity (for example, distinguishing functions-as-data from functions-as-computation [24]) to ensure that a meta-language using this form of representation for binders can conveniently express the wide range of first-order syntax-manipulating algorithms commonly employed in informal practice. See for example Savary-Belanger *et al.* [50].

Miller *et al.*'s work on proof theory for higher-order abstract syntax features a proof-level quantifier for locally scoped variables, the ∇-quantifier [33]. It shares some connections with the freshness quantifier $\#$ from Definition 2.3.16, yet the two quantifiers represent different concepts and have different behaviour. Cheney and Gabbay [18, Section 7] relate the quantifiers on the basis of a nominal sequent calculus.

6.1.2 Nominal representation

The nominal approach to object-level syntax utilised in this thesis lies somewhere between the first-order and higher-order representation. Free and especially bound objects are named, leading to a concrete representation of object-level syntax that is faithful to informal practice. Yet we keep the advantage that α-equivalence classes of object-level syntax are uniquely represented in the meta-language, via name abstractions.

Partiality of concretion The concretion operation (as in Definition 2.3.23) is how name abstractions are deconstructed. However, concretion is only defined on fresh names and this partiality causes problems in a meta-language. In fact, Pitts [44, Example 11.7] shows that denotationally the concretion operation is not even monotone (as a partial function). Meta-languages with nominal representation deal with partiality of concretion in different ways. Several languages [10, 45, 51, 63] ensure freshness (in the sense that $a \# [\![e]\!]$ whenever we apply a concretion $e @ a$) statically through the type system. However, freshness in the $a \# [\![e]\!]$ sense is a purely semantic notion, and hence it is undecidable for the usual recursion-theoretic reasons. Therefore static type systems always have to over-approximate freshness, rejecting more programs than necessary. How much static freshness inference impairs programming depends on the specific type system and application; Shinwell *et al.* [57, Section 6] discuss this issue in the context of FreshML 2000[1] [45].

Generative names If we want to keep a standard type system (without freshness in it), then another approach is to have a local scoping construct (for atomic names, written $\nu a.e$ here) in the meta-language. Local scoping corresponds semantically to a name restriction operation (Definition 2.3.26), which can be used to make concretion total as in Proposition 2.3.27. Pitts [43, Section 5] gives a brief history of name restrictions in nominal sets. The most common way to implement local scoping of names, used for example in the `gensym` construct of Lisp, is the so-called *dynamic allocation of names* or *generative names*: we carry around a state of names and dynamically allocate a new name at the local scoping construct. In the style of operational semantics from Section 4.4.2 (with a state of names $A \subseteq_{\mathrm{f}} \mathbb{A}$), generative names can be described by the rule

$$\frac{A \cup \{a\}, e \Downarrow A', c}{A, \nu a.e \Downarrow A', c}\ (a \notin A).$$

The statefulness of generative names can lead to very complicated behaviour when combined with higher-order functions, as Pitts and Stark show with their ν-calculus [46]. This is the cost of using generative names for ensuring freshness of concretion as well as keeping a standard type system (as we do in FreshML [57] and Fresh O′Caml [55]). See also Pitts [44, Section 10.13] for an illustrative discussion of name-induced statefulness in FreshML-like languages. However, if we are willing to switch from the

[1] FreshML 2000 is called FreshML in [45], however the authors decided to rename it, in order to promote the simpler language FreshML from [57].

well-known generative names to less familiar local scoping constructs, then there is in fact a state-free (also called pure or referentially transparent) way to make concretion total, which does not impact the type system: the *Odersky-style local names* we use in this thesis.

Odersky-style local names What is known today as Odersky-style local names were introduced by Odersky [36] as a tool for studying the foundations of functional and imperative programming. Their characteristic features are that they push the local scoping into products ($\nu a.(e_1\,,e_2) \Downarrow (\nu a.e_1\,,\nu a.e_2)$) and functions ($\nu a.\lambda x:\tau \to e \Downarrow \lambda x:\tau \to \nu a.e$) and disappear at fresh names ($\nu a.a' \Downarrow a'$ if $a \neq a'$). In PNA these rules are implemented by the rules in Figure 4.12. Odersky shows that this form of local scoping gives a conservative extension of the λ-calculus [36, Section 5]. Furthermore, our previous work [25] shows that Odersky-style local names are as expressive as generative names in languages with higher-order functions, in the sense that there is a continuation-passing style translation from generative names to Odersky-style local names. Odersky also gives a denotational semantics based on locally complete partial orders [31] to a version of PCF with his style of local names and shows that it is computationally adequate [36, Section 6]. The denotational semantics in this thesis is different, as we discuss below.

Odersky-style local names did not receive much attention when they were introduced in 1994, perhaps because of their unintuitive operational rules. Pitts [42, 43] rediscovered them in the context of nominal sets, while looking for a operational version of the name restriction operations from Lemmas 2.3.28, 2.3.29 and 2.3.30. So in Pitts' work [42, 43] and in this thesis, Odersky-style local names are denoted by name-restriction operations as in Lemmas 2.3.28, 2.3.29 and 2.3.30. The full abstraction results herein show that this is a good match. In [27, Section 8] we show that our full abstraction results specialise to a version of PCF with only Odersky-style local names and without name abstractions.

By using Odersky-style local names for nominal representation, we can keep a state-free semantics and a standard type system, while using name abstractions with a total concretion operation. Additionally the representation of object-level syntax in PNA is '*junk-free*' in the sense that every variable-closed canonical form of type `term` corresponds exactly to an α-equivalence class of the λ-calculus (called λ-term) as given in Example 2.3.22, and name-closed canonical forms correspond exactly to closed λ-terms. This is a consequence of Lemmas 4.2.8 and 4.3.11. By contrast, the generative names of FreshML do not enjoy the 'junk-free' property [57, Remark 5.7].[2]

Meta-level binding In his survey about metaprogramming, Sheard [53, Section 13] writes:

> "We believe the trick to representing object-level binding is to use a binding mechanism of the meta-language."

[2]The calculi in [42, 43] are also not junk-free, even though they use Odersky-style local names in the fashion of PNA. The reason is that they take $\nu a.a$, the anonymous name, to be a distinct canonical form, whereas in PNA $\nu a.a$ is stuck (i.e. it does not evaluate) and denotes bottom $[\![\nu a.a]\!] = \bot$.

We agree and consider PNA to be a case study for this approach. It is crucial for the correctness of syntax-representation in PNA that name abstraction $\alpha a.e$ and local scoping $\nu a.e$ are both binders of the name a in the expressions e, and that meta-level substitution is not just capture-avoiding for variables, but also for names. For example, the correctness of the encoding of object-level substitution for the λ-calculus from (4.1) relies on the fact that whatever we substitute in the meta-language for the variables f and z will not be captured by the name a in $\alpha a.f(z \mathbin{@} a)$.

6.1.3 Domain theory with nominal sets

The idea of using nominal sets in denotational semantics and in particular domain theory is not new. Already the first meta-language using nominal representation, FreshML 2000, was designed with a denotational semantics with name abstractions in mind [15, Remark 21.7] [45, Section 9].

Shinwell-Pitts domain theory A detailed construction of such a denotational semantics with name abstractions is given by Shinwell and Pitts [56, Section 3][54, Chapter 4] for the later language FreshML [57]. Their domain theory is based on nominal posets (Definition 3.1.1) that have joins for all finitely-supported and countable chains of elements.[3] By Proposition 3.2.3 this notion corresponds to nominal posets possessing joins of all sets that are directed, uniformly supported and countable. Generative names are modelled via a continuation monad and this gives a computationally adequate semantics to FreshML [56, Corollary 4.4]. However, full abstraction fails for this semantics, as Shinwell and Pitts [56, Section 4] state. They conjecture that some contextual equivalences of Stark's thesis [59] cannot be validated by their domain theory.

Free restriction semantics Pitts [44, Theorem 9.15] shows that there is a *free nominal restriction set* for each nominal set in the sense of a left adjoint to the forgetful functor from the category **Res** (nominal restriction sets with restriction-preserving and equivariant functions) to **Nom**. This gives rise to a free restriction monad in **Nom** that Pitts [44, Section 9.6] uses to give a monadic denotational semantics to generative names in form of the (recursion-free) ν-calculus [46]. Pitts [private communication] states that, via the categorical equivalences of **Nom** with continuous G-sets and the Shanuel topos (Pitts [44, Sections 6.2 and 6.3]), this free restriction semantics of the ν-calculus is equivalent to the semantics in Stark [59, Section 3.7]. Therefore, by the results in Stark [59], the free restriction semantics is computationally adequate, but not fully abstract. If we wanted to extend the free restriction semantics to languages with recursion, then we probably would have to develop a theory of free uniform-continuous name restrictions for nominal Scott domains. We conjecture this is feasible but leave it for future work, see Open Problem 6.2.2.

[3] Shinwell and Pitts [54, 56] call such a poset FM-cpo and nominal sets are called FM-sets.

Turner-Winskel domain theory HOPLA [35] is a concurrent meta-language possessing a fully abstract domain theory based on down-closed subsets of preorders that form prime-algebraic complete lattices. The Shinwell-Pitts domain theory from above was a motivation for Turner and Winskel [62, 63] to remodel the domain theory of HOPLA with nominal techniques, in order to gain access to name abstraction constructs. Through the permutation action in (2.11) we can consider every nominal set to be a set with empty support; relaxing this to finite support gives us the notion of *FM set* (a precise definition is given for example in Pitts [44, Definition 2.28]). Turner and Winskel base their 'nominal domain theory for concurrency' on (down-closed and finitely supported subsets of) FM-preorders, that is, nominal preorders in the sense of Definition 3.1.1 whose underlying set is an FM-set. Modulo countability, their category $\mathbf{FMCts}_{\emptyset}$ is a full subcategory of **Nsd**. It was this work in which the notion of 'uniform-directedness' and a characterisation of uniform-compact elements in terms of the 'hull' construct (from Definition 2.3.35) first appeared.

Turner and Winskel derive the language Nominal HOPLA from their domain theory. Nominal HOPLA has name abstractions, concretions and nondeterministic sums over names; partiality of concretion is dealt with by having freshness assumptions in the type system. The Nominal HOPLA denotational semantics is computationally adequate, but not fully abstract [62, Section 7.1.1]. More information about ongoing work on the full abstraction problem for Nominal HOPLA is given in Open Problem 6.2.6.

Orbit-finite subsets and hulls The notion of orbit-finite subset (Definition 2.3.34) is central for the nominal domain theory of this thesis, as visualised in (1.1). This relaxation from 'finite' to 'finite modulo symmetry' proved to be useful for automata over infinite alphabets and led to the development of what one might call orbit-finite automata theory [9, 19, 34, 65].

Hulls (Definition 2.3.35) provide a finite representation of orbit-finite subsets that is suitable for computation. The notion of hulls was introduced independently by Turner [63, Definition 3.4.3.2], Gabbay [16, Section 3.3] [17, Definition 3.1] [19, Definition 3.1] and Bojańczyk *et al.* [8, Section 8], whose 'hull' terminology we adopt here. Furthermore, the 'closures' of Ciancia and Montanari [11, Definition 6.10] are hulls of the form $\mathrm{hull}_{\mathrm{supp}\, x - \{a\}}\{x\}$.

The hull characterisation of orbit-finite subsets (Theorem 2.3.38) was observed by Bojańczyk *et al.* in a generalised version of nominal sets over any 'Fraïssé symmetry' [8, Lemma 6]. Turner and Winskel discovered a characterisation of the uniform-compact elements in their domain theory (see above) in terms of the hull construct [63, Lemma 1]. Our Theorem 3.3.6 draws a new connection between these results, relating orbit-finiteness with uniform-compactness.

Full abstraction for local names A defining feature of domain theory is that expressions of function type are denoted by ordinary functions. Those functions are *extensional* in the sense that they are equal if and only if they give the same results for every input. Other variants of denotational semantics are more *intensional*, meaning

that expressions of function type are denoted by mathematical objects that are not solely determined by their input-output behaviour. For example, in game semantics quotienting of game strategies by an equivalence relation is needed in the denotations to make equality in the model coincide with contextual equivalence.

To the best of our knowledge, the results of this thesis provide the first full abstraction result for languages combining higher-order functions with some form of locally scoped names (generative or Odersky-style) which uses a denotational semantics based on extensional functions. So far the only denotational model of local scoping with higher-order functions that is known to be fully abstract makes use of game semantics [2, 23, 34, 64]. Those game-semantics-based full abstractions results target generative names and we discuss game semantics for Odersky-style local names in Open Problem 6.2.3.

It is worth noting that the full abstraction results in this thesis depend crucially upon the fact that PNA uses Odersky-style local names, rather than generative ones. There is no extensional full abstraction result known for FreshML [57] or the simpler ν-calculus, which use generative rather than Odersky-style local names to implement the features that PNA provides for programming with name abstractions; and yet we believe that PNA (extended with recursive types) is in principle as expressive as FreshML, in light of our previous work [25].

6.2 Open problems

We list a number of open problems related to this thesis. They identify potential directions for future research.

Open Problem 6.2.1. *For an arbitrary* PNA^+ *type* τ*, are the uniform-compact elements of* $[\![\tau]\!]$ PNA^+*-definable?*

In Theorem 5.5.18 we show that definability of uniform-compact elements holds for the restricted set of simple types, and this suffices for our proof of full abstraction. What about PNA^+-types that are not simple types? Remark 5.5.21 points out that if our definable retracts from Proposition 5.5.12 were actually definable embedding-projection pairs, then this would prove definability at all types. However, the same remark shows that (5.49) and (5.50) do *not* form an embedding-projection pair, so it seems that PNA^+-types are not definably embeddable into simple types. As a result uniform-compact definability at all types remains an open problem for PNA^+.

Open Problem 6.2.2. *What recursive domain equations can be solved in* **Nsd***?*

In his thesis, Shinwell [54, Section 4.5] shows that the traditional method for constructing minimally invariant solutions for locally continuous functors of mixed variance can be applied to the simple notion of nominal domain from the Shinwell-Pitts domain theory above. Pitts [44, Section 11.4] extends this to udcpos. An interesting alternative approach is to develop a nominal version of Scott's information systems [52] and construct solutions for recursive domain equations via inductively defined nominal sets of information tokens. We have begun to develop such a theory of *nominal Scott information systems* in which the role of finite subsets is replaced by orbit-finite subsets. From a logical point of view [1], nominal information systems are

presentations of non-trivial nominal posets *with all orbit-finite meets*, rather than just finite meets. We expect this machinery can be used to good effect for the orbit-finite power domain construct mentioned in Open Problem 6.2.4, as well as for a version for nominal Scott domains of the free name restriction monad from Section 6.1.3.

Open Problem 6.2.3. *Is there a fully abstract model of* PNA *based on games in nominal sets?*

Just as PCF is of more interest from a programming point of view than PCF+por, we regard PNA to be interesting in its own right as a functional metaprogramming language. Game semantics provided an interesting solution for the original full abstraction problem for PCF [3, 22], and its nominal version has provided computationally useful, fully abstract models of generative local state [2, 23, 34, 64]. Can nominal game semantics provide a similar thing for PNA with its Odersky-style local names?

Open Problem 6.2.4. *Is there a nominal Scott domain semantics for the form of nominal computation embodied by the* Nλ *language?*

With their language Nλ [8], Bojańczyk *et al.* extend the simply-typed λ-calculus with a collection type representing orbit-finite subsets via a syntax for hulls (Definition 2.3.35).[4] It is natural to consider adding fixed point recursion to this language, with a denotational semantics using nominal Scott domains rather than nominal sets. The denotational semantics of such an extension of Nλ will require the development of orbit-finite power domains $\mathsf{F}_{\mathrm{n}}D$ in **Nsd**, whose uniform-compact elements are orbit-finite subsets of the uniform-compact elements of D.

Open Problem 6.2.5. *Do Odersky-style local names give a well-behaved semantics for call-by-value and call-by-need?*

This thesis presents evidence that Odersky-style local names (see Sections 4.4.1 and 6.1.2) are a good match for call-by-name languages. In PNA, Odersky-style local names enable state-free metaprogramming, which could also be beneficial for call-by-value languages (such as OCaml [`http://ocaml.org`]) or call-by-need languages (such as Haskell [`http://www.haskell.org`]), potentially leading to more 'real world' applications of Odersky-style local names[5]. How do we check if the resulting languages are well-behaved? It is always a good idea to evaluate some example programs (such as the ones from Section 4.5.2) to see if they give the expected outcome. Alternatively, we can check if semantic results can be carried over from the base language to the extended language. For example, Sieber [58] shows that a call-by-value version of PCF is fully abstract for a denotational semantics based on (non-pointed) cpo's and partial continuous functions. Can we replicate this result with a call-by-value version of PNA?

Open Problem 6.2.6. *Is there a fully abstract domain theory for a version of* HOPLA *with name abstractions?*

[4] Their paper [8] is concerned with general 'Fraïssé nominal sets'. Here we restrict our attention to the 'equality symmetry' and nominal sets in the original sense.

[5] In PNA, evaluation of the expression $va.a$ is undefined (i.e. it is 'stuck'), which suffices for the semantic results in this thesis. In a language intended for practical use, evaluating $va.a$ should preferably result in a comprehensible error message.

The concurrent meta-language HOPLA [35] mentioned in Section 6.1.3 cannot express languages that use locally scoped names (or channels), such as the π-calculus. The development of the Turner-Winskel domain theory from Section 6.1.3 was motivated by the goal of extending HOPLA with name abstractions for binding, while carrying over the full abstraction results from HOPLA to its extension. The resulting language Nominal HOPLA [63] can express the π-calculus, but fails to be fully abstract. In hitherto unpublished work, the author of this thesis has begun to simplify the Turner-Winskel domain theory, through switching from FM-preorders to nominal preorders as underlying construct. The hope is that this will lead to a version of Nominal HOPLA that has a fully abstract domain theory. There are promising partial results, in particular, name abstractions are well-behaved in this novel domain theory. However, for making concretion total in the style of this thesis (Section 3.6.3), one needs to find an appropriate name restriction operation, and this has been difficult thus far. Avoiding this issue via freshness assumptions in the type system (as it is done in [63]) seems not to be viable, as consequently some expressions needed for full abstraction fail to typecheck.

6.3 Summary

The results in this thesis provide further evidence for how a semantic theory (domain theory in this case) is enhanced by using nominal sets: we gain the ability to model constructs involving names and their symmetries while preserving most aspects of the classical theory. The complications arising from our nominal approach can be feasibly dealt with and are somehow orthogonal to the other developments.

At the same time, the use of nominal sets gives access to new constructs that are far from trivial. This is the case for the notion of *orbit-finite subset*, which formalises the important idea of finiteness modulo symmetry within nominal sets. The nominal constructs in the domain theory guide the design of a state-free extension of PCF with metaprogramming constructs: the language PNA. Its state-freeness gives PNA good semantic properties, in particular, many program equivalences are easy to prove by using the denotational semantics. Furthermore, the object-level representation of syntax in PNA is 'junk-free' in the sense of Section 6.1.2.

We claim that PNA metaprogramming constructs are natural to use and that they successfully hide α-equivalence issues from the programmer. For example, the definition of object-level capture-avoiding substitution is pleasingly simple in PNA (4.1), especially when using some syntactic sugar (4.12).

Our quest for full abstraction led to two new programming language constructs, existential quantification over names and definite description over names. They play a similar role for PNA as 'parallel-or' does for PCF. For the extended language PNA^+, the full abstraction property ensures that every contextual equivalence is (in principle) provable by checking equalities in our nominal domain theory. In this sense full abstraction shows that our nominal domain theory is the 'best' denotational semantics possible for PNA^+.

APPENDIX A

PROOF DETAILS

A.1 Proof of Lemma 3.1.7

Let a nominal poset D and $S \subseteq_{\text{fs}} [\mathbb{A}]D$ be given. For all $a \in \mathbb{A}$ we define the sets

$$S'_a \triangleq \{d \in D \mid \langle a \rangle d \in S\}$$

Each S'_a is a finitely supported subset of D and for $a, a' \# S$ we have $(a\ a') \cdot S'_a = S'_{a'}$. If for some $a \# S$ the join $\bigsqcup S'_a$ exists in D, then by Proposition 3.1.4 we know that for any $a' \# S$ the join $\bigsqcup S'_{a'}$ exists and satisfies $\bigsqcup S'_{a'} = (a\ a') \cdot \bigsqcup S'_a$. With this we can apply Lemma 2.3.15 and Definition 2.3.16 to get that

$$\text{fresh } a \text{ in } \langle a \rangle (\bigsqcup S'_a) \tag{A.1}$$

is a well-defined element of $[\mathbb{A}]D$. We claim that $\bigsqcup S =$ (A.1). To see that (A.1) is an upper bound, let any $e \in S$ be given, and pick $a \# S, e$. By Lemma 2.3.25 we have $\langle a \rangle (e @ a) = e \in S$ and hence $e @ a \in S'_a$, which implies $e @ a \sqsubseteq \bigsqcup S'_a$. By (3.5) this leads to $e = \langle a \rangle (e @ a) \sqsubseteq \langle a \rangle (\bigsqcup S'_a) =$ (A.1). For showing that (A.1) is the join, let any other upper bound $e \in [\mathbb{A}]D$ be given and pick any $a \# S, e$. For any $d \in D$ we have that $\langle a \rangle d \in S$ implies $\langle a \rangle d \sqsubseteq e = \langle a \rangle (e @ a)$ and hence by (3.5) we have $d \sqsubseteq e @ a$. As d is arbitrary we get $\bigsqcup S'_a \sqsubseteq e @ a$ and again with (3.5) we obtain (A.1) $= \langle a \rangle \bigsqcup S'_a \sqsubseteq \langle a \rangle (e @ a) = e$, which was to show.

What remains to show is that if S is bounded/directed/uniformly supported then so is S'_a (with $a \# S$). For boundedness, let e be the upper bound of S and let any $d \in S'_a$ be given. We will show that $(a\ b) \cdot (e @ b)$ is an upper bound for S'_a for any $b \# S, e$. We know $\langle a \rangle d \in S$, so $\langle b \rangle (a\ b) \cdot d \in S$ by $a, b \# S$. As e is an upper bound for S and $b \# e, \langle b \rangle (a\ b) \cdot d$ we get $(a\ b) \cdot d \sqsubseteq e @ b$, which implies $d \sqsubseteq (a\ b) \cdot (e @ b)$. A very similar argument gives the proof for directedness. For the uniform support property, let A be a uniform support of S. We show that $A \cup \{a\}$ is a uniform support of S'_a, so let $\pi \# A \cup \{a\}$ and $d \in S'_a$ be given. As $\pi \# A$ and $\langle a \rangle d \in S$, we get $\langle a \rangle d = \pi \cdot \langle a \rangle d = \langle \pi\, a \rangle (\pi \cdot d) = \langle a \rangle (\pi \cdot d)$, so $d = \pi \cdot d$ follows by Lemma 2.3.21.

If S is uniformly supported, we can describe its join in an alternative way. By the above argument we have $\bigsqcup S = \langle a \rangle (\bigsqcup \{d \in D \mid \langle a \rangle d \in S\})$ for some/any $a \# S$ and

by Lemma 2.3.41 we know $a \# e$ for all $e \in S$. With Lemma 2.3.25 this leads to the following characterisation

$$\bigsqcup S = \text{fresh}\, a \text{ in } \langle a \rangle (\bigsqcup \{ e @ a \mid e \in S \}) \qquad \text{for uniformly supported } S \tag{A.2}$$

assuming that $\{ e @ a \mid e \in S \}$ has a join in D. □

A.2 Proof of Lemma 3.4.3

Let $D \in \mathbf{Nsd}$ and $S \subseteq_{\text{fs}} D$ with upper bound $d \in D$ be given, and define

$$B \triangleq \{ u \in \mathrm{K}D \mid (\exists s \in S)\, u \sqsubseteq s \}$$
$$C \triangleq \{ \bigsqcup \text{hull}_{\text{supp}\, S} F \mid F \subseteq_{\text{f}} B \}.$$

For any $F \subseteq_{\text{f}} B$ the set $\text{hull}_{\text{supp}\, S} F$ is bounded by d, is orbit-finite (by Theorem 2.3.38) and consists of uniform-compact elements (by Lemma 3.3.3). Therefore $\bigsqcup \text{hull}_{\text{supp}\, S} F$ exists by the definition of nominal Scott domains, so C is well-defined.

Next we show that C is uniform-directed, thereby proving that $\bigsqcup C$ exists in D. C is uniformly supported by $\text{supp}\, S$, because by (3.3) and (2.27) we have for any $F \subseteq_{\text{f}} B$ that $\text{supp}(\bigsqcup \text{hull}_{\text{supp}\, S} F) \subseteq \text{supp}(\text{hull}_{\text{supp}\, S} F) \subseteq \text{supp}\, S$. It is directed, because any two $\text{hull}_{\text{supp}\, S} F_1, \text{hull}_{\text{supp}\, S} F_1 \in C$ have the upper bound $\text{hull}_{\text{supp}\, S}(F_1 \cup F_2) \in C$.

What remains to be shown is that $\bigsqcup C = \bigsqcup S$. First, we prove that $\bigsqcup C$ is an upper bound for S and therefore $\bigsqcup S \sqsubseteq \bigsqcup C$. Let any $s \in S$ be given, by Lemma 3.3.2 we have $s = \bigsqcup \{ u \in \mathrm{K}D \mid u \sqsubseteq s \}$. For any $u \in \mathrm{K}D$ with $u \sqsubseteq s$ we know $u \in \text{hull}_{\text{supp}\, S} \{u\}$ and we also know by $u \in B$ that $\bigsqcup \text{hull}_{\text{supp}\, S} \{u\} \in C$, so overall we get $u \sqsubseteq \bigsqcup \text{hull}_{\text{supp}\, S} \{u\} \sqsubseteq \bigsqcup C$. This shows that $\bigsqcup C$ is an upper bound for $\{ u \in \mathrm{K}D \mid u \sqsubseteq s \}$ and hence $s \sqsubseteq \bigsqcup C$ holds.

For proving $\bigsqcup C \sqsubseteq \bigsqcup S$ let x' be any upper bound of S. For any $F \subseteq_{\text{f}} B$ we get that x' is also an upper bound of $\text{hull}_{\text{supp}\, S} F$, so $\bigsqcup \text{hull}_{\text{supp}\, S} F \sqsubseteq x'$. As F was chosen arbitrarily, x' is furthermore an upper bound of C and this gives $\bigsqcup C \sqsubseteq x'$. □

A.3 Proof of Proposition 3.4.27

Proposition 3.4.18 shows that $D_1 \to_{\text{uc}} D_2$ is a pointed udcpo that is bounded-complete. For algebraicity, we show that any $f \in D_1 \to_{\text{uc}} D_2$ satisfies

$$f = \bigsqcup S_f \tag{A.3}$$

where

$$S_f \triangleq \{ \bigsqcup \text{hull}_{\text{supp}\, f} F \mid F \subseteq_{\text{f}} K_f \}$$
$$K_f \triangleq \{ (u_1 \searrow u_2) \in (D_1 \to_{\text{step}} D_2) \mid (u_1 \searrow u_2) \sqsubseteq f \}.$$

It is easy to see that for every $F \subseteq_{\mathrm{f}} K_f$ the set $\mathrm{hull}_{\mathrm{supp}\, f} F$ is bounded by f and is through (2.27) supported by $\mathrm{supp}\, f$. Therefore $\bigsqcup \mathrm{hull}_{\mathrm{supp}\, f} F$ exists and is uniform-compact by Lemmas 3.3.3, 3.3.4 and 3.4.23. This shows that S_f consists of uniform-compact elements, is bounded by f, is uniformly supported by $\mathrm{supp}\, f$ and is directed (since $\mathrm{hull}_A(\_)$ preserves inclusions), so its join exists and satisfies $\bigsqcup S_f \sqsubseteq f$.

To prove (A.3) we still need $f \sqsubseteq \bigsqcup S_f$, which by Lemma 3.4.19 holds if $(\forall u_1 \in \mathrm{K}D_1)\ f\, u_1 \sqsubseteq (\bigsqcup S_f)\, u_1$. Let any $u_1 \in \mathrm{K}D_1$ be given, since D_2 is algebraic, we have $f\, u_1 = \bigsqcup U_2$ for some uniform-directed subset $U_2 \subseteq \mathrm{K}D_2$. For each $u_2 \in U_2$, $u_2 \sqsubseteq \bigsqcup U_2 = f\, u_1$ and hence $(u_1 \searrow u_2) \sqsubseteq f$. Therefore $(u_1 \searrow u_2) \in K_f$; and since $(u_1 \searrow u_2) \in \mathrm{hull}_{\mathrm{supp}\, f}\{(u_1 \searrow u_2)\}$ we get $(u_1 \searrow u_2) \sqsubseteq \bigsqcup \mathrm{hull}_{\mathrm{supp}\, f}\{(u_1 \searrow u_2)\} \in S_f$. Hence $(u_1 \searrow u_2) \sqsubseteq \bigsqcup S_f$ and thus $u_2 = (u_1 \searrow u_2)\, u_1 \sqsubseteq (\bigsqcup S_f)\, u_1$. Since this is true for each $u_2 \in U_2$, we get $f\, u_1 = \bigsqcup U_2 \sqsubseteq (\bigsqcup S_f)\, u_1$, as required.

What remains to be shown is the characterisation of $\mathrm{K}(D_1 \to_{\mathrm{uc}} D_2)$ from (3.18), which is then a countable set by Corollary 2.3.39. Lemma 3.4.23 together with Lemma 3.3.4 gives the right-to-left inclusion of (3.18). For the left-to-right inclusion, let any $f \in \mathrm{K}(D_1 \to_{\mathrm{uc}} D_2)$ be given. By (A.3) and uniform-compactness of f we know that $f \sqsubseteq \bigsqcup \mathrm{hull}_{\mathrm{supp}\, f} F$ for some $F \subseteq_{\mathrm{f}} K_f$, and by the above argument $\bigsqcup \mathrm{hull}_{\mathrm{supp}\, f} F \sqsubseteq f$. Hence $f = \bigsqcup \mathrm{hull}_{\mathrm{supp}\, f} F$ and by Theorem 2.3.38, (3.15) and Lemma 3.4.24 we have that $\mathrm{hull}_{\mathrm{supp}\, f} F$ is an orbit-finite and consistent set of step functions. □

A.4 Proof of Lemma 5.3.16

For showing that (5.34) implies (5.33), we use transitivity of $\leqq_\tau^{\mathrm{wk}}$ (Lemma 5.3.9) and prove $e \leqq_\tau^{\mathrm{wk}} \nu a'.(a\, a') \cdot (\alpha a.\, e \mathbin{@} a') \leqq_\tau^{\mathrm{wk}} \alpha a.\, e \mathbin{@} a$ for some/any $a' \# a, e$. The second pre-order relation is a consequence of (5.34), whereas the first one can be proved directly by using Lemmas 4.4.2, 4.4.3, 4.4.4, 4.4.10 and (5.27).

For the proof of (5.34), suppose $a' \# (a, e)$ and

$$\nu a'.(a\, a') \cdot (e \mathbin{@} a') \Downarrow c\,. \tag{A.4}$$

We have to show $e \mathbin{@} a \Downarrow c'$, for some c' with $c \leq_\tau^{\mathrm{wk}} c'$. But (A.4) can only hold because for some c_1 we have

$$(a\, a') \cdot (e \mathbin{@} a') \Downarrow c_1 \tag{A.5}$$

$$a' \backslash\backslash c_1 := c\,. \tag{A.6}$$

From (A.5) we get (by equivariance of $\Downarrow$, Lemma 4.4.10) that $e \mathbin{@} a' \Downarrow (a\, a') \cdot c_1$; and hence we must have for some $a'' \# (a, a', e, c, c_1)$ and c_2 that

$$e \Downarrow \alpha a''.\, c_2 \tag{A.7}$$

$$a'' \backslash\backslash (a''\, a') \cdot c_2 := (a\, a') \cdot c_1\,. \tag{A.8}$$

Since $a' \# e$ we get $a' \# c_2$ from (A.7) and Lemma 4.4.11, and hence $a'' \# (a''\ a') \cdot c_2$. Then by (5.27) and (A.8) we have $(a\ a') \cdot c_1 \leq_\tau^{\mathrm{wk}} (a''\ a') \cdot c_2$ and hence by Lemma 5.3.10 also

$$c_1 \leq_\tau^{\mathrm{wk}} (a\ a')(a''\ a') \cdot c_2. \tag{A.9}$$

Through applying (5.22) to (A.6) and (A.9) we know that there exists c' with

$$a' \backslash\backslash (a\ a')(a''\ a') \cdot c_2 := c' \ \wedge\ c \leq_\tau^{\mathrm{wk}} c'. \tag{A.10}$$

Note that since $a' \# c_2$ it holds that $\alpha a''.\, c_2 = \alpha a'.\, (a''\ a') \cdot c_2$, and therefore from (A.7), $e \Downarrow \alpha a'.\, (a''\ a') \cdot c_2$. Combining this with (A.10) we get $e \mathbin{@} a \Downarrow c'$ and $c \leq_\tau^{\mathrm{wk}} c'$, as required. □

A.5 Proof of Lemma 5.4.2

The proof works by induction on i. As we identify configurations by α-equivalence, we may assume that all the binding names in F are chosen to be fresh for a (and hence fresh for $\vec{e}$), so by Lemma 5.4.1 it holds that $\langle F, e\rangle[\vec{e}/\vec{x}] = \langle F[\vec{e}/\vec{x}], e[\vec{e}/\vec{x}]\rangle$. We also know for this choice of bound names that $a \# \langle F, e\rangle$ holds if and only if $a \# F, e$.

In the base case of the induction $i = 0$, we must have $|F[\vec{e}/\vec{x}]| = |\mathrm{Id}|$ by Lemma 4.4.18, so $F = \mathrm{Id}$, and hence we must also have $e[\vec{e}/\vec{x}] = c$ for $c \in \mathrm{Can}^{\mathsf{PNA}^+}(\gamma)$. By typing we know that e cannot be a variable (otherwise $e[\vec{e}/\vec{x}]$ would be ill-typed), thus we get $e = c$ and the property follows from that.

For the inductive step, assume $\langle F[\vec{e}/\vec{x}], e[\vec{e}/\vec{x}]\rangle \to \langle F', e'\rangle \to^i \langle \mathrm{Id}, c\rangle$ and let $e'_1, \ldots, e'_j \in \mathit{KBot}$ be given. We make a case distinction on the first transition and then we analyse the structure of F, e, F' and e' based on the form of the transition, using the determinacy of the frame-stack transitions (Lemma 5.2.6). We need to take into account the possibility that e is a variable. This possibility will be treated later, so for now assume $e \notin \mathbb{V}$.

We prove the case for the successor transition in detail: Assume that the first transition $\langle F[\vec{e}/\vec{x}], e[\vec{e}/\vec{x}]\rangle \to \langle F', e'\rangle$ is of the form $\langle F_1, \mathtt{S}\, e_1\rangle \to \langle F_1 \circ \mathtt{S}\cdot, e_1\rangle$. We know by the definition of substitution that $e = \mathtt{S}\, e'_1$ and $e'_1[\vec{e}/\vec{x}] = e_1$, and by the rules of the frame-stack transitions we know $\langle F[\vec{e}/\vec{x}], \mathtt{S}\, e'_1[\vec{e}/\vec{x}]\rangle \to \langle F[\vec{e}/\vec{x}] \circ \mathtt{S}\cdot, e'_1[\vec{e}/\vec{x}]\rangle$ and $\langle F[\vec{e}/\vec{x}] \circ \mathtt{S}\cdot, e'_1[\vec{e}/\vec{x}]\rangle = \langle F \circ \mathtt{S}\cdot, e'_1\rangle[\vec{e}/\vec{x}]$ by the definition of substitution for frame-stacks and configurations. Determinacy of the frame-transitions (Lemma 5.2.6) implies $\langle F', e'\rangle = \langle F \circ \mathtt{S}\cdot, e'_1\rangle[\vec{e}/\vec{x}]$ and we know by $a \# F, \mathtt{S}\, e'_1$ that $a \# F \circ \mathtt{S}\cdot, e'_1$ so we can apply induction to obtain $\langle F \circ \mathtt{S}\cdot, e'_1\rangle[\vec{e'}/\vec{x}] \to^* \langle \mathrm{Id}, c\rangle$. By the frame-stack transition rules, substitution for configurations and $\mathrm{fn}\, \vec{e'} = \emptyset$ we also know that $\langle F, \mathtt{S}\, e'_1\rangle[\vec{e'}/\vec{x}] \to \langle F \circ \mathtt{S}\cdot, e'_1\rangle[\vec{e'}/\vec{x}]$ which concludes this case.

The other cases where $e \notin \mathbb{V}$ work in a similar way. Several cases involving canonical forms, such as the case for $\langle F \circ \mathtt{S}\cdot, c\rangle$, apply Lemma 4.1.6 to get $e[\vec{e}/\vec{x}] \in \mathrm{Can}^{\mathsf{PNA+the}} \Leftrightarrow e[\vec{e'}/\vec{x}] \in \mathrm{Can}^{\mathsf{PNA+the}}$. The cases whose frame-stack transitions are defined via substitutions, such as the case for $\langle F \circ \cdot\, e_2, \lambda x : \tau \to e\rangle$, use Lemma 4.1.3 and Lemma 4.1.4. The case for $\langle F \circ (a_1 \doteq a_2)\cdot, c\rangle$ uses Lemma 4.1.3. The cases for $\langle F \circ \nu a'.\cdot, c\rangle$ and $\langle F \circ \alpha a'.\, c \mathbin{@} \cdot, a''\rangle$ use Lemma 4.4.9.

The case for $\langle F[\vec{e}/\vec{x}], e[\vec{e}/\vec{x}]\rangle = \langle F_1, \mathtt{the}\, y.\, e_1\rangle$ where $e \notin \mathbb{V}$ is the most subtle one, therefore we give a detailed argument for it. We may assume by a similar argument as above that $y \notin \vec{x}$, $a \# F, \mathtt{the}\, y.\, e$ and that the transitions are of the form

$$\langle F[\vec{e}/\vec{x}], \mathtt{the}\, y.\, e[\vec{e}/\vec{x}]\rangle \to \langle F[\vec{e}/\vec{x}], \nu a_{m+1}.\, \mathtt{case}_{\mathtt{bool}}(e[\vec{e}/\vec{x}][a_1/y], e[\vec{e}/\vec{x}][a_2/y], \ldots, e[\vec{e}/\vec{x}][a_m/y], e[\vec{e}/\vec{x}][a_{m+1}/y])\ \mathtt{of}\ ((\mathtt{T},\mathtt{F},\ldots,\mathtt{F},\mathtt{F}) \to a_1 \mid (\mathtt{F},\mathtt{T},\ldots,\mathtt{F},\mathtt{F}) \to a_2 \mid \ldots \mid (\mathtt{F},\mathtt{F},\ldots,\mathtt{T},\mathtt{F}) \to a_m \mid \_ \to \mathtt{bot}_{\mathtt{name}})\rangle \to^i \langle \mathrm{Id}, c\rangle$$

where $a_1, \ldots, a_m = \mathrm{fn}(e[\vec{e}/\vec{x}])$ and $a_{m+1} \# e[\vec{e}/\vec{x}]$, so potentially $a \in \{a_1, \ldots, a_m\}$ and we may assume $a \neq a_{m+1}$. Note that $\mathrm{fv}\,\vec{e} = \emptyset$ and $y \notin \vec{x}$, so $e[\vec{e}/\vec{x}][a_j/y] = e[a_j/y][\vec{e}/\vec{x}]$ holds for all $j \in \{1, \ldots, m+1\}$ by Lemma 4.1.4. By Lemma 4.4.23 and typing it must also hold that

$$\langle \mathrm{Id}, \nu a_{m+1}.\, \mathtt{case}_{\mathtt{bool}} \ldots\rangle \to^{i'} \langle \mathrm{Id}, a'\rangle \ \wedge\ \langle F[\vec{e}/\vec{x}], a'\rangle \to^{i-i'} \langle \mathrm{Id}, c\rangle$$

In this situation we make the following observations.

- By the definition of $\mathtt{case}_{\mathtt{bool}}$ (4.6) it must be that $a' \in \{a_1, \ldots, a_m\}$, and assume without loss of generality $a' = a_1$. Again by the definition of $\mathtt{case}_{\mathtt{bool}}$ it then must hold that $\langle \mathrm{Id}, e[a_1/y][\vec{e}/\vec{x}]\rangle \to^{i_1} \langle \mathrm{Id}, \mathtt{T}\rangle$ and for all $m' \in \{2, \ldots, m+1\}$ that $\langle \mathrm{Id}, e[a_{m'}/y][\vec{e}/\vec{x}]\rangle \to^{i_{m'}} \langle \mathrm{Id}, \mathtt{F}\rangle$, where altogether $i_1 + \ldots + i_{m+1} < i'$.
- We show $a \neq a_1$ by contradiction. If $a = a_1$ was true, then by Theorem 4.4.26 we would have $e[a/y][\vec{e}/\vec{x}] \Downarrow \mathtt{T}$. By the above bullet point we have $\langle \mathrm{Id}, e[a_{m+1}/y][\vec{e}/\vec{x}]\rangle \to^{i_{m+1}} \langle \mathrm{Id}, \mathtt{F}\rangle$ and $a \# \mathrm{Id}, e[a_{m+1}/y]$, so we can apply induction on this. It gives us for any $\vec{e'} \in KBot^j$ that $\langle \mathrm{Id}, e[a_{m+1}/y][\vec{e'}/\vec{x}]\rangle \to^* \langle \mathrm{Id}, \mathtt{F}\rangle$, hence by Theorem 4.4.26 we have $e[a_{m+1}/y][\vec{e'}/\vec{x}] \Downarrow \mathtt{F}$. We know $a, a_{m+1} \# e, \vec{e'}$ and with Lemmas 4.4.10 and 4.1.3 this gives $e[a/y][\vec{e'}/\vec{x}] \Downarrow \mathtt{F}$. It holds also that $[\![\vec{e'}]\!] = \bot$, so with Lemma 5.3.21 we obtain $e[a/y][\vec{e}/\vec{x}] \Downarrow \mathtt{F}$. However, this gives a contradiction to Lemma 4.4.13 with our earlier assumption $e[a/y][\vec{e}/\vec{x}] \Downarrow \mathtt{T}$.
- Similarly we can show for any $b \# e, a$ that $\langle \mathrm{Id}, e[b/y][\vec{e'}/\vec{x}]\rangle \to^* \langle \mathrm{Id}, \mathtt{F}\rangle$.
- As $a \neq a_1$ we can apply induction on $\langle F[\vec{e}/\vec{x}], a'\rangle \to^{i-i'} \langle \mathrm{Id}, c\rangle$ and $\langle \mathrm{Id}, e[a_1/y][\vec{e}/\vec{x}]\rangle \to^{i_1} \langle \mathrm{Id}, \mathtt{T}\rangle$ to obtain $\langle F[\vec{e'}/\vec{x}], a'\rangle \to^* \langle \mathrm{Id}, c\rangle$ as well as $\langle \mathrm{Id}, e[a_1/y][\vec{e'}/\vec{x}]\rangle \to^* \langle \mathrm{Id}, \mathtt{T}\rangle$. By combining the arguments above we also know for all $b \in \mathbb{A} - \{a_1\}$ that $\langle \mathrm{Id}, e[b/y][\vec{e'}/\vec{x}]\rangle \to^* \langle \mathrm{Id}, \mathtt{F}\rangle$. Furthermore by $a \neq a_1$ and $a_1 \in \mathrm{fn}(e[\vec{e}/\vec{x}])$ we get $a_1 \in \mathrm{fn}\, e$ and hence also $a_1 \in \mathrm{fn}(e[\vec{e'}/\vec{x}])$.

Overall we then observe by the definition of the frame-stack transitions that

$$\langle F[\vec{e'}/\vec{x}], \mathtt{the}\, y.\, e[\vec{e'}/\vec{x}]\rangle \to \langle F[\vec{e'}/\vec{x}], \nu a'_{m'+1}.\, \mathtt{case}_{\mathtt{bool}} \ldots\rangle \to^* \langle F[\vec{e'}/\vec{x}], a_1\rangle \to^* \langle \mathrm{Id}, c\rangle$$

which concludes this case.

Let us now focus on the cases where $e \in \mathbb{V}$, say $e = x$. We know that $x \in \vec{x}$ because otherwise $F[e]$ would be ill-typed, and from this it follows that $e[\vec{e}/\vec{x}] = \mathtt{eqBot}_{a,n}$

for some $n \in \mathbb{N}$. By typing we only need to consider the following four cases for the initial configuration: $\langle (F \circ \cdot\, e_2)[\vec{e}/\vec{x}], \mathtt{eqBot}_{a,n}\rangle$, $\langle (F \circ (a_1 \rightleftharpoons a_2)\cdot)[\vec{e}/\vec{x}], \mathtt{eqBot}_{a,n}\rangle$, $\langle (F \circ \nu a'.\cdot)[\vec{e}/\vec{x}], \mathtt{eqBot}_{a,n}\rangle$ and $\langle (F \circ \alpha a'.\cdot)[\vec{e}/\vec{x}], \mathtt{eqBot}_{a,n}\rangle$.

- In the first case assume $\langle (F \circ \cdot\, e_2)[\vec{e}/\vec{x}], \mathtt{eqBot}_{a,n}\rangle \to^{i+1} \langle \mathrm{Id}, c\rangle$. We will end up with a contradiction that shows that this cannot be true. By the determinacy of the frame-stack transitions, Lemma 5.2.6, we know that the transitions are of the following form

  $$\begin{aligned}&\langle (F \circ \cdot\, e_2)[\vec{e}/\vec{x}], \mathtt{eqBot}_a^n\rangle\\ &\to^{n+3} \langle \underbrace{F[\vec{e}/\vec{x}] \circ \nu b_1.\cdot \circ \ldots \circ \nu b_n.\cdot \circ \mathtt{if}\cdot\mathtt{then\ T\ else\ bot}_{\mathtt{bool}} \circ \cdot = a}_{\triangleq F'}, e_2[\vec{e}/\vec{x}]\rangle\\ &\to^{i-n-2} \langle \mathrm{Id}, c\rangle.\end{aligned}$$

  By Lemma 4.4.23 and typing this can only hold if $\langle \mathrm{Id}, e_2[\vec{e}/\vec{x}]\rangle \to^m \langle \mathrm{Id}, a'\rangle$ and $\langle F', a'\rangle \to^{i-n-2-m} \langle \mathrm{Id}, c\rangle$ for some $a' \in \mathbb{A}$. We know $a \# e_2$ by assumption and certainly $m < i$, so we can use induction on $\langle \mathrm{Id}, e_2[\vec{e}/\vec{x}]\rangle \to^m \langle \mathrm{Id}, a'\rangle$ to obtain $\langle \mathrm{Id}, e_2[\vec{e'}/\vec{x}]\rangle \to^* \langle \mathrm{Id}, a'\rangle$, and by Theorem 4.4.26 this gives $e_2[\vec{e'}/\vec{x}] \Downarrow a'$. Note that $a \# e_2[\vec{e'}/\vec{x}]$ as $a \# e_2, \vec{e'}$, and hence $a \neq a'$ because of Lemma 4.4.11. However, that means that

  $$\langle F', a'\rangle \to^2 \langle F[\vec{e}/\vec{x}] \circ \nu b_1.\cdot \circ \ldots \circ \nu b_n.\cdot, \mathtt{bot}_{\mathtt{bool}}\rangle$$

  so $\langle F', a'\rangle$ diverges and never evaluates to a canonical form as all transitions involved are deterministic. This is a contradiction to $\langle F', a'\rangle \to^{i-n-2-m} \langle \mathrm{Id}, c\rangle$ above.

- In the second case the transitions must be of the form $\langle (F \circ (a_1 \rightleftharpoons a_2)\cdot)[\vec{e}/\vec{x}], \mathtt{eqBot}_{a,n}\rangle \to \langle F[\vec{e}/\vec{x}], (a_1\ a_2)\cdot \mathtt{eqBot}_{a,n}\rangle \to^i \langle \mathrm{Id}, c\rangle$. By assumption we know $a \# F \circ (a_1 \rightleftharpoons a_2)$, so $a \# a_1, a_2$ and therefore $(a_1\ a_2)\cdot \mathtt{eqBot}_{a,n} = \mathtt{eqBot}_{a,n}$. The rest of the case follows from that with induction.

- In the third case we know that $\langle (F \circ \nu a'.\cdot)[\vec{e}/\vec{x}], x[\vec{e}/\vec{x}]\rangle = \langle (F \circ \nu a'.\cdot)[\vec{e}/\vec{x}], \mathtt{eqBot}_{a,n}\rangle \to^{i+1} \langle \mathrm{Id}, c\rangle$ and also $a \# F, a'$. Let also $\vec{e'} \in KBot^j$ be given and let $\mathtt{kBot}_{n'} \triangleq x[\vec{e'}/\vec{x'}]$. The transitions must be of the form $\langle (F \circ \nu a'.\cdot)[\vec{e}/\vec{x}], \mathtt{eqBot}_{a,n}\rangle \to \langle F[\vec{e}/\vec{x}], \lambda x : \mathtt{name} \to \nu a'.\nu b_1.\ldots\nu b_n.\, \mathtt{if}(x = a)\ \mathtt{then\ T\ else\ bot}_{\mathtt{bool}}\rangle = \langle F[\vec{e}/\vec{x}], \mathtt{eqBot}_{a,n+1}\rangle \to^i \langle \mathrm{Id}, c\rangle$. Choose any $x' \notin x_1, \ldots x_j$ and define $\vec{e_2} \triangleq e_1, \ldots e_j, \mathtt{eqBot}_{a,n+1}$, $\vec{x'} \triangleq x_1, \ldots, x_j, x'$ and $\Gamma' \triangleq \Gamma, x' : \mathtt{name} \to \mathtt{bool}$. It follows that $F[\vec{e}/\vec{x}] = F[\vec{e_2}/\vec{x'}]$ and $x'[\vec{e_2}/\vec{x'}] = \mathtt{eqBot}_{a,n+1}$. So we have $\langle F[\vec{e_2}/\vec{x'}], x'[\vec{e_2}/\vec{x'}]\rangle \to^i \langle \mathrm{Id}, c\rangle$ and we can use induction on this to obtain $(\forall \vec{e'} \in KBot^{j+1})\ \langle F[\vec{e'}/\vec{x'}], x'[\vec{e'}/\vec{x'}]\rangle \to^* \langle \mathrm{Id}, c\rangle$. Instantiate this with $\vec{e'_2} \triangleq e'_1, \ldots, e'_j, \mathtt{kBot}_{n'+1}$ to get $\langle F[\vec{e'_2}/\vec{x'}], x'[\vec{e'_2}/\vec{x'}]\rangle = \langle F[\vec{e'}/\vec{x}], \mathtt{kBot}_{n'+1}\rangle \to^* \langle \mathrm{Id}, c\rangle$. It is easy to check that $\langle (F \circ \nu a'.\cdot)[\vec{e'}/\vec{x}], x[\vec{e'}/\vec{x}]\rangle = \langle (F \circ \nu a'.\cdot)[\vec{e'}/\vec{x}], \mathtt{kBot}_{n'}\rangle \to \langle F[\vec{e'}/\vec{x}], \mathtt{kBot}_{n'+1}\rangle$ and this concludes this case.

- Finally for the fourth case we know that $a \# (F \circ \alpha a'.\cdot)$ and that the transitions must be of the form

$$\langle F[\vec{e}/\vec{x}] \circ \alpha a'.\cdot, x[\vec{e}/\vec{x}]\rangle \to \langle F[\vec{e}/x], \alpha a'.x[\vec{e}/\vec{x}]\rangle \to^{i} \langle \mathrm{Id}, c\rangle\,.$$

  It follows that $a \# F, \alpha a.\, x$ and hence we obtain $\langle F[\vec{e'}/x], \alpha a'.\, x[\vec{e'}/\vec{x}]\rangle \to^{*} \langle \mathrm{Id}, c\rangle$ by induction. We conclude this case by observing $\langle F[\vec{e'}/\vec{x}] \circ \alpha a'.\cdot, x[\vec{e'}/\vec{x}]\rangle \to \langle F[\vec{e'}/x], \alpha a'.\, x[\vec{e'}/\vec{x}]\rangle$ and this also concludes the entire proof.

□

A.6 Proof of Lemma 5.5.13

Assume $(\forall \sigma' \in \mathrm{Styp}^{\mathsf{PNA}^+})\,(\mathrm{DEF}_{\sigma'})$ and note that $\Gamma \vdash e \not\lesssim_{\mathsf{PNA}^+} e' : \sigma$ holds if and only if there is a context $C : (\Gamma \triangleright \sigma) \rightsquigarrow (\emptyset \triangleright \mathtt{bool})$ that separates e and e', that is $C[e] \Downarrow \mathtt{T}$ and $\neg(C[e'] \Downarrow \mathtt{T})$.

We first prove the property for closed expressions $(\forall \sigma \in \mathrm{Styp}^{\mathsf{PNA}^+})\,(\mathrm{FA}_{\emptyset|\sigma})$ by the classical argument as surveyed by Curien [12, Criterion 2.2]. So we prove by induction on σ that $[\![e]\!] \not\sqsubseteq [\![e']\!] \Rightarrow \emptyset \vdash e \not\lesssim_{\mathsf{PNA}^+} e' : \sigma$. We do not need Proposition 5.5.12 for this part of the argument.

- *Case* $\sigma = \gamma \in \{\mathtt{nat}, \mathtt{name}\}$*:* By (5.39) it suffices to show $e \not\lesssim^{k} e'$ and this a consequence of the fundamental property (Proposition 5.3.5), the definition of the $\triangleleft_{\gamma}$ (Definition 5.3.1) and soundness (Proposition 4.4.16).

- *Case* $\sigma = \sigma_1 \times \sigma_2$*:* It must hold that $proj_i\,[\![e]\!] \not\sqsubseteq proj_i\,[\![e']\!]$ for $i \in \{1,2\}$ and assume without loss of generality that $i = 1$. Hence $[\![\mathtt{fst}\ e]\!] \not\sqsubseteq [\![\mathtt{fst}\ e']\!]$ and by induction we know that there is a $C : (\emptyset \triangleright \sigma_1) \rightsquigarrow (\emptyset \triangleright \mathtt{bool})$ that separates $\mathtt{fst}\ e$ and $\mathtt{fst}\ e'$. Then $C' \triangleq C[\mathtt{fst}\ -]$ is a context that separates e and e'.

- *Case* $\sigma = \sigma_1 \to \mathtt{nat}$*:* Lemma 3.4.19 tells us that there must be $u \in \mathrm{K}[\![\sigma_1]\!]$ such that $[\![e]\!]\,u \not\sqsubseteq [\![e']\!]\,u$. By $(\mathrm{DEF}_{\sigma_1})$ there is $e_1 \in \mathrm{Exp}^{\mathsf{PNA}^+}(\sigma_1)$ such that $[\![e_1]\!] = u_1$. It follows that $[\![e\,e_1]\!] \not\sqsubseteq [\![e'\,e_1]\!]$, by induction there is a $C : (\emptyset \triangleright \sigma_1) \rightsquigarrow (\emptyset \triangleright \mathtt{bool})$ that separates $e\,e_1$ and $e'\,e_1$ and hence $C' \triangleq C[-e_1]$ separates e and e'.

Having established $(\mathrm{FA}_{\emptyset|\sigma})$ we now show that this implies $(\mathrm{FA}_{\Gamma|\sigma})$ with Proposition 5.5.12. So assume $[\![e]\!] \not\sqsubseteq [\![e']\!] \in [\![\Gamma]\!] \to_{\mathrm{uc}} [\![\sigma]\!]$, let $\Gamma = \{x_1 : \tau_1, \ldots, x_n : \tau_n\}$ and define $\tau = \tau_1 \times \ldots \times \tau_n$. We have $[\![\Gamma]\!] = [\![\tau]\!]$ and by Proposition 5.5.12 there is $\sigma_1 \in \mathrm{Styp}^{\mathsf{PNA}^+}$ such that $\tau \preceq \sigma_1$ with expressions i, r. By Lemma 3.4.20 there must be a $u_1 \in \mathrm{K}[\![\sigma_1]\!]$ such that $[\![e]\!]([\![r]\!]\,u_1) \not\sqsubseteq [\![e]\!]([\![r]\!]\,u_1) \in [\![\sigma]\!]$ and by $(\mathrm{DEF}_{\sigma_1})$ there is $e_1 \in \mathrm{Exp}^{\mathsf{PNA}^+}(\sigma_1)$ such that $[\![e_1]\!] = u'$. Therefore we have $[\![(\lambda \vec{x} : \tau \to e)(r\,e_1)]\!] \not\sqsubseteq [\![(\lambda \vec{x} : \tau \to e')(r\,e_1)]\!] \in [\![\sigma]\!]$, where $\vec{x} = (x_1, \ldots, x_n)$ and we use some syntactic sugar for λ-abstraction of tuples. By $(\mathrm{FA}_{\emptyset|\sigma})$ there is a context C separating the two expressions. Therefore $C' \triangleq C[(\lambda \vec{x} : \tau \to -)(r\,e_1)]$ satisfies $C' : (\Gamma \triangleright \sigma) \rightsquigarrow (\emptyset \triangleright \mathtt{bool})$ and separates e and e'. □

A.7 Proof of Lemma 5.5.16

Let $\tau \in \mathrm{Typ}^{\mathsf{PNA}^+}$, $u \in \mathrm{K}[\![\tau]\!]$, $A \subseteq_{\mathrm{f}} \mathbb{A}$ and $e \in \mathrm{Exp}^{\mathsf{PNA}^+}(\tau \to \mathtt{bool})$ be given such that $[\![e]\!] = (u \searrow \mathrm{true})$. We have to find $e' \in \mathrm{Exp}^{\mathsf{PNA}^+}(\tau \to \mathtt{bool})$ satisfying

$$[\![e']\!] = \bigsqcup \mathrm{hull}_A\{(u \searrow \mathrm{true})\}. \tag{A.11}$$

Define for each $f \in \mathrm{Perm}(\mathbb{A}) \to_{\mathrm{uc}} 2_\perp$ the analogue of (3.24) for permutations:

$$\mathit{exists}_{\mathrm{Perm}(\mathbb{A})} f \triangleq \begin{cases} \mathrm{true} & \text{if } (\exists \pi \in \mathrm{Perm}(\mathbb{A}))\, f\,\pi = \mathrm{true} \\ \mathrm{false} & \text{if } (\forall \pi \in \mathrm{Perm}(\mathbb{A}))\, f\,\pi = \mathrm{false} \\ \perp & \text{otherwise.} \end{cases} \tag{A.12}$$

This allows us to characterise $\bigsqcup \mathrm{hull}_A\{(u \searrow \mathrm{true})\}$ by

$$\begin{aligned} &\bigsqcup \mathrm{hull}_A\{(u \searrow \mathrm{true})\} \\ &= \lambda d \in [\![\tau]\!] \to \begin{cases} \mathrm{true} & \text{if } (\exists \pi \in \mathrm{Perm}(\mathbb{A}))\, \pi \# A \wedge \pi \cdot u \sqsubseteq d \\ \perp & \text{otherwise} \end{cases} \\ &= \lambda d \in [\![\tau]\!] \to \mathit{exists}_{\mathrm{Perm}(\mathbb{A})}(\lambda \pi \in \mathrm{Perm}(\mathbb{A}) \to \pi \# A \wedge (\pi \cdot u \searrow \mathrm{true})\, d). \end{aligned} \tag{A.13}$$

Applying Lemma 2.3.14 to (A.13), together with the denotational semantics of locally scoped names (Figure 4.11 and Theorem 4.3.1), it follows that (A.11) holds with

$$\begin{aligned} e' \triangleq \lambda x : \tau \to \mathtt{ex}\, y_1. \ldots \mathtt{ex}\, y_n.(\mathtt{distinct}\ \vec{y})\ \mathtt{and}\ (\vec{y}\ \mathtt{freshfor}\ A) \\ \mathtt{and}\ \nu b_1. \ldots \nu b_n.(((\vec{y} \rightleftharpoons \vec{b})(\vec{a} \rightleftharpoons \vec{b})\, e)\, x) \end{aligned}$$

where $[\![e]\!] = (u \searrow \mathrm{true})$ as above, $\vec{a} \triangleq \mathrm{supp}\, u - A$, and $\vec{b}$ are distinct atomic names satisfying $\vec{b} \# u, A$. For better readability, we used some syntactic sugar from Section 4.5.1. □

A.8 Proof of Lemma 5.5.17

Suppose we are given $\tau \in \mathrm{Typ}^{\mathsf{PNA}^+}$ such that

$$(\forall v, v' \in \mathrm{K}[\![\tau]\!])\, v \not\asymp v' \Rightarrow (\exists e \in \mathrm{Exp}^{\mathsf{PNA}^+}(\tau \to \mathtt{bool}))\, [\![e]\!] = (v \searrow \mathrm{true}) \sqcup (v' \searrow \mathrm{false}) \tag{A.14}$$

and also $u, u' \in \mathrm{K}[\![\tau]\!]$ and $A \subseteq_{\mathrm{f}} \mathbb{A}$ such that

$$(\forall \pi \in \mathrm{Perm}(\mathbb{A}))\, \pi \# A \Rightarrow u \not\asymp \pi \cdot u'. \tag{A.15}$$

We have to find $e \in \mathrm{Exp}^{\mathsf{PNA}^+}(\tau \to \mathtt{bool})$ satisfying

$$[\![e]\!] = \bigsqcup \mathrm{hull}_A\{(u \searrow \mathrm{true}), (u' \searrow \mathrm{false})\}. \tag{A.16}$$

Without loss of generality we may assume

$$(\operatorname{supp} u \cap \operatorname{supp} u') - A = \emptyset \tag{A.17}$$

because otherwise we can take $u'' \triangleq (\vec{a}\ \vec{b}) \cdot u$ instead of u', where $\vec{a} \triangleq (\operatorname{supp} u \cap \operatorname{supp} u') - A$ and $\vec{b}$ are some distinct and fresh atomic names. (We use the notation for swapping lists of atomic names from Notation 2.3.13.) It is easy to show that $(\operatorname{supp} u \cap \operatorname{supp} u'') - A = \emptyset$. We can replace u' with u'' because $(\vec{a}\ \vec{b}) \# A$ implies that $\bigsqcup \operatorname{hull}_A\{(u \searrow \text{true}), (u' \searrow \text{false})\} = \bigsqcup \operatorname{hull}_A\{(u \searrow \text{true}), (u'' \searrow \text{false})\}$ and that (A.14) as well as (A.15) hold for u''.

Note that (A.15) entails that whenever there is a $\pi \# A$ with $\pi \cdot u \sqsubseteq x$, then for all $\pi' \# A$ it is the case that $\pi' \cdot u \not\sqsubseteq x$ (and symmetrically with u and u' interchanged). Hence we get:

$$\begin{aligned}
&\bigsqcup \operatorname{hull}_A\{(u \searrow \text{true}), (u' \searrow \text{false})\} \\
&= \lambda d \in [\![\tau]\!] \to \begin{cases} \text{true} & \text{if } (\exists \pi \in \operatorname{Perm}(\mathbb{A}))\, \pi \# A \wedge \pi \cdot u \sqsubseteq d \\ \text{false} & \text{if } (\exists \pi' \in \operatorname{Perm}(\mathbb{A}))\, \pi' \# A \wedge \pi' \cdot u' \sqsubseteq d \\ \bot & \text{otherwise} \end{cases} \\
&= \lambda d \in [\![\tau]\!] \to \\
&\begin{cases} \text{true} & \text{if } (\exists \pi \in \operatorname{Perm}(\mathbb{A}))\, \pi \# A \wedge \pi \cdot u \sqsubseteq d \wedge (\forall \pi' \in \operatorname{Perm}(\mathbb{A}))\, \pi' \# A \Rightarrow \pi' \cdot u' \not\sqsubseteq d \\ \text{false} & \text{if } (\exists \pi' \in \operatorname{Perm}(\mathbb{A}))\, \pi' \# A \wedge \pi' \cdot u' \sqsubseteq d \wedge (\forall \pi \in \operatorname{Perm}(\mathbb{A}))\, \pi \# A \Rightarrow \pi \cdot u \not\sqsubseteq d \\ \bot & \text{otherwise} \end{cases} \\
&= \lambda d \in [\![\tau]\!] \to \mathit{exists}_{\operatorname{Perm}(\mathbb{A})}(\lambda \pi \in \operatorname{Perm}(\mathbb{A}) \to \pi \# A \wedge \mathit{all}_{\operatorname{Perm}(\mathbb{A})}(\lambda \pi' \in \operatorname{Perm}(\mathbb{A}) \to \pi' \# A \\
&\qquad \Rightarrow ((\pi \cdot u \searrow \text{true}) \sqcup (\pi' \cdot u' \searrow \text{false}))\, d))
\end{aligned} \tag{A.18}$$

where

$$\mathit{all}_{\operatorname{Perm}(\mathbb{A})} f \triangleq \begin{cases} \text{true} & \text{if } (\forall \pi \in \operatorname{Perm}(\mathbb{A}))\, f\ \pi = \text{true} \\ \text{false} & \text{if } (\exists \pi \in \operatorname{Perm}(\mathbb{A}))\, f\ \pi = \text{false} \\ \bot & \text{otherwise} \end{cases} \tag{A.19}$$

is the dual of (A.12).

Following Appendix A.7, we can replace the quantifications over π and π' in (A.18) by multiple quantifications over atomic names. Suppose $\operatorname{supp} u - A$ consists of the distinct atomic names $\vec{b} = (b_1, \ldots, b_n) \in \mathbb{A}^{\#n}$ and $\operatorname{supp} u' - A$ consists of the distinct atomic names $\vec{c} = (c_1, \ldots, c_m) \in \mathbb{A}^{\#m}$; so from (A.17) we have $\vec{b} \# \vec{c}$. Given $\pi, \pi' \# A$, as in Lemma 2.3.14 we have

$$\begin{aligned}
\pi \cdot u &= ((\vec{b}'\ \vec{d}) \circ (\vec{b}\ \vec{d})) \cdot u \\
\pi' \cdot u' &= ((\vec{c}'\ \vec{d}') \circ (\vec{c}\ \vec{d}')) \cdot u'
\end{aligned}$$

with $\vec{b}' = \pi \cdot \vec{b}$, $\vec{c}' = \pi' \cdot \vec{c}$ and $\vec{d}, \vec{d}'$ chosen suitably fresh. Therefore in view of (A.18),

to solve (A.16) we can take e to be

$$\begin{aligned} e \triangleq\ & \lambda x : \tau \to \mathtt{ex}\, x_1. \ldots \mathtt{ex}\, x_n. (\mathtt{distinct}\ \vec{x})\ \mathtt{and}\ (\vec{x}\ \mathtt{freshfor}\ A) \\ & \mathtt{and}\ \mathtt{not}\ (\mathtt{ex}\, y_1. \ldots \mathtt{ex}\, y_m. (\mathtt{distinct}\ \vec{y})\ \mathtt{and}\ (\vec{y}\ \mathtt{freshfor}\ A)\ \mathtt{and}\ \mathtt{not}\ (e'\, x)) \end{aligned} \tag{A.20}$$

under the condition that we can find an expression e' that satisfies $x_1 : \mathtt{name}, \ldots, x_n : \mathtt{name}, y_1 : \mathtt{name}, \ldots, y_m : \mathtt{name} \vdash e' : \tau \to \mathtt{bool}$ and for each environment ρ that

$$[\![e']\!]\rho = ((([\![\vec{x}]\!]\rho\ \vec{d}) \circ (\vec{b}\ \vec{d})) \cdot u \searrow \mathrm{true}) \sqcup ((([\![\vec{y}]\!]\rho\ \vec{d'}) \circ (\vec{c}\ \vec{d'})) \cdot u' \searrow \mathrm{false}) \tag{A.21}$$

for suitably fresh $\vec{d} \in \mathbb{A}^{\#n}$ and $\vec{d'} \in \mathbb{A}^{\#m}$. The syntactic sugar from Section 4.5.1 is used in (A.20) as well as in the rest of this proof.

Giving such an e' might seem easy at first, since in view of (A.14) and (A.15), $(\pi \cdot u \searrow \mathrm{true}) \sqcup (\pi' \cdot u' \searrow \mathrm{false})$ is PNA^+-definable for any particular $\pi, \pi' \# A$. It is harder than it seems though: the problem is that e' has $\vec{x}, \vec{y}$ as free variables and we need to be parametric with respect to whatever atomic names those free variables get assigned to by a given environment ρ. Specifically, to define e' we need to consider all ways in which atomic names assigned to $\vec{x}$ and $\vec{y}$ may overlap. Fortunately each way corresponds to a partial bijection from $\{1, \ldots, n\}$ to $\{1, \ldots, m\}$ and there are only finitely many of them, N say. Thus we define e' by

$$\begin{aligned} e' \triangleq\ & \mathtt{if}\ x_1 \neq y_1\ \mathtt{and}\ x_1 \neq y_2\ \mathtt{and} \ldots \mathtt{and}\ x_n \neq y_m\ \mathtt{then}\ e_1 \\ & \mathtt{else\ if}\ x_1 = y_1\ \mathtt{and}\ x_2 \neq y_2\ \mathtt{and} \ldots \mathtt{and}\ x_n \neq y_m\ \mathtt{then}\ e_2 \\ & \quad \vdots \\ & \mathtt{else\ if}\ x_1 = y_1\ \mathtt{and}\ x_2 = y_2\ \mathtt{and} \ldots \mathtt{and}\ x_n = y_m\ \mathtt{then}\ e_N \\ & \mathtt{else}\ \mathtt{bot}_{\tau \to \mathtt{bool}} \end{aligned} \tag{A.22}$$

and show how to define the expressions $e_1, \ldots, e_N$ in such a way that for each ρ, $[\![e']\!]\rho = [\![e_i]\!]\rho$ for the i corresponding to the overlap conditions ρ induces between $\vec{x}$ and $\vec{y}$. Then (A.21) holds because the `if`-clauses in (A.22) are exhaustive; in particular the $\mathtt{bot}_{\tau \to \mathtt{bool}}$ case will never be reached.

Let f_i be the partial bijection corresponding to e_i, and define $\vec{b_i} \triangleq \{b_j \mid j \in \mathrm{dom} f_i\}$, $\vec{c_i} \triangleq \{c_{f_i(j)} \mid j \in \mathrm{dom} f_i\}$, $\vec{c'_i} \triangleq \{c_j \mid j \notin \mathrm{im} f_i\}$, $\vec{x_i} \triangleq \{x_j \mid j \in \mathrm{dom} f_i\}$, $\vec{y_i} \triangleq \{y_{f_i(j)} \mid j \in \mathrm{dom} f_i\}$, $\vec{y'_i} \triangleq \{y_j \mid j \notin \mathrm{im} f_i\}$. By (A.14) let $e'_i \in \mathrm{Exp}^{\mathsf{PNA}^+}(\tau \to \mathtt{bool})$ be given such that $[\![e'_i]\!] = (u \searrow \mathrm{true}) \sqcup ((\vec{c_i}\ \vec{b_i}) \cdot u' \searrow \mathrm{false})$. This allows us to define

$$e_i \triangleq \nu d_1. \ldots \nu d_n. \nu d'_1. \ldots \nu d'_{m - |\mathrm{dom} f_i|}. (\vec{y'_i} \rightleftharpoons \vec{d'})(\vec{x} \rightleftharpoons \vec{d})(\vec{c'_i} \rightleftharpoons \vec{d'})(\vec{b} \rightleftharpoons \vec{d})\, e'_i \tag{A.23}$$

where $\vec{d}, \vec{d'}$ consist of distinct atomic names that satisfy $\vec{d}, \vec{d'} \# A, \vec{b}, \vec{c}$; through α-renaming they may also be chosen to satisfy $\vec{d}, \vec{d'} \# [\![\vec{x}]\!]\rho, [\![\vec{y}]\!]\rho$ once we are given a

particular environment ρ. Define also $\vec{d}_i = \{d_j \mid j \in \mathrm{dom} f_i\}$. With (3.2), (3.15), (2.25),

$$\begin{aligned}
&(\llbracket \vec{y}'_i \rrbracket \rho\ \vec{d}') \circ (\llbracket \vec{x} \rrbracket \rho\ \vec{d}) \circ (\vec{c}'_i\ \vec{d}') \circ (\vec{b}\ \vec{d}) \cdot u \\
&= (\llbracket \vec{y}'_i \rrbracket \rho\ \vec{d}') \circ (\llbracket \vec{x} \rrbracket \rho\ \vec{d}) \circ (\vec{b}\ \vec{d}) \cdot u && \text{as } (\vec{c}'_i\ \vec{d}') \# \vec{d} \\
&= (\llbracket \vec{x} \rrbracket \rho\ \vec{d}) \circ (\vec{b}\ \vec{d}) \cdot u && \text{as } (\llbracket \vec{y}'_i \rrbracket \rho\ \vec{d}') \# \llbracket \vec{x} \rrbracket \rho
\end{aligned}$$

and

$$\begin{aligned}
&(\llbracket \vec{y}'_i \rrbracket \rho\ \vec{d}') \circ (\llbracket \vec{x} \rrbracket \rho\ \vec{d}) \circ (\vec{c}'_i\ \vec{d}') \circ (\vec{b}\ \vec{d}) \circ (\vec{c}_i\ \vec{b}_i) \cdot u' \\
&= (\llbracket \vec{y}'_i \rrbracket \rho\ \vec{d}') \circ (\llbracket \vec{x} \rrbracket \rho\ \vec{d}) \circ (\vec{c}'_i\ \vec{d}') \circ (\vec{c}_i\ \vec{d}_i) \circ (\vec{b}\ \vec{d}) \cdot u' && \text{as } \pi \circ (a\ b) = (\pi a\ \pi b) \circ \pi \\
&= (\llbracket \vec{y}'_i \rrbracket \rho\ \vec{d}') \circ (\llbracket \vec{x} \rrbracket \rho\ \vec{d}) \circ (\vec{c}'_i\ \vec{d}') \circ (\vec{c}_i\ \vec{d}_i) \cdot u' && \text{as } (\vec{b}\ \vec{d}) \# u' \\
&= (\llbracket \vec{y}'_i \rrbracket \rho\ \vec{d}') \circ (\llbracket \vec{x}_i \rrbracket \rho\ \vec{d}_i) \circ (\vec{c}'_i\ \vec{d}') \circ (\vec{c}_i\ \vec{d}_i) \cdot u' && \text{as } \vec{x} = \vec{x}_i \cup \vec{x}'_i \\
&= (\llbracket \vec{y}'_i \rrbracket \rho\ \vec{d}') \circ (\llbracket \vec{y}_i \rrbracket \rho\ \vec{d}_i) \circ (\vec{c}'_i\ \vec{d}') \circ (\vec{c}_i\ \vec{d}_i) \cdot u' && \text{as } \llbracket \vec{x}_i \rrbracket \rho = \llbracket \vec{y}_i \rrbracket \rho \\
&= (\llbracket \vec{y} \rrbracket \rho\ \vec{d}'') \circ (\vec{c}\ \vec{d}'') \cdot u' && \text{as } \vec{y} = \vec{y}_i \cup \vec{y}'_i, \text{ with } \vec{d}'' = \vec{d}' \cup \vec{d}_i
\end{aligned}$$

we obtain that $\llbracket e_i \rrbracket \rho = \llbracket e' \rrbracket \rho$ as required. □

Since the last part of the proof is combinatorially complicated, we illustrate the constructions for a simple instance.

Example A.8.1 (**construction illustration**). Suppose we are in the special case of Lemma 5.5.17 where $A = \{a\}$, $\mathrm{supp}\, u = b_1, b_2$ and $\mathrm{supp}\, u' = c_1, c_2$. In this setting, the expression e satisfying $\llbracket e \rrbracket = \bigsqcup \mathrm{hull}_{\{a\}}\{(u \searrow \mathrm{true}), (u' \searrow \mathrm{false})\}$ is defined by

$$\begin{aligned}
e \triangleq\ & \lambda x : \tau \to \mathtt{ex}\, x_1 . \mathtt{ex}\, x_2 . x_1 \neq x_2 \text{ and } x_1 \neq a \text{ and } x_2 \neq a \\
& \text{and } \mathtt{not}\,(\mathtt{ex}\, y_1 . \mathtt{ex}\, y_2 . x_1 \neq x_2 \text{ and } y_1 \neq a \text{ and } y_2 \neq a \text{ and } \mathtt{not}\,(e'\, x))
\end{aligned}$$

where the expression e' reads

$$\begin{aligned}
e' \triangleq\ & \mathtt{if}\ x_1 \neq y_1 \text{ and } x_2 \neq y_2 \text{ and } x_1 \neq y_2 \text{ and } x_n \neq y_m \ \mathtt{then}\ e_1 \\
& \mathtt{else\ if}\ x_1 = y_1 \text{ and } x_2 \neq y_2 \ \mathtt{then}\ e_2 \ \mathtt{else\ if}\ x_1 \neq y_1 \text{ and } x_2 = y_2 \ \mathtt{then}\ e_3 \\
& \mathtt{else\ if}\ x_1 = y_2 \text{ and } x_2 \neq y_1 \ \mathtt{then}\ e_4 \ \mathtt{else\ if}\ x_1 \neq y_2 \text{ and } x_2 = y_1 \ \mathtt{then}\ e_5 \\
& \mathtt{else\ if}\ x_1 = y_1 \text{ and } x_2 = y_2 \ \mathtt{then}\ e_6 \ \mathtt{else\ if}\ x_1 = y_2 \text{ and } x_2 = y_1 \ \mathtt{then}\ e_7 \\
& \mathtt{else}\ \mathtt{bot}_{\tau \to \mathtt{bool}}.
\end{aligned}$$

Compare this to (A.20) and (A.22). Let us now define e_3 explicitly as in (A.23). The corresponding partial bijection is $f_3 = \{(2,2)\}$ and by (A.14) we may assume the existence of e'_3 with $\llbracket e'_3 \rrbracket = (u \searrow \mathrm{true}) \sqcup ((c_2\ b_2) \cdot u' \searrow \mathrm{false})$. Then we can define

$$e_3 \triangleq \nu d_1 . \nu d_2 . \nu d'_1 . (y_1 \rightleftharpoons d'_1)(x_1 \rightleftharpoons d_1)(x_2 \rightleftharpoons d_2)(c_1 \rightleftharpoons d'_1)(b_1 \rightleftharpoons d_1)(b_2 \rightleftharpoons d_2) e'_3$$

for which $\llbracket e_3 \rrbracket \rho = ((\llbracket x_1 \rrbracket \rho\ d_1) \circ (\llbracket x_2 \rrbracket \rho\ d_2) \circ (b_1\ d_1) \circ (b_2\ d_2) \cdot u \searrow \mathrm{true}) \sqcup ((\llbracket y_1 \rrbracket \rho\ d'_1) \circ (\llbracket y_2 \rrbracket \rho\ d_2) \circ (c_1\ d'_1) \circ (c_2\ d_2) \cdot u' \searrow \mathrm{false})$ holds for any ρ under the conditions that $\llbracket x_1 \rrbracket \rho \neq \llbracket y_1 \rrbracket \rho$ and $\llbracket x_2 \rrbracket \rho = \llbracket y_2 \rrbracket \rho$.

A.9 Proof of Theorem 5.5.18

We follow the structure of Plotkin [47, Lemma 4.5] and Streicher [61, Theorem 13.9] by proving that the following statements simultaneously hold for all $u, u' \in \mathrm{K}[\![\sigma]\!]$:

(a) u is PNA^+-definable.

(b) $(u \searrow \mathrm{true})$ is PNA^+-definable.

(c) If $u \not\parallel u'$, then $(u \searrow \mathrm{true}) \sqcup (u' \searrow \mathrm{false})$ is PNA^+-definable.

The statements above are proved by structural induction on $\sigma \in \mathrm{Styp}^{\mathsf{PNA}^+}$. Note that it holds for any σ that $\bot$ is defined by $\mathtt{bot}_\sigma$ and $(\bot \searrow \mathrm{true})$ is defined by $\lambda x : \sigma \to \mathtt{T}$. Throughout the proof we use Notation 3.4.25 and the syntactic sugar from Sections 4.5.1 and 5.2.4.

Case $\sigma = \mathtt{name}$*:* Let $u, u' \in \mathrm{K}[\![\mathtt{name}]\!] = \mathrm{K}(\mathbb{A}_\bot) = \mathbb{A}_\bot$ be given, the above properties (a), (b) and (c) are proved one by one:

(a) $u = a \in \mathbb{A}$ is defined by the expression a.

(b) If $u = a \in \mathbb{A}$, then the step function $(u \searrow \mathrm{true})$ is defined by $\lambda x : \mathtt{name} \to \mathtt{if}\ x = a\ \mathtt{then}\ \mathtt{T}\ \mathtt{else}\ \mathtt{bot}_{\mathtt{bool}}$.

(c) We know that $u \not\parallel u'$ holds, which simply means $u = a \in \mathbb{A}$ and $u = a' \in \mathbb{A}$ with $a \neq a'$. Then $\lambda x : \mathtt{name} \to \mathtt{if}\ x = a\ \mathtt{then}\ \mathtt{T}\ \mathtt{else}\ (\mathtt{if}\ x = a'\ \mathtt{then}\ \mathtt{F}\ \mathtt{else}\ \mathtt{bot}_{\mathtt{bool}})$ defines $(u \searrow \mathrm{true}) \sqcup (u' \searrow \mathrm{false})$.

Case $\sigma = \mathtt{nat}$*:* Let $u, u' \in \mathrm{K}[\![\mathtt{nat}]\!] = \mathrm{K}(\mathbb{N}_\bot) = \mathbb{N}_\bot$ be given.

(a) $u = n \in \mathbb{N}$ is defined by $\mathtt{S}^n\,\mathtt{0}$.

(b) $\lambda x : \mathtt{nat} \to \mathtt{if}\ x =_{\mathtt{nat}} (\mathtt{S}^n\,\mathtt{0})\ \mathtt{then}\ \mathtt{T}\ \mathtt{else}\ \mathtt{bot}_{\mathtt{bool}}$ defines $(u \searrow \mathrm{true})$ for $u = n \in \mathbb{N}$.

(c) By $u \not\parallel u'$ we know $u = n \in \mathbb{N}$, $u' = n' \in \mathbb{N}$ and $n \neq n'$. $(u \searrow \mathrm{true}) \sqcup (u' \searrow \mathrm{false})$ is then defined by $\lambda x : \mathtt{nat} \to \mathtt{if}\ x =_{\mathtt{nat}} (\mathtt{S}^n\,\mathtt{0})\ \mathtt{then}\ \mathtt{T}\ \mathtt{else}\ (\mathtt{if}\ x =_{\mathtt{nat}} (\mathtt{S}^{n'}\,\mathtt{0})\ \mathtt{then}\ \mathtt{F}\ \mathtt{else}\ \mathtt{bot}_{\mathtt{bool}})$.

Case $\sigma = \sigma_1 \times \sigma_2$*:* By Proposition 3.4.11 a uniform-compact element of product type $u \in \mathrm{K}[\![\sigma_1 \times \sigma_2]\!]$ is always of the form $u = (u_1, u_2)$ with $u_1 \in \mathrm{K}[\![\sigma_1]\!]$ and $u_2 \in \mathrm{K}[\![\sigma_2]\!]$.

(a) Let $u = (u_1, u_2)$ be given. By induction there are expressions e_1, e_2 so that $[\![e_1]\!] = u_1$ and $[\![e_2]\!] = u_2$. Thus the expression (e_1, e_2) defines u.

(b) Given $u = (u_1, u_2)$, we know by induction that there are expressions e_1, e_2 satisfying $[\![e_1]\!] = (u_1 \searrow \mathrm{true})$ and $[\![e_2]\!] = (u_2 \searrow \mathrm{true})$. We can define $((u_1, u_2) \searrow \mathrm{true})$ by $\lambda x : \sigma_1 \times \sigma_2 \to (e_1(\mathtt{fst}\ x))$ and $(e_2(\mathtt{snd}\ x))$.

(c) Let $u = (u_1, u_2), u' = (u_1', u_2') \in \mathrm{K}[\![\sigma_1 \times \sigma_2]\!]$ be given. By $u \not\sqsubset u'$ it follows that $u_1 \not\sqsubset u_1'$ or $u_2 \not\sqsubset u_2'$. Assume without loss of generality that $u_1 \not\sqsubset u_1'$. Then by induction there is an expression e_1 satisfying $[\![e_1]\!] = (u_1 \searrow \mathrm{true}) \sqcup (u_1' \searrow \mathrm{false})$. By induction we also have expressions e_2 and e_2' that satisfy $[\![e_2]\!] = (u_2 \searrow \mathrm{true})$ and $[\![e_2']\!] = (u_2' \searrow \mathrm{true})$. An expression defining $(u \searrow \mathrm{true}) \sqcup (u' \searrow \mathrm{false})$ is then given by $\lambda x : \sigma_1 \times \sigma_2 \to \mathtt{if}\ e_1(\mathtt{fst}\ x)\ \mathtt{then}\ e_2(\mathtt{snd}\ x)\ \mathtt{else}\ \mathtt{not}(e_2'(\mathtt{snd}\ x))$.

Case $\sigma = \sigma_1 \to \mathtt{nat}$*:* By Theorem 3.4.28 every uniform-compact element $u \in \mathrm{K}[\![\sigma_1 \to \mathtt{nat}]\!]$ is of the form $u = \bigsqcup \mathrm{hull}_A F$, where $A \subseteq_{\mathrm{f}} \mathbb{A}$ and $F = \{(u_1 \searrow n_1), \ldots, (u_k \searrow n_k)\}$ is a finite set of step functions with $u_1, \ldots, u_k \in \mathrm{K}[\![\sigma_1]\!]$ and $n_1, \ldots, n_k \in \mathbb{N}_\perp$. More precisely, we may actually assume $n_1, \ldots, n_k \in \mathbb{N}$ because if $n_i = \perp$ then $\bigsqcup \mathrm{hull}_A F = \bigsqcup \mathrm{hull}_A F - \{(u_i \searrow n_i)\}$, which can be easily verified via the characterisation of joins of step functions in the proof of Lemma 3.4.24.

(a) Let any $u = \bigsqcup \mathrm{hull}_A F$ be given as above and prove definability of u by induction on the size of F. For the induction base we have $F = \emptyset$, and in this case u is defined by $\mathtt{bot}_{\sigma_1 \to \mathtt{nat}}$. Assume for the inductive step that $F \neq \emptyset$, say $|F| = k$. Proceed by the following case distinction:

- *If for all* $(u_i \searrow n_i), (u_j \searrow n_j) \in F$ *there is a permutation* $\pi \# A$ *such that* $u_i \uparrow \pi \cdot u_j$*:* By Lemma 3.4.24 it must hold that $n_i \uparrow \pi \cdot n_j$, so $n_i \uparrow n_j$ and therefore $n_i = n_j$. This means that $n_1 = n_2 = \ldots = n_k$. Define this number to be n and select any $(u_i \searrow n) \in F$. By induction over the types we know that $(u_i \searrow \mathrm{true})$ is definable and via Lemma 5.5.16 it follows that $\bigsqcup \mathrm{hull}_A(u_i \searrow \mathrm{true})$ is definable. So let $e \in \mathrm{Exp}^{\mathsf{PNA}^+}(\sigma_1 \to \mathtt{bool})$ be given such that $[\![e]\!] = \bigsqcup \mathrm{hull}_A(u_i \searrow \mathrm{true})$. Through induction over F it follows that there is also $e' \in \mathrm{Exp}^{\mathsf{PNA}^+}(\sigma_1 \to \mathtt{nat})$ such that $[\![e']\!] = \bigsqcup \mathrm{hull}_A(F - \{(u_i \searrow n)\})$. We can now give the PNA^+-expression defining $\bigsqcup \mathrm{hull}_A F$ by $\lambda x : \sigma_1 \to \mathtt{pif}_{\mathtt{nat}}\, e\, x\ \mathtt{then}\ \mathtt{S}^n\,\mathtt{0}\ \mathtt{else}\ e'\, x$, where it is crucial that we use the 'parallel-if' from (5.18) and not the usual `if`-construct.
- *If there are* $(u_i \searrow n_i), (u_j \searrow n_j) \in F$ *such that for all* $\pi \# A$ *it holds that* $u_i \not\sqsubset \pi \cdot u_j$*:* We know that $u_i, u_j \in \mathrm{K}[\![\sigma_1]\!]$ and by induction over types we know that property (c) holds for σ_1. With that we can apply Lemma 5.5.17 with u_i and u_j and obtain that there is an expression e satisfying $[\![e]\!] = \bigsqcup \mathrm{hull}_A\{(u_i \searrow \mathrm{true}), (u_j \searrow \mathrm{false})\}$. By induction over F we also know that there are expressions e', e'' such that $[\![e']\!] = \bigsqcup \mathrm{hull}_A(F - \{(u_j \searrow n_j)\})$ and $[\![e'']\!] = \bigsqcup \mathrm{hull}_A(F - \{(u_i \searrow n_i)\})$., and it follows that the expression $\lambda x : \sigma_1 \to \mathtt{pif}_{\mathtt{nat}}\, e\, x\ \mathtt{then}\ e'\, x\ \mathtt{else}\ e''\, x$ defines $\bigsqcup \mathrm{hull}_A F$.

(b) Let $u = \bigsqcup \mathrm{hull}_A F$ be given and proceed by induction over F. If $F = \emptyset$ then $(u \searrow \mathrm{true})$ is defined by $\lambda(x : \sigma_1 \to \mathtt{nat}) \to \mathtt{T}$. For the inductive step choose any

$(u_i \searrow n_i) \in F$ and define $F' \triangleq F - \{(u_i \searrow n_i)\}$. Observe that

$$
\begin{array}{l}
(\bigsqcup \mathrm{hull}_A F \searrow \mathrm{true}) \\
= \lambda f \in [\![\sigma_1 \to \mathtt{nat}]\!] \to \\
\begin{cases} (\bigsqcup \mathrm{hull}_A F' \searrow \mathrm{true}) f & \text{if } (\forall \pi \in \mathrm{Perm}(\mathbb{A}))\ \pi \# A \Rightarrow (\pi \cdot u_i \searrow n_i) \sqsubseteq f \\ \bot & \text{otherwise} \end{cases} \\
= \lambda f \in [\![\sigma_1 \to \mathtt{nat}]\!] \to \\
\begin{cases} (\bigsqcup \mathrm{hull}_A F' \searrow \mathrm{true}) f & \text{if } \mathit{all}_{\mathrm{Perm}(\mathbb{A})}(\lambda \pi \in \mathrm{Perm}(\mathbb{A}) \to \pi \# A \Rightarrow n_i = f(\pi \cdot u_i)) \\ \bot & \text{otherwise} \end{cases}
\end{array}
$$

where the last equality uses (A.19) and (3.16). By induction on F there is an expression e such that $[\![e]\!] = (\bigsqcup \mathrm{hull}_A F' \searrow \mathrm{true})$ and by induction on types there is an expression e' such that $[\![e']\!] = u_i$. We can encode the quantification over permutations with quantification over atomic names via Lemma 2.3.14 as it is done in Sections A.7 and A.8. With that, $(u \searrow \mathrm{true})$ can be defined by

$$
\begin{array}{l}
\lambda(x : \sigma_1 \to \mathtt{nat}) \to \mathtt{if}\ \mathtt{not}(\mathtt{ex}\, y_1 . \ldots \mathtt{ex}\, y_k . \mathtt{not}(\mathtt{distinct}\, \vec{y})\ \mathtt{or} \\
\mathtt{not}(\vec{y}\ \mathtt{freshfor}\ A)\ \mathtt{or}\ \nu b_1 . \ldots \nu b_k . \mathtt{S}^{n_i}\,\mathtt{0} =_{\mathtt{nat}} x\,((\vec{y} \rightleftharpoons \vec{b})(\vec{a} \rightleftharpoons \vec{b})\, e'))) \\
\mathtt{then}\ e\ x\ \mathtt{else}\ \mathtt{bot}_{\mathtt{bool}}.
\end{array}
$$

where $\vec{a} \triangleq \mathrm{supp}\, u_i - A \in \mathbb{A}^{\#k}$ and $\vec{b}$ are distinct atomic names satisfying $\vec{b} \# u_i, A$.

(c) Let any $u = \bigsqcup \mathrm{hull}_A F$ and $u' = \bigsqcup \mathrm{hull}_{A'} F'$ satisfying $u \not\asymp u'$ be given. $\mathrm{hull}_A F$ and $\mathrm{hull}_{A'} F'$ are sets of step functions so we can apply Lemma 3.4.26 and obtain that there are $(u_i \searrow n_i) \in \mathrm{hull}_A F$ and $(u'_j \searrow n'_j) \in \mathrm{hull}_{A'} F'$ so that $u_i \uparrow u'_j$ and $n_i \neq n'_j$. By induction on types we know that there is $e \in \mathrm{Exp}^{\mathsf{PNA}^+}(\sigma_1)$ satisfying $[\![e]\!] = u_i \sqcup u'_j$ and by case (b) we know that there are $e', e'' \in \mathrm{Exp}^{\mathsf{PNA}^+}((\sigma_1 \to \mathtt{nat}) \to \mathtt{bool})$ satisfying $[\![e']\!] = (u \searrow \mathrm{true})$ and $[\![e'']\!] = (u' \searrow \mathrm{true})$. This allows us to define $(u \searrow \mathrm{true}) \sqcup (u' \searrow \mathrm{false})$ by

$$
\begin{array}{l}
\lambda(x : \sigma_1 \to \mathtt{nat}) \to \mathtt{if}(x\, e =_{\mathtt{nat}} \mathtt{S}^{n_i}\,\mathtt{0})\ \mathtt{then}\ e'\, x\ \mathtt{else} \\
\mathtt{if}(x\, e =_{\mathtt{nat}} \mathtt{S}^{n'_j}\,\mathtt{0})\ \mathtt{then}\ \mathtt{not}(e''\, x)\ \mathtt{else}\ \mathtt{bot}_{\mathtt{bool}}.
\end{array}
$$

For correctness note that $u \sqsubseteq f$ implies $(u_i \searrow n_i) \sqsubseteq f$ and therefore by (3.16) we have $n_i \sqsubseteq f(u_i) \sqsubseteq f(u_i \sqcup u'_j)$ which gives $n_i = f(u_i \sqcup u'_j)$ (as $[\![\mathtt{nat}]\!]$ is a flat domain and $n_i \neq \bot$). Similarly $u' \sqsubseteq f$ leads to $n'_j = f(u_i \sqcup u'_j)$.

□

INDEX

List of notation

$=_\alpha$: explicit α-equivalence relation on syntax. 31, 86

$\mathbb{A}$: the set of atomic names. 24, 62

$\mathbb{B}$: the two-element set of booleans. 25

$\bot$: least element of a poset. 23

$\mathrm{Can}^{\mathsf{PNA}}$: canonical forms of PNA. 66

$\circ$: composition operation between functions or morphisms of a category. 19

$\mathrm{Config}^{\mathsf{PNA}}$: configurations of the PNA abstract machine. 87

$\cong_{\mathsf{PNA}}$: contextual equivalence of PNA. 96

$\lesssim_{\mathsf{PNA}}$: contextual preorder of PNA. 96

$\mathrm{Cont}^{\mathsf{PNA}}$: contexts of PNA. 67

$\preceq$: definable retract relation on PNA^+ types. 118

$[\![p]\!]$: denotation of the program p. 12

$\mathrm{Env}^{\mathsf{PNA}}$: typing environments of PNA. 71

$\mathrm{Exp}^{\mathsf{PNA}}$: expressions of PNA. 63

fn : free atomic names. 62

$\mathrm{Frame}^{\mathsf{PNA}}$: frames of PNA. 68

fresh a in $F\,a$: the unique element for some/any fresh a. 29

$\#$: freshness relation. 28

$X \to Y$: set of all functions between the sets X and Y. 22

$X \to_{\mathrm{fs}} Y$: set of finitely supported functions between the nominal sets X and Y. 26

$X \rightharpoonup Y$: set of partial functions between the sets X and Y. 27

$X \rightharpoonup_{\mathrm{fs}} Y$: set of finitely supported partial functions between the nominal sets X and Y. 27

$D_1 \to_{\mathrm{step}} D_2$: set of step functions between the nominal Scott domains D_1 and D_2. 50

fv : free variables. 62

$\simeq$: generalised α-equivalence. 30

$\mathrm{Gnd}^{\mathsf{PNA}}$: ground types of PNA. 69

$\mathrm{Grnd}^{\mathsf{PNA}}$: extended ground types of PNA. 69

$\cong$: isomorphism relation between two sets. 22

$s_1 \sqcup s_2$: join of the set $\{s_1, s_2\}$. 22

$\bigsqcup S$: join of the set S. 22

$=^k$: Kleene equivalence. 107

$\leqq^k$: Kleene preorder. 107

KD : set of uniform-compact elements of the udcpo D. 43

$[t]_\alpha$: syntactical term of the λ-calculus. 32

$\lhd_\tau$: logical relation at type τ. 105

$\mathbb{N}$: the set of natural numbers. 25

Perm($\mathbb{A}$) : finite permutations of atomic names. 24

PNA : The programming language called Programming with Name Abstractions. 61

PNA+ex : PNA with existential quantification over names. 104

PNA^+ : PNA with definite description and existential quantification. 103

PNA+the : PNA with definite description over names. 104

$\sqsubseteq_{\mathrm{supp}}$: partial order of $\sqsubseteq$ and subset inclusion of the support. 45

P : powerset. 21

P_{f} : finite powerset. 21

P_{fs} : finitely supported powerset. 27

P_{of} : orbit-finite powerset. 35

$proj_1$: first projection function. 27

$proj_2$: second projection function. 27

$\backslash\backslash$: operational name restriction of PNA. 80

$\backslash$: name restriction operation. 33, 57

$\mathrm{Stack}^{\mathsf{PNA}}$: stack-frames of PNA. 68

$(u_1 \searrow u_2)$: step function between compact elements. 50

$\subseteq$: subset relation. 21

$\subseteq_{\mathrm{f}}$: finite subset relation. 21

$\subseteq_{\mathrm{fs}}$: finitely supported subset relation. 27

$\subseteq_{\mathrm{of}}$: orbit-finite subset relation. 35

$\mathrm{Subst}^{\mathsf{PNA}}(\Gamma)$: set of all Γ-substitutions. 99

$\mathsf{S}^n\,\mathsf{0}$: successor applied n times to zero. 72

supp : least support. 25

$(a\ b)$: swapping of the atomic names a and b. 24

$\mathrm{Styp}^{\mathsf{PNA}^+}$: simple types of PNA^+. 118

$\mathrm{Typ}^{\mathsf{PNA}}$: types of PNA. 69

$\mathbb{V}$: set of variables. 62

$\leq^{\mathrm{wk}}$: weak Kleene preorder on canonical forms. 108

$=^{\mathrm{wk}}$: weak Kleene equivalence. 108

List of figures

BIBLIOGRAPHY

[1] Samson Abramsky. Domain theory in logical form. *Annals of Pure and Applied Logic*, 51:1–77, 1991.

[2] Samson Abramsky, Dan R. Ghica, Andrzej S. Murawski, Chih-Hao Luke Ong, and Ian D. B. Stark. Nominal games and full abstraction for the nu-calculus. In *19th IEEE Symposium on Logic in Computer Science (LICS 2004)*, pages 150–159, 2004.

[3] Samson Abramsky, Radha Jagadeesan, and Pasquale Malacaria. Full abstraction for PCF. *Information and Computation*, 163(2):409–470, 2000.

[4] Samson Abramsky and Achim Jung. Domain theory. In Samson Abramsky, Dov Gabbay, and Thomas Stephen Edward Maibaum, editors, *Handbook of Logic in Computer Science*, volume 3, pages 1–168. Clarendon Press, 1994.

[5] Steve Awodey. *Category Theory*, volume 52 of *Oxford Logic Guides*. Oxford University Press, 2nd edition, 2010.

[6] Brian Aydemir, Arthur Charguéraud, Benjamin C Pierce, Randy Pollack, and Stephanie Weirich. Engineering formal metatheory. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '08)*, pages 3–15, 2008.

[7] Brian E. Aydemir, Aaron Bohannon, Matthew Fairbairn, J Nathan Foster, Benjamin C. Pierce, Peter Sewell, Dimitrios Vytiniotis, Geoffrey Washburn, Stephanie Weirich, and Steve Zdancewic. Mechanized metatheory for the masses: The POPLmark challenge. In *18th International Conference on Theorem Proving in Higher Order Logics (TPHOLs '05)*, pages 50–65, 2005.

[8] Mikolaj Bojanczyk, Laurent Braud, Bartek Klin, and Slawomir Lasota. Towards nominal computation. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '12)*, pages 401–412, 2012.

[9] Mikolaj Bojanczyk, Bartek Klin, and Slawomir Lasota. Automata theory in nominal sets. To appear in Logical Methods in Computer Science.

[10] James Cheney. A simple nominal type theory. *Electronic Notes in Theoretical Computer Science*, 228:37–52, 2009.

[11] Vincenzo Ciancia and Ugo Montanari. Symmetries, local names and dynamic (de)-allocation of names. *Information and Computation*, 208(12):1349–1367, 2010.

[12] Pierre-Louis Curien. Definability and full abstraction. *Electronic Notes in Theoretical Computer Science*, 172:301–310, 2007.

[13] Nicolaas Govert De Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser Theorem. *Indagationes Mathematicae (Proceedings)*, 75(5):381–392, 1972.

[14] Matthias Felleisen and Robert Hieb. The revised report on the syntactic theories of sequential control and state. *Theoretical Computer Science*, 103:235–271, 1992.

[15] Murdoch J. Gabbay. *A Theory of Inductive Definitions with alpha-Equivalence*. PhD thesis, University of Cambridge, 2001.

[16] Murdoch J. Gabbay. A study of substitution, using nominal techniques and Fraenkel-Mostowski sets. *Theoretical Computer Science*, 410(12-13):1159–1189, 2009.

[17] Murdoch J. Gabbay. Foundations of nominal techniques: logic and semantics of variables in abstract syntax. *Bulletin of Symbolic Logic*, 17(2):161–229, 2011.

[18] Murdoch J. Gabbay and James Cheney. A sequent calculus for nominal logic. In *Proceedings of the 19th Annual IEEE Symposium on Logic in Computer Science (LICS '04)*, pages 139–148, 2004.

[19] Murdoch J. Gabbay and Vincenzo Ciancia. Freshness and name-restriction in sets of traces with names. In *Proceedings of the 14th International Conference on Foundations of Software Science and Computation Structures (FOSSACS '11)*, volume 6604 of *Lecture Notes in Computer Science*, pages 365–380, 2011.

[20] Murdoch J. Gabbay and Andrew M. Pitts. A new approach to abstract syntax with variable binding. *Formal Aspects of Computing*, 13(3-5):341–363, 2001.

[21] Robert Harper. *Practical Foundations for Programming Languages*. Cambridge University Press, 2013.

[22] J. M. E. Hyland and C.-H. L. Ong. On full abstraction for PCF: I, II and III. *Information and Computation*, 163(2):285–408, 2000.

[23] Jim Laird. A game semantics of names and pointers. *Annals of Pure and Applied Logic*, 151(2):151–169, 2008.

[24] Daniel R. Licata and Robert Harper. A universe of binding and computation. In *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming (ICFP '09)*, pages 123–134, 2009.

[25] Steffen Lösch and Andrew M. Pitts. Relating two semantics of locally scoped names. In *Computer Science Logic (CSL'11) - 25th International Workshop/20th Annual Conference of the EACSL*, volume 12 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 396–411, 2011.

[26] Steffen Lösch and Andrew M. Pitts. Full abstraction for nominal Scott domains. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '13)*, pages 3–14, 2013.

[27] Steffen Lösch and Andrew M. Pitts. Denotational semantics with nominal Scott domains. *Journal of the ACM*, 61(4):27:1–27:46, 2014.

[28] Saunders Mac Lane. *Categories for the Working Mathematician*. Springer-Verlag, 1971.

[29] George Markowsky. Chain-complete posets and directed sets with applications. *Algebra Universalis*, 6(1):53–68, 1976.

[30] Conor McBride and James McKinna. Functional pearl: I am not a number—I am a free variable. In *Proceedings of the 2004 ACM SIGPLAN Workshop on Haskell (Haskell '04)*, pages 1–9, 2004.

[31] Albert R. Meyer and Kurt Sieber. Towards fully abstract semantics for local variables. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '88)*, pages 191–203, 1988.

[32] Dale Miller. Abstract syntax for variable binders: An overview. In *Proceedings of the First International Conference on Computation Logic (CL '00)*, volume 1861 of *Lecture Notes in Computer Science*, pages 239–253, 2000.

[33] Dale Miller and Alwen Tiu. A proof theory for generic judgments. *ACM Transactions on Computational Logic*, 6(4):749–783, 2005.

[34] Andrzej S. Murawski and Nikos Tzevelekos. Algorithmic games for full ground references. In *Proceedings of the 39th International Colloquium Conference on Automata, Languages, and Programming - Volume Part II (ICALP '12)*, pages 312–324, Berlin, Heidelberg, 2012.

[35] Mikkel Nygaard and Glynn Winskel. Domain theory for concurrency. *Theoretical Computer Science*, 316(1-3):153–190, 2004.

[36] Martin Odersky. A functional theory of local names. In *Proceedings of the 21st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '94)*, pages 48–59, 1994.

[37] Daniela L. Petrisan. *Investigations into Algebra and Topology over Nominal Sets*. PhD thesis, University of Leicester, 2011.

[38] Frank Pfenning and Conal Elliott. Higher-order abstract syntax. In *Proceedings of the ACM SIGPLAN Conference on Programming language Design and Implementation (PLDI '88)*, pages 199–208, 1988.

[39] Andrew M. Pitts. Operational semantics and program equivalence. In G Barthe, P Dybjer, and J Saraiva, editors, *Applied Semantics, Advanced Lectures, International Summer School, APPSEM 2000, Caminha, Portugal*, volume 2395 of *Lecture Notes in Computer Science, Tutorial*, pages 378–412. Springer-Verlag, 2002.

[40] Andrew M. Pitts. Typed operational reasoning. In Benjamin C. Pierce, editor, *Advanced Topics in Types and Programming Languages*, chapter 7, pages 245–289. The MIT Press, 2005.

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

[42] Andrew M. Pitts. Nominal System T. In *Proceedings of the 37th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL '10)*, pages 159–170, 2010.

[43] Andrew M. Pitts. Structural recursion with locally scoped names. *Journal of Functional Programming*, 21(03):235–286, 2011.

[44] Andrew M. Pitts. *Nominal Sets: Names and Symmetry in Computer Science*, volume 57 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 2013.

[45] Andrew M. Pitts and Murdoch J. Gabbay. A metalanguage for programming with bound names modulo renaming. In *Mathematics of Program Construction (MPC '00)*, volume 1837 of *Lecture Notes in Computer Science*, pages 230–255, 2000.

[46] Andrew M. Pitts and Ian D. B. Stark. Observable properties of higher order functions that dynamically create local names, or: What's new? In *Mathematical Foundations of Computer Science (MFCS '93), Proceedings of the 18th International Symposium*, volume 711 of *Lecture Notes in Computer Science*, pages 122–141, 1993.

[47] Gordon D. Plotkin. LCF considered as a programming language. *Theoretical Computer Science*, 5:223–255, 1977.

[48] Gordon D. Plotkin. A structural approach to operational semantics. Technical report, Computer Science Department, Aarhus University, 1981. DAIMI FN-19.

[49] Davide Sangiorgi, Naoki Kobayashi, and Eijiro Sumii. Logical bisimulations and functional languages. In Farhad Arbab and Marjan Sirjani, editors, *International Symposium on Fundamentals of Software Engineering*, volume 4767 of *Lecture Notes in Computer Science*, pages 364–379. Springer-Verlag, 2007.

[50] Olivier Savary-Belanger, Stefan Monnier, and Brigitte Pientka. Programming type-safe transformations using higher-order abstract syntax. In Georges Gonthier and Michael Norrish, editors, *Certified Programs and Proofs*, volume 8307 of *Lecture Notes in Computer Science*, pages 243–258. Springer-Verlag, 2013.

[51] Ulrich Schöpp and Ian D. B. Stark. A dependent type theory with names and binding. In *Computer Science Logic (CSL '04): Proceedings of the 18th International Workshop*, volume 3210 of *Lecture Notes in Computer Science*, pages 235–249, 2004.

[52] Dana S. Scott. Domains for denotational semantics. In *Proceedings of the 9th International Colloquium on Automata, Languages and Programming (ICALP '82)*, pages 577–613, 1982.

[53] Tim Sheard. Accomplishments and research challenges in meta-programming. In Walid Taha, editor, *Semantics, Applications, and Implementation of Program Generation*, volume 2196 of *Lecture Notes in Computer Science*, pages 2–44. Springer-Verlag, 2001.

[54] Mark R. Shinwell. *The Fresh Approach : Functional Programming with Names and Binders*. PhD thesis, University of Cambridge, 2005. Available as University of Cambridge Computer Laboratory Technical Report UCAM-CL-TR-618.

[55] Mark R. Shinwell and Andrew M. Pitts. Fresh Objective Caml user manual. Technical report, University of Cambridge, 2005. Available as University of Cambridge Computer Laboratory Technical Report UCAM-CL-TR-621.

[56] Mark R. Shinwell and Andrew M. Pitts. On a monadic semantics for freshness. *Theoretical Computer Science*, 342:28–55, 2005.

[57] Mark R. Shinwell, Andrew M. Pitts, and Murdoch J. Gabbay. FreshML: Programming with binders made simple. In *Eighth ACM SIGPLAN International Conference on Functional Programming (ICFP '03)*, pages 263–274, 2003.

[58] Kurt Sieber. Relating full abstraction results for different programming languages. In *Foundations of Software Technology and Theoretical Computer Science*, volume 472 of *Lecture Notes in Computer Science*, pages 373–387, 1990.

[59] Ian D. B. Stark. *Names and Higher-Order Functions*. PhD thesis, University of Cambridge, 1994. Available as University of Cambridge Computer Laboratory Technical Report UCAM-CL-TR-363.

[60] Viggo Stoltenberg-Hansen, Ingrid Lindström, and Edward R. Griffor. *Mathematical Theory of Domains*. Cambridge University Press, 1994.

[61] Thomas Streicher. *Domain-Theoretic Foundations of Functional Programming*. World Scientific Publishing Company, 2006.

[62] David C. Turner. *Nominal Domain Theory for Concurrency*. PhD thesis, University of Cambridge, 2009. Available as University of Cambridge Computer Laboratory Technical Report UCAM-CL-TR-751.

[63] David C. Turner and Glynn Winskel. Nominal domain theory for concurrency. In Erich Grädel and Reinhard Kahle, editors, *Computer Science Logic*, volume 5771 of *Lecture Notes in Computer Science*, pages 546–560, 2009.

[64] Nikos Tzevelekos. *Nominal Game Semantics*. PhD thesis, University of Oxford, 2008.

[65] Nikos Tzevelekos. Fresh-register automata. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '11)*, pages 295–306, 2011.

[66] Christian Urban, Andrew M. Pitts, and Murdoch J. Gabbay. Nominal unification. *Theoretical Computer Science*, 323:473–497, 2004.