What Does this Notation Mean Anyway?

Interpreting BNF-Style Notation as it is Used in Practice

Dee Quinlan

Submitted for the degree of Doctor of Philosophy

Heriot-Watt University

School of Mathematics and Computer Science

January 2023

The copyright in this thesis is owned by the author. Any quotation from the thesis or use of any of the information contained in it must acknowledge this thesis as the source of the quotation or information.

Abstract

BNF (Backus Naur Form) notation, as introduced in the Algol 60 report, was followed by numerous notational variants (EBNF ISO (1996), ABNF Crocker et al. (2008), etc.), and later by a new metalanguage which is used for discussing structured objects in Computer Science and Mathematical Logic. We call this latter offspring of BNF *MBNF* (Math BNF). MBNF is sometimes called "abstract syntax". MBNF can express structured objects that cannot be serialised as finite strings. What MBNF and other BNF variants share is the use of production rules, whose form is given below, which state that "every instance of $\circ_i$ for $i \in \{1, \ldots, n\}$ is also an instance of $\bullet$".

$$\bullet ::= \circ_1 \mid \cdots \mid \circ_n$$

This thesis studies BNF and its variant forms and contrasts them with MBNF production rules. We show via a series of detailed examples and lemmas that MBNF, differs substantially from BNF and its variants in how it is written, the operations it allows, and the sets of entities it defines. We demonstrate with an example and a proof that MBNF has features that, when combined, could make MBNF rule sets inconsistent.

Readers do not have a document which tells them how to read MBNF and have to learn MBNF through a process of cultural initiation. We propose a framework, MathSyn, that handles most uses of MBNF one might encounter in the wild.

Research Thesis Submission

Please note this form should be bound into the submitted thesis.

Name:	Dee Quinlan		
School:	Mathematics and Computer Science		
Version: *(i.e. First, Resubmission, Final)*	Final	Degree sought:	*PhD in Computer Science*

Declaration

In accordance with the appropriate regulations I hereby submit my thesis and I declare that:

1. The thesis embodies the results of my own work and has been composed by myself
2. Where appropriate, I have made acknowledgement of the work of others
3. The thesis is the correct version for submission and is the same version as any electronic versions submitted*.
4. My thesis for the award referred to, deposited in the Heriot-Watt University Library, should be made available for loan or photocopying and be available via the Institutional Repository, subject to such conditions as the Librarian may require
5. I understand that as a student of the University I am required to abide by the Regulations of the University and to conform to its discipline.
6. I confirm that the thesis has been verified against plagiarism via an approved plagiarism detection application e.g. Turnitin.

ONLY for submissions including published works

Please note you are only required to complete the Inclusion of Published Works Form (page 2) if your thesis contains published works)

7. Where the thesis contains published outputs under Regulation 6 (9.1.2) or Regulation 43 (9) these are accompanied by a critical review which accurately describes my contribution to the research and, for multi-author outputs, a signed declaration indicating the contribution of each author (complete)
8. Inclusion of published outputs under Regulation 6 (9.1.2) or Regulation 43 (9) shall not constitute plagiarism.

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

Signature of Candidate:	L. D. Quinlan	Date:	04/01/23

Submission

Submitted By *(name in capitals)*:	DEE QUINLAN
Signature of Individual Submitting:	L. D. Quinlan
Date Submitted:	04/01/23

For Completion in the Student Service Centre (SSC)

Limited Access	Requested	Yes		No		Approved	Yes		No
E-thesis Submitted (***mandatory for final theses***)									
Received in the SSC by *(name in capitals)*:						Date:			

Inclusion of Published Works

Please note you are only required to complete the Inclusion of Published Works Form if your thesis contains published works under Regulation 6 (9.1.2)

Declaration

This thesis contains one or more multi-author published works. In accordance with Regulation 6 (9.1.2) I hereby declare that the contributions of each author to these publications is as follows:

Citation details	Dee Quinlan, Joe B Wells and Fairouz Kamareddine, BNF-Style Notation as It Is Actually Used, Intelligent Computer Mathematics, LNCS volume 11617, 03/07/2019
Author 1	Dee Quinlan
Author 2	Joe B Wells
Signature:	Lilith Dee Quinlan
Date:	31/08/21

Citation details	Dee Quinlan, Joe B Wells, Fairouz Kamareddine and Sebastien Carlier, What Does This Notation Mean Anyway, arXiv:1806.08771, 12/06/2019
Author 1	Dee Quinlan
Author 2	Joe B Wells
Signature:	Lilith Dee Quinlan
Date:	31/08/21

Citation details	e. g. Author 1 and Author 2, Title of paper, Title of Journal, X, XX-XX (20XX)
Author 1	Contribution….
Author 2	Contribution….
Signature:	
Date:	

Please included additional citations as required.

Acknowledgements

I would like to thank Fairouz Kamaraddine and Joe Wells for their excellent supervision and help with this thesis. I would also like to thank my family with all the support they have offered me. There is no way this would have been possible without you.

Contents

1 Introduction **1**

1.1 What is MBNF and Why Does it Need a Definition? 1

1.2 Contributions and Structure of this Thesis 7

1.3 Notational Conventions 10

2 Key Terminology and Concepts **11**

2.1 Basic Logic and Mathematics 12

2.1.1 Metavariable Conventions 12

2.1.2 Sets 12

2.1.3 Pairs 13

2.1.4 Binary Relations 13

2.1.5 Functions 14

2.1.6 Sequences 15

2.1.7 Serialisation 15

2.1.8 Lexicographic Order 16

2.1.9 Lattices 16

2.1.10 Ordinals 16

2.1.11 Cardinals 17

2.1.12 Trees 20

2.2 How BNF Works 20

2.3 How MBNF Works 22

3 BNF and Its Variants: A Literature Review **25**

3.1 Backus and Naur 26

3.1.1 Backus 26

3.1.2 Naur 26

3.2 Extensions to BNF 27

3.2.1 EBNF 27

3.2.2 PEGs 32

3.2.3 ABNF 36

3.2.4 RBNF 37

3.2.5 LBNF 38

3.2.6 TBNF 38

3.3 A Brief History of BNF and Its Variants 40

3.4 Limitations of BNF and Its Variants 44

4 Introducing MBNF 45

4.1 Key Differences Between MBNF and BNF 47

4.1.1 Where BNF and Its Variants Use Strings, MBNF Uses Math Text 47

4.1.2 MBNF Is Aimed Exclusively at Human Readers 49

4.1.3 MBNF Uses Operators That Stand For Chains of Syntax 50

4.1.4 MBNF Allows Powerful Operators e.g. Context Hole Filling . . 51

4.1.5 MBNF Mixes Mathematical Language With BNF-Style Notation 54

4.1.6 MBNF Has at Least the Power of Indexed Grammars 55

4.1.7 MBNF Has a Native Concept of Binding 58

4.1.8 MBNF Allows "Arbitrary" Side Conditions on Production Rules 59

4.1.9 MBNF Can Contain Very Large Infinite Sets Within the "Syntax" 61

4.1.10 MBNF May Produce Large Sets of Undecidable Syntax 62

4.1.11 MBNF Allows Infinitary Operators 62

4.1.12 MBNF Allows Coinductive Definitions 64

4.1.13 MBNF May Be Considered up to "Arbitrary" Equivalences . . . 65

4.2 A Brief History of MBNF 68

4.3 Some Mathematical Properties of MBNF 71

4.3.1 Constraints Provided by an MBNF Rule-Set May Not be Solvable 71

4.3.2 An Incompleteness Result for MBNF 74

4.4 Conclusions for this Chapter 78

5 Introducing MathSyn 79

5.1 What Tools Does MathSyn Offer? 82

5.1.1 Objects, Arrangements, Symbols and Pointers 82

5.1.2 Inclusion of Mathematical Entities Outside of Syntactic Objects 91

5.1.3 Syntactic Equivalences in MathSyn 92

5.1.4 Hole Filling 96

5.1.5 Contexts and Compatible Closure 102

5.1.6 Sub-Objects and Primitive Constructor Decomposition 103

5.1.7 α-Equivalence, Names and Binding 114

5.1.8 Substitution, Sub-Object, Sub-Arrangement 121

5.2 The Model for Object and Arrangement 123

5.3 What Was Covered by This Chapter 131

6 Interpreting Production Rules in MathSyn 133

6.1 What is an MBNF Production Rule 133

6.2 One Way in Which Production Rules are to be Safely Defined 139

6.3 Notes on the Coinductive Case 145

6.4 What Was Covered by This Chapter 146

7 The λ-Calculus and Extensions in MathSyn v BNF 147

7.1 The Lambda Calculus in BNF 147

7.1.1 BNF Alone Only Permits String Equality 148

7.1.2 BNF Must Provide "Arbitrary" Non-Terminals as Sets of Strings 148

7.1.3 BNF Might Not Produce a Unique Abstract Syntax Tree 149

7.2 The Lambda Calculus in MathSyn 150

7.2.1 Adding α-Equivalence 151

7.2.2 Adding β and η rewriting 153

7.3 The λ-Calculus in MathSyn With Records and Generalised β-Reduction 153

8 Related Work, Conclusions and Future Work 158

8.1 Advance Over State of the Art 158

8.2 Conclusions 160

8.3 Future Work 162

8.3.1 Coinduction Analogues in MathSyn 162
8.3.2 Parsing Analogues in MathSyn 163

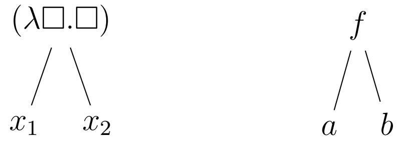
Chapter 1

Introduction

1.1 What is MBNF and Why Does it Need a Definition?

In this thesis we discuss a form of BNF-style notation which is sometimes called abstract syntax, but which we term Math-BNF (hereafter MBNF). MBNF is used because authors want to use the power and conventions of mathematics along with and inside BNF-style grammars in a way whose use is not covered by existing BNF variants. MBNF is sometimes called "abstract syntax". We avoid that name, because MBNF is in fact a somewhat concrete form with a more abstract form. For example, consider an abstract syntax tree representing $\lambda x.e$. An *abstract syntax tree* is a tree where each node is either a piece of syntax which does not stand for any function (in which case it is a terminal node) or a function taking pieces of syntax (in which case it is an interior node). Each branch in an abstract syntax tree represents the assignation of an occurrence of a term to a value. In an abstract syntax tree for $\lambda x.e$, we would not be interested that the x and the e are arranged with a dot between them and a λ in front of them. Rather, $(\lambda\Box.\Box)$ would just be a name for a particular function taking two arguments of an appropriate type.

Example 1.1. The two structures below are both similar abstract syntax trees, but the first is more verbose and includes concrete syntax which might be given as part of an MBNF which included the expression $\lambda x.e$ and the production rule $e ::= x$.



We also avoid the name “absract syntax”, because MBNF can express things that cannot be serialised as finite strings.

MBNF differs from BNF, notably:

1. MBNF and its concrete syntax are written using what we call *math text* (i.e., text with tree-like structure, which might include super/subscripting, overbarring, etc.). More information on the layout of math text can be found in The TeXbook Knuth (1986) or in the Presentation MathML Ion et al. (2001) and OpenDocument ISO (2015) standards. More information on math text can be found in Section 4.1.1 of this thesis, while the structures we give t model math text are given in Subsection 5.1.1.

2. MBNF comes with implicit rules which allow omitting parentheses without specifying this in the rules of the grammar.

3. MBNF may work with math text modulo arbitrary equivalences. Examples include α-equivalence, reordering of fields in records, and structural congruence of π-calculus, Milner et al. (1992), and related systems.

4. MBNF production rules can feature math computations, including splicing (i.e., context hole filling).

5. Metavariables in MBNF production rules can depend on sets defined using standard math notation. These can in turn depend on sets defined using MBNF production rules. There can be cycles. The problems that may arise from this are usually avoided by having the math notation and production rues being either inductively defined ore recursively picked out of a set of syntactic objects.

6. MBNF may include very large mathematical objects within the syntax, including some that cannot possibly be represented by a finite string.

7. MBNF can be defined coinductively as well as inductively. Coinductive definitions are given for syntax trees with infinite depth and may relate to some equivalences as well, for example, those in the π-calculus.

MBNF does not correspond to any existing formal definition. We give a more detailed account of these differences and why they represent a significant departure from BNF in chapter 4. Here is a brief motivating example from that chapter:

Example 1.2.

$$\begin{array}{rcl} e & ::= & x \mid \lambda x.e \mid ee \\ A & ::= & [\,] \mid A[\lambda x.A]e \end{array}$$

Here the square brackets with nothing between them represents a hole at the object level of the syntax whereas the square brackets with text between them represents a computation on the mathematical meta level, filling the leftmost hole in the tree with the text between the square brackets.

This example includes a mathematical operator for hole filling that performs a splicing operation on the syntax tree as a part of the production rules. It also extends the power of BNF, producing a grammar which cannot be thought of as context free. Finally it contains syntax that is treated as quotiented by alpha-equivalence (e in the example) nested within syntax that is to be treated as literal (the syntax appearing above the context hole in the syntax tree for A). We cover this example in more detail in SubSection 4.1.4. The treatment of MBNF which this paper provides can deal with these issues.

MBNF is important to interpreting papers in theoretical computer science. Out of the 30 papers in the ESOP 2012 proceedings Seidl (2012), 19 used MBNF and none used BNF. We chose ESOP 2012 but we could equally choose any other conferences. Because the first book we chose contained an abundance of challenging instances of MBNF, our wider searching has mainly been to find even more challenging examples.

We will be happy to receive pointers to additional interesting cases. We also checked the POPL 2017 proceedings Fluet (2017) and found that, out of 46 papers using BNF-style notation, not one used notation exactly corresponding to the EBNF ISO (1996), ABNF Crocker et al. (2008) or RBNF Farrel (2009) standards and only one Grigore (2017) could possibly be thought of as EBNF or ABNF with variant syntax. Out of the other 45 POPL 2017 papers featuring BNF-style notation, 44 use what we call MBNF. Not only is MBNF widely used; it can define a far greater variety of structures than those defined using BNF (or its most common extensions) and easily incorporates mathematics. In addition, it allows authors to write their grammars in a more abridged form than BNF and some of its extensions might. It is not used for automation, though formalising MBNF could be a necessary first step towards automating it.

In Section 4.3.1 we show that MBNF rule sets may provide constraints that have no solution. To deal with this we introduce a system called MathSyn that deals with a consistent subset. MBNF merits a different treatment than extensions of BNF which add one or two formal rules to BNF for building strings or parse trees. MBNF differs significantly in its mathematical structure from other BNF variants whose main focus is term rewriting on strings, or the parse trees generated from them.[1] These differences prevent easy translation of MBNF rule sets to the grammars of other BNF variants. This also makes trying to define MBNF by extending some BNF variant with a pre-existing formal definition a daunting and onerous process in which many of the tools available are ill-suited to the job. Throughout this paper we use MBNF to refer to the pre-existing notation which is used in a variety of papers in the computer science literature and which has no formal definition, exactly as it might be encountered in the wild. We use MathSyn to refer to the model we give for a large consistent subset of MBNF.

Steele (2017) describes computer science metanotation (CSM), which features MBNF. Steele's work underlines the importance of MBNF to the computer science literature and the need for a formal definition of this notation, but he does not go into the same depth about the underlying mathematical differences between BNF and MBNF we do

[1]A discussion of how BNF can sometimes be used to generate parse trees and how BNF grammars relate to rewriting on strings can be found in Section 2.2

in this thesis. As a result, readers learning about CSM from Steele may not realise just how distant some uses of CSM are from BNF variants which have been given a full formal definition. Although the primary focus of MathSyn is on MBNF, some of the features of MathSyn are designed to be integrated into other parts of CSM. An example of another part of CSM some of MathSyn may be helpful with is the use of syntactic objects in languages defined by a system of inference rules which contain syntax separated by a horizontal bar and a name of a judgement γ and which can be read loosely as, "The syntax above and below the bar are instances of the same language and are related by the relation γ". We do not supplant existing notation. We intend a number of MBNF rule sets to be read using MathSyn, without much adjustment. As MBNF is widely used, we expect parts of MathSyn to be familiar to many readers. We remind these readers that the only way to gain familiarity with this notation is by a long process of cultural initiation and that there is no other definition that for those wishing to avoid inconsistent MBNF rule sets.

Converting mathematical text into a form where it can be checked by a proof assistant involves human input and intermediary translations. MathSyn focuses on the translation, by the reader, of MBNF production rules together with associated pieces of mathematical text. It aims to help the reader translate these, as they appear in a document, to a formal structure independent of any theorem prover format. The proofs in this document therefore follow mathematical convention, rather than being written in a theorem prover. Specifically they are translated to structures in ZFC which can be defined using a set theoretic proof. Projects like MathLang Kamareddine et al. (2014) for computerising mathematical text may eventually help moving from the set theoretic proofs in this document to a full computer formalisation. There are several reasons we have focused on this part of the translation, many of which are reasons MBNF itself is used instead of theorem prover text:

- MBNF is often used to write rule sets which already have a lengthy computer specification, because, even though it lacks any sort of definition, MBNF is quite compact and readable. Jumping straight back into a lengthy definition which requires familiarity with a theorem prover or a parser generator would defeat the purpose of writing a guide to this notation.

- MBNF is wide ranging in its permissible application and work documenting this notation has only just begun. Even for the parts of the notation which are reasonably well documented, definitions are scattered throughout the literature. We are not aware of any resource which pulls them together in the way we do. Jumping straight to a computer specification of a small part of MBNF would be premature. Scaling the specification down in order to deal only with the most computationally friendly and well defined parts of MBNF would leave a lot of the space of MBNF rule sets unexplored and it wouldn't highlight many of the important differences between MBNF and other pre-existing BNF variants.

- The languages of the metavariables in some MBNF grammars are not serialisable Toronto & McCarthy (2012), Mislove (1995), Eberhart et al. (2015), Frumin et al. (2019),[2] but we would still like tools which help us reason about them. We say that a set, S, is *serialisable* if there exists a function, f, which takes each of the elements of S to a unique finite string, and both f and the inverse of f are computable.

- It is our aim to explain MBNF as it is generally used (although we stick to a standardised syntax in the document, which may vary in the wild, and we include some pieces of clarifying syntax, which may not always appear in the wild, but we do allow these to be omitted by conventions we provide). Translation of this notation to a computer would require significant rewriting of any document which features it. Most of this translation is not necessary to understanding it, from the perspective of the reader.

- MBNF borrows heavily from mathematics. It is likely that, even with a very comprehensive and complete translation of various MBNF features to some sort of computer specification, authors would still want to mix in new features which they will define using mathematics. In this case it would be more helpful to

[2]The specific details of why the grammars cited here are not serialisable can be found in subsections 4.1.9, 4.1.10 and 4.1.12. Generally, MBNF grammars are not serialisable, because of the inclusion of very large pieces of infinite syntax.

provide a clear picture of where MBNF sits in a formal structure like the ones used by mathematicians.

Since, in principle, MBNF may feature almost any part of mathematics, MathSyn cannot realistically aim at a comprehensive documentation of anything which might appear in an MBNF rule set. Instead the goal is to provide a picture of the machinery commonly employed as part of MBNF (and CSM more broadly) in such a way that it integrates clearly into the language of mathematics.

The main goal of MathSyn is to provide tools that enable readers unfamiliar with MBNF to read it as it is written. The secondary goal of MathSyn is to provide sufficient tools to determine whether an MBNF grammar has a solution and to show how that solution might be constructed. The tertiary goal of MathSyn is to document concepts relating to syntax that are often employed in MBNF and other forms of CSM (some of which also incorporate mathematics and lack a formal definition), but which are used less often in math as a whole. Finally, MathSyn provides machinery for discussing syntax which remains largely hidden when MathSyn is used to interpret MBNF, but which authors may find helpful, if they wish to make parts of their discussion more explicit.

1.2 Contributions and Structure of this Thesis

The contributions of this thesis are as follows:

1. The demarcation of MBNF as a notation distinct from BNF, which merits individual study. As part of this:

   (a) We argue that the entities which MBNF handles cannot always be expressed in terms of rewriting relations on strings and the parse trees derived from them and, as such, are not comparable to the terms handled by BNF and its notational variants (Section 4.4). For example, MBNF deals with infinite trees, which can have infinite depth (Subsection 4.1.12) or infinite breadth (4.1.11) and some of which may be irregular or undecidable (subsections 4.1.10, 4.1.11 and example 4.25). In addition, MBNF rule sets may contain mathematical entities like real numbers (Subsection 4.1.10) and sets, including infinite, uncountable sets, which may even be as large as an inaccessible cardinal (Subsection 4.1.9), as part of the syntax. As well as this, MBNF rule sets may feature arithmetic (Subsection 4.1.6) and set-building operations (Subsection 4.1.5) which are not encoded as operations on strings.

   (b) We show, via a series of detailed examples and lemmas, that MBNF, while superficially similar to BNF, differs substantially from BNF and its variants in how it is written, the operations it allows, and the sets it defines (Section 4.1).

2. We give reasons that the combination of MBNF features is a process which requires careful consideration, despite the lack of in-depth treatments of MBNF rule sets which show how features are meant to be combined. As part of this:

   (a) We outline multiple ways in which the features of MBNF may be combined in such a way that the resulting set of MBNF rules does not define a language (Subsection 4.3.1).

   (b) We give an incompleteness result for MBNF (Subsection 4.3.2). That is to say, given a definition, Def, of MBNF, then either there exists an MBNF grammar which has a solution, but which Def cannot prove has a solution, or there exists a grammar such that, given a proof that it has a solution, one can construct a proof that it does not and vice versa. This provides motivation for giving a definition for a large consistent subset of MBNF and a reason we do not and cannot cover every imaginable use.

3. We give a system for translating MBNF rule sets to the sets of syntactic objects[3] they define, which we call MathSyn. Here is what MathSyn offers:

   (a) A model for the set of equivalences over "syntax" (i.e., the equivalences which mostly relate to literal marks on a page and ways of positioning them).

[3]A syntactic object is either an instance of the context hole or else it consists of a set of countably many arrangements which themselves consist of ways of typesetting symbols together with pointers to syntactic objects.

This means MathSyn is well-disposed towards integration with tools from set theory, which are already used quite regularly alongside, and sometimes within MBNF production rules. It also means that most of the set theoretic language which is used alongside MBNF production rules can be interpreted literally. Finally, it means that syntactic equivalence is the same as $=$ when it relates syntactic "objects" regardless of the context in which they are placed.

(b) A demonstration of one way in which this set can be safely extended with mathematical objects not traditionally regarded as "syntax."

(c) A set of tools for letting relations over syntax "descend" inside syntax, so that constraints detailing these relations can be written with the shorthand already popular amongst users of MBNF.

(d) A strategy for "solving" sets of production rules (i.e., showing we can construct each of the sets which each metavariable ranges over such that all constraints hold).

(e) A treatment of names, binding, and α-equivalence which allows for bound names to belong to distinct sets and which allows α-equivalence to be interpreted in more or less the syntax the authors use, without need for translation to some more parser friendly format.

(f) A definition of other operations appearing as part of MBNF which are fairly specific to CSM (hole filling and capture avoiding substitution).

4. A case study for MathSyn in which we look at how MathSyn can be used to read an MBNF rule set for the λ-calculus with records and generalised β-reduction. As part of this:

   (a) We give a practical example where MathSyn must support mathematical operators appearing in the production rules themselves (in this case hole-filling)

(b) We give a practical example where MathSyn must support the mixing of MBNF production rules with set theory notation and the inclusion of sets as part of the syntax.

(c) We give an account of why there is no clear way to translate the features required by the λ-calculus with records and generalised β-reduction into the grammar of some other formal BNF variant.

1.3 Notational Conventions

By the *object-level* text of a paper, we mean text that in some sense stands for itself where it appears and forms the object of discussion. By the *meta-level* text of a paper, we mean text that tells us that we have to follow some additional chain of reasoning in order to arrive at the object of discussion. The meta-level syntax does stand for itself when you see it and it does not appear in the object of discussion. This is not too different to how object level and meta level are commonly used. We only include the above descriptions of object and meta level for clarity. Since the text of other documents' meta-levels is part of the object level of this one, we introduce the following notation. We use "boxes" for both inline and block text.

> “ Text placed in a quotebox (aside from this one) is quoted directly from another document. ”

> Text placed in an undecorated box (aside from this one) is intended to imitate the text of other documents which we may look to deal with and is usually derived from things written elsewhere, but it is not a direct quote.

“ This is used as an inline version of quotebox wherever it appears after this point in the document.

Chapter 2

Key Terminology and Concepts

It is necessary to outline some core concepts and terminology before we have given a thorough overview of BNF, MBNF, and the differences between them. We have tried to avoid making this too confusing, but the reader can skim this section and revisit it, if they encounter terminology they are unsure how to interpret later in the paper.

Throughout this thesis we use a variety of common mathematical concepts, such as relations, functions, sequences, etc. Sometimes, there can be several ways of representing these concepts and these may not all work in exactly the same ways in proofs and models. In the interests of keeping our notation clear and keeping this paper as self-contained as possible, we give definitions for the mathematical concepts we make use of in this chapter, in Section 2.1. Also in this chapter, in Section 2.2, we give an overview of how BNF rules are to be read. This includes key pieces of terminology that also applies to the variants discussed in chapter 3. Despite the fact that there is no pre-existing standard that systematises how MBNF production rules are to be read and the only standard there is will be given later in chapters 5 and 6, we will first sketch how MBNF production rules may be thought of as operating and provide some important pieces of terminology in this chapter, in Section 2.3.

2.1 Basic Logic and Mathematics

2.1.1 Metavariable Conventions

A metavariable is a variable at the meta-level which denotes something at an object-level. For this section, v stands for an arbitrary metavariable (a meta-metavariable). Statements of the form "let v range over $\mathcal{C}$" declare and define v as a metavariable that stands for some element of the class $\mathcal{C}$. The class $\mathcal{C}$ will often be defined as part of the same statement, and its definition may depend on the metavariable declaration.

We use single letters (either Roman or Greek) for metavariables.

Whenever we declare a metavariable v as ranging over a class, this also defines as ranging over that class all variants of v obtained by either (1) adding a subscript $i \in \mathbb{N}$ to v to produce v_i (e.g., v_0, v_1, v_2, etc), (2) adding a single, double, or triple prime to v, producing respectively in v', v'', and v''', or (3) a combination of (1) and (2).

In contrast, we use superscripts (e.g., v^1, v^2) and accents (e.g., $\bar{v}$, $\tilde{v}$) to distinguish metavariables that are in some way related to the corresponding undecorated metavariable, but not necessarily ranging over the same class. For example, if we have declared v to range over the set S, we might have v^0 ranging over S^0, v^1 ranging over S^1, and $S^1 \subset S^0 \subset S$.

2.1.2 Sets

The mathematical foundation we use is set theory with choice. ZFC Frenkel et al. (1973) is suitable. So are other variants. If $P(X)$ is a proposition of first-order logic that mentions X, then (1) $P(Y)$ differs from $P(X)$ only by mentioning Y instead of X, and (2) the notation $\{X \mid P(X)\}$ stands for $\{X \in S \mid P(X)\}$ for some set S which is left to the reader to infer from the context of discussion. Given some expression $f(X_1, \ldots, X_n)$ mentioning variables X_1, …, X_n, we use the notation $\{f(X_1, \ldots, X_n) \mid P(X_1, \ldots, X_n)\}$ for $\{Y \mid \exists X_1, \ldots, X_n.\ Y = f(X_1, \ldots, X_n) \wedge P(X_1, \ldots, X_n)\}$. Given two sets X and Y we use the notation $X \perp Y$ to mean 'X and Y are disjoint.'

2.1.3 Pairs

We rely on a operator $(\,\cdot\,,\,\cdot\,)$ for building *ordered pairs* and corresponding *projection* operators Fst and Snd, such that if $Z = (X, Y)$, then $\mathsf{Fst}(Z) = X$ and $\mathsf{Snd}(Z) = Y$. We require that it is impossible for a pair to also be a set of pairs and that the natural numbers do not overlap with pairs. We can work in a set theory with a primitive pairing operator. Alternatively we could choose to encode pairs in set theory, in which case we must take more care. We therefore can not use the encoding of pairs from Kuratowski (1921) where $(X, Y) = \{\{X\}, \{X, Y\}\}$, because (for example) $\{(X, X)\} = \{\{\{X\}\}\} = (\{X\}, \{X\})$. Similarly, we can not use the "short" encoding where $(X, Y) = \{X, \{X, Y\}\}$ together with von Neumann's encoding of natural numbers (actually of all ordinal numbers) where $0 = \emptyset$ and $i + 1 = i \cup \{i\}$ because $(0, 0) = \{0, \{0, 0\}\} = \{\emptyset, \{\emptyset, \emptyset\}\} = \{\emptyset, \{\emptyset\}\} = \{\emptyset\} \cup \{\{\emptyset\}\} = 1 \cup \{1\} = 2$. We can use the encoding of pairs in Wiener (1967) where $(X, Y) = \{\{\{X\}, \emptyset\}, \{\{Y\}\}\}$, because in this encoding a pair can not be a set of pairs, a set of sets of pairs, or a von Neumann ordinal number. Given two sets S and T, the *product set* $S \times T$ is the set of pairs $\{(X, Y) \mid X \in S \text{ and } Y \in T\}$. Let *tuple* notation be defined so that $(X_1, X_2, X_3, \ldots, X_n) = ((X_1, X_2, X_3, \ldots, X_{n-1}), X_n)$.

2.1.4 Binary Relations

Let R range over sets of pairs. The statement $(X, Y) \in R$ can be written with three kinds of alternate notation: $R(X, Y)$, and $X\ R\ Y$, and $X \overset{R}{\to} Y$.

A relation R is *reflexive with respect to* S if and only if $R \supseteq \{(X, X) \mid X \in S\}$. As is common practice, if we mention that a relation is reflexive without saying what set S this is with respect to, this means we are leaving it to the reader to infer from the context of discussion which set S to use.

Let R^* be the reflexive and transitive closure of R and let $R^=$ be the reflexive, symmetric, and transitive closure of R; in both cases we use the above-mentioned convention that the reader must infer the set S with respect to which to take the reflexive closure. Let $X \overset{R}{\to}_* Y$ mean $X \overset{R^*}{\to} Y$, and let $X \overset{R}{=} Y$ mean $X \overset{R^=}{\to} Y$.

A relation is an *equivalence* if and only if it is reflexive, symmetric and transitive. Given an equivalence relation R, let $[X]_R = \{Y \mid (X,Y) \in R\}$ be the *equivalence class of* X *with respect to* R and let $[X]_R$ be an equivalence class of R.

A relation R is *terminating* if and only if there is no infinite sequence X_1, X_2, ... such that $X_1 \overset{R}{\to} X_2 \overset{R}{\to} \cdots$. If $X \overset{R}{\to}_* Y$, and there exists no Z such that $Y \overset{R}{\to} Z$, then we call Y *an* R*-normal form of* X. If R is terminating, then it can be used for *induction*: If it can be shown that R is terminating and $\forall X \in S.\ (\forall Y \in S.\ X \overset{R}{\to} Y \Rightarrow P(Y)) \Rightarrow P(X)$, then it follows that $\forall X \in S.\ P(X)$.

A relation is *a partial order* on S if and only if it is transitive and antisymmetric. A partial order is *strict* if and only if it is irreflexive. A non-strict partial order, $\leq$, is a *total order* on S if and only if for all $X, Y \in S$ either $X \leq Y$ or $Y \leq X$. A strict partial order, $<$, is a *strict total order* on S if and only if for all $X, Y \in S$ such that $X \neq Y$ either $X < Y$ or $Y < X$.

2.1.5 Functions

A *function* is a relation f such that for all X, Y, and Z, if $\{(X,Y),(X,Z)\} \subseteq f$ then $Y = Z$. Let $S \to T = \{f \mid f \subseteq S \times T \text{ and } f \text{ is a function}\}$. Let f be *from* S *to* T if and only if $f \in S \to T$. A function f is *injective* if and only if f^{-1} is a function. If $(X,Y) \in f$ for some Y, then $f(X)$ denotes Y, otherwise $f(X)$ is undefined. A function f is *total* on S if and only if $f(X)$ is defined for all $X \in S$. The *domain* of f, $\mathsf{domain}(f)$, is the largest S such that f is total on S and the *range* of f, $\mathsf{range}(f)$, such that f is total on S is $\{f(x) \mid x \in S\}$.

A *fixed point* of a function f is some X for which $f(X) = X$. If the set of fixed points of f has a greatest lower bound which is itself a fixed point, then we call this the *least fixed point* of f and if it has a least upper bound which is itself a fixed point, then we call this the *greatest fixed point* of f.

A function is f *order preserving* w.r.t a partial ordering $\leq$ if $f(X) \leq f(Y)$ if and only if $X \leq Y$.

2.1.6 Sequences

Given a set S which is not a relation (if S contains only pairs then instead the notation refers to the definition of R^* from Section 2.1.4, the reflexive and transitive closure of R), let S^*, the set of *finite sequences of elements in* S, be the set of all finite functions f such that $\mathsf{range}(f) \subseteq S$, and $\mathsf{domain}(f) \subseteq \mathbb{N}$, and $m < n \in \mathsf{domain}(f)$ implies $m \in \mathsf{domain}(f)$.

Convention 2.1 (Metavariables over Sequences)**.** If v is declared to range over S, then $\vec{v}$ is automatically declared to range over S^*. □

The notation $[v_0, \ldots, v_n]$ stands for the least-defined function $\vec{v}$ such that $\vec{v}(i) = v_i$ for all $i \in \{0, \ldots, n\}$. For example, the singleton sequence $[v]$ containing v as its only element is $\{(0, v)\}$, and we have $[v_0, v_1, v_2] = \{(0, v_0), (1, v_1), (2, v_2)\}$. The *component of a sequence* $\vec{v}$ *at index* i is simply $\vec{v}(i)$. Note that the first component of a sequence is at index 0, and that the *empty sequence* ε is merely the empty set. The *length* of a sequence $\vec{v}$, $|\vec{v}|$ is the smallest $n \in \mathbb{N}$ which is larger than the index of all elements of $\mathsf{domain}(\vec{v})$. The *concatenation* of sequences $\vec{v}_1$ and $\vec{v}_2$ is $\vec{v}_1 \cdot \vec{v}_2 = \vec{v}_1 \cup \{(|\vec{v}_1| + i, v) \mid (i, v) \in \vec{v}_2\}$.

Note that $(S^*, \cdot, \varepsilon)$ forms a monoid.

$$\varepsilon \cdot \vec{v} = \vec{v} \qquad \vec{v} \cdot \varepsilon = \vec{v} \qquad (\vec{v}_1 \cdot \vec{v}_2) \cdot \vec{v}_3 = \vec{v}_1 \cdot (\vec{v}_2 \cdot \vec{v}_3)$$

Figure 2.1: Equalities on a monoid

2.1.7 Serialisation

Let F be some finite set (for simplicity say $F \subseteq \mathbb{N}$). Given a set S, if an author provides a computable injective function $f \in S \to F^*$, which is total on S and whose inverse is also computable (assuming any of the usual notions of a computable function), we say that it S serialised. If this can be done, we say that S is serialisable. A set, S, can be serialised only if it is countable.

2.1.8 Lexicographic Order

Let the *lexicographic order* $\prec$ be the order on $\mathbb{N}^*$ (sequences of natural numbers) defined such that $\vec{m} \prec \vec{n}$ holds if and only if there exists some i such that $\vec{m}(i) < \vec{n}(i)$ and for all $j < i$, $\vec{m}(j) = \vec{n}(j)$. Note that the function which takes each ordinal number to its successor, Succ (defined in SubSection **??**), is terminating when restricted to sequences of length at most $n \in \mathbb{N}$. [1]

Convention 2.2 (Induction on Tuples)**.** When we state that induction is being done on a tuple $(n_1, \ldots, n_m)$ of natural numbers, the order used is $\prec$ on the sequence $[n_1, \ldots, n_m]$. For example, if induction is on a tuple (m, n), the induction hypothesis may be used on any (m', n') where $m' < m$ or where $m' = m$ and $n' < n$. $\square$

2.1.9 Lattices

A *lattice* Davey & Priestley (2002) is a system $X = (S, \leq)$ where S is a set, $\leq$ is a partial order on S and for any two elements $a, b \in S$ there is a least upper bound (join) $a \vee b$ and a greatest lower bound (meet) $a \wedge b$.

A lattice $X = (S, \leq)$ is called *complete* if every subset of S has a least upper bound and a greatest lower bound in S.

2.1.10 Ordinals

We use the encoding of ordinals in von Neumann (1923): a set S is an ordinal if and only if S is strictly well-ordered with respect to set membership (i.e. the set membership relation is a strict total order on S and every non-empty subset of S has a least element in the ordering given by the set membership relation) and every element of S is also a subset of S.

For any ordinals X and Y such that $X \in Y$ we define $\mathsf{Succ}(X)$ to be the least upper bound of X in Y.

[1](Note that the above definition of lexicographic order is equivalent to the following more relaxed definition which can sometimes be more convenient: Let $\vec{m} \prec \vec{n}$ hold if and only if $\exists i.\ x_m(i) < x_n(i) \wedge (\forall j < i.\ x_m(j) \leq x_n(j))$.)

A *limit ordinal* is any ordinal which is neither a successor or 0 (the empty set).

Addition is defined inductively for ordinals α and β:

1. $\alpha + 0 = \alpha$
2. $\alpha + \mathsf{Succ}(\beta) = \mathsf{Succ}(\alpha + \beta)$
3. if β is a limit ordinal then $\alpha + \beta$ is the least upper bound of $\alpha + \delta$ in γ such that $\alpha + \delta \in \gamma$ for $\delta < \beta$.

Multiplication is defined inductively for ordinals α and β:

1. $\alpha \cdot 0 = 0$
2. $\alpha \cdot \mathsf{Succ}(\beta) = \alpha \cdot \beta + \alpha$
3. if β is a limit ordinal then $\alpha \cdot \beta$ is the least upper bound of $\alpha \cdot \delta$ in γ such that $\alpha \cdot \delta \in \gamma$ for $\delta < \beta$.

Unlike in cardinal arithmetic, neither addition nor multiplication need be commutative.

Exponentiation is defined inductively for ordinals α and β:

1. $\alpha^0 = 1$
2. $\alpha^{\mathsf{Succ}(\beta)} = (\alpha^\beta) \cdot \alpha$
3. if β is a limit ordinal then α^β is the least upper bound of α^δ in γ such that $\alpha^\delta \in \gamma$ for $\delta < \beta$.

In contrast to cardinal arithmetic, α^β need not yield an ordinal of the same cardinality as $|\beta \to \alpha|$.

2.1.11 Cardinals

We use the cardinal assignment from von Neumann (1923): For a well-ordered set S, we define its cardinal number, $|S|$, to be the smallest ordinal number equinumerous to

S, using the Von Neumann definition of an ordinal number. Since we are using choice and every S can be given a well-ordering every S has an associated ordinal number.

Addition is the cardinality of the disjoint union of the sets being added, $|\alpha|+|\beta| = |\alpha \sqcup \beta|$, where $\sqcup$ denotes disjoint union. Alternatively, we may write $|\alpha| + |\beta| = |\,|\alpha| +^* |\beta|\,|$ where $+^*$ denotes ordinal addition.[2]

The *axiom of choice* states that given a set there exists a choice function which takes that set and outputs a single member. The following hold of cardinal addition without the axiom of choice:

1. Zero is an additive identity.

2. Addition is associative.

3. Addition is commutative.

4. Addition is non-decreasing in both arguments.

In the presence of the axiom of choice, if either α or β is infinite, $|\alpha| + |\beta| = \mathsf{max}(\alpha, \beta)$. We can also consider infinite sums. Let γ be an ordinal and $\{\alpha_i \mid i < \gamma\}$ a sequence of ordinals. Then $\sum_{i<\gamma} \alpha_i$ is the ordinal resulting from concatenating the α_i in the given order; formally, we consider the disjoint union $\bigcup_{i\in\gamma} \alpha_i \times \{i\}$ and well-order it by setting $(\beta, i) < (\delta, j)$ if and only if $i < j$ or else $i = j$ and $\beta < \delta$. Let I be a set and let $\{\kappa_i \mid i \in I\}$ be an I-indexed family of cardinals. Then $\sum_{i\in I} \kappa_i = |\sum_{a<|I|} \kappa_{\pi(a)}|$ where $\pi \in |I| \to I$ is a bijection, and the later sum is the infinitary addition of ordinals defined above.

Multiplication is the cardinality of the cross product of the sets being multiplied, $|\alpha| \cdot |\beta| = |\alpha \times \beta|$.

The following hold of cardinal multiplication without the axiom of choice:

1. One is a multiplicative identity.

2. Multiplication is associative.

[2]The additional outer bars are to take the answer to the lowest ordinal of the same cardinality.

3. Multiplication is commutative.

4. Multiplication is non-decreasing in both arguments.

5. Multiplication distributes over addition.

6. $|\alpha| + |\beta| \leq |\alpha \times \beta|$ if both α and β are at least 2.

In the presence of the axiom of choice we have the result that if either α or β is infinite and both are non-zero, $|\alpha| \cdot |\beta| = \mathsf{max}(\alpha, \beta)$. Let I be a set and let $\{\kappa_i \mid i \in I\}$ be an I-indexed family of cardinals. Then $\prod_{i \in I} \kappa_i = |\prod^*_{i \in I} \kappa_i|$ where $\prod^*$ denotes the product of the sets κ_i, the set of all functions $f : I \to \bigcup_i \kappa_i$ such that for all $i \in I$, $f(i) \in \kappa_i$.

Exponentiation is given by the cardinality set of functions from the exponent into the set it acts on $|\alpha|^{|\beta|} = |\beta \to \alpha|$.

The following hold of exponentiation without the axiom of choice:

1. $\alpha^0 = 1$.

2. If $1 \leq \beta$, then $0^\beta = 0$.

3. $1^\beta = 1$.

4. $\alpha^1 = \alpha$.

5. $\alpha^{\beta+\gamma} = \alpha^\beta \cdot \alpha^\gamma$.

6. $\alpha^{\beta \cdot \gamma} = (\alpha^\beta)^\gamma$.

7. $(\alpha \cdot \kappa)^\beta = \alpha^\beta \cdot \kappa^\beta$.

8. Exponentiation is non-decreasing in both arguments.

The following hold of exponentiation with the axiom of choice:

1. If α and γ are both finite and greater than 1, and β is infinite, then $\alpha^\beta = \gamma^\beta$.

2. If α is infinite and β is finite and non-zero, then $\alpha^\beta = \alpha$.

3. If $2 \leq \alpha$ and $1 \leq \beta$ and at least one of them is infinite, then $\mathsf{max}(\alpha, 2^\beta) \leq \alpha^\beta \leq \mathsf{max}(2^\alpha, 2^\beta)$.

We use $\aleph_0$ to refer to the first infinite cardinal and $\aleph_1$ to refer to the second infinite cardinal.

2.1.12 Trees

A tree T is a partially ordered set $(T, <)$ such that:

1. For each $t \in T$, the set $\{s \in T \mid s < t\}$ is well-ordered by the relation $<$.

2. For each $a, b \in T$ there exists $x \in T$, such that $x \leq a$ and $x \leq b$.

A *finite tree* is one containing finitely many elements.

An *interior node* of a tree T is some $x \in T$, such that there exists $y \in T$ such that $y < x$ and a *leaf node* of a tree T is some $x \in T$ which is not an interior node. The *root* of a tree T is some element $x \in T$, such that for all $y \in T$, $y < x$.

2.2 How BNF Works

BNF can be thought of as a game. You start with a "non-terminal" and are then given rules for what you can replace this with. The value of the "non-terminal" defined by the BNF grammar is just the set of all things you can produce by applying these rules to each "non-terminal", provided you reach a string entirely composed of "terminal" symbols after a finite number of steps.

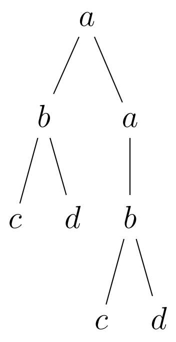
The rules are called production rules and normally look like this:

$$\bullet ::= \circ_1 \mid \cdots \mid \circ_n$$

A production rule simply states that the symbol on the left-hand side of the ::= must be replaced by one of the alternatives on the right hand side. For example, the non-terminal $\langle a \rangle$ in $\boxed{\langle a \rangle ::= \langle b \rangle \mid \langle b \rangle\langle a \rangle}$ would range over things of the form $\langle b \rangle$, $\langle b \rangle\langle b \rangle$,

$\langle b\rangle\langle b\rangle\langle b\rangle$, etc. If we were also given $\boxed{\langle b\rangle ::= cd}$, then it would range over cd, $cdcd$, etc. The alternatives are separated by $\boxed{|}$. Alternatives usually consist of "non-terminals" and "terminals". *Non-terminals* are strings that get replaced in the course of applying production rules. All non-terminals appear on the left hand side of a $\boxed{::=}$ in a production rule. Terminals are simply pieces of the final string that are not "non-terminals". They are called terminals because there are no production rules for them, so they terminate the production process. We can write a tree (sometimes called an Abstract syntax tree) to show how BNF style notation produces syntax as an instance of a non-terminal (where each child node is an instance of its parent).

Example 2.3. Here is how we would write $cdcd$ as an instance of a given the rules $\langle a\rangle ::= \langle b\rangle \mid \langle b\rangle\langle a\rangle$ and $\langle b\rangle ::= cd$:



We begin with the non-terminal a. This may either be of the form $\langle b\rangle$ or $\langle b\rangle\langle a\rangle$. We choose the latter. The b in our selection of $\langle b\rangle\langle a\rangle$ consists of two terminal symbols c and d. This time, for the a in our selection of $\langle b\rangle\langle a\rangle$, we choose $\langle b\rangle$. This gives us $cd\langle b\rangle$. The b in $cd\langle b\rangle$ also consists of two terminal symbols c and d, so we are left with $cdcd$.

Non-terminals are distinguished from terminals either by placing them in triangular brackets or by surrounding terminals by quotes and using either a comma or a space to separate both non-terminals and terminals. The *language* of $\bullet$ is the set of all things of the form $\circ_i$ for $1 < i < n$. In the example where $\langle a\rangle ::= \langle b\rangle \mid \langle b\rangle\langle a\rangle$ and $\langle b\rangle ::= cd$ the language of a would be $\{(cd)^n \mid n \in \mathbb{N} \wedge n > 0\}$ where $\mathbb{N}$ is the set of natural numbers and $(cd)^n$ denotes something of the form cd concatenated with itself n times. $\square$

Definition 2.4 (Terminal and Non-Terminal Strings)**.** We write $a \cdot b$ for a concatenated with b.

- The empty string is a terminal string.
- If S is a terminal string and T is a terminal, $S \cdot T$ is a terminal string.
- If X is a non-terminal, X is a non-terminal string.
- If A and B are non terminal strings and T is a terminal string each of the following are non-terminal strings: $A \cdot B$, $A \cdot T$, and $T \cdot A$.
- Nothing else is a terminal or non-terminal string.

$\square$

A *rewriting* relation is a relation, R, from strings to strings. The set of constraints on R can be given as a set of pairs of the form (x, y), where x is a string containing at least 1 character and y is a string which may be empty. What such a pair says is for all strings of the form uxv, where u is a string preceding x, which may be empty, and v is a string following x, which may be empty, we have $uxv \overset{R}{\to} uyv$. BNF can be thought of as expressing a specific kind of rewriting relation on strings where non-terminals are rewritten with alternatives of production rules. Given a non-terminal $\bullet$ and an alternative $\circ_i$ such that $\bullet ::= \ldots \mid \circ_i \mid \ldots$. We may add $(\bullet, \circ_i)$ to the constraints governing R. The language of a BNF non-terminal $\bullet$ are all the strings s such that $\bullet \overset{R^*}{\to} s$ and s is in R-normal form.

We refer to the languages which can be produced by BNF as *context free.*

2.3 How MBNF Works

We use an MBNF *rule set* to mean a set of MBNF production rules, together with an additional set of constraints. Additional constraints are given using ordinary mathematical language. Some define relations on "syntax" and others define sets of "syntax" which are mutually recursively defined alongside production rules. An MBNF rule set is sometimes referred to as a grammar. This is an unusual definition of grammar for anyone accustomed to thinking of grammars either as rules for sentence construction, but many of the entities MBNF deals with are not strings nor are they anything like sentences in any spoken language. For this reason, we try to refer to MBNF rule sets instead of MBNF grammars.

Each MBNF production rule is usually of roughly this form:

$$\bullet \in S ::= \mathcal{A}_1 \mid \cdots \mid \mathcal{A}_n$$

Where $\bullet$ is a metavariable, S is a set, over which $\bullet$ ranges ($\in S$ may be omitted from the written rule for brevity) and each $\mathcal{A}$ is an alternative indicating a constraint. In this context the constraint given by $\mathcal{A}_i$ is something like "each instance of $\mathcal{A}_i$ is also an instance of S."

An alternative $\mathcal{A}$ may be of roughly the form:

$$\circ_1 \star_1 \cdots \star_{n-1} \circ_n \qquad\qquad \text{where}\, c$$

Where each $\circ_i$ is a metavariable, each $\star_j$ is a mathematical operator (which builds MBNF syntax and which may or may not be written using syntax building operations inherited from BNF), and c is an expression (which may use the full power of mathematics and which represents a side condition).[3] Normally what such an expression says is: "For some assignment of a set of MBNF syntax for each of $\circ_1, \cdots, \circ_n$, which we call the *language* of $\circ_i$, for each choice of $\circ_i$ from its language, if c holds, then $\circ_1 \star_1 \cdots \star_{n-1} \circ_n \in S$."

[3]The text $\boxed{\text{where}\, c}$ is a part of the MBNF grammar. For our purposes side conditions may use any part of mathematics, as there is a wide range of mathematical expressions used in side conditions in the literature and no obvious limitations in how they are employed. Examples of the range of mathematical expressions used in side conditions appear in SubSection 4.1.8.

An example of an alternative like the one above comes from Chang & Felleisen (2012):

Example 2.5.

"
$$\hat{A}[A_1[\lambda x.\check{A}[E[x]]]A_2[v]] = \hat{A}[A_1[A_2[\check{A}[E[x]]x := v]]] \qquad \text{where } \hat{A}[\check{A}] \in A$$
"

Here they use a mixture of BNF operators (concatenation with non-terminals) and binary tree splicing/hole filling operators from mathematics. It also has a side condition that depends on tree splicing and a set membership condition.

In the place of any $\circ_i$, the expression $\mathcal{A}$ may include $\underset{j \in J}{(\heartsuit)}\, e$ where $\heartsuit$ is an operator of flexible arity, possibly of infinite arity, which builds MBNF syntax, and e is an expression consisting of metavariables (some of which may be indexed with indexes taken from index set J) and mathematical operators, which build MBNF syntax. We will see examples of infinitary operators later in the paper. They are not particularly important to the terminology and concepts we are introducing here. We include them only for completeness.

Each constraint appearing in the rules of the grammar is "solved" simultaneously. Usually what we mean when we say that a set of constraints is *solved* is that for each metavariable $\bullet$ appearing anywhere in one of the production rules, we assign some set S, such that $\bullet$ ranges over S and all the constraints in the grammar are true.

For each metavariable $\bullet$ we refer to the set it ranges over as *the language of* $\bullet$. Some authors use $\bullet$ interchangeably to mean the metavariable $\bullet$ and the language of $\bullet$, which can be confusing. We will point out when this occurs.

MBNF *syntax* refers to any mathematical object which may appear inside the language of a metavariable occurring anywhere within a production rule in the grammar. It is worth noting that MBNF syntax may look wildly different from what we often consider syntax. For example, unlike BNF syntax, which consists of strings only, MBNF syntax may contain trees of infinite breadth or depth and mathematical objects like sets and numbers.

Chapter 3

BNF and Its Variants: A Literature Review

We give a brief overview of BNF and its more popular variants which remains broad enough to cover how BNF variants normally work, the kinds of entity they work with, and the kinds of operations they usually allow. According to Zaytsev (2012):

“The grammarware technological space is commonly perceived as mature and drained of any scientific challenge, but provides many unsolved problems for researchers who are active in that field.”

We agree with Zaytev that there are still numerous reasons for a deeper comparison of these relatives to BNF. For example, one which might touch on more obscure examples and order them in terms of how quickly they may be used to build syntax or their expressive power. However, this is not what we aim to provide here. The main purpose of this section is as a preliminary to our examination of MBNF. The idea we intend readers to take away from this section is that, while the notations examined here allow for variation between them and while other fully formal BNF variants exist, they are still more like one another than they are like MBNF. Notably each grammar in this section is for building sets of strings and produces a language quotiented up to string equality. Where they do deal with trees it is with unambiguous parse trees for the strings produced. MBNF by contrast deals with tree like structures, which may start off quotiented by equivalences that are seemingly arbitrary. These may or may not be finite.

3.1 Backus and Naur

To illustrate what the original BNF looked like we present an example of BNF as it was used by Backus and BNF as it was used by Naur.

3.1.1 Backus

Backus (1959) used "$:\equiv$" to symbolise a production and "$\overline{or}$" to separate production rules. He distinguished non-terminals from terminals by surrounding them with angle brackets.

Example 3.1.

"
$$\begin{aligned}\langle digit\rangle &:\equiv O\,\overline{or}\,1\,\overline{or}\,2\,\overline{or}\,3\,\overline{or}\,4\,\overline{or}\,5\,\overline{or}\,6\,\overline{or}\\ &\quad 7\,\overline{or}\,8\,\overline{or}\,9\\ \langle integer\rangle &:\equiv \langle digit\rangle\,\overline{or}\,\langle integer\rangle\langle digit\rangle\end{aligned}$$
"

□

Here, $\langle digit\rangle$ ranges over the set of symbols {"0, "1", "2", "3", "4", "5", "6", "7", "8", "9"}. $\langle integer\rangle$ ranges over the set of strings one would use to write the non-negative integers using digits 0 to 9.

3.1.2 Naur

Naur, in Backus et al. (1963), used "::=" instead of "$:\equiv$" and "|" instead of "$\overline{or}$". Other than that the grammar is the same.

Example 3.2.

“
$$\langle \textit{digit} \rangle ::= 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$$
$$\langle \textit{unsigned integer} \rangle ::= \langle \textit{digit} \rangle \mid \langle \textit{unsigned integer} \rangle \langle \textit{digit} \rangle$$
”

□

Here ⟨*digit*⟩ ranges over the same set ⟨*digit*⟩ ranged over in the previous example and ⟨*unsigned integer*⟩ ranges over the same set ⟨*integer*⟩ did.

3.2 Extensions to BNF

The following are extensions to BNF, which, unlike MBNF up until this point, have a formal definition.

3.2.1 EBNF

(Extended Backus-Naur Form) adds facilities for dealing with repetition of syntactic rules (braces around repeated text), special sequences (Two ?s around names of special characters), choice of syntactic rules (square brackets around optional text), and exceptions to syntactic rules (written $R - E$ where R is a rule and E an exception). For example we might write:

$$\begin{aligned} X &::= \text{‘}a\text{'} \mid X, \text{‘}b\text{'} \\ Y &::= \text{‘}a\text{'} \mid \text{‘}b\text{'} \mid Y, Y \\ Z &::= Y - X, X \end{aligned}$$

Here $Y - X$ tells us to lead with string that do not consist of a single a, followed by some number of bs. Instead of having non-terminal symbols surrounded by angle brackets, terminal symbols are surrounded by single quotes and all symbols are separated by commas. Each line is ended in a semicolon. Parentheses may be placed around groups of syntax. This doesn't alter the language produced, but aids in parsing. A full copy of the syntax for EBNF is found in ISO (1996).

Example 3.3. In EBNF, the terminating decimals DI can be written as:

“
$$DI ::= [\text{‘}-\text{’}], D, \{D\}, [\text{‘.’}, D, \{D\}];$$
$$D ::= \text{‘0’} \mid \text{‘1’} \mid \text{‘2’} \mid \text{‘3’} \mid \text{‘4’} \mid \text{‘5’} \mid \text{‘6’} \mid \text{‘7’} \mid \text{‘8’} \mid \text{‘9’};$$
”

□

We read the rule $DI ::= [\text{‘}-\text{’}], D, \{D\}, [\text{‘.’}, D, \{D\}]$; as giving the following instructions for producing something of the form DI: First, begin with an optional minus, $[\text{‘}-\text{’}]$, followed by a choice of D, D, followed by any number of choices of D, $\{D\}$, followed by an optional selection of the entire sequence produced by the text in the square brackets, $[\text{‘.’}, D, \{D\}]$. This group consists of things produced with the following instructions: begin with a dot, ‘.’, followed by a choice of D, D, followed by any number of choices of D, $\{D\}$.

Here is how we would generate 13.33 from example 3.3:

We choose not to include the ‘ $-$ ’ from $[\text{‘}-\text{’}]$, giving $D, \{D\}, [\text{‘.’}, D, \{D\}]$. We choose the D at the start to be ‘1’, giving ‘1’$, \{D\}, [\text{‘.’}, D, \{D\}]$. We choose the $\{D\}$ at the start to give only one other D, giving ‘1’$, D, [\text{‘.’}, D, \{D\}]$. We choose this to be ‘3’, giving ‘13’$, [\text{‘.’}, D, \{D\}]$. We choose to include the ‘.’$, D, \{D\}$ from $[\text{‘.’}, D, \{D\}]$, giving ‘13.’$, D, \{D\}$. After this choice we must include the ‘.’ and a terminating string derived from $D, \{D\}$ which must include a selection for the 1st D and which includes a selection of how many times to repeat D thereafter. That is, after making the optional choice of the string ‘.’$, D, \{D\}$, we parse it as normal. Each member of the sequence is not optional. We choose the D at the start to be ‘3’, giving ‘13.3’$, \{D\}$. We choose the $\{D\}$ to give only one other D, giving ‘13.3’$, D$. We choose this to be ‘3’, giving 13.33.

EBNF without exceptions to syntactic rules is not more powerful than BNF in terms of what sets of strings it can generate (see Lemma 3.5), but it is more convenient and the parse trees it generates may look different. The above example is much more cumbersome to write using BNF:

Example 3.4.

$$\begin{aligned}
\langle DI \rangle &::= \langle PD \rangle \mid -\langle PD \rangle \mid \langle PD \rangle.\langle PD \rangle \mid -\langle PD \rangle.\langle PD \rangle \\
\langle PD \rangle &::= \langle D \rangle \mid \langle PD \rangle\langle D \rangle \\
\langle D \rangle &::= 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9
\end{aligned}$$

□

Here is how we would generate 13.33 from example 3.4:

We choose $\langle DI \rangle$ to be $\langle PD \rangle.\langle PD \rangle$. We choose the first $\langle PD \rangle$ to be $\langle PD \rangle\langle D \rangle$, giving $\langle PD \rangle\langle D \rangle.\langle PD \rangle$. We choose the first $\langle PD \rangle$ to be $\langle D \rangle$, giving $\langle D \rangle\langle D \rangle.\langle PD \rangle$. We choose the first $\langle D \rangle$ to be 1, giving $1\langle D \rangle.\langle PD \rangle$. We choose the $\langle D \rangle$ to be 3, giving $13.\langle PD \rangle$. We choose the $\langle PD \rangle$ to be $\langle PD \rangle\langle D \rangle$, giving $13.\langle PD \rangle\langle D \rangle$. We choose the $\langle PD \rangle$ to be $\langle D \rangle$, giving $13.\langle D \rangle\langle D \rangle$. We choose the first $\langle D \rangle$ to be 3, giving $13.3\langle D \rangle$. We choose the first $\langle D \rangle$ to be 3, giving 13.33. For the following standard lemma we follow the presentation of Attenborough (2003), who provides the same result for these functions as they appear in EBNF, although she incorrectly characterises EBNF as context free.

Lemma 3.5 (Repetition and Choice Don't Add Generative Power to BNF)**.** *Repetition and choice can be translated into an equivalent set of BNF productions.*

Proof. We outline the process:

- Convert every repetition { E } to a fresh non-terminal X and add

  X = ε | X E.

- Convert every option [E] to a fresh non-terminal X and add

  X = ε | E.

  (We can convert X = A [E] B. to X = A E B | A B as an inlining of the above.)

- Convert every group (E) to a fresh non-terminal X and add

  X = E.

(The parentheses do not change the string output, but separating it out this way may allow the (E) to be parsed as a group.)

□

Although this much of EBNF is context free, EBNF also features exceptions to syntactic rules, which prevent it from being context free. Hopcroft & Ullman (1979) show that the context-free languages are not closed under intersection. We adapt their proof to show that the set of languages produced by EBNF is larger than the set of languages produced by BNF. We base our proof on the version of this result given by Hopcroft and Ullman, because it is widely referenced and quite concise in its presentation.

To help show this we also introduce some notational conventions, a definition, and a lemma. If both a and b denote strings and n is a natural number, then we use ab to denote a concatenated with b, $|a|$ to denote the length of a and a^n to denote a concatenated with itself n times. In order to show that EBNF is not context free this we make use of the pumping lemma for context-free languages. We use the version given by Berstel et al. (2009), as many earlier versions of this lemma leave out some cases where the outer part of the string remains constant, to which the version we give here applies.

Definition 3.6 (Substring)**.** For all strings a, b, and c which may be empty, if $s = abc$, then a, b, and c are all substrings of s. □

Lemma 3.7 (The Pumping Lemma for Context Free Languages)**.** *If a language L is context-free, then there exists some integer $p \geq 1$ (called a "pumping length") such that every string s in L that has a length of p or more symbols (i.e., with $|s| \geq p$) can be written as*

$$s = uvwxy$$

with substrings u, v, w, x, and y such that

1. $|vx| \geq 1$

2. $|vwx| < p$

3. $\forall n > 0$, $uv^n wx^n y \in L$

Proof. In Berstel et al. (2009). $\square$

Lemma 3.8 (EBNF is Not BNF)**.** *Production rules of the form $R - E$ have no BNF equivalent.*

Proof. First consider two EBNF languages L_1 and L_2 as follows:

1. The language of $L_1 = \{a^n b^n a^m \mid n, m \geq 0\}$ generated by:

$$\begin{aligned}\langle L_1\rangle &::= \langle X\rangle\langle A\rangle\\ \langle X\rangle &::= a\langle X\rangle b \mid \epsilon\\ \langle A\rangle &::= \langle A\rangle a \mid \epsilon\end{aligned}$$

2. The language of $L_2 = \{a^n b^m a^m \mid n, m \geq 0\}$ is generated by:

$$\begin{aligned}\langle L_2\rangle &::= \langle A\rangle\langle X\rangle\\ \langle X\rangle &::= a\langle X\rangle b \mid \epsilon\\ \langle A\rangle &::= \langle A\rangle a \mid \epsilon\end{aligned}$$

3. $L_1 \cap L_2 = \{a^n b^n a^n \mid n \geq 0\}$ is not context free by the pumping lemma Berstel et al. (2009) since, for a given $p \geq 1$, we can choose $n > p$, such that $s = a^n b^n a^n$ is in our language and we cannot choose any substring q of s (corresponding to vwx), such that q is of length p, $s = xqy$ for some strings x and y, and $xv^n wx^n y \in L_1 \cap L_2$ (since q would have to take in at least one a and should take in the same number of as on the left as on the right of the bs).

It follows easily that rules of the form $R - E$ where R and E are non terminals cannot be modelled in BNF, since BNF only generates the context free grammars.

$$\begin{array}{rcl}\langle P\rangle & ::= & \langle L_1\rangle - \langle L_2\rangle \\ \langle Q\rangle & ::= & \langle L_1\rangle - \langle P\rangle\end{array}$$

The language of $\langle Q\rangle$ is $\{a^n b^n a^n \mid n \geq 0\}$. □

As we see in lemma 3.8 there are some languages represented in EBNF which are not represented in BNF.

3.2.2 PEGs

(Parsing Expression Grammars) Ford (2004) have many of the same facilities as EBNF, but contain an ordered choice operator which indicates parsing preference between options. For example, the EBNF rules $A = a, b \mid a$, and $A = a \mid a, b$ are equivalent, but the PEG rules $A \leftarrow a\, b \;/\; a$, and $A \leftarrow a \;/\; a\, b$ are different. The second alternative in the latter PEG rule will never succeed because the first choice is always taken if the input string to be recognized begins with 'a'. PEG rules also add and, '&', and not, '!', syntactic predicates which match a pattern only if a certain context is present. The expression '&e' attempts to match pattern e, then unconditionally backtracks to the starting point, preserving only the knowledge of whether e succeeded or failed to match. Conversely, the expression '!e' fails if e succeeds, but succeeds if e fails. For example, '!EndOfLine .' matches any single character so long as the nonterminal EndOfLine does not match starting at the same position as that character in the parse tree. Below is a PEG that parses the positive integers without additional leading zeros.

Example 3.9.

$$\begin{array}{r}\langle PD\rangle ::= (\langle PD\rangle\, \langle D\rangle)\ !(0\ \langle PD\rangle) \;/\; \langle D\rangle \\ \langle D\rangle ::= 0\ \;/\; 1 \;/\; 2 \;/\; 3 \;/\; 4 \;/\; 5 \;/\; 6 \;/\; 7 \;/\; 8 \;/\; 9\end{array}$$

□

PEG provides us with slightly more power than EBNF, at least with regards to parsing, see Lemma 3.12. However, the languages the grammar produces (as opposed to the languages it unambiguously parses) are the same for PEGs and EBNF, see Lemma 3.11. Understanding PEG rules rests on the user's intuitive understanding of parsing and string recognition. The body of literature for this is very large. In addition there is not a particularly close correspondence between the extra operators provided by PEG and anything in the syntax of MBNF.

We give some basic translations from EBNF to PEGs and show how PEGs offer capacities EBNF does not Redziejowski (2013) has a more in depth treatment of the translation of EBNF to PEG that answers the question of when an EBNF can be treated as its own PEG. We start by defining some extra syntactic machinery that will help with proofs later.

Definition 3.10 (Alternative Assignments for Instances of Non-Terminals)**.** The existing syntax of PEGs allows us to create expressions that match a given non-terminal, X, appearing in an alternative A only in the case where a particular alternative Y succeeds for X. We define some additional syntax to allow us to discuss this process more easily. For a non-terminal X appearing in an alternative A, let X_1 refer to the 1st occurrence of X in A, X_2 to the second, and so on. If we write $A[X_n = Y]$, where Y is one of the alternatives for X, this is to be read as the expression produced by replacing the nth occurrence of X in A with $(X\&Y)$. If we are performing more than 1 of these replacements, for non-terminals $X, X', X'', ...$ and alternatives $Y, Y', Y'', ...$, then, instead of writing $A[X_n = Y][X'_m = Y'][X''_p = Y'']...$, we can write $A[X_n = Y, X'_m = Y', X''_p = Y'', ...]$. In a similar vein we write $A[X_n \neq Y]$ as the expression produced by replacing the nth occurrence of X in A with $(X!Y)$. $\square$

We may consider the "language" of a PEG non-terminal to be the set of strings that are parsed as that non-terminal. The following lemma is folklore, but we could not find a source stating it (perhaps because the proof is cumbersome and the result, unsurprising):

Lemma 3.11 (PEGs Cannot Generate More Languages Than EBNF)**.** *The set of languages of PEG non-terminals is identical to that of EBNF ones.*

Proof. We begin by showing production rules of the form $R - E$ can be used to generate production rules of the form $R\&E$ and $R!E$ and vice versa. $R - E$ gives all instances of R which are not also instances of E, which can all be parsed by $R!E$. Let $F ::= R - E$, then $R - F$ gives only those instances of R which are also instances of E (i.e., those parsed by $R\&E$).

Since no other rule of EBNF provides more power than BNF and the remaining syntax of PNGs are written as a BNF with additional parsing precedence, all that remains is to show that there is a way of writing the rules in a PEG such that every alternative succeeds on some pieces of syntax, unless it is identical to another alternative.

First consider 2 alternatives for a production rule for V, A_1 and A_2, such that, if A_1 succeeds, then A_2 succeeds. In this case, either A_1 and A_2 are identical, or we can write $V \leftarrow A_1 \;/\; A_2$ to ensure both succeed on some pieces of syntax.

We now consider the case where whether an alternative succeeds if another succeeds may depend on the assignment of non-terminals in that alternative. Consider 2 alternatives for a production rule for V, A_1 and A_2, such that, if

$$A_1[P_1 = Q, P_2 = Q'..., P_n = Q^n, P'_1 = Q^{n+1}, ..., P'_m = Q^{n+m}, ...]$$

succeeds then

$$A_2[R_1 = S, R_2 = S'..., R_i = S^i, R'_1 = S^{i+1}, ..., R'_j = Q^{i+j}, ...]$$

succeeds. In this case, either

$$A_1[P_1 = Q, P_2 = Q'..., P_n = Q^n, P'_1 = Q^{n+1}, ..., P'_m = Q^{n+m}, ...]$$

and

$$A_2[R_1 = S, R_2 = S'..., R_i = S^i, R'_1 = S^{i+1}, ..., R'_j = Q^{i+j}, ...]$$

are identical, or we can write:

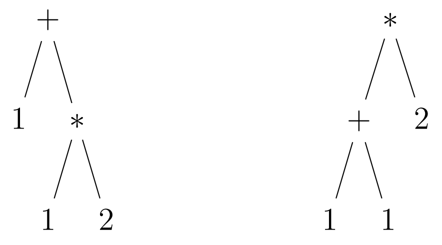
$$\begin{aligned}
V \;\leftarrow\;\; & A_2[R_1 \neq S, R_2 \neq S'..., R_i \neq S^i, R'_1 \neq S^{i+1}, ..., R'_j = Q^{i+j}, ...] \\
& /\ A_1[P_1 \neq Q, P_2 \neq Q'..., P_n \neq Q^n, P'_1 \neq Q^{n+1}, ..., P'_m \neq Q^{n+m}, ...] \\
& /\ A_1[P_1 = Q, P_2 = Q'..., P_n = Q^n, P'_1 = Q^{n+1}, ..., P'_m = Q^{n+m}, ...] \\
& /\ A_2[R_1 = S, R_2 = S'..., R_i = S^i, R'_1 = S^{i+1}, ..., R'_j = Q^{i+j}, ...]
\end{aligned}$$

to ensure both will succeed on some pieces of syntax. □

The following lemma is an obvious corollary to the results presented in Redziejowski's work on EBNF and PEGs Redziejowski (2013). We do not make use of any of the proofs appearing in Redziejowski's work here, because the lemma is basic enough that it can be proven very straightforwardly without them. In EBNF any alternative in a production rule for a non-terminal may be "parsed" as that non-terminal.

Lemma 3.12 (PEGs Can Parse More Strings Than EBNF Does)**.** *Strings which appear ambiguous to parsers in EBNF can be parsed by PEGs.*

Proof. Consider the EBNF $e ::= e * e \mid e + e \mid v$ where v ranges over the strings used to write the positive integers. The string $1 + 1 * 2$ cannot be unambiguously parsed. We could have either of the following:



In the PEG $e ::= e * e \ /\ e + e \ /\ v$ only the first of these parse trees would be an option.

□

The reader may notice that the language of an EBNF non-terminal may be larger than the set of strings that can be unambiguously parsed as that non-terminal, so lemmas 3.11 and 3.12 may both hold.

3.2.3 ABNF

(Augmented Backus-Naur Form) Crocker et al. (2008) Contains most of the facilities of EBNF aside from the ability to write exceptions to syntactic rules. ABNF can generate the same syntax as BNF. We include it here only for completeness. The following lemma has not been stated explicitly anywhere we are aware of, although it is obviously very similar to the result given by Attenborough (2003), which we have already given in Lemma 3.5:

Lemma 3.13 (ABNF Grammars are Context-Free)**.** *The set of languages ABNF grammars can generate is the same those as BNF ones can.*

Proof. We provide translations from ABNF expressions into EBNF expressions. These translations can be performed in reverse for EBNF expressions written without exceptions to get ABNF expressions. The symbol / in ABNF is translated to | in EBNF. The expression $*E$ in ABNF, where E is an expression, is translated to $\{E'\}$ in EBNF, where E' is E after any translation which can be applied to E has been. The expression $n * E$, where E is an expression and n is a natural number, is translated to n lots of E' concatenated with $\{E'\}$, where E' is E after any translation which can be applied to E has been. The expression $n * mE$, where E is an expression and n and m are natural numbers, is translated to a list of $m - n$ alternatives. The first of these alternatives has n lots of E' concatenated together in the place of $n * mE$ and the ith with $n + i - 1$ lots of E' in place of $n * mE$, where E' is E after any translation which can be applied to E has been. Strings of binary, decimal, and hexadecimal characters are translated to alphanumeric strings surrounded by quotation marks if possible. Otherwise, they are translated to whichever special character(s) they indicate surrounded by ?s (and concatenated together if more than 1 special character is specified this way). Strings surrounded by quotation marks which aren't preceded by %s are translated to a finite list of alternatives with all possible variations of capitalisation. Strings surrounded by quotation marks which are preceded by %s in ABNF are no longer preceded by %s in EBNF.

Since ABNF has no equivalent for production rules of the form $R - E$, it generates the same languages as BNF by lemma 3.5. □

Here is an ABNF for the terminating decimals:

Example 3.14.

$$DI ::= [\%d150], D, *D, [\%d46, D, *D];$$
$$D ::= \text{‘0’ / ‘1’ / ‘2’ / ‘3’ / ‘4’ / ‘5’ / ‘6’ | ‘7’ / ‘8’ / ‘9’};$$

□

3.2.4 RBNF

(Routing Backus-Naur Form) Farrel (2009) Contains most of the facilities of EBNF aside from the ability to write exceptions to syntactic rules. RBNF can generate the same syntax as BNF. We include it here only for completeness. The following lemma has not been stated explicitly anywhere we are aware of, although it is obviously very similar to the result given by Attenborough (2003), which we have already given in Lemma 3.5:

Lemma 3.15 (RBNF Grammars are Context-Free)**.** *The set of languages RBNF grammars can generate is the same those as BNF ones can.*

Proof. We provide translations from RBNF expressions into EBNF expressions. These translations can be performed in reverse for EBNF expressions written without exceptions to get RBNF expressions. $<$ and $>$ in RBNF are replaced by commas. Strings of consecutive non-terminals are grouped together and surrounded by quotation marks.

Since RBNF has no equivalent for production rules of the form $R - E$, it generates the same languages as BNF by lemma 3.5. □

Here is a RBNF for the terminating decimals:

Example 3.16.

$$<DI> ::= [-]<D>\{<D>\}, [.<D>\{<D>\}];$$
$$<D> ::= 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9;$$

□

3.2.5 LBNF

(Labelled BNF) Forsberg & Ranta (2005) extends EBNF with functionality for dealing with polymorphic lists of rules. It also provides a few pre-defined sets such as characters, integers, strings, and identifiers. It also provides labels which deal with higher order abstract syntax Pfenning & Elliott (1988). However, this is not intended to be used in LBNF grammars written by hand, but instead in ones generated from the grammar formalism GF (Grammatical Framework) Ranta (2004).

This means there is no clear mathematical model of this functionality to aid in human understanding. Analysing the functions provided by LBNF requires an understanding of the programs used to compile expressions in the grammar.

Since LBNF is not designed with human readers in mind, we do not include any lemmas about the languages it generates and the strings it is able to parse in this paper. Proving lemmas about LBNF would be cumbersome and involve delving into compiler code.

While LBNF is a significant expansion of BNF, we include LBNF here, because it still deals with sets of strings generated by term rewriting rules and does not allow the incorporation of mathematical objects and expressions into the syntax. In addition the syntax of MBNF does not typically include anything like the labels provided by LBNF.

3.2.6 TBNF

(Translational Backus-Naur Form) Mann (2006) Puts non terminals in the place of internal nodes and terminals in place of external (leaf) nodes on a tree (which we call an Abstract syntax tree or AST). When the resulting syntax is parsed, the abstract syntax tree it creates is traversed adding to EBNF additional production arrows listed in Figure 3.1 below.

[~>]	Reverse Production Arrow. An arrow preceding the right side of a rule for which you want the nodes to be arranged in reverse order.
[=>]	Call A Function. This means to call a function when a rule in the grammar has been recognized. A rule in the grammar may have multiple function calls.
[+>]	Make A Node. This means to make a node corresponding to this rule in the grammar. During AST traversal this node will be processed with a built-in default node processing function.
[*>]	Make A Node and during AST traversal, call a function with the same name as this node, instead of calling the default node processing function. This allows customization of the code generation process.
[**$1**]	Parse Stack Position. This refers to the symbol in parse stack position #1, the first symbol in the right side of the rule. $n refers to the *n*th position.
[**..***]	Node Traversal Indicator. Indicates when processing for this node should occur, at top- down, pass-over, or bottom-up time, respectively. *.. indicates top-down only. ..* indicates bottom-up only. *.* indicates top-down and bottom-up.
[**(...)**]	The Arguments. Arguments are used for function calls (=>) and node processing (+>). For node processing the arguments apply to the relative '*' in the Node Traversal Indicator. *-* would require two string arguments.
[**& 0**]	Counter Indicator. When the AST processor enters a node, it increments a counter for the node and puts in on a stack. The '& 0' indicates the current count for the node taken from the stack. A '& 1' means the counter for the parent node on the stack and '& 2' means the counter of the grandparent. This provides a unique number for labels.

Figure 3.1: Additional production arrows

Figure 3.1 demonstrates that TBNF provides a very rich extension to EBNF which is particularly well suited for relating expressions to their abstract syntax trees. However a clear mathematical model of the trees generated by TBNF is not provided alongside its syntax (again these are created in a compiler rather than presented in a form intuitive to human beings). In addition while it covers some of the functionality authors expect when they use MBNF it does not particularly resemble the way in which it is written.

We do not include any lemmas about the languages TBNF generates and the strings it is able to parse in this thesis, because TBNF is not designed with human readers in mind, so proving these would be cumbersome and involve delving into compiler code.

3.3 A Brief History of BNF and Its Variants

The following history of BNF variants is mostly a paraphrasing of one given in a talk by Steele (2017). We have chosen this as one of the more comprehensive histories available provided by someone with a specific interest in notation.

- 6th–4th century BCE: Aggarwal (n.d.) writes the Aṣṭādhyāyī, a formal grammar containing numerous concise technical rules that describe Sanskrit linguistics, syntax, and semantics unambiguously and in particular describe its morphology completely. This book describes the production of strings by the substitution of non-terminals (Although it does not use the word "non-terminal" nor does it use a notation particularly similar to BNF). Pāṇini used the method of "auxiliary symbols", in which new affixes are designated to mark syntactic categories and the control of grammatical derivations. This technique, rediscovered by the logician Emil Post, became a standard method in the design of computer programming languages. Bhate & Kak (1993) Kadvany (2007)
- 1914: Thue (1914) studies string-rewriting systems defined by rewrite rules. In this paper, Thue introduces a system consisting of pairs of corresponding strings over a fixed alphabet:

  $$(A_1, B_1), (A_2, B_2), ..., (A_n, B_n)$$

  and poses the following problem:

  Definition 3.17 (The word problem for semigroups)**.** Given two arbitrary strings P and Q, can we get one from the other by replacing some substring A_i or B_i by its corresponding string in the other half of the pair? This replacement may take place in either direction (for example, if P was 'mcm' and Q was 'ncn' and the answer would be "yes" for a Thue system containing pairs ('m', 'q'), ('l', 'q'), and ('l', 'n')). □

BNF is sometimes called a "Semi-Thue" system. I.e., a system in which this replacement of substrings happens in one direction only. Every BNF is a Semi-Thue system, but not every Semi-Thue system is a BNF.

- 1920s: Post (1943) studies "tag systems" in which symbols are repeatedly replaced by associated strings (this work is not published until 1943). These "tag systems" are later referred to as "post production systems" and they correspond exactly to the Semi-Thue systems.
- 1947: Markov (1947) and Post (1947) independently prove that the word problem for semigroups (a problem posed by Thue) is undecidable.
- 1956: Chomsky (1956) publishes "Three Models for the Description of Language," which describes grammars with production rules and what we now call the "Chomskian hierarchy of grammars". The grammars written using BNF corresponds exactly to the context free grammars in Chomsky's hierarchy.
- 1959: Backus (1959) uses a specific syntax to write production rules for a context-free grammar for the International Algorithmic Language, which is influenced by the "Post productions" of Emil Post. A single production may contain multiple alternatives. This notation is discussed in Section 3.1.
- 1960: The "Report on Algol 60," edited by Peter Naur Backus et al. (1963), appears in CACM. It uses a slightly prettier (and easier to typeset) variant of the Backus notation. Naur introduces use of ::= and | and makes names of nonterminals identical to equivalent English phrases used in the text. This notation is discussed in Section 3.1.
- 1965: IBM's PL/I specification Radin & Rogoway (1965) combines BNF with COBOL metanotation. An ellipsis indicates a nonzero number of repetitions of the preceding item. Here is an example of the features added:

“

expression ::= union | {expression || union}

expression-1 ::=

$$\left[\begin{array}{ll} \text{TO expression-2} & \text{[BY expression-3]} \\ \text{BY expression-3} & \text{[TO expression-2]} \end{array}\right] \text{[WHILE expression-4]}$$

DECLARE [level] name [attribute] ...
[, [level] name [attribute] ...] ...;

”

- 1965: PL360 Wirth (1968) used a parametrised form of BNF. If in the denotations of the constituents of the rule the script letters $\mathcal{A}$, $\mathcal{K}$, or $\mathcal{J}$ occur more than once, they must be replaced consistently (or possibly according to further rules given in the accompanying text). As an example, the syntactic rule

  “ $\langle\mathcal{K}$ register$\rangle$ ::= $\langle\mathcal{K}$ register identifier$\rangle$ ”

  is an abbreviation for the set of rules

  “
  ⟨long real register⟩ ::= ⟨long real register identifier⟩
  ⟨integer register⟩ ::= ⟨integer register identifier⟩
  ⟨real register⟩ ::= ⟨real register identifier⟩
  ”

- 1968: Adriaan van Wijngaarden et al. Mailloux et al. (1969) describe Algol 68 using a two-level grammar: one grammar has an infinite set of productions, which are generated by another grammar. The notation used is powerful enough to define the type 0 grammars in the Chomsky hierarchy.
- 1970: The BLISS language Wulf et al. (1971) is described using BNF, but with a right-arrow instead of "::=". This notation is taken for granted.

  “
  block → begin declarations compoundexpression end
  declarations →| declaration; | declarations; declaration;
  compoundexpression →| e; | e; compoundexpression;
  begin → BEGIN
  end → END
  ”

- 1976: Stanford's SAIL Reiser (1976) language uses BNF with repeated "::=" and no "|".
- 1978: CMU Alphard project Shaw et al. (1977) uses regular expressions in BNF with $*$, $+$, and #.
- 1980: Ada specification Barnes (1980) Boute (1980) uses BNF, but with "is" for "::=" and "or" for "|".
- 1981: CMU Shaw (1981) FEG and IDL use regular expressions in BNF with * , + , and ? .
- 1981: The British Standards Institution publishes BS 6154 Scowan (1981) a precursor to the EBNF standard.
- 1984: C: A Reference Manual Harbison (1984) uses REs in BNF.
- 1990: Common Lisp: The Language Steele (1990) uses REs in BNF.
- 1995: Python Reference Manual (Release 1.2) Rossum (1995) uses * and + in BNF, but brackets (rather than ?) for optional items.
- 1996: The ISO standard for EBNF ISO (1996) is published.
- 1997: The first (now deprecated) RFC standard for ABNF Crocker & Overell (1997) is published.
- 1998: Haskell 98 Report Jones et al. (1999) uses BNF, with -> for ::=, and also uses ellipsis.
- 1998: Ruby Language Reference Manual (1.4.6) Matsumoto (1988) uses * and + in "pseudo BNF" (somewhat like WSN), but brackets (rather than ?) for optional items.
- 2004: Ford (2004) introduces the concept of parsing expression grammars.
- 2005: Forsberg & Ranta (2005) document the Labelled BNF formalism.
- 2006: Mann (2006) documents the Translational BNF notation.
- 2008: The most recent RFC standard for ABNF Crocker et al. (2008) is published.
- 2009: The RFC standard for RBNF Farrel (2009) is published.

3.4 Limitations of BNF and Its Variants

An important thing to note when considering whether a piece of BNF-like syntax can be readily converted into BNF is that BNF is a language for building sets of strings and the only notion of equality it deals with easily is string equality. It is possible to derive a notion of tree equality from the parse trees generated by an EBNF grammar, but there is no guarantee that parse trees will be unambiguous. Unless an author writes their grammar with a parser in mind, inferring a sensible parse tree from a set of production rules and an input string is non-trivial.

BNF can describe exactly the context-free grammars in Chomsky's hierarchy Chomsky (1956). Non-context-free grammars cannot be written without extending BNF in some way.

Chapter 4

Introducing MBNF

In this chapter, we highlight ways in which the notation we call MBNF differs from BNF and its variants. We demonstrate that MBNF is non-trivially different to the existing extensions of BNF, which deal only with sets of strings and the parse trees deriving from production rules and input strings, and that it should not be thought of as dealing with the same kinds of entity as these extensions do. The following key features of MBNF are covered in this chapter:

1. Where BNF and its variants use strings, MBNF uses math text.
2. MBNF is not intended for parsers.
3. MBNF uses operators that stand for chains of syntax.
4. MBNF allows powerful operators e.g. context hole filling.
5. MBNF mixes mathematical language with BNF-style notation.
6. MBNF has at least the power of indexed grammars.
7. MBNF has a native concept of binding.
8. MBNF allows “arbitrary” side conditions on production rules.
9. MBNF can contain very large infinite sets within the “syntax.”

10. MBNF may produce large sets of undecidable syntax.

11. MBNF allows infinitary operators.

12. MBNF allows coinductive definitions.

13. MBNF may be considered up to “arbitrary” equivalences.

MBNF is widely used as a more expressive alternative to BNF notation. MBNF can be used to write all of the grammars BNF and its variants can produce, but it also defines grammars BNF cannot define. There are a number of reasons we think the features discussed in this chapter should be sensibly grouped under the common heading “MBNF”:

1. Most of the features discussed in this chapter are inherited from math and come about from the mixing of BNF-style notation with math text. This gives MBNF additional expressive power.

2. Some authors use more than one of the features discussed in this chapter at a time. For example Chang & Felleisen (2012) use features discussed in Subsections 4.1.4, 4.1.7 and 4.1.8.

3. Our work is novel in that there is no existing standard for the features discussed in this chapter as they appear alongside BNF-style notation.

4. There is no document we are aware of that prohibits the use of any of the features discussed in this chapter together with one another.

5. MBNF-style featurs seem to be commonly accepted in the community, judging from papers using this notation informally are regularly accepted in conferences and journals in this field.

4.1 Key Differences Between MBNF and BNF

4.1.1 Where BNF and Its Variants Use Strings, MBNF Uses Math Text

Instead of using strings of left to right ordered symbols MBNF uses what we are calling math text, i.e. text which one arranges as one might arrange mathematics. Parentheses are often used to disambiguate between similar chunks of text. But these are not needed in MBNF and when a grammar specifies such parentheses they can often be omitted without any need to explain. For example, writing $\boxed{(\lambda x.xx)\lambda y.y}$ instead of $\boxed{((\lambda x.(xx))(\lambda y.y))}$. MBNF is a "structured" as opposed to a "linear" notation. It is in that sense similar to BNFs, but fundamentally stronger in which kind of structures to express (more than just finite trees) and the contents in each of the nodes (arbitrary equivalences corresponding to math structures). While some BNF grammars can produce parse trees MBNF differs in that it can use structure implicit in the layout of symbols on the page when features like superscripting and overbarring are used. In this way it is closer to the structure a mathematician might use. For example, in $\boxed{\overline{f_x^{n+1} + \overline{y \cdot fj}}}$ the overbar can be thought of as being higher in the "tree" than the text occurring beneath it and the superscripting operation can be thought to be higher in the tree than the syntax inside the superscript.

Instead of non-terminal symbols, MBNF uses metavariables, which appear in what we call *math text* and obey the conventions of mathematical variables. Recall a metavariable is a variable at the meta-level which denotes something at an object-level. The object level of MBNF is called *concrete syntax* if it has no metavariables appearing in it. Concrete syntax in MBNF loosely corresponds to the notion of a string of terminal symbols in BNF. However, in BNF syntax must always be reduced to a string of terminal symbols and fed into a parser before further computations can be performed on it, whereas this is not always the case for MBNF. Frequently, MBNF uses metavariables which have not been assigned any concrete syntax and reasoning about a grammar is performed assuming some arbitrary metavariable assignment satisfying the constraints. In fact, this practice is so commonplace each of the examples we give in this chapter which are taken from other papers appear in grammars where some metavariables are not assigned a set of concrete syntax which they range over. In MBNF metavariables are not distinguished from other symbols by annotating them as BNF and its notational variants do, but by math text features such as font, spacing, or merely by tradition.

In addition to arranging symbols from left to right on the page, math text allows arranging text by subscripting, superscripting, pre-subscripting, pre-superscripting, and placing text above or below other text. It also allows for marking whole segments of text, for example with an overbar (a vinculum). Readers can find more detailed information on how math text can be laid out in The TeXbook Knuth (1986), or the Presentation MathML Ion et al. (2001) and OpenDocument ISO (2015) standards. Here is a nonsense piece of math text to illustrate how it may be laid out:

Example 4.1 (Example of Math Text Layout)**.**

$$ {}^{c}{\downarrow}\, a' = \check{p}\langle v''_x \odot a^{2+1}\rangle - \overline{f^n_x + \overline{\overline{y \cdot fj}}} + \sum_{i=0}^{\infty} s_{i\in 1\ldots n} \frac{\leftarrow}{a, b, c} b\hat{a} $$

□

Lemma 4.2 (Optional Parentheses Are Not Supported by Formal BNF Variants)**.** *Neither BNF nor the variants in Section 3.2 deal with optional parentheses.*

Proof. For both BNF and the variants in Section 3.2 the only way of parsing parentheses as part of a grammar is to include them as part of the literal syntax. While PEGs and TBNF provide tools for parser preference which enable them to express an order of operation that may involve parentheses, they do not allow for automatic addition and removal of parentheses. □

The optional addition of parentheses allows authors who use MBNF to write in a compact and easily readable form and one which corresponds more closely to how mathematics is written. This is a motivation for how we will go on to model syntax. We will give a tree like structure with internal equivalence classes around syntax boundaries, which is similar to the treatment of parentheses in math text.

Lemma 4.3. *[Math Text Layout Is Not Supported by Formal BNF Variants] Neither BNF nor the variants in Section 3.2 deal with the layout of math text.*

Proof. Both BNF and the variants in Section 3.2 only parse text that is joined together by concatenation. Therefore they do not handle overbarring, underbarring, subscripting, or superscripting. While TBNF provides tools for building tree-like structures it still does so by parsing strings. □

4.1.2 MBNF Is Aimed Exclusively at Human Readers

MBNF can be used to write all of the grammars BNF and its variants can produce, but it also defines grammars BNF cannot define. Unlike many BNF variants, MBNF is meant to be interpreted by humans, not computers/parser generators, as it has not been adequately formalised yet. Authors may give an MBNF grammar in an article for humans and a separate grammar for use with a parser generator to build a corresponding implementation.

Definition 4.4. When we say a grammar is serialized we mean each of the terms generated by that grammar is given a unique numeric encoding (usually if every term is given a corresponding binary representation in order to represent them on a computer). A grammar is serialisable if such an encoding is possible. □

MBNF defines entities not intended or expected to be serialised or parsed.

Example 4.5. We find an example in Dolan & Mycroft (2017)

“

$$\begin{aligned}
e &::= x \mid \lambda x.e \mid e_1 e_2 \mid \{\ell_1 = e_1, ..., \ell_n = e_n\} \mid e.\ell \mid \mathit{true} \mid \mathit{false} \\
&\quad \mid \mathit{if}\, e_1\, \mathit{then}\, e_2\, \mathit{else}\, e_3 \mid \hat{\mathbf{x}} \mid \mathit{let}\hat{\mathbf{x}} = e_1 \mathit{in} e_2 \\
\Gamma &::= \epsilon \mid \Gamma, x : \tau \mid \Gamma, \hat{\mathbf{x}} : \forall\vec{\alpha}.\tau \\
\tau &::= \mathit{bool} \mid \tau_1 \rightarrow \tau_2 \mid \{\ell_1 : \tau_1, ..., \ell_n : \tau_n\} \mid \alpha \mid \top \mid \bot \mid \tau \sqcup \tau \mid \tau \sqcap \tau \\
\Delta &::= \epsilon \mid \Delta, x : \tau \\
\Pi &::= \epsilon \mid \Pi, \hat{\mathbf{x}} : [\Delta]\tau
\end{aligned}$$

”

□

In this example Γ, Δ, and Π are never intended to be serialised. If an implementation did end up actually serializing instances of Γ, Δ, and Π to store them in a file or send them across the network, then the parser would almost certainly not use the arrangement of symbols suggested by this grammar, or a renamed version of them. The authors provide an implementation in OCaml which looks very little like the above syntax:

`https://github.com/stedolan/mlsub`

Most MBNF grammars are missing features needed to disambiguate complex terms (for example, notation for separating metavariables from concrete syntax and from other kinds of evaluated syntax (like $\langle$ and $\rangle$ do in BNF)), bracketing (as covered in Section 4.1.1) and notation for declaring operator precedence (for the example above Dolan & Mycroft (2017), no rules are given for the order in which patterns should be matched)). Papers often put complicated uses of the mathematical metalanguage inside MBNF grammars (for example hole filling, which uses syntax splicing operations that may not be parser friendly).

Definition 4.6. A piece of syntax is *undecidable* if you cannot read it using any parser (i.e., if its Abstract syntax tree is not regular). ☐

SubSection 4.1.10 of this thesis contains undecidable syntax; i.e., syntax for which no possible parser could be built, even if it were written very differently. Subsections 4.1.9 4.1.11 and 4.1.12 contain infinite syntax (i.e., syntax which can be written as an infinitely large graph), which cannot be serialised unless it also has a finite representation (e.g. as a regular tree as is the case for example 4.26).

4.1.3 MBNF Uses Operators That Stand For Chains of Syntax

In BNF, all concrete syntax (i.e. the string of symbols which can be given an encoding through a given data-type) are generated by non-terminals. In MBNF we can have operators which stand for chains of binary operators, which each take 2 terms and insert concrete syntax between them.

Example 4.7. We find an example in Milner (1999) that appears as part of a calculus for analysing properties of concurrent communicating processes:

“

$$P ::= A\langle a_1, \ldots, a_n \rangle \mid \sum_{i \in I} \alpha_i.P_i \mid P_1 | P_2 \mid \mathsf{new}\, a\, P$$

where I is any finite indexing set. ”

□

Here, choosing our finite indexing set to be the natural numbers less than n, $\sum_{i \in I} \alpha_i.P_i$ is meant to be read as standing for $\alpha_1.P_1 + \cdots + \alpha_n.P_n$ where each $+$ is concrete syntax. This notation is taken for granted. We will see more examples of operators which build chains of syntax, where one term stands for a family of MBNF operators, in subsections 4.1.11 and 4.1.12.

4.1.4 MBNF Allows Powerful Operators e.g. Context Hole Filling

Chang & Felleisen (2012) present a grammar defining the λ-term contexts with one hole where the spine is a balanced segment. We define the *spine* as follows: The root node is on the spine. If A is applied to B by an application on the spine, then the root node of A is on the spine and the root node of B is not. If a node on the spine is an abstraction each of its children is on the spine (i.e., every node appearing on the furthest left hand side of the tree is on the spine). We define a *balanced segment* as follows: A balanced segment is one where each application has a matching abstraction and where each application/abstraction pair contains a balanced segment.

For explanatory purposes, when presenting this grammar, we write $e @ e$ instead of $e\, e$ and add parentheses. Concrete syntax and BNF-style notation are olive. Metavariables are blue. Tree-splicing operators (aka operators like hole filling which replace one node on the tree with of another tree) are red. We have also slightly standardised their notation to use “::=.” Apart from these adjustments we use the same syntax Chang and Felleisen use.

Example 4.8.

$$e ::= x \mid (\lambda x.e) \mid (e@e)$$
$$A ::= [\,] \mid (A[(\lambda x.A)]@e)$$

□

One can think of the context hole filling operation ($\boxed{[\,]}$ in $\boxed{(A[(\lambda x.A)]@e)}$) as performing tree splicing operations within the syntax. Here the green square brackets with nothing between them represents a hole at the object level of the syntax whereas the red square brackets with text between them represents a computation on the mathematical meta level, filling the leftmost hole in the tree with the text between the square brackets. Figure 4.1 illustrates steps in building syntax trees for A (the operation $\bullet@\bullet$ is higher up the tree than $\bullet[\,\bullet\,]$ because of parsing precedence inherited from math):

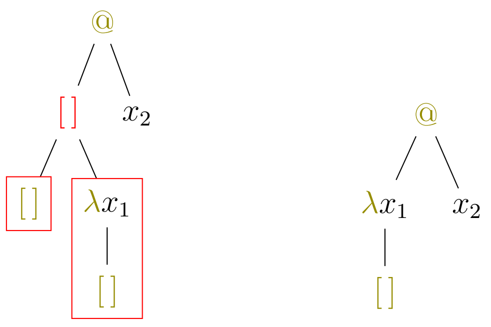


Figure 4.1: Steps in building a syntax tree for A

Figure 4.1. shows the result of the second rule where each A is $\boxed{[\,]}$ and e is a variable. The tree on the left is the tree corresponding to $\boxed{A[\lambda x.A]@e}$ before the hole filling operation is performed, where the first A is assigned $[\,]$. The tree on the right represents an unparsing of the typical syntax tree for $\boxed{((\lambda x_1.[\,])@x_2)}$. x_1 and x_2 are disambiguated instances of x. A metavariable assigned a value won't appear in the final tree. If it is not a terminal node, $[\,]$ tells us to fill in the leaf in the frame on the left with the tree in the frame on the right. Once performed, $[\,]$ disappears.

We can show that, unlike BNF, the language of the metavariable/non-terminal A (the set of strings derived from A using roughly the rules of BNF plus hole filling) is not context-free and so MBNF certainly isn't, as we show in Lemma 4.9.

For this section we use ToStr informally to mean a function which takes pieces of MBNF syntax, provided it can be written as a chunk of math text whose only operation is concatenation and whose only equivalence is syntactic equivalence, and takes them to the fully parenthesised string one might normally use to refer to it. For example, if we let $O = (\lambda a.[\,])@b$ declare the object O, then $\mathsf{ToStr}(O) =$ "$((\lambda a.[\,])@b)$", where "$((\lambda a.[\,])@b)$" is the symbol "(" concatenated with another "(" concatenated with "λ" etc., and O is the data structure $(\lambda a.[\,])@b$ represents. In order to show that MBNF is not context free, we make use of the pumping lemma for context-free languages, which appears in Berstel et al. (2009) and which we reproduce in this paper as lemma 3.7 for ease of reference.

Lemma 4.9 (Hole Filling is Not "Context-free")**.** *For A given by the MBNF grammar in example 4.8, the language given by $\{x \in \mathsf{String} \mid x = \mathsf{ToStr}(A)\}$ is not Context-free.*

proofsketch. We need to show that for any given $p \geq 1$ we can produce an $s = \mathsf{ToStr}(A)$ such that no substring of s can be "pumped" (some non-empty part of one or both of its outermost substrings repeated) to give another string in the language $\{x \in \mathsf{String} \mid x = \mathsf{ToStr}(A)\}$.

Since each A has a balanced segment along the spine we must be expected to keep count of both abstractions and applications. Parentheses must also be balanced. Getting the desired result is as simple as selecting an $s = \mathsf{ToStr}(A)$ such that the abstraction at the bottom of the spine of A is more than p abstractions away from the application closest to the bottom of the spine of A and such that A contains no e long enough to be pumped. Since parentheses are balanced, the only possible section we might choose to "pump" is around the hole. Since there are p abstractions before we reach an application, there is no way that "pumping" this could give us a balanced segment. □

Corollary 4.10 (Hole Filling Adds Power to BNF and its Close Variants)**.** *Hole filling can generate languages that BNF, ABNF, and RBNF cannot.*

Proof. BNF generates the context free languages. By lemmas 3.13 and 3.15 ABNF and RBNF can also only generate the context free languages. By lemma 4.9 Hole filling can generate languages BNF, ABNF, and RBNF do not. □

EBNF ISO (1996), parsing expression grammars, LBNF Forsberg & Ranta (2005), and TBNF Mann (2006) do not have an analogue for context-hole filling.

4.1.5 MBNF Mixes Mathematical Language With BNF-Style Notation

Germane & Might (2017) mix BNF-style notation freely with mathematical notation in such a way that the resulting grammar relies upon both sets produced with mathematical notation and MBNF production rules which use metavariables defined using mathematical notation:

Example 4.11.

“
$$
\begin{array}{rcl@{\qquad}rcl}
u \in \mathit{UVar} &=& \text{a set of identifiers} & \mathit{ccall} \in \mathit{CCall} &::=& (q\, e^*)_\gamma \\
k \in \mathit{CVar} &=& \text{a set of identifiers} & e, f \in \mathit{UExp} &=& \mathit{UVar} + \mathit{ULam} \\
\mathit{lam} \in \mathit{Lam} &=& \mathit{ULam} + \mathit{CLam} & q \in \mathit{CExp} &=& \mathit{CVar} + \mathit{CLam} \\
\mathit{ulam} \in \mathit{ULam} &::=& (\lambda e(u^* k)\mathit{call}) & \ell \in \mathit{ULab} &=& \text{a set of labels} \\
\mathit{clam} \in \mathit{CLam} &::=& (\lambda_\gamma(u^*)\mathit{call}) & \gamma \in \mathit{CLab} &=& \text{a set of labels} \\
\mathit{call} \in \mathit{Call} &=& \mathit{UCall} + \mathit{CCall} & \mathit{ucall} \in \mathit{UCall} &::=& (f e^* q)_\ell
\end{array}
$$
”

□

The results of math computations are interleaved with MBNF production rules. They are not just applied after the results of the production rules have been obtained. This grammar uses $\bullet_1 \in \bullet_2$ to mean “$\bullet_2$ is the language of $\bullet_1$” (this is the case in both the MBNF production rules (::=) and the math itself (=)).

In the MBNF grammar above $+$ is used as union and we have the additional requirement that there must exist some procedure for choosing sets fulfilling these constraints such that, if, for some terms X and Y, $X + Y$ appears in the grammar, then X and Y do not intersect. This is not stated in the grammar the authors expect us to infer it from the use of $+$. Here, the requirement is most likely fulfilled by the author following the convention that any arbitrary sets declared separately are disjoint. However, in order to check that the rules of grammars like the one above can be satisfied, we would still need a general procedure for checking that $ULam$ and $CLam$ do not overlap, if such a convention is chosen.

It is worth mentioning briefly, that, unlike in many MBNF grammars where parentheses may be omitted, the parentheses here are mandatory. The syntax u^* indicates a repetition, $\vec{u}$ is a slightly more common way of writing this in MBNF.

BNF, EBNF, ABNF, RBNF, PEGs, LBNF, and TBNF do not have a concept of disjoint union and do not allow one to interleave set theoretic operations on the language of a non-terminal with ordinary BNF definitions.

4.1.6 MBNF Has at Least the Power of Indexed Grammars

Every language that can be produced by an indexed grammar can also be produced by an MBNF. Inoue & Taha (2012) use this grammar:

Example 4.12.

“ $$\mathcal{E}^{\ell,m} \in ECtx^{\ell,m}_{\mathbf{n}} ::= \cdots \mid \langle \mathcal{E}^{\ell+1,m} \rangle \mid \cdots$$ ”

□

This suggests that MBNF deals with some close relative of the family of indexed grammars Hopcroft et al. (2006), which is yet another reason it is not context-free. The $\ell + 1$ is a calculation that is not intended to be part of the syntax. The production rule above defines an infinite set of metavariables ranging over different sets.

One possible definition of an Indexed Grammar is given by Aho (1968) (from whom we take both of the following definitions and the resulting lemma).

Definition 4.13 (Indexed Grammars)**.** An indexed grammar is a 5-tuple, given by $G = (N, T, F, P, S)$, in which:

1. N is a finite nonempty set of symbols called the nonterminal alphabet.

2. T is a finite set of symbols called the terminal alphabet.

3. F is a set of so-called index symbols, or indices.

4. In productions as well as in derivations of indexed grammars, a string (or "stack"), $\sigma \in F^*$, of index symbols is attached to every nonterminal symbol $A \in N$, denoted by $A[\sigma]$. For an index stack $\sigma \in F^*$ and a string $\alpha \in (N \cup T)^*$ of nonterminal and terminal symbols, $\alpha[\sigma]$ denotes the result of attaching $[\sigma]$ to every nonterminal in α. Using this notation, each production in P has to be one of the following forms:

   (a) $A[\sigma] ::= \alpha[\sigma]$

   (b) $A[\sigma] ::= B[f\sigma]$

   (c) $A[f\sigma] ::= \alpha[\sigma]$

   Where $A, B \in N$, $f \in F$, $\sigma \in F^*$ is a string of index symbols, and $\alpha \in (N \cup T)^*$ is a string of nonterminal and terminal symbols.

5. S, the sentence symbol, is a distinguished symbol in N.

□

The $\ell + 1$ in the example above is loosely comparable to adding an index to the stack. We give the definition above only to relate the concept of an indexed grammar to a context free grammar and to make it clear that indexed grammars have a historical definition like the one above, which is already one step removed from the indexing in MBNF.

Definition 4.14 (Direct Derivation, Language of an Indexed Grammar)**.** Let $G = (N, T, F, P, S)$ be an indexed grammar the relation $\Rightarrow$ ("direct derivation") is defined on the set $(N[F^*] \cup T)^*$ of "sentential forms" as follows:

Let $\overset{*}{\Rightarrow}$ be the reflexive transitive closure of direct derivation $\Rightarrow$.

1. If $A[\sigma] ::= \alpha[\sigma]$ is a production of type a and $S \overset{*}{\Rightarrow} \beta A[\phi]\gamma$, then $\beta A[\phi]\gamma \Rightarrow \beta\alpha[\phi]\gamma$. That is, the rule's left hand side's index stack ϕ is copied to each nonterminal of

the right hand side.

2. If $A[\sigma] ::= B[f\sigma]$ is a production of type b and $S \overset{*}{\Rightarrow} \beta A[\phi]\gamma$, then $\beta A[\phi]\gamma \Rightarrow \beta B[f\phi]\gamma$. That is, the right hand side's index stack is obtained from the left hand side's stack ϕ by pushing f onto it.

3. If $A[f\sigma] ::= \alpha[\sigma]$ is a production of type 3 and $S \overset{*}{\Rightarrow} \beta A[\phi]\gamma$, then $\beta A[f\phi]\gamma \Rightarrow \beta\alpha[\phi]\gamma$ That is, the first index f is popped from the left hand side's stack, which is then distributed to each nonterminal of the right hand side.

The language $L(G) = \{w \in T^* : S \overset{*}{\Rightarrow} w\}$ is the set of all strings of terminal symbols derivable from the start symbol. ☐

Lemma 4.15. *[Indexed Grammars are not Context Free] The language of a metavariable appearing in an indexed grammar may not be context free.*

Proof. Consider the grammar $G = (\{S, T, A, B, C\}, \{a, b, c\}, \{f, g\}, P, S)$. This produces the language $\{a^n b^n c^n : n \geq 1\}$, where the production set P consists of:

$$\begin{array}{lll} S[\sigma] ::= T[g\sigma] & A[f\sigma] ::= aA[\sigma] & A[g\sigma] ::= a \\ T[\sigma] ::= T[f\sigma] & B[f\sigma] ::= bB[\sigma] & B[g\sigma] ::= b \\ T[\sigma] ::= A[\sigma]B[\sigma]C[\sigma] & C[f\sigma] ::= cC[\sigma] & C[g\sigma] ::= c \end{array}$$

This language is not context free by the pumping lemma, since for any pumping length p we can select $n > p$ such that no string of length p can encompass a, b, and c and thus cannot be "pumped." ☐

Corollary 4.16. *[Indexed Grammars Add Power to BNF and its Close Variants] Indexed grammars can generate languages BNFBackus et al. (1963), ABNFCrocker et al. (2008), and RBNFFarrel (2009) cannot.*

Proof. Follows trivially from previous lemmas 4.15, 3.13, 3.15. ☐

Lemma 4.17. *[Indexed Grammars Generate Languages EBNF Cannot] We can find a language generated by the indexed grammars that is not in the context free grammars plus their complements.*

proofsketch. It is known that the language $\{ww \mid w \in \{a, b\}^*\}$ can be generated by the indexed grammars Vijay-Shanker & Weir (1994). There is no way to generate this language from the context free grammars plus their complements. □

4.1.7 MBNF Has a Native Concept of Binding

In Germane & Might (2017) we found the following:

Example 4.18.

“ $$pr \in Pr = \{ulam : ulam \in Ulam, \text{closed}(ulam)\}$$ ”

□

In order to perform this evaluation of the set *Ulam* we must recognise which variables in each term ranged over by metavariable *ulam* are bound.

In addition we need a notion of binding to deal with some of the issues surrounding α-equivalence that often arise when authors start working with the grammar they define as part of a reduction system. Chang & Felleisen (2012) give the following axiom:

Example 4.19.

“ $$\hat{A}[A_1[\lambda x.\check{A}[E[x]]]A_2[v]] = \hat{A}[A_1[A_2[\check{A}[E[x]]x := v]]] \qquad \text{where } \hat{A}[\check{A}] \in A$$ ”

□

Each pair of square brackets denote a hole being filled. The notation $x := v$ denotes a capture avoiding substitution of x with v. In the side condition the reader is meant to infer that $\hat{A}$ and $\check{A}$ are metavariables, wheras A is meant to denote a language A, A_1 and A_2 belong to. I.e., A is overcoded both as a language and as a metavariable.

Here we are meant to recognise an implicit convention, known as the Barendregt convention, on the terms we are β-reducing over. In this case the Barendregt convention would require that we choose terms from the α-equivalence class of $\hat{A}[A_1[\lambda x.\check{A}[E[x]]]A_2[v]]$ such that no bound variable of $A_1[\lambda x.\check{A}[E[x]]]$ is a free variable in $A_2[v]$ and none of the bound variables in $A_2[v]$ are free variables in $\check{A}[E[x]]]$.[1] Since Chang and Felleisen also expect the Church-Rosser property to hold of their reduction relations, terms are identified up to α-equivalence again after performing the reduction and filling the holes.

Many different kinds of binding need to be handled, some of them quite complicated. For example: Dami (1998), uses an MBNF grammar to talk about dynamic binding, which is beyond the scope of what can be represented using Higher Order Abstract Syntax.

Example 4.20.

“

$$
\begin{array}{rcll}
a & ::= & x_l & \textit{labelled variable} \\
 & \mid & \lambda x.a & \textit{abstraction} \\
 & \mid & a(l = b) & \textit{bind expression} \\
 & \mid & a! & \textit{close expression} \\
 & \mid & \varepsilon & \textit{runtime error}
\end{array}
$$

”

□

A notion of binding is not native to BNF, EBNF, ABNF, RBNF, TBNF, or PEGs but must be defined after the grammar.

4.1.8 MBNF Allows “Arbitrary” Side Conditions on Production Rules

An example of a production rule with a side condition can be found in Chang & Felleisen (2012):

[1] Actually a slightly weaker condition than the one we give here is probably sufficient for the Barendregt convention to hold, but it would be more complicated to state.

> “ $E = [\,] \mid Ee \mid A[E] \mid \hat{A}[A[\lambda x.\check{A}[E[x]]]E] \qquad \text{where } \hat{A}[\check{A}] \in A$ ”

Note that in the above example the two xs are the same, as are the $\check{A}$s and the $\hat{A}$s but the Es and As are all different. Likewise, the A appearing in the side condition stands for the language of A, but, elsewhere, E, e, A, x $\check{A}$, and $\hat{A}$ are all metavariables. The author does not mention this anywhere. Instead, the reader is supposed to intuit it.

It is possible to make side conditions that prevent MBNF production rules from having a solution. We give a detailed proof of this in Subsection 4.3.1. A definition for MBNF can help in finding conditions on side conditions that ensure MBNF rules actually define something. We believe that authors often have some heuristic in mind which allows them to avoid cross reference of the sort in Subsection 4.3.1, but do not know of a definition which explicitly says what's allowed. Neither BNF nor any of its notational variants allow arbitrary side conditions on production rules.

Features of Mathematics Used in Side Conditions

Here are some parts of mathematics which might appear in side conditions:

1. Set membership Chang & Felleisen (2012) $\hat{A}[\check{A}] \in A$ and Frumin et al. (2019) $\xi \in \{L, U\}$
2. Hole filling Chang & Felleisen (2012) $\hat{A}[\check{A}] \in A$
3. Equals on index variable Inoue & Taha (2012) $m = \ell$
4. Greater than on index variable Inoue & Taha (2012) $\ell > 0$
5. Natural numbers (i.e., 0) Inoue & Taha (2012) $\ell > 0$
6. Set literals Frumin et al. (2019) $\xi \in \{L, U\}$
7. Real intervals Frumin et al. (2019) $q \in (0, 1]$
8. Types Forster et al. (2020) $x : \mathbb{N}, f : \mathcal{F}_\Sigma, P : \mathcal{P}_\Sigma$
9. Intersection Garnock-Jones & Felleisen (2016) $\pi_{add} \cap \pi_{del} = \emptyset$

10. Emptyset Garnock-Jones & Felleisen (2016) $\pi_{add} \cap \pi_{del} = \emptyset$

11. Equals on sets Garnock-Jones & Felleisen (2016) $\pi_{add} \cap \pi_{del} = \emptyset$

12. Sets appearing in side conditions with additional constraints involving =, and $\forall$ Hausmann & Schröder (2019) $\heartsuit \in \Lambda$ with additional constraints

> “for each modal operator $\heartsuit \in \Lambda$, there is a dual $\overline{\heartsuit} \in \Lambda$, such that $\overline{\overline{\heartsuit}} = \heartsuit$ for all $\heartsuit \in \Lambda$”

These suggest that very nearly the full power of mathematics may appear in a side condition. Indeed in subsection 4.3.1 we show that the parts of maths appearing in these side conditions are sufficient that constraints provided by an MBNF rule set may not be solvable.

4.1.9 MBNF Can Contain Very Large Infinite Sets Within the "Syntax"

Any set that is larger than countable is larger than we would typically think of data streams and the sets that house them as being, but in this case very large is the size of an inaccessible cardinal. Toronto & McCarthy (2012) write:

Example 4.21.

> “
> $$e ::= \cdots \mid \langle t_{set}, \{e^{*\kappa}\}\rangle$$
>
> Here $\{e^{*\kappa}\}$ means sets comprised of no more than κ terms from the language of e.
>
> ...The language of $v ::= \langle t_{set}, \{v^{*\kappa}\}\rangle$ is comprised of the encodings of all the hereditarily accessible sets. ”

□

The authors do not state what κ is, but elsewhere in the paper it is an inaccessible cardinal. It seems as though κ is also intended to be an inaccessible cardinal here.

It is worth noting that the sets contained in syntax produced by the above grammar are too large to be serialisable under any encoding.

BNF and its notational variants, by contrast, only deal with strings of finite length.

4.1.10 MBNF May Produce Large Sets of Undecidable Syntax

Frumin et al. (2019) use an MBNF grammar whose syntax contains metavariables which can be taken from a real interval

Example 4.22.

“ $$P, Q \in \mathsf{Prop} ::= \cdots \mid l \overset{q}{\mapsto}_{\xi} v \mid \cdots \qquad (q \in (0,1])(\xi \in \{L, U\})$$ ”

$\square$

If $(0, 1]$ is taken to be the real interval, then Prop includes syntax that is undecidable.

BNF and its notational variants, by contrast, are computationally decidable and do not produce uncountably large sets of syntax.

4.1.11 MBNF Allows Infinitary Operators

Llana Díaz & Núñez (1997) write a grammar which includes an infinitary operator:

Example 4.23.

“ $$P ::= \cdots \mid \bigsqcap_{i \in I} P_i \mid \cdots$$

...But, for instance in our language we have the term

$$\bigsqcap_{n \in \mathbb{N}}$$

where each P_n is born at time n, and so P is born at time $\omega + 1$. ”

“ ...So, to fully formalize the set of valid expressions, we begin by bounding the size of the possible sets of indices I, and that of the set of actions Act by some infinite cardinal κ. The functional governing the right hand side of the equation is clearly monotone, but it is not so obvious whether it has any fixpoint. Fortunately it has. Besides, it is guaranteed that it is reached after (at most) λ iterations, where λ is the smallest regular cardinal bigger than κ. Then, the principle of structural induction is valid and corresponds to the principle of transfinite induction. ”

$\square$

We can assume without loss of generality that I is an ordinal. Then $\underset{i \in I}{\sqcap} P_i$ stands for $P_1 \sqcap P_2 \sqcap \cdots \sqcap P_\omega \sqcap \cdots$ where $\forall i < I$, P_i appears somewhere in this chain.

Another example of a grammar which includes infinitary operators is given by Mislove (1995):

Example 4.24.

“

$$P ::= \ldots \mid \bigoplus_{i \in I} P_i \mid \ldots$$

...While $\bigoplus$ is defined to be an operator just like the others, there is one delicate point here. Namely all other operators have finite arity, but $\bigoplus$ is assumed to apply to any index set I. ...we can fix a regular cardinal κ that is larger than the cardinality of the family A of atomic actions in our language, the family X of variables, and ω, the first infinite cardinal. Then we can assume all index sets I satisfy card $I < \kappa$. ”

$\square$

We can assume without loss of generality that I is an ordinal. Then $\bigoplus_{i \in I} P_i$ stands for $P_1 \oplus P_2 \oplus \cdots \oplus P_\omega \oplus \cdots$ where $\forall i < I$, P_i appears somewhere in this chain.

The terms constructed by both of the infinitary operators given here are too large to be serialisable. This is an issue if we are imagining the syntactic objects produced by a set of MBNF production rules to be data structures held in some computer.

We may think of infinitary operators as defining trees of infinite breadth (i.e., trees whose internal nodes may have infinitely many direct children), where BNF and its notational variants deal with finite data structures (with a string representation).

4.1.12 MBNF Allows Coinductive Definitions

We are not aware of any fairly general attempts to formally define coinduction on the language of an MBNF although we sketch what this may mean and some of the issues which it may entail later in the document. Coinduction is normally defined on a stream of data which has a head and a possibly countable tail. It is the counterpart to parsing for infinite streams of data.

Eberhart et al. (2015) write:

Example 4.25.

> “ We consider processes to be infinite terms as generated by the grammar:
>
> $$P, Q ::= \Sigma_{i \in n} G_i \mid (P|Q) \qquad G ::= \overline{a}\langle b \rangle.P \mid a(b).P \mid \nu a.P \mid \tau.P \mid \heartsuit.P$$
>
> up to renaming of bound variables as usual. Such a coinductive definition... ”

□

We can assume without loss of generality that n is an ordinal. Then $\Sigma_{i \in n} G_i$ stands for $G_1 + G_2 + \cdots + G_\omega + \cdots$ where $\forall i < n$, G_i appears somewhere in this chain.

The syntax generated by the above grammar is not serialisable.

Castagna et al. (2009) also use MBNF coinductively:

Example 4.26.

“ The set of contracts Σ is the set of possibly infinite terms coinductively generated by the following grammar:

$$\begin{array}{rcll} \alpha & ::= & a \mid \overline{a} & a \in \mathcal{N} \\ \sigma & ::= & \mathbf{0} \mid \alpha.\sigma \mid \sigma \oplus \sigma \mid \sigma + \sigma & \end{array}$$

”

□

The above grammar would typically be read inductively and it is only the natural language text accompanying it that tells us to read it otherwise. In this example the author expects us to recognise that Σ denotes the set of contracts σ ranges over.

The coinductive use of MBNF is further complicated by the fact that there are papers that depend on both the inductive and coinductive readings and they might even be tangled together. For example, Castagna et al. use the grammar above alongside recursive contracts and recursive filters which they define inductively.

The grammar given by Castagna et al. is meant to produce the regular trees, so it is serialisable, although not under some encodings. These serialisation of these terms cannot be the syntax one would typically associate with this grammar (it could be given by a grammar, but it would have to be a different grammar). This is in contrast to the grammar given by Eberhart et al. which is unserialisable because it produces uncountably many terms.

We may think of coinductive definitions as allowing us to define trees of infinite depth (i.e., trees in which paths may pass through infinitely many nodes), where BNF and its notational variants deal with finite data structures.

4.1.13 MBNF May Be Considered up to “Arbitrary” Equivalences

As well as α-equivalence and binding, the syntax created by an MBNF grammar may be considered up to various other equivalences. For example, associativity and composition with a 0 element (as in the π-calculus Milner et al. (1992)), equivalence up to the exchanging of labels (as in the λ-calculus with records Pierce (2002)) equivalence up to repetition of elements (as with set-like syntactic objects), and additional equivalences which may be defined by the author.

Example 4.27. Tobisawa Tobisawa (2015) defines equivalences $\simeq_s$ and $\simeq_t$. First he introduces the operator $\cdot\langle\cdot,\cdot\rangle$.

“
$$\begin{array}{rcl} \mathrm{id}\langle v, d\rangle & := & v^d[\mathrm{id}], \\ ({}^w{\downarrow}(M)\cdot\sigma)\langle v, d\rangle & := & \begin{cases} M & \text{If } v = w \text{ and } d = 0, \\ \sigma\langle v, d - \delta_{vw}\rangle & \text{otherwise,} \end{cases} \\ (\uparrow_w \cdot\sigma)\langle v, d\rangle & := & \sigma\langle v, d + \delta_{vw}\rangle \end{array}$$
”

where δ_{vw} is the integer defined by

“
$$\delta_{vw} := \begin{cases} 1 & \text{If } v = w \\ 0 & \text{otherwise.} \end{cases}$$
”

Then $\simeq_s$ and $\simeq_t$ are defined inductively.

“
$$\begin{array}{rcll} \mathrm{id} & \simeq_s & \mathrm{id} & \\ \sigma & \simeq_s & \tau & \text{if } \sigma\langle v, d\rangle \simeq_t \tau\langle v, d\rangle \text{ for any } v, d \\ v^d[\sigma] & \simeq_t & v^d[\tau] & \text{if } \sigma \simeq_s \tau \\ \lambda v.M & \simeq_t & \lambda v.N & \text{if } M \simeq_t N \\ M_1 \mathbin{@_\ell} M_2 & \simeq_t & N_1 @_\ell N_2 & \text{if } M_1 \simeq_t N_1 \text{ and } M_2 \simeq_t N_2 \end{array}$$
”

□

The operator $\cdot\langle\cdot,\cdot\rangle$ does not appear in the production rules of the grammar. Both $\cdot\langle\cdot,\cdot\rangle$ and large parts of the syntax which occurs inside equivalences $\simeq_s$ and $\simeq_t$ are intended to be given using the quoted text alone.

It may not be obvious to the reader that MBNF equivalences cannot simply be applied after the MBNF grammar has been solved by quotienting on the literal syntax. Indeed, in the grammar above this might work. However, we may use existing features of MBNF to construct an example where this is not possible. In the grammar given by Tobisawa equivalences are given over terms that do not occur within the literal syntax of the language given by the MBNF (so we know this is permissible). Given the side conditions covered in Subsection 4.1.8, it also seems reasonable to assume that we are allowed to use $\notin$ as part of a side condition. We may use these assumptions to prove equivalences must be calculated as an MBNF grammar is solved. I.e., MBNF does not just rest on inductive definitions, but on inductive recursive definitions too. Further discussion of induction recursion is given by Dybjer & Setzer (2003). Our model for defining nested equivalences over syntax given in section 5.2 relates to induction recursion as the set Object is inductively defined with recursively defined set $\mathsf{Pointer}$ acting as a decoding function. Like induction recursion in set theory the set Object is captured formally as the least fixed point of a monotone operator on the lattice of subsets of a sufficiently large base set. A more thorough investigation of the relation between induction recursion and MBNF is left for future work.

Lemma 4.28 (Equivalences Must be Calculated as an MBNF Grammar is Solved)**.** *Quotienting by syntactic equivalence cannot always be done over a set of "literal" terms generated by an MBNF grammar (i.e., those terms where every part of their tree is alone in its equivalence class), but must be done as the MBNF grammar is solved.*

Proof. Consider the following MBNF grammar:

$$\begin{array}{rcll} a \in A & ::= & b & \qquad\qquad \text{where } b \notin C \\ b \in B & ::= & \heartsuit \mid \spadesuit \mid \langle b \rangle b & \\ c \in C & ::= & \heartsuit \mid c@c & \end{array}$$

With no further equivalences the only element of B, which is not also an element of A, is $\heartsuit$. Suppose now we add the following equivalence:

$$c_1@c_2 \approx \langle c_2 \rangle c_1$$

Now $\langle\heartsuit\rangle\heartsuit$, for example, is no longer a member of A, whereas it was before the equivalence was added. In this case, the only members of B which are allowed in A after adding the equivalence are those pieces of syntax containing at least one instance of $\spadesuit$.

Since adding equivalences changes the syntax that can belong to an MBNF grammar, equivalences must be calculated as the grammar is solved or, as a bare minimum, grammars must be recalculated after equivalences are added. $\square$

Even if we wanted to exclude grammars like the one in the proof above, we would still need a notion of "safe" equivalences to do so, which would require consideration of equivalences as part of the grammar.

A sufficiently general notion of equivalence is not native to BNF and its notational variants but must be defined after the grammar.

4.2 A Brief History of MBNF

What follows is our best attempt at tracing a history of MBNF to give a general background of this notation. One difficulty we had in putting together this history is that MBNF was not recognised as a separate entity to BNF until this thesis. Therefore we welcome readers to send us earlier examples if they have found any.

- 1964: McCarthy (1964) introduces the notion of "abstract syntax" as opposed to syntax as functions on strings. He uses it to describe a subset of ALGOL called Microalgol. He does not use BNF-style syntax, but has been credited with coining the term.

- 1968: While the hole filling and BNF are defined separately Morris (1968) uses the notion of context hole filling alongside BNF-like syntax to analyse recursive definitions and type declarations in the λ-calculus, as shown in the following extracts:

“

$$< \text{wfe} > ::= \lambda < \text{variable} > . < \text{wfe} > | < \text{combination} >$$
$$< \text{combination} > ::= < \text{combination} > < \text{atom} > | < \text{atom} >$$
$$< \text{atom} > ::= < \text{variable} > | < \text{constant} > | \ (< \text{wfe} >)...$$

...$E[\lambda x.M]$ may be converted to $E[\lambda y.M']$ if y is not free in M and M' results from M by replacing every free occurrence of x by y. We write $E[\lambda x.M] =_\alpha E[\lambda y.M']$...

...$E[(\lambda x.M)N]$ is β-reducible if no variable which occurs free in N is bound in M. (This proviso prevents the ”capturing” of free variables.) Specifically, $E[(\lambda x.M)N]$ is reducible to $E[M']$ where M' results from the replacement of all free occurrences of x...

...A wfe $E[(\lambda x.M)]$ where x is not free in M is reducible $(>_\eta)$to $E[M]$. ”

He also identifies syntax up to α-equivalence while establishing Church-Rosser properties, without explicitly stating that he’s doing so. He does not use any hole filling in the BNF-like syntax itself, though. We include it as an early case of BNF and hole filling appearing in the same context before the merging of the two.

- 1975: Mosses (1975) uses syntax he describes as BNF-like. This has ellipses inside production rules.

“

$$\begin{array}{lll} \varepsilon \in \mathsf{ExpB} & \varepsilon ::= & \lambda\beta.\varepsilon \mid \mathsf{val}\beta.\varepsilon \mid \mathsf{fix}\beta.\varepsilon \mid \varepsilon \to \varepsilon, \varepsilon \mid \\ & & \varepsilon\omega\varepsilon \mid \omega\varepsilon \mid \varepsilon(\varepsilon) \mid \langle \varepsilon, ..., \varepsilon \rangle \mid \langle\,\rangle \mid \\ & & \iota_i \mid \nu_i \mid \omega_i \mid \text{true} \mid \text{false} \mid \text{err} \\ \beta \in \mathsf{Bvs} & \beta ::= & \iota_i \mid \langle \beta, ..., \beta \rangle \mid \langle\,\rangle \end{array}$$

”

In the above, he omits angle brackets around production rules and refers to them as metavariables.

- 1994: Ariola & Felleisen (1994) use hole filling operations as part of the meta-level BNF-style syntax.

  > “ $$E ::= [\,] \mid EM \mid (\lambda x.E)M \mid (\lambda x.E[x])E$$ ”

  Both Ariola and Felleisen have earlier articles covering similar ground, but this is the first where hole filling is used within BNF-style notation as a meta-level syntax building operation.

- 1995: Mislove (1995) uses BNF style syntax with an infinitary operator drawing from index sets smaller than a large cardinal, as shown in the following extracts:

  > “ $$P ::= \mathrm{STOP} \mid \mathrm{SKIP} \mid a \to P \mid P\backslash a \mid P;P \mid P||P \mid \bigoplus_{i\in I} P_i \mid x \mid \mu x.P...$$
  >
  > ...While $\oplus$ is defined to be an operator just like the others, there is one delicate point here. Namely all other operators have finite arity, but $\oplus$ is assumed to apply to any indexset I. Of course such an operator is not well-defined; there is no set of all sets to use as a basis for such a definition. But there is a way around this problem... ...we can fix a regular cardinal κ that is larger than the cardinality of the family A of atomic actions in our language, the family X of variables, and ω, the first infinite cardinal. Then we can assume that all index sets satisfy card $I < \kappa$. ”

- 2007: Sewell et al. (2007) develop Ott. This is a tool for specifying languages with a few of the capacities of MBNF. Ott is a metalanguage with relatively lightweight expressions for binding and list forms that produces an output in both MBNF and theorem prover formats. It is possible to generate MBNF specifications from Ott specifications, but not the other way around.

- 2009: Castagna et al. (2009) use BNF-style production rules coinductively, as indicated by a piece of natural language text accompanying the MBNF grammar.

“The set of contracts is the set of possibly infinite terms coinductively generated by the following grammar:

$$\begin{array}{lcll} \alpha & := & a \mid \overline{a} & a \in N \\ \sigma & ::= & 0 \mid \alpha.\sigma \mid \sigma \oplus \sigma \mid \sigma + \sigma & \end{array}$$
”

They use this alongside BNF-style production rules which are meant to be read inductively.

- 2017: Steele (2017) begins documenting uses of MBNF as part of his efforts to document computer science metanotation (CSM). He points out the need for a greater understanding of this notation.

4.3 Some Mathematical Properties of MBNF

4.3.1 Constraints Provided by an MBNF Rule-Set May Not be Solvable

We consider 2 different cases whereby the constraints given by the production rules appearing in a grammar may not have a solution. The first case we will consider is where the grammar features side conditions which prevent it from having a solution. This is not identical to an empty inductive type instead its a definition of sets whose constraints demand they be populated, but for which no stable solution can be found. The second case we will consider is where infinitary operators give syntax that is too large for the MBNF grammar to have a solution within any standard foundation of mathematics. The examples given in this section are based on known constraints on set-theoretic constructions, although we develop them for MBNF ourselves.

Case 1: Unsolvable Side Conditions

Consider again the following by Chang & Felleisen (2012):

“ $$E = [\,] \mid Ee \mid A[E] \mid \hat{A}[A[\lambda x.\check{A}[E[x]]]E] \qquad \text{where } \hat{A}[\check{A}] \in A$$ ”

Side conditions like this one may cause problems in making sure that an MBNF grammar defines a language. We offer a set of assumptions about what one may be allowed to do in MBNF which are separately plausible and unproblematic, but which allow us to create a grammar which does not define a language, if we use all of them unrestrictedly. Cases like this provide motivation for a definition of when an MBNF grammar is safely defined.

Where we are allowed to use $\in$, we are usually allowed to use $\notin$. The side condition of the MBNF production rule for E has a term that is created by filling a hole in a instance of $\hat{A}$ with a instance of $\check{A}$. This suggests that we may be allowed to use mathematical operations similar to those used in the production rules themselves to create the terms featuring in the side conditions of the production rules. We may conclude that, provided we have a production rule of the form $b \in B ::= \cdots$, we can have $\heartsuit b$ in one of our side conditions where $\heartsuit b$ is just a syntax building operation applied to b. MBNF, like BNF, also allows us to have production rules that reference themselves, either directly or indirectly. By allowing all these assumptions, we can make a grammar that does not have a solution. The grammar below is slightly more complex than $\boxed{a \in A ::= t \text{ where False}}$. We wanted to showcase a grammar whose side conditions used operations you could find in the wild. We also wanted a language where each set could not be statically defined, rather than where one was defined as empty.

Lemma 4.29 (Not Every Use of MBNF Which Features Side Conditions Has a Solution)**.** *Unrestricted application of side conditions that use the full power of mathematics and syntax building notation can provide constraints which have no solution when used alongside BNF-style notation.*

Proof. Consider the following MBNF grammar:

$$\begin{aligned} a \in A &::= \circ \mid \heartsuit a \mid b && \text{where } \heartsuit b \notin C \\ b \in B &::= \circ \mid a \mid \spadesuit b \\ c \in C &::= a \mid b \end{aligned}$$

Observe that $\spadesuit\circ \in B$.

Suppose that $\heartsuit\spadesuit\circ \notin C$. Then $\spadesuit\circ \in A$. Then $\heartsuit\spadesuit\circ \in A$, because for all $a \in A$ it holds that $\heartsuit a \in A$. Then $\heartsuit\spadesuit\circ \in C$, because $C ::= a \mid b$.

Suppose instead that $\heartsuit\spadesuit\circ \in C$. Then either $\heartsuit\spadesuit\circ \in A$, or $\heartsuit\spadesuit\circ \in B$. Then $\heartsuit\spadesuit\circ \in A$, because every term in B is either an $\circ$, or else it begins with $\spadesuit$, or else it is also in A. Then $\spadesuit\circ \in A$, because $\heartsuit y \in A$ if and only if $y \in A$. Then $\spadesuit\circ$ can only be produced by the production rule

$$a \in A ::= \cdots \mid b \qquad \text{where } \heartsuit b \notin C,$$

because $\spadesuit\circ$ is not of the form $\circ$ or $\heartsuit A$. Then $\heartsuit\spadesuit\circ \notin C$, because otherwise this production rule could not produce $\spadesuit\circ$.

So we have that, if $\heartsuit\spadesuit\circ \in C$, then $\heartsuit\spadesuit\circ \notin C$ and if $\heartsuit\spadesuit x \notin C$ then $\heartsuit\spadesuit\circ \in C$.

So there is no assignment of sets of terms to A, B, and C we can choose that satisfies the rules of the grammar. $\square$

It is worth mentioning that, in the above case, we can't isolate any particular production rule which causes the problem. Each rule alone may be fine within the context of a slightly different grammar.

Readers may suspect that $\heartsuit b \notin C$ is not a permissible side condition, but since set literals, equals on sets, intersection, and emptyset may all be used in side conditions we may write $\{\heartsuit b\} \cup C = \emptyset$ instead. The guidance we will give on side conditions (to structure them such that sets are built up in a monotonic fashion) would rule out the above case, as we will demonstrate in Example 6.5 once we have provided our construction.

Case 2: Infinitary Operators

Consider again the following from Llana Díaz & Núñez (1997):

> “ $$P ::= \cdots \mid \bigsqcap_{i \in I} P_i \mid \cdots$$ ”

This suggests it is possible to have infinitary operators whose arguments are taken from some subset of a language whose members can be given indexes by some index set I. In most instances where sets and metavariables appear in an MBNF grammar, we allow them to be generated by MBNF production rules. As mentioned in Case 1, MBNF, like BNF, allows us to have production rules that reference themselves, either directly or indirectly.

We use $\mathcal{P}X$ to denote the powerset of X. The following grammar gives a non strictly positive inductive type.

Lemma 4.30 (Not Every Use of MBNF Which Features Infinitary Operators Has a Solution)**.** *Unrestricted application of infinitary operators can provide contraints which have no solution.*

Proof. Consider the following MBNF grammar:

$$\begin{array}{rcl} p \in P & ::= & \bullet \mid \bigsqcap_{j \in I} p_j \\ i \in I & ::= & \circ \mid p \end{array}$$

Since $P \subseteq I$ we have $|I| \geq |P|$. Since $\bigsqcap_{j \in I} p_j$ denotes an infinitary operator $\sqcap$ whose arguments are taken from those subsets of P which can be given indexes in I, $|\bigsqcap_{j \in I} p_j| = |\mathcal{P}P|$. Since $\bigsqcap_{j \in I} p_j \subseteq P$ we have $|P| \geq |\mathcal{P}P|$. This is impossible as a set cannot be the same cardinality as its powerset in any standard set theory. $\square$

Again, in the above case, we can't isolate any particular production rule which causes the problem. Each rule alone may be fine within the context of a slightly different grammar. This prevents us from saying one problematic term doesn't contribute anything to the grammar.

4.3.2 An Incompleteness Result for MBNF

We consider some properties it might be desirable for a definition of MBNF to have. Ideally, MBNF would exist within some foundation already well known to people who do not have a deep familiarity with the theoretical computer science literature, as it is often an entry point to said literature. Ideally, in most cases where authors write some MBNF grammar that they believe to define some languages, a definition of MBNF would allow us to derive what those languages are. We have no reason to believe a definition fulfilling these properties cannot be given.

Ideally we would like for a definition of MBNF to be able to tell us in every instance whether a set of MBNF production rules is satisfied by some assignation of languages to metavariables, or whether it is not. We show that this cannot be done, (i.e., that any definition of MBNF is either inconsistent or incomplete). The general form of our incompleteness result is as follows:

Lemma 4.31 (An Incompleteness Result for MBNF)**.** *Given a definition, Def, of MBNF, either 1) Def is incomplete (i.e., there exists a grammar which has a solution, but which Def cannot prove has a solution), or 2) Def is inconsistent (i.e., there exists a grammar such that, given a proof that it has a solution, one can construct a proof that it does not and vice versa).*

Proof. Suppose that F is some foundation we use to model MBNF and G is a Gödel sentence in F Consider $U(y)$, with open variable y. $U(y)$ is defined as "For all x, x does not code for a sequence of numbers that constitutes a proof of the diagonalization of the well formed formula coded for by y." The Gödel sentence is the diagonalization of $U(y)$. Meaning, the Gödel sentence is "For all x, x does not code for a sequence of numbers that constitutes a proof of the diagonalization of $U(y)$." Now consider the following MBNF grammar:

$$\begin{array}{rcll} x \in X & ::= & \circ & \text{where } G \text{ is defined} \\ & | & \text{undefined} & \text{otherwise} \end{array}$$

The constraint undefined here stands in as a shorthand for some set of rules which lacks a solution. If we can prove that the above grammar has a solution then we can derive a proof that it does not and if we cannot prove that the above grammar has a solution we can show that letting the language of X equal $\{\circ\}$ ought to satisfy the above constraints. □

It may not be immediately obvious to the reader that the grammar below (seen in the proof of lemma 4.31) is really an allowable MBNF grammar.

$$\begin{array}{rcll} X & ::= & \circ & \text{where } G \text{ is defined} \\ & \mid & \text{undefined} & \text{otherwise} \end{array}$$

The reader may question whether we are allowed to include production rules that state that a grammar is undefined. They might also question whether our foundation can necessarily be given a Gödel encoding, or that "G is defined" is a side condition MBNF allows readers to write. We offer a few remarks and a more complex example to convince the reader that a grammar similar to this one can be written with established features of MBNF.

Remark 4.32. There exist ways of writing sets of MBNF production rules and formal proofs involving them as a string of symbols (for example, the latex we might use to write the math text). We consider it fairly safe to assume that a proof involving sets of MBNF production rules and their mathematical properties may be written using some syntax which can be given a Gödel numbering. □

Remark 4.33. MBNF production rules may be given side conditions that depend upon arithmetic computations.

For example Inoue & Taha (2012) have side conditions in the production rules of their MBNF grammar which depend on the truth of various arithmetic statements and Tobisawa (2015) includes side conditions on equivalence relations and reduction rules, which depend on the truth of arithmetic statements.

It is therefore reasonable to assume that, at the very least, *Def* is able to tell us when an arithmetic statement holds within a side condition. □

Remark 4.34. Natural numbers are allowed to appear in MBNF. Tobisawa (2015) uses natural numbers inside MBNF production rules which also appear in the resulting grammar. $\square$

Example 4.35. Let us consider a Gödel encoding of various statements in arithmetic which are either provable or not in *Def*, where *Def* is a definition of MBNF. Let $G(n, s)$ denote "s is provable in *Def* and n is a Gödel encoding of s". For an arithmetic statement s we can alter the MBNF grammar from lemma 4.29 to create a grammar that is well-defined (i.e., there are set and metavariable assignations which satisfy its constraints) if and only if s is provable in *Def*.

$$\begin{array}{rcll}
n \in \mathbb{N} & & & \\
a^s \in A^s & ::= & \circ \mid \heartsuit a^s \mid d^s & \\
b^s \in B^s & ::= & \circ \mid a^s \mid \spadesuit b^s & \\
c^s \in C^s & ::= & a^s \mid b^s & \\
d^s \in D^s & ::= & b^s & \text{where } \heartsuit b^s \notin C^s \text{ and } \forall n \in \mathbb{N}, \neg G(n, s) \\
 & \mid & n & \text{where } G(n, s)
\end{array}$$

Consider the case where s is not provable in *Def*. Then there is not a natural number n corresponding to a Gödel encoding of s such that $G(n, s)$. Therefore, D^s is the set of all b^s such that $\heartsuit b^s \notin C^s$ holds. In this case A^s, B^s, and C^s are syntactically equivalent to A, B, and C from the MBNF in lemma 4.29. Therefore they do not have a solution.

Consider the case where s is provable in *Def*. Then D consists of a single natural number. If we take x to be a terminal symbol, then we can read this in more or less the same way as we read a piece of ordinary BNF. When s is provable in *Def*, the only differences between a set of BNF production rules and the above MBNF grammar are that, rather than non-terminals, we have unbracketed and decorated metavariables and the set produced is of mixed sequences of numbers and characters, rather than of strings. These are fairly trivial semantic differences, so we can be confident that, if s is provable in *Def*, the above set exists.

Therefore, the above MBNF has a solution, if and only if s is provable in *Def*.

Suppose *Def* can prove all such rule sets have a solution. Then "s is provable" is equivalent to A^s has a solution. Take, s to be the Gödel sentence in the encoding defined by *Def*. Then, if A^s provably has a solution, then we can prove that it does not have a solution and vice versa. So, either there exists a grammar which is satisfied by some choice of languages for each of the metavariables in each rule of the grammar and that *Def* cannot prove is satisfied in this way, or *Def* is inconsistent. $\square$

4.4 Conclusions for this Chapter

BNF and the other variants dealt with in chapter 3 either deal with a concept of language as a set of strings (which is usually context-free), or else the parse trees derived from a set of strings. SubSection 4.1.1 shows that MBNF is written using math text, which has a richer structure than a string. Subsections 4.1.7 and 4.1.13 show that pieces of math text may have complex equivalences over them, which are nonetheless considered part of the MBNF "syntax." Subsections 4.1.9, 4.1.10, 4.1.11 and 4.1.12 show that MBNF may have infinitely large structures inside its syntax, which cannot possibly be represented by strings. MBNF may include operations that go beyond the scope of the context-free languages (proofs of this appear in Section 4.1.4 and 4.1.6, but many of the other features discussed put BNF far beyond the scope of the context-free languages, simply because they cannot be represented in terms of sets of strings). MBNF may include operations which are not just operations on strings, such as operations on sets (subsections 4.1.5, 4.1.7 and 4.1.8) and arithmetic operations (Subsection 4.1.6). Sets, natural numbers which have not been given an encoding as strings, and real numbers may also appear inside MBNF syntax (subsections 4.1.6, 4.1.9, 4.1.10 and 4.1.13).

Chapter 5

Introducing MathSyn

In this chapter we define some of the underlying machinery of the model for understanding MBNF which we call MathSyn. The material in this chapter doesn't include the actual MBNF production rules used in picking out syntactic objects. This is described in chapter 6. Instead, it offers a basic conception of what a syntactic object is and what sort of operations may be performed on one, which is applicable even to grammars produced by those parts of what Steele (2017) calls computer science metanotation (CSM) which are sometimes used outside of MBNF production rules.

Before we give examples of how MathSyn is used to define MBNF rule sets, we sketch some of the machinery it uses. The contributions given in this section are an overview of some tools MathSyn offers for discussing syntactic objects picked out by MBNF grammars and a proof that objects in MathSyn have a model. Syntactic objects and arrangements in MathSyn are an abstraction of math layout allowing for nested equivalences.

While we acknowledge that, in MBNF, constraints providing relations can be given using any part of mathematics, some of the tools for doing so which are more popular in CSM are given special treatment by MathSyn.

MathSyn provides special facilities for relations defined using constraints of the form:

$$\circ_1 \approx \circ_n$$

which are briefly discussed in Subsection 5.1.3 under the heading "Syntactic Equivalences in MathSyn." Such a constraint says something like, for any pair (A_1, A_2) such that A_1 satisfies $\circ_1$ and A_2 satisfies $\circ_2$, A_1 and A_2 are syntactically equivalent for any context in which they appear.

Design Decision 5.1. We include holes and hole filling in MathSyn for several reasons. Hole filling operations are popular in computer science literature discussing syntax. They are used extensively in CSM and far less outside it, so it makes sense that we make an effort to define them, rather than defer to mathematics for their meaning. Hole filling operations can appear inside MBNF production rules (e.g., Chang & Felleisen (2012) write $A = [\,] \mid A[\lambda x.A]\,e$). We aim to explain what it means when they do. Hole filling is already used alongside MBNF when providing contexts for syntax evaluation. We want to explain this use. MathSyn makes use of this feature in defining the compatible closure of a relation with respect to some syntactic context. This notion of compatible-closure allows us to define production rules and equivalences as operating "inside" other syntactic contexts (allowing relations to descend through the syntax tree to the point where they apply). The presence of a hole effects our notions of syntactic equivalence. E.g., bound variables above a hole cannot be renamed until the hole is filled. A concept of hole filling is useful for a rigorous treatment of substitution and binding. Hole filling allows us to write n-ary functions which build objects from other objects which do not abstract away the arrangements and symbols used. E.g., if we wanted to write $(\mathsf{x} \to \mathsf{y}) \to \mathsf{z}$ as a series of functions on syntax with a tree-like ordering, terminating with symbols we could write $(\Box \to \Box)[(\Box \to \Box)[\mathsf{x}, \mathsf{y}], \mathsf{z}]$. These functions provide a tool to disambiguate syntax boundaries, which is used in the notion of primitive constructor decomposition we introduce in Subsection 5.1.6. Since hole filling on objects coerces them into an equivalence class (see definition 5.25.3) this means we can represent an object with any such construction appearing in its equivalence class (e.g., $\lambda x.x\mathsf{y}$ may alternately be represented $(\lambda\Box.\Box)[\,\mathsf{x}, (\Box\,\Box)[\mathsf{x}, \mathsf{y}]\,]$, $(\lambda\Box.\Box)[\,\mathsf{z}, (\Box\,\Box)[\mathsf{z}, \mathsf{y}]\,]$, etc.).

We cover context hole filling in Subsection 5.1.4 and make use of it throughout this chapter.

MathSyn also provides special facilities for relations defined using constraints of the form:

> Let A bind the object placed in the nth of its holes in the nth, ith, jth, kth, ...,etc. of its holes.
> Let $\bullet \in S$ be identified up to α-equivalence.

which are briefly discussed in Subsection 5.1.7 under the heading "α-Equivalence, Names and Binding." Constraints of this form also apply to the relation $\approx$. Such a constraint says something like, for any pair (A_1, A_2) such that A_2 is "α-equivalent" to A_1 by the constraint given above, A_1 and A_2 are syntactically equivalent for any context in which they appear.

Additionally MathSyn provides special facilities for relations defined using constraints of the form:

> Our rewriting rules for S are:
> $$\circ_1 \overset{*}{\rightarrow} \circ_2$$

which are briefly discussed in Subsection 5.1.5 under the heading "Contexts and Compatible Closure." Such a constraint says something like, for all $\bullet_1, \bullet_2 \in S$, $\bullet_1 \overset{*}{\rightarrow} \bullet_2$ if and only if there exists a pair (A_1, A_2) and a context C, such that $\bullet_1$ consists of A_1 appearing in the context C, $\bullet_2$ consists of A_2 appearing in the context C, A_1 satisfies $\circ_1$ and A_2 satisfies $\circ_2$.

While we acknowledge that MBNF may feature operations from any part of mathematics, some of the tools for doing so which are more popular in CSM are given special treatment by MathSyn. MathSyn covers operations which may be used to position math text. This is briefly discussed in Subsection 5.1.1 under the heading "Objects, Arrangements, Symbols and Pointers" and one way of representing these operations is given in the model in Section 5.2. MathSyn covers what it means when an MBNF grammar uses hole filling operations on syntax. This is briefly discussed in Subsection 5.1.4 under the heading "Hole Filling." MathSyn covers what it means when an MBNF grammar uses Capture Avoiding Substitution. This is briefly discussed in Subsection 5.1.8 under the heading "Substitution, Sub-Object, Sub-Arrangement."

MathSyn provides machinery for discussing syntax which remains largely hidden when MathSyn is used to interpret MBNF, but which authors may find helpful, if they wish to make parts of their discussion more explicit. We will mention where notation is optional, convention enables authors to abbreviate what they write, or machinery is hidden.

Most of the lemmas and proofs in this section refer to the inductive case only. However, the construction we give for syntax doesn't preclude putting similar treatments for the co-inductive case on top of what we have now.

5.1 What Tools Does MathSyn Offer?

5.1.1 Objects, Arrangements, Symbols and Pointers

MathSyn consists of syntactic *objects* belonging to the set Object, these contain syntactic *arrangements* which belong to the set $\mathsf{Arrangement}$. The set Symbol of *symbols* may be thought of as the "glue" that holds objects together. Each symbol is a mark that can be made on the page, which is not used to represent a syntactic equivalence, a syntactic boundary, a pointer to a piece of syntax or a hole and which can be arranged alongside a collection of symbols in any way one can arrange text in mathematics (or wrapped round it as may be the case with overbarring). The set $\mathsf{Arrangement}$ may be thought of as corresponding to pieces of syntax which are joined together with symbols in any way permitted by mathematical text (e.g. by concatenation, subscripting, superscripting, overbarring, underbarring etc.). One can think of an arrangement as a tree with internal nodes for sequencing, subscripting, etc. The underlying set-theoretic representation will need to capture this tree-like structure. Sometimes symbols are used like non terminals, but often they appear as parts of arrangements higher up the syntax "tree." The set Object may be thought of as corresponding to the set of equivalences over arrangements of syntax (where these equivalences can also be applied to syntax boundaries where it is relevant to do so). This quotienting by equivalences, which may occur at any syntax boundary (analogous to a node) in the tree-like structure of an arrangement is why we refer to arrangements as tree-like rather than as trees.

There are many challenges we face when constructing objects and arrangements.

- Syntax is not always represented as strings of symbols concatenated together. Ideally we would want a model that covered every way in which an author might arrange text.

- The context hole and the empty arrangement are, in some sense, not concrete syntax, but their inclusion in sets of syntax still renders them non-empty. Ideally our model would give them special treatment. Hole filling does not work like substitution or rewriting (both of which take somewhat arbitrary syntax and neither of which has an inherent order). We would like to distinguish the context hole from other symbols. Empty sequences may appear in arrangements at any point and their addition does nothing (i.e. even when regarding syntax as literal we still need to deal with syntax building operations that are idempotent (which may include more than just concatenation) we also need a way to represent an empty sequence that sets it apart from symbols).

- Objects should be identical to equivalences over syntactic arrangements. It is common practice to write two pieces of syntax next to one another with an =, which we would like to understand as equality, even though how they are written differs. Sometimes demonstrating properties, (e.g., confluence) requires working modulo some equivalence, yet these are presented as properties of the syntax itself.

- MBNF users do not always tell us how their equivalences should be serialised. Usually equivalences are given with math. Our model should allow most syntactic equivalences to be treated as an object.

- Objects and arrangements should resemble trees and incorporate syntax boundaries. Syntax is often treated as having boundaries corresponding to a tree like structure. Equivalences like associativity are difficult to express without a tree like ordering of operations. Manipulation of syntax (e.g., capture avoiding substitution, hole filling) can often be thought of in terms of operations on trees. Language and concepts associated with parse trees may be employed alongside MBNF grammars for which no parser exists.

- Objects should be nested so equivalences can appear at each node of a syntax tree. Some terms in MBNF grammars only feature equivalences on sub-trees (e.g. α-equivalence in terms with a hole). The treatment of syntax as trees and the treatment of the same syntax as equivalences can occur in the same paper without any switch being acknowledged, much less problematised. An MBNF grammar may perform an operation on a tree and expect it to be simultaneously coerced into an equivalence.

- Our model should use set theory. MBNF grammars are discussed and interleaved with set theory. Some use sets inside syntax.

- Ideally, there should be a set of all objects. MBNF features constraints that evaluate over the universe of syntax. MBNF grammars use mathematical and syntactic functions to build sets in a monotonic increasing way and expect the universe of syntax to give an upper bound for a fixed point. Some are defined co-inductively over all syntax.

- Objects should be allowed to feature themselves. Some MBNF grammars feature syntax that is idempotent (e.g., the 0 element of the π-calculus). Some include regular trees in their syntax.

Naively, we might try to meet these requirements as follows. Objects and arrangements are encoded as sets. There is a set of all objects and a set of all arrangements. Each object is a subset of $\mathsf{Arrangement}$, representing an equivalence. Each arrangement to takes a number of objects and concrete symbols and arranges them in any way we might arrange math text. There can be cycles in which objects contain themselves. We run into various problems with this. Here, the set of objects is the powerset of the set of arrangements, yet we also expect objects to be contained within arrangements. This is not possible if we want Object and $\mathsf{Arrangement}$ to be sets within a standard set theory. We want to allow objects to have arrangements as elements which may also have those objects inside them. While there are some set theories that allow this, they tend to be unfamiliar to the average reader and restrictive in ways that prove awkward. They also sacrifice set comprehension, which is a tool MBNF grammars use.

Design Decision 5.2. We fix this by limiting objects to countable subsets of $\mathsf{Arrangement}$, since even authors who do not show their syntactic equivalences are serialisable do not usually require that each equivalence feature uncountably many pieces of syntax. This commits us to ZFC, or similar, because we require cardinal arithmetic to show that a set can be equinumerous with countable equivalences over itself. This does mean that Object and $\mathsf{Arrangement}$ are uncountable, however this is ultimately less problematic than restricting allowable equivalences further. Having done this, the least complicated way of dealing with objects containing themselves is to have them contain pointers to themselves.

Definition 5.3 (Symbol)**.** The set Symbol of syntactic symbols (or simply symbols) is a non-empty countable set of syntactic symbols (arbitrary marks on the page) such that: Symbol does not contain $\square, \varepsilon, [,], (,), \{, \}$, and $\approx$.[1] Symbol does not contain syntax consisting of symbols positioned around one another using the positioning operations of math text (e.g. for $A, A' \in \mathsf{Symbol}$, none of the following are elements of Symbol: AA', $A^{A'}$, $A_{A'}$, and neither is any way of wrapping symbols around A like $\overline{A}$). $\square$

Definition 5.4 (Objects, Arrangements, Pointers and Arity)**.** Object is a set of syntactic *objects*. $\mathsf{Arrangement}$ is a set of syntactic *arrangements*. Let O range over Object and let A range over $\mathsf{Arrangement}$. The set $\mathsf{Pointer}$ contains pointers to objects. We use the notation P_O for the pointer that indicates O. Our model requires we choose some

[1] This is purely so as not to cause any confusion with our use of $\square, \varepsilon, [,], (,), \{, \}$, and $\approx$ in the meta-level notation of this paper. There is nothing especially conceptually wrong with these marks on the page being represented somehow in Symbol, but, in this paper, placing them inside Symbol instead of reading them as meta-level syntax would require we mark them in some way. If an author wants to include these as part of Symbol, this should be fine so long as they do not also want to use them like we do here and any reference to our use of them in this paper can be clearly distinguished from the author's object-level syntax.

bijection $\mathsf{ptr} \in \mathsf{Pointer} \to \mathsf{Object}$,[2] P_O is then defined in such a way that, for $O \in \mathsf{Object}$ there exists a unique $P_O \in \mathsf{Pointer}$ such that $\mathsf{ptr}(P_O) = O$. We do this to avoid any ambiguity in context hole filling which is defined in Subsection 5.1.4.

We define the sets Object and Arrangement as the smallest sets satisfying the following conditions.

1. The *empty arrangement* ϵ is in Arrangement.
2. The core items of arrangements are symbols, pointers to objects, natural numbers, and underlined or overlined arrangements. For any symbol x, pointer P_O, number $n \in \mathbb{N}$, and non-empty arrangement $A \neq \epsilon$, all of the following are in Arrangement: x, P_O, n, $\underline{A}$, and $\overline{A}$. Furthermore, these are all *core arrangements*, which are ranged over by the metavariable $\hat{A}$.
3. Left-to-right sequencing allows appending additional core arrangements to a non-empty arrangement. For any arrangement $A \neq \epsilon$ and core arrangement $\hat{A}$, it holds that $A\hat{A}$ is in Arrangement.
4. Superscripting, subscripting, pre-subscripting and pre-superscripting, etc. are supported. For non-empty arrangements A, A_1 and A_2, all of the following are in Arrangement: A^{A_1}, A_{A_2}, $A^{A_1}_{A_2}$, ${}^{A_1}A$ etc.
5. If $S \subseteq \mathsf{Arrangement}$ does not contain any arrangements consisting of a single pointer to an object with no other syntax surrounding it, S is non-empty, and $|S| \leq \aleph_0$, then $S \in \mathsf{Object}$. The cardinality constraint here enables us to find a fixed point satisfying all of these constraints. If we did not have the cardinality constraint here there would be no model for this definition in ZFC.
6. The syntax $\square$ is in Object. It indicates a *hole* to place an object in.
7. The number of $\square$ in an object or an arrangement is its *arity*. I.e., let $\mathsf{Ari}(A)$ denote the arity of A, then:

[2]Technically we only need a 1 to 1 function as all we are interested in is that an inverse can be found for every mapping, but a bijection makes it so that every arrangement containing a pointer has a corresponding object, which makes the definition more convenient.

(a) Symbols and natural numbers have arity 0.

(b) $\square$ has arity 1

(c) If $A \in O$, then O has the same arity as A. If there are multiple A of different arity, then O has arity 1.[3]

(d) P_O has the same arity as O

(e) $\underline{A}$, and $\overline{A}$ have the same arity as A

(f) $A\hat{A}$ has an arity of $\mathsf{Ari}(A) + \mathsf{Ari}(\hat{A})$

(g) $\mathsf{Ari}(A^{A_1}) = \mathsf{Ari}(A) + \mathsf{Ari}(A_1), \qquad \mathsf{Ari}(A_{A_2}) = \mathsf{Ari}(A) + \mathsf{Ari}(A_2),$
$\mathsf{Ari}(A^{A_1}_{A_2}) = \mathsf{Ari}(A) + \mathsf{Ari}(A_1) + \mathsf{Ari}(A_2)$
...

$\square$

Design Decision 5.5. We use pointers to objects inside arrangements rather than the objects themselves, since we want to allow objects to be nested within themselves, provided some syntax is added. As we are using sets to represent arrangements and sets to represent objects, without the use of pointers it would be the case that, reading the above definition inductively, at each step and each limit point, the set of arrangements made with all countable equivalences over prior arrangements would yield new arrangements. As such, without pointers, $\mathsf{Arrangement}$ could not be constructed as a set as we would have a chain of sets of arrangements which is not bounded above and so no limit point may be found.

Lemma 5.6. *Pointers enable us to include all equivalences over regular trees in* Object.

Proof. For every finite number of pointers there is a countable number of arrangements containing them. Each arrangement can be thought of as having branches wherever there is a pointer to an object below (which may include any arrangement), as such each

[3]We need not concern ourselves with objects containing arrangements of different arity, we include them here to make sure our definition is complete, but later definitions will exclude them from $\approx$-well-formed objects.

regular tree is modelled. The set of regular trees is countable, as such each equivalence over it is countable. Each countable equivalence over arrangements is an object. As such every equivalence over regular trees has a model in Object. $\square$

Arrangements do not replace existing standards, such as OpenMathISO (2015), MathML Presentation Ion et al. (2001), Latex Knuth (1986), etc. They are an abstraction of math layout allowing for nested equivalences.

Design Decision 5.7. There are a few reasons for building equivalence classes into arrangements rather than making them identical to literal syntax. We want to allow object-to-object operations in production rules. When we define equivalences inductively over arrangements we want some of that structure to be represented by our model. As demonstrated in Subsection 4.1.13 some features of MBNF can also be combined to give a rule set which necessitates that nested equivalences are calculated as the rule set is solved.

Design Decision 5.8. Object, Arrangement and Pointer are uncountable, because we want syntactic objects to represent all the countable equivalences over syntax. It did not make sense to give further restrictions as:

1. Authors define syntactic equivalences with mathematical language and only sometimes provide a representation of these equivalences as strings (or similar finite data structures). If this is given at all, it is usually in a separate computer implementation and much of the detail is not included in the paper with the MBNF grammar.

2. Where authors do provide tools for serialising equivalences, these frequently require significant rewriting of the syntax (e.g. writing α-equivalence with De-Bruijn indices often requires changes to the syntax as written in the MBNF). The exact method used to deal with an equivalence is often the messiest part of the implementation of an MBNF rule set. Authors usually will not explain how they perform the translation from a syntactic equivalence given by an MBNF rule set in their paper to the version used in their computer representation.

3. For almost every kind of equivalence authors use a different method. The approaches which are used for α-equivalence are not used for associative operators, different approaches again may be used for operators that are commutative or idempotent. Sometimes a canonical representation may be given, other times it may be necessary to perform a calculation to check two pieces of syntax are structurally congruent (i.e. equivalent). Providing documentation for each and every one of these tactics and when they can be effectively employed could easily be several years of work. Before this work can take place, though, we must first provide a thorough account of the objects given by an MBNF rule set, which are then translated to a computer implementation. An account of how these objects may be derived is part of what MathSyn provides. Definition 5.4 covers how countable equivalences are worked into the structure of syntactic objects. Subection 5.1.3 deals with what it means to work modulo an equivalence.

4. MBNF syntax can include uncountably large sets anyway. It seems pointless to limit ourselves to countable representations of syntax when MBNF does not. If we did so, we would still need to account for how our ideas may be extended to deal with MBNF rule sets whose syntax does not have a countable representation. This means that, regardless of whether our set of syntactic objects can be serialised, our methodology for reading production rules would still need to work on sets that are not serialised (so we couldn't provide an account which tells us whether an MBNF rule set has a solution and which mathematical entities could satisfy it when it does, if this account also relies on every piece of syntax within that solution being computable).

5. We do not need to be able to serialise Object to prove an MBNF grammar has a solution. Our focus is to provide readers with tools to check the consistency of MBNF rule sets as they are given in the papers they appear in and to give them some way of finding mathematical entities which can be thought to satisfy these rule sets under an abstract encoding of "syntax," which we provide in Definition 5.4 and cover in more depth in Section 5.2. It is not to use an MBNF rule set to independently reproduce an implementation which may already be given

separately to that rule set and which usually looks quite different to it. For example, in this paper it is not necessary to build a parser for all of Object and this is not necessary to understand how to read MBNF grammars. If authors require a parser for their MBNF grammar we expect it to exist in some form, the issue we aim to deal with here usually being the difference between implementation and what is given on the page for the reader.

A Pair of Examples to Illustrate the Use of Symbols, Objects and Arrangements, Etc.

Example 5.9. Consider, for example, the object $\lambda x.x$, generated by the following:

$$e \in \mathsf{exp} ::= v \mid \lambda v.e \mid e\, e$$

In MBNF and, likewise, in MathSyn, 'λ' and '.' would normally be symbols (and they are in this case). Each x likely represents a pointer to an object, in this case the same object consisting of a literal symbol (which need not be 'x').

Example 5.10.

• Symbol could include a, C, λ, $\mathsf{\Gamma}$, $\langle$, $\rangle$, etc. In this case, none of these symbols can be used unambiguously in the place of objects or pointers.

• Pointers include $P_\Box$, $P_{\{P_\Box P_\Box\}}$, $P_{\{P_{\{P_\Box P_\Box\}} P_{\{P_\Box P_\Box\}}\}}$.

• Core arrangements could include all above symbols and pointers in addition to 1, 2, 3, $\overline{\mathsf{a}}$, $\overline{\mathsf{C}}$, $\overline{\lambda}$, $\overline{\mathsf{\Gamma}}$, $\overline{1}$, $\overline{2}$, $\overline{P_\Box}$, $\overline{P_{\{P_\Box P_\Box\}}}$, $\overline{P_{\{P_{\{P_\Box P_\Box\}} P_{\{P_\Box P_\Box\}}\}}}$.

• Arrangements could include all above core arrangements in addition to ways of mixing them like $\overline{P_\Box}^{\overline{P_\Box}}$, $\overline{P_\Box}_{\overline{P_\Box}}$, $\overline{P_\Box}^{\overline{P_\Box}}_{\overline{P_\Box}}$.

• Objects include $\Box$, $\{\overline{P_\Box}^{\overline{P_\Box}}\}$, $\{P_\Box\}$, $\{\mathsf{a}, \mathsf{aa}, \mathsf{aaa}, ...\}$ and $\{\overline{P_\Box}_{\overline{P_\Box}}, \overline{P_\Box}^{\overline{P_\Box}}_{\overline{P_\Box}}\}$.[4] $\Box$

[4]Although, when we come to definition 5.18, the last of these objects is not well-formed

5.1.2 Inclusion of Mathematical Entities Outside of Syntactic Objects

Design Decision 5.11. Pointers also enable us to extend the set Object with a set of mathematical entities which are disjoint from Object, while still allowing these to appear inside arrangements. Suppose we had a set of mathematical entities (i.e. entities which may be defined using any part of math, provided they can be placed in a set), M, such that M is disjoint from Object the cardinality of M is no larger than $\aleph_1$ (i.e., we require M to be no larger than $\mathsf{Pointer}$) and no element of M was also an element of $\mathsf{Pointer}$. We use Object_M to denote a set which has the same mathematical properties as Object, but where arrangements may also include pointers to mathematical entities taken from M.

The definition below enables the inclusion of math directly in the syntax, which is one of the features of MBNF and related grammars. This accounts for grammars where authors mix mathematical definitions of entities with more standard syntactic definitions of grammars.

Definition 5.12 (Object_M, ptr_M)**.** Object_M and ptr_M are defined as follows:

- $\mathsf{ptr}_M : \mathsf{Pointer} \to \mathsf{Object}_M$ is a bijective function from $\mathsf{Pointer}$ to Object_M.
- If $S \subseteq \mathsf{Arrangement}$ does not contain any arrangements consisting of a single pointer to an object, S is non-empty, and $|S| \leq \aleph_0$, then $S \in \mathsf{Object}_M$.[5]
- $\square$ is in Object_M it indicates a *hole* to place an object in.
- $M \subseteq \mathsf{Object}_M$

$\square$

Example 5.13. We could extend the set of objects Object with the real interval $[0, 1)$ as follows: Let $\mathsf{Object}_{[0,1)} = \mathsf{Object} \cup [0, 1)$. Let $\mathsf{ptr}_{[0,1)} : \mathsf{Pointer} \to \mathsf{Object}_{[0,1)}$ be a

[5]We continue to use arrangements from the set $\mathsf{Arrangement}$ here, rather than supplying some new $\mathsf{Arrangement}_M$, as the construction of Object_M does not affect the construction of $\mathsf{Arrangement}$. The only difference is in the function ptr_M (which is a bijection from Object_M instead of Object) and in Object_M itself. Likewise we do not need to construct a $\mathsf{Pointer}_M$ as the only constraint on $\mathsf{Pointer}$ is its cardinality, which remains the same.

bijective function from $\mathsf{Pointer}$ to $\mathsf{Object}_{[0,1)}$. Define $\mathsf{Arrangement}$ as normal.

In section 5.2. we will prove that Object, $\mathsf{Arrangement}$ and $\mathsf{Pointer}$ have a model. For now, we will assume that this is true and demonstrate that, given this assumption we can prove ptr_M and Object_M can also be given a model.

Theorem 5.14. *If* Object*,* $\mathsf{Arrangement}$*,* M *and* $\mathsf{Pointer}$ *have a model in ZFC, then so do* ptr_M *and* Object_M*.*

Proof. For all $S \subseteq \mathsf{Arrangement}$ such that S does not contain any arrangements consisting of a single pointer to an object, S is non-empty, and $|S| \leq \aleph_0$, $S \in \mathsf{Object}$. We also have that $\Box \in \mathsf{Object}$ and nothing else is a member of Object. As such, $\mathsf{Object}_M = \mathsf{Object} \cup M$ which is well defined if Object and M are. If Object, $\mathsf{Pointer}$ and M are of cardinality $\aleph_1$, then there exists a bijection ptr_M between $\mathsf{Pointer}$ and $\mathsf{Object} \cup M$. Consider a bijection $b : \mathsf{Pointer} \to \kappa$ from $\mathsf{Pointer}$ to limit ordinal κ. Let $c : \kappa \to \kappa$ map each ordinal of the form $\alpha + n$ where α is a limit ordinal and n is a natural number to $\alpha + n \cdot 2$. Let $\mathsf{ptr}_M(O) = b^{-1}(c(b(\mathsf{ptr}(O))))$. Let $\mathsf{ptr}_M(M)$ be a mapping to the largest subset of $\mathsf{Pointer}$ whose elements are not of the form $\mathsf{ptr}_M(O)$ (which is of the same cardinality as $\mathsf{Pointer}$). This is true of M and happens to be true of Object and $\mathsf{Pointer}$ as well as is shown to be the case in Section 5.2. $\Box$

In this way we can readily support the inclusion of more arbitrarily defined mathematical objects in MBNF grammars, provided these objects can be placed in a set and we have a proof it is the correct cardinality. These objects can safely include syntactic objects, consisting of sets of arrangements of symbols and pointers in their construction, even if these syntactic objects reference members of M (this is because the sets Object_M and $\mathsf{Object} \cup M$ are identical, essentially the only difference is between ptr and ptr_M).

5.1.3 Syntactic Equivalences in MathSyn

In MBNF it is necessary to consider syntax up to various syntactic equivalences simultaneously to constructing the grammar. One might associate these equivalences with the idea of quotient inductive types, but as our model is in ZFC and not homotopy type theory, we would also include the law of excluded middle and the axiom of choice in our reasoning. In order to support α-conversion and operators that are associative, commutative, idempotent, etc., objects are defined in effect to work modulo an equivalence relation $\approx$ on arrangements, which is defined as part of the MBNF rule set. In MathSyn, if A is an arrangement, then $[A]_{\approx}$ can optionally[6] be used to denote the object whose elements are every arrangement which is syntactically equivalent to A by some equivalence $\approx$ on the set $\mathsf{Arrangement}$. If $\approx$ is not defined on an arrangement, A, then by convention $A \approx A$ and $\forall A' \in \mathsf{Arrangement}$ if $A' \neq A$, then A' is not equivalent to A by $\approx$.

Definition 5.15 ($\mathsf{Arrangement}$-Equivalence $\approx$)**.** An $\mathsf{Arrangement}$-Equivalence $\approx$ can be any equivalence relation on $\mathsf{Arrangement}$ such that:

- If $A \approx A'$ then either both A and A' are arrangements of arity 0 or $A = A'$.[7]
- For every arrangement A, $|[A]_{\approx}| \leq \aleph_0$ where $[A]_{\approx}$ is the $\approx$-class of A.[8]

□

The following lemma tells us when $[A]_{\approx}$ is an object.

Lemma 5.16. *Let $\approx$ be an* $\mathsf{Arrangement}$*-Equivalence and let $A \in \mathsf{Arrangement}$.*

1. *$[A]_{\approx} \in \mathsf{Object}$.*
2. *If A is of arity ≥ 1, then, $[A]_{\approx}$ is an object iff $[A]_{\approx} = \{A\}$.*

Proof. 1. By Definition 5.15, $|[A]_{\approx}| \leq \aleph_0$. Hence, $[A]_{\approx} \in \mathsf{Object}$.
2. By definition of $[\cdot]_{\approx}$ and of objects (see Definitions 5.15 and 5.4). □

We build equivalence classes into arrangements and define the well- and ill-formed arrangements and objects. Objects that are well-formed can be considered to represent

[6]We say "optionally" here because there are syntactic conventions that allow it not to be written, but we include it for disambiguation.

[7]We do this to avoid any ambiguity in context hole filling. Arrangements above a hole cannot form an equivalence class where one would not have a single path to descend which has a leftmost hole.

[8]This is the same size constraint we have on objects so that each instance of $[A]_{\approx}$ is also a member of Object.

equivalence classes of $\approx$, whereas objects that are ill-formed cannot. An arrangement or an object is $\approx$-ill-formed iff it is not $\approx$-well-formed.

Design Decision 5.17. We use $\approx$ to distinguish between the syntactic equivalences an author may provide for an arrangement as opposed to $=$, which we reserve for the identity relation over $\mathsf{Arrangement}$ and Object. It is not unusual to coerce non-identical arrangements into the $\approx$-well formed objects that can be thought to represent their $\approx$-equivalence classes. As objects quotient arrangements into equivalence classes at syntactic boundaries, so are our notions of $\approx$-well-formedness built up inductively over these boundaries.

Definition 5.18 ($\approx$-well-/ill-formed, Preserves $\approx$-well-formedness)**.** For each choice of $\mathsf{Arrangement}$-Equivalence $\approx$ (which the author must indicate they require at a given point in a document), we define $\approx$-well-formed objects and arrangements simultaneously as follows:

- ϵ, and all elements of $\mathsf{Symbol} \cup \mathbb{N}$ are $\approx$-well-formed arrangements.
- If O is a well-formed object then P_O is a $\approx$-well-formed arrangement.
- If $A \neq \epsilon$ is a $\approx$-well-formed arrangement then $\overline{A}$ is a $\approx$-well-formed arrangement.
- For core arrangement $\hat{A}$ and non-empty arrangements A, A_1 and A_2, if $\hat{A}$, A, A_1 and A_2 are $\approx$-well-formed arrangements then all of $A\hat{A}$, A^{A_1}, A_{A_2}, $A^{A_1}_{A_2}$ are $\approx$-well-formed arrangements.
- $\square$ is a $\approx$-well-formed object.
- If O is an $\approx$-equivalence class, every element of O is a $\approx$-well-formed arrangement, and $O \cap \mathsf{Pointer} \neq O$, then O is a $\approx$-well-formed object.

• An $\mathsf{Arrangement}$-Equivalence $\approx$ preserves $\approx$-well-formedness iff (for all A, A_1 such that $A \approx A_1$ and A is $\approx$-well-formed, we also have A_1 is $\approx$-well-formed). $\square$

Design Decision 5.19. Whether or not an object/arrangement is $\approx$-well-formed depends on the value chosen for $\approx$ at a given point in the document. We will use this notion later to describe how an arrangement A may be coerced into an $\approx$-equivalence class of A at any point where it is necessary to do so and how objects can be re-defined up to equivalence following a substitution. There are various reasons why we have built equivalence classes into arrangements rather than making them identical to math-text. We want to eventually support mathematical language in syntax, with mathematical language containing objects not arrangements. We want to allow object-to-object operations in production rules. When we define equivalences inductively over arrangements we want some of that structure to be represented by our model. When we come to hole filling in the next section it will also become apparent that holes always need to be on their own in their equivalence class, but that larger equivalence classes are allowed to appear in an arrangement beside them. There is no way this can be accomplished without allowing equivalence classes to appear within arrangements.

The following lemma tells us that no well formed object has a pointer to an object of arity ≥ 1 in an equivalence class of more than one arrangement (i.e. there is a sense in which context holes always appear on their own within their equivalence classes).

Lemma 5.20. *Let $\approx$ be an* $\mathsf{Arrangement}$-*Equivalence. An Object O of arity ≥ 1 is $\approx$-well-formed iff one of the following holds:*

1. $O = \Box$.
2. $O = \{A\} = [A]_\approx$ *for some $\approx$-well-formed A of arity ≥ 1 such that $A \notin \mathsf{Pointer}$.*

Proof. $\Leftarrow$) is by definition of $\approx$-well-formedness (see Definition 5.18).
$\Rightarrow$) Assume $O \neq \Box$, $O \subseteq \mathsf{Arrangement}$, $O \neq \emptyset$ and $|O| \leq \aleph_0$. Since the arity of $O \geq 1$, let $A \in O$ such that the arity of $A \geq 1$. Since O is well-formed, then O is an equivalence class and for all $A' \in O$, $A' \approx A$. Since A has at least one $\Box$, by definition of $\approx$, for all $A' \in O$, $A' = A$. Hence $O = \{A\} = [A]_\approx$ where A is $\approx$-well-formed. Since $O \cap \mathsf{Pointer} \neq O$, $A \notin \mathsf{Pointer}$. $\Box$

Example 5.21. Depending on the constraints placed on $\approx$, $\approx$-well-formed objects might include $\Box$, $\{\mathsf{x}_i \mid i \in \mathbb{N}\}$, $\{\overline{P_\Box}^{\overline{P_\Box}}\}$, $\{P_{O_1} \mid P_{O_2}, P_{O_2} \mid P_{O_1}\}$ and $\{P_{O_1} \mid P_{O_2}, P_{O_3}\}$

where O_1, O_2 and O_3 are $\approx$-well-formed objects. Non $\approx$-well-formed objects include $\{\overline{P_\square}_{\overline{P_\square}}, \overline{P_\square}^{\overline{P_\square}}_{\overline{P_\square}}\}$ and $\{P_\square P_\square, \heartsuit P_\square\}$. $\square$

In most papers authors do not write $[A]_\approx$ to denote a syntactic object and MathSyn allows it to be left out. We include it only to disambiguate objects and arrangements in our discussion. An upright font may optionally be used to denote symbols and a slanted font to denote metavariables, (e.g. x for a metavariable and x for a symbol). Again not all authors do this and MathSyn allows it to be left out, but we include it for disambiguation. If O is an object, then $\mathsf{ptr}(O)$, or P_O may optionally be used to denote the pointer to O. Again not all authors do this and MathSyn allows it to be left out, but we include it for disambiguation.

Design Decision 5.22. All objects and sub-objects in an MBNF rule set are instances of $[A]_\approx$ for whatever value of $\approx$ is given at that stage in the paper. MathSyn interprets grammars dynamically throughout a document as new rules are read, so authors may work with one equivalence, prove some results about it, and switch to another, in which case the rule set is recalculated with the new equivalence.

Example 5.23. For example, consider the following:

$$e \in \mathsf{exp} ::= v \mid \lambda v.e \mid e\, e$$

Given no further information, $\approx$ for exp is the identity relation on $\mathsf{Arrangement}$. An author may want to make a number of statements about, e.g., the sub-terms of elements of exp, for which they want $\approx$ to be the identity relation. They might then want to talk about, e.g., the Church-Rosser property, for which they want $\approx$ to be α-equivalence.

5.1.4 Hole Filling

We now define $\approx$-*context-hole filling* for arbitrary objects and arrangements (although it is usually only useful for $\approx$ equivalence classes of objects and arrangements with the correct arity). Not only do authors make use of context-hole filing in their own papers, we will also use it in this paper to describe how objects appear inside one another and to describe "primitive constructors," which are used for disambiguating functions on syntax. As such, we deal with hole filling on both the object and meta-levels of this paper.

Design Decision 5.24. As previously mentioned in Definition 5.4, we use the special object $\Box$ to indicate a hole to be filled. Hole filling is an operation that descends the arrangement tree in a left to right order, filling holes in the order it encounters them, and producing an object corresponding to some $[A]_{\approx}$. Filling O_1 with O_2 is usually denoted by $O_1[O_2]$. While we permit this denotation in other texts, in our definition we disambiguate between objects followed by sequences and hole filling operations by writing $\mathsf{fill}_{\approx}$ to denote any non-zero number of filling operations and $O\vec{O}^{\approx}$ to be the result of filling the holes of an object with every object in the sequence $\vec{O}$. We support hole filling in arrangements whose parts are arranged by orderings other than left to right (E.g. subscripting, superscripting...), but we do not make use of this feature here and omit it, for simplicity. Hole filling offers some motivation for nesting equivalences in MathSyn, as parts of a tree occurring above $\Box$ can only be considered up to literal equivalence on syntax, but parts of a tree featuring $\Box$, which do not occur above it, may occur in larger equivalences.

Definition 5.25 (Contexts and $\approx$-Hole Filling)**.**
Let $\approx$ be an $\mathsf{Arrangement}$-Equivalence and $\vec{O}$ be a sequence of objects.

- A *context* is an object with arity ≥ 1.
- We define hole filling inside arrangements and objects as follows:

1. $\mathsf{fill}_{\approx}(\Box, [O] \cdot \vec{O}) = (O, \vec{O})$[9].

2. $\mathsf{fill}_{\approx}(O, \vec{O}) = (O, \vec{O})$, if O is not a context[10] or $\vec{O} = [\,]$, where $[\,]$ denotes the empty sequence

3. $\mathsf{fill}_{\approx}(\{A\}, \vec{O}) = ([A']_{\approx}, \vec{O}')$[11] if the arity of $A \geq 1$, and $\mathsf{fill}_{\approx}(A, \vec{O}) = (A', \vec{O}')$.[12]

[9]Each hole uses up one of the replacements. Recall $[O] \cdot \vec{O}$ is the result of concatenating the sequence $[O]$ with $\vec{O}$.

[10]The only way context-hole filling skips embedded objects which are non-singleton equivalence classes of arrangements.

[11]Note that we do not define $\mathsf{fill}_{\approx}(O, \vec{O})$ where O is a context and $|O| > 1$ as such O are not well-formed unless they do not contain $\Box$.

[12]Context-hole filling in a well formed context only descends inside an arrangement that is alone in its equivalence class, else ambiguities may arise over whether free variables in an inserted object

This is similar to the previous example except it refers to the cases where O is a context.

4. $\mathsf{fill}_{\approx}(\epsilon, \vec{O}) = (\epsilon, \vec{O})$ $\quad$ $\mathsf{fill}_{\approx}(s, \vec{O}) = (s, \vec{O})$ $\quad$ $\mathsf{fill}_{\approx}(n, \vec{O}) = (n, \vec{O})$.

5. $\mathsf{fill}_{\approx}(\overline{A}, \vec{O}) = (\overline{A'}, \vec{O}')$ if $A \neq \epsilon$ and $\mathsf{fill}_{\approx}(A, \vec{O}) = (A', \vec{O}')$

6. For[13] non-empty arrangements A_1, A_2, A_3, core arrangement $\hat{A}$, and lists of objects $\vec{O}_1$, $\vec{O}_2$, $\vec{O}_3$, $\vec{O}_4$, if $\mathsf{fill}_{\approx}(\hat{A}, \vec{O}_2) = (\hat{A}', \vec{O}_3)$and for all $1 \leq i \leq 3$, $\mathsf{fill}_{\approx}(A_i, \vec{O}_i) = (A'_i, \vec{O}_{i+1})$ and A'_i is non-empty then:[14]

$$\begin{array}{ll} \mathsf{fill}_{\approx}(A_1\hat{A}, \vec{O}_1) = (A'_1\hat{A}', \vec{O}_3) & \mathsf{fill}_{\approx}(A_1{}^{A_2}_{A_3}, \vec{O}_1) = (A'_1{}^{A'_2}_{A'_3}, \vec{O}_4) \\ \mathsf{fill}_{\approx}(A_{1A_2}, \vec{O}_1) = (A'_{1A'_2}, \vec{O}_3) & \mathsf{fill}_{\approx}(A_1{}^{A_2}, \vec{O}_1) = (A'_1{}^{A'_2}, \vec{O}_3) \end{array}$$

7. $\mathsf{fill}_{\approx}(P_O, \vec{O}) = (P_{O'}, \vec{O}')$ if $\mathsf{fill}_{\approx}(O, \vec{O}) = (O', \vec{O}')$.[15]

8. We define notation for when no hole filling operations remain and we may display only the object or arrangement relating to the hole filling operation without displaying the sequence still. Let O (resp. A) be such that $\mathsf{fill}_{\approx}(O, \vec{O}) = (O', [\,])$ (resp. $\mathsf{fill}_{\approx}(A, \vec{O}) = (A', [\,])$). We define the object (resp. arrangement) which fills the holes $\Box$ reachable from O (resp. A) with the objects in the sequence $\vec{O}$ as $O\vec{O}^{\approx} = O'$ (resp. $A\vec{O}^{\approx} = A'$).

9. We say that $\approx$ is closed under hole filling if for any $\approx$-well-formed A, A', and any sequences of $\approx$-well-formed objects $\vec{O}, \vec{O}'$, if $A\vec{O}^{\approx} \approx A'\vec{O}'^{\approx}$ then $A \approx A'$ and $\vec{O} = \vec{O}'$.[16] For example if the equivalence relation $\approx$ had equivalence up to reordering of objects for an arrangement with multiple holes, say $a, b \approx b, a$ for the filled arrangement $\Box, \Box$ then for two sequences of non identical $\approx$-equivalence classes $[O_1, O_2]$ and $[O_2, O_1]$, we could have $\Box, \Box[O_1, O_2]^{\approx} \approx \Box, \Box[O_1, O_2]^{\approx}$ where $[O_1, O_2] \neq [O_2, O_1]$, so $\approx$ wouldn't be closed under hole filling in this case.

become bound (e.g., $\mathsf{fill}_{\approx}(\{\lambda x.\Box, \lambda y.\Box\}, [\{x\}])$ would create ambiguities of whether the inserted X was bound by the λ). This is why $\approx$ must not relate distinct arrangements containing holes.

[13]Context-hole filling essentially traverses arrangement trees left-to-right filling holes in the reached order.

[14]These $\vec{O}_i$ are threaded through a series of calls to fill

[15]Context-hole filling descends object pointers until it encounters a hole.

[16]This is needed to show the unique decomposition of an object into a context and a list of objects.

Note that if $O\vec{O}_{\approx} = O'$ then also $P_O\vec{O}^{\approx} = P_{O'}$. Note additionally that most authors dealing with hole filling drop the $^{\approx}$ off of the notation $A\vec{O}^{\approx}$ and $O\vec{O}^{\approx}$. This is because the works we tend to deal with assume working modulo $\approx$ equivalence. $\square$

Example 5.26. Let $\approx$ be an $\mathsf{Arrangement}$-Equivalence. Here are some examples of context-hole filling (use Lemma 5.20):[17]

- Arrangement $\square\,\square$ stands for $P_\square\,P_\square$. Hence
  $\mathsf{fill}_{\approx}(\square\square, [O, O, O, O]) = \mathsf{fill}_{\approx}(P_\square P_\square, [O, O, O, O]) = (P_O P_O, [O, O]) = (OO[O, O])$.[18]
  And, $\mathsf{fill}_{\approx}(\square\square, [O, O]) = \mathsf{fill}_{\approx}(P_\square P_\square, [O, O]) = (P_O P_O, []) = (OO, [])$.
  Hence, the arrangement $(\square\,\square)[O, O]^{\approx} = O\,O$.

- Object $\square\,\square$ stands for $[P_\square\,P_\square]_{\approx}$. Hence object $(\square\,\square)[O, O]^{\approx} = [P_\square\,P_\square]_{\approx}[O, O]^{\approx} = [P_O\,P_O]_{\approx} = O\,O$.

- Arrangement $\square \to O_1[O_2 \to O_2]^{\approx} = (P_\square \to P_{O_1})[P_{O_2} \to P_{O_2}]^{\approx} = P_{[P_{O_2} \to P_{O_2}]_{\approx}} \to P_{O_1} = O_2 \to O_2 \to O_1$.

- Object $\square \to O_1[O_2 \to O_2]^{\approx} = \{P_\square \to P_{O_1}\}[P_{O_2} \to P_{O_2}]^{\approx} = [P_{[P_{O_2} \to P_{O_2}]_{\approx}} \to P_{O_1}]_{\approx} = (O_2 \to O_2) \to O_1$.[19]

- Object $(\square := \square, \square)[O_1, O_2, O_3]^{\approx} = \{P_\square := P_\square, P_\square\}[O_1, O_2, O_3]^{\approx} = [P_{O_1} := P_{O_2}, P_{O_3}]_{\approx} = (O_1 := O_2, O_3)$.

- Object $(\square := \square, \square)[O_1, O_2]^{\approx} = \{P_\square := P_\square, P_\square\}[O_1, O_2]^{\approx} = [P_{O_1} := P_{O_2}, P_\square]_{\approx} = (O_1 := O_2, \square)$.

- Arrangement $(\lambda \mathsf{x}.\mathsf{x})[]^{\approx} = \lambda \mathsf{x}.\mathsf{x}$ and object $[\lambda \mathsf{x}.\mathsf{x}]_{\approx}[]^{\approx} = [\lambda \mathsf{x}.\mathsf{x}]_{\approx}$.

- Pointer $P_{[\lambda \mathsf{x}.\mathsf{x}]_{\approx}}[]^{\approx} = P_{[\lambda \mathsf{x}.\mathsf{x}]_{\approx}}$.

We show that hole filling is a well defined partial function that reduces the arity of an object or arrangement by the size of the sequence of objects holes are filled with.

[17]To illustrate the different ways hole filling may be written we also write down object names in the place of pointers and leave arrangement coercions $[A]_{\approx}$ as optional. The rules for doing this will be discussed in Convention 5.46.

[18]Note (O O[O,O]) here is not another $\mathsf{fill}_{\approx}$ operation, but a result of the hole filling operation being applied to a sequence longer than the arity of the arrangement being filled.

[19]The curly braces here denote a set containing the arrangement $P_\square \to P_{O_1}$.

Lemma 5.27. *Let an* $\mathsf{Arrangement}$-*Equivalence* $\approx$*, a sequence of objects* $\vec{O}$*, an object* O *and an arrangement* A*. The following hold.*

1. *If* $\mathsf{fill}_{\approx}(A, \vec{O}) = (A', \vec{O}')$ *then* $A \notin \mathsf{Pointer}$ *iff* $A' \notin \mathsf{Pointer}$.

2. $\mathsf{fill}_{\approx}$ *is a well defined partial function which when* $\approx$ *is defined, takes an object (resp. an arrangement) and a list of objects and returns an object (resp. an arrangement) and a list of objects.*

3. *Let* $\mathsf{fill}_{\approx}(\heartsuit, \vec{O}) = (\heartsuit', \vec{O}')$*. If* $\heartsuit$ *is an object (resp. an arrangement) then* $\heartsuit'$ *is an object (resp. an arrangement) and* $\vec{O} = \vec{O}'' \cdot \vec{O}'$ *for some* $\vec{O}''$*. Furthermore:*

   (a) *If* $\mathsf{Ari}(\heartsuit) \leq |\vec{O}|$ *then* $|\vec{O}''| = \mathsf{Ari}(\heartsuit)$*, and for any list of objects* $\vec{O}'''$ *we have* $\mathsf{fill}_{\approx}(\heartsuit, \vec{O} \cdot \vec{O}''') = (\heartsuit', \vec{O} \cdot' \vec{O}''')$.

   If $\mathsf{Ari}(\heartsuit) = |\vec{O}|$ *then* $\vec{O}' = []$*, else if* $\mathsf{Ari}(\heartsuit) < |\vec{O}|$ *the* $\vec{O}' \neq []$.

   (b) *If* $\mathsf{Ari}(\heartsuit) > |\vec{O}|$ *then* $\mathsf{Ari}(\heartsuit') \geq 1$ *and* $\vec{O}' = []$.

4. *Let* $\heartsuit$ *be an object (resp. arrangement). Then:*

   (a) $\heartsuit\vec{O}^{\approx}$ *is well defined iff (*$\mathsf{fill}_{\approx}(\heartsuit, \vec{O})$ *is well defined and* $\mathsf{Ari}(\heartsuit) \geq |\vec{O}|$*).*

   (b) *If* $\mathsf{fill}_{\approx}(\heartsuit, [\,]) = (\heartsuit', \vec{O}')$ *then* $\heartsuit = \heartsuit'$ *and* $\vec{O}' = []$.

   (c) $\mathsf{fill}_{\approx}(\heartsuit, \vec{O}) = (\heartsuit, \vec{O})$ *iff* $\mathsf{fill}_{\approx}(\heartsuit, \vec{O})$ *is well defined and (the arity of* $\heartsuit = 0$ *or* $\vec{O} = []$*).*

5. *If* $\mathsf{fill}_{\approx}(A, \vec{O}) = (A', [\,])$ *then* $\mathsf{fill}_{\approx}([A]_{\approx}, \vec{O}) = ([A']_{\approx}, [\,])$.

Proof. 1. By induction on the structure of the derivation $\mathsf{fill}_{\approx}(A, \vec{O}) = (A', \vec{O}')$.
2. By simutaneous induction on the structure of O and A.
3. By induction on the derivation $\mathsf{fill}_{\approx}(\heartsuit, \vec{O}) = (\heartsuit', \vec{O}')$.
4a. By definition 5.25 and 3. above.
4b. By induction on the derivation $\mathsf{fill}_{\approx}(\heartsuit, \vec{O}) = (\heartsuit', \vec{O}')$.
4c. By induction on the structure of A.
5. If the arity of $A \geq 1$ then since $[A]_{\approx}$ is an object, by Lemma 5.16.2, $[A]_{\approx} = \{A\}$ and

$A \notin \mathsf{Pointer}$. Then by definition, $\mathsf{fill}_\approx([A]_\approx, \vec{O}) = \mathsf{fill}_\approx(\{A\}, \vec{O}) = ([A']_\approx, [\,])$.
If the arity of $A = 0$ then by 7. above, $\mathsf{fill}_\approx(A, \vec{O}) = (A, \vec{O})$. Hence $(A, \vec{O}) = (A', [\,])$ and so, $A = A'$ and $\vec{O} = [\,]$. Since the arity of A = the arity of $[A]_\approx = 0$, we get $\mathsf{fill}_\approx([A]_\approx, \vec{O}) = ([A]_\approx, \vec{O})$. Since $A = A'$ and $\vec{O} = [\,]$, we have $\mathsf{fill}_\approx([A]_\approx, \vec{O}) = ([A']_\approx, [\,])$. □

We show that hole filling preserves $\approx$-well formedness on objects or arrangements that are $\approx$-well formed.

Lemma 5.28. *Let an* Arrangement-*Equivalence* $\approx$*, a sequence of objects* $\vec{O}$*, an object* O *and an arrangement* A*. The following hold.*

1. *If* O *(resp.* A*) is* $\approx$*-well-formed then* $\mathsf{fill}_\approx(O, \vec{O})$ *(resp.* $\mathsf{fill}_\approx(A, \vec{O})$*) is well defined.*[20]

2. *If* $\mathsf{fill}_\approx(\heartsuit, \vec{O}) = (\heartsuit', \vec{O}')$ *where* $\heartsuit'$ *is* $\approx$*-well-formed and* $\vec{O}'$ *is a sequence of* $\approx$*-well-formed objects, then* $\heartsuit$ *is* $\approx$*-well-formed and* $\vec{O}$ *is a sequence of* $\approx$*-well-formed objects.*

3. *If* $\vec{O}$ *is a sequence of* $\approx$*-well-formed objects and* $\approx$ *preserves* $\approx$*-well-formedness then:*

   (a) If O *is* $\approx$*-well-formed then* $\mathsf{fill}_\approx(O, \vec{O}) = (O', \vec{O}')$ *where* O' *is* $\approx$*-well-formed.*

   (b) If A *is* $\approx$*-well-formed then* $\mathsf{fill}_\approx(A, \vec{O}) = (A', \vec{O}')$ *where* A' *is* $\approx$*-well-formed.*

4. *If the arity of* $O \geq |\vec{O}|$ *and* O *is* $\approx$*-well-formed then* $O\vec{O}^\approx$ *is well defined. If also* $\vec{O}$ *is a sequence of* $\approx$*-well-formed objects and* $\approx$ *preserves* $\approx$*-well-formedness, then* $O\vec{O}^\approx$ *is* $\approx$*-well-formed.*

Proof. 1. By simutaneous induction on the structure of O and A.
2. By induction on the derivation $\mathsf{fill}_\approx(\heartsuit, \vec{O}) = (\heartsuit', \vec{O}')$.
3. By simutaneous induction on the structure of O and A. We only do one case. Assume a $\approx$-well-formed $O \neq \Box$ of arity ≥ 1. By Lemma 5.20, $O = \{A\} = [A]_\approx$ for some $\approx$-well-formed A of arity ≥ 1 such that $A \notin \mathsf{Pointer}$. By IH, $\mathsf{fill}_\approx(A, \vec{O}) = (A', \vec{O}')$ where

[20]We mean well defined not $\approx$-well-formed here as $\mathsf{fill}_\approx(O, \vec{O})$ (resp. $\mathsf{fill}_\approx(A, \vec{O})$ is not necessarily an object.

A' is $\approx$-well-formed. Hence by definition, $\mathsf{fill}_{\approx}(\{A\}, \vec{O}) = ([A']_{\approx}, \vec{O'})$. By Lemma 5.27.1, $A' \notin \mathsf{Pointer}$. Since $\approx$ preserves $\approx$-well-formedness, $[A']_{\approx}$ is $\approx$-well-formed.
4. The first part is a consequence of 1. above and Lemma 5.27.4a. The second part follows from 3. above. □

5.1.5 Contexts and Compatible Closure

Next we define the set of contexts O of arity 1 which are functions from a set of objects S_1 to a set of objects S_2, as well as *S-compatible closure*. An *S-compatible closure* describes the set of relations produced when a given relation is allowed to descend inside some set of contexts S. This gives a tool for describing relations that "descend" inside syntactic objects. For example rewriting relations make use of the notion of *S-compatible closure* to perform rewriting on sub-objects in a grammar. Example 5.30 will give an example of how *S-compatible closure* are used for rewriting relations, but first we define what an *S-compatible closure* is.

Definition 5.29 ($(S_1, S_2)-\mathsf{Context}$, $S-\mathsf{Context}$, *S-compatible closure*)**.**
Let $\approx$ be an $\mathsf{Arrangement}$-Equivalence which preserves $\approx$-well-formedness. Let S_1, S_2, S be sets of $\approx$-well-formed objects.

- We define a kind of function space $(S_1, S_2)-\mathsf{Context} =$
  $\{O_{\mathsf{c}} \mid \mathsf{Ari}(O_{\mathsf{c}}) = 1 \land O_{\mathsf{c}}$ is $\approx$-well-formed object $\land \forall O' \in S_1, O_{\mathsf{c}}[O']^{\approx} \in S_2\}$.

- We write $S-\mathsf{Context}$ for $(S, S)-\mathsf{Context}$.

- For relation R where $(\mathsf{domain}(R) \cup \mathsf{range}(R)) \subseteq S \subseteq \mathsf{Object}$, define the *$S$-compatible closure of R*, $[R]^S$ by: if $O_{\mathsf{c}} \in S-\mathsf{Context}$ and there exists O_1 and O_2 such that, $O_1 \overset{R}{\to} O_2$,[21] then $O_{\mathsf{c}}[O_1] \overset{[R]^S}{\to} O_{\mathsf{c}}[O_2]$. Write $[R]$ for $[R]^S$ for some S which can be inferred from the context of discussion. The above should define *S-compatible* and then rely on our earlier definition of closure to get *S-compatible closure of R.*

□

[21] $O_1 \overset{R}{\to} O_2$ is an alternative notation for $(O_1, O_2) \in R$.

By Lemma 5.28, all of $(S_1, S_2)-\mathsf{Context}$, $S-\mathsf{Context}$ and S-*compatible closure* are well-defined.

Contexts allow relations to descend inside of sets, which can be particularly relevant for rewriting. For example, we will model the η-reduction relation on the λ-calculus.

Example 5.30. Recall the grammar exp given by:

$$e \in \mathsf{exp} ::= v \mid \lambda v.e \mid e\, e$$

We may write $e_c \in \mathsf{exp}\text{-}\mathsf{Context} ::= \square \mid \lambda v.e_c \mid e_c\, e \mid e\, e_c$. Suppose we define the relation then η as the smallest relation (in the ordering given by $\subseteq$) satisfying the constraint $[\lambda v.e_1 v]_{\approx} \overset{\eta}{\to} e_1$, where $e_1 \in \mathsf{exp}$ and v is an element of a countable set of names which is not free in e_1. Then the exp-compatible closure of η, $[\eta]^{\mathsf{exp}}$ is the smallest relation s.t. $e_c[\lambda v.e_1 v] \overset{[\eta]^{\mathsf{exp}}}{\to} e_c[e_1]$ and v is not free in e_1.

Convention 5.31 (Choosing contexts for rewriting rules on a set)**.** For a set S, and a relation $* : S \to S$ and a set of constraints on $*$[22], if we write that $*$ is a rewriting rule on S then $*$ is the S-compatible closure of the least $*$ satisfying these constraints.

5.1.6 Sub-Objects and Primitive Constructor Decomposition

Next, we will define 2 related notions, primitive constructors and Primitive constructor decomposition.

Design Decision 5.32. Primitive constructors may be thought of as representing applications of syntactic "functions" to a list of objects in order to produce another syntactic object which has each of the objects in the list as an immediate sub-object. A primitive constructor decomposition gives a notion of how a syntactic object may be broken down into sets of syntactic "functions" applied to symbols. We start by the definition of an immediate sub-object.

[22]E.g., statements of the form $\circ_1 \overset{*}{\to} \circ_2$ if c, where each $\circ_1$ and $\circ_2$ is an element of S and c is an optional side condition s.t. each c yields a boolean. If so, they are read as "Each $(\circ_1, \circ_2)$ is an element of $*$ if c holds."

Definition 5.33 (Immediate sub-objects isub, ln)**.** We define immediate sub-objects isub on objects and arrangements as follows:

- $\mathsf{isub}(\Box) = \mathsf{isub}(\epsilon) = \mathsf{isub}(s) = \mathsf{isub}(n) = \emptyset$.
- $\mathsf{isub}(P_O) = \{O\}$.
- $\mathsf{isub}(\overline{A}) = \mathsf{isub}(A)$.
- $\mathsf{isub}(A\hat{A}) = \mathsf{isub}(A) \cup \mathsf{isub}(\hat{A})$.
- $\mathsf{isub}(A^{A_1}) = \mathsf{isub}(A_{A_1}) = \mathsf{isub}(A) \cup \mathsf{isub}(A_1)$.
- $\mathsf{isub}(A^{A_1}_{A_2}) = \mathsf{isub}(A) \cup \mathsf{isub}(A_1) \cup \mathsf{isub}(A_2)$.
- If $O \neq \Box$ then $\mathsf{isub}(O) = \bigcup_{A \in O} \mathsf{isub}(A)$.

One may think of ln as counting the number of arrangements that make up an object. We define ln of an object and an arrangement as follows:

- $\mathsf{ln}(\heartsuit) = 1$ if $\heartsuit = \Box$ or $\heartsuit$ is an arrangement.
- If $O \neq \Box$ then $\mathsf{ln}(O) = |O|$

$\Box$

Lemma 5.34. *1.* isub *is well defined and* $\mathsf{isub}(O) \cup \mathsf{isub}(A) \subseteq \mathsf{Object}$.

2. Let $\approx$ *be an* $\mathsf{Arrangement}$*-Equivalence. For all* $O \in \mathsf{Object}$ $A \in \mathsf{Arrangement}$*, if* O *and* A *are* $\approx$*-well-formed then* $\mathsf{isub}(O) \cup \mathsf{isub}(A) \subseteq \{O' \mid O' \textit{ is } \approx\textit{-well-formed}\}$.

Proof. Easy. $\Box$

We can now give the following convention:

Convention 5.35 (Choosing contexts for rewriting rules where no set is given)**.** There is also a similar convention that can be used for rewriting rules where S is not given. This helps with practices in CSM (e.g., where an author declares a number of relations with a horizontal bar, but doesn't tell us what set they are drawn from). In the case where S is not given, then it is assumed to be the largest subset S of Object such that:[23]

1. Its elements and their sub-objects are equivalence classes of $\approx$.
2. For all $A \in \mathsf{Arrangement}$ whose immediate sub-objects are holes, and $O_1, \ldots, O_n \in \mathsf{Object}$ $A[O_1, \ldots, O_n]$ is only contained within an element of S, or within a sub-object of an element of S, if, there exist $x_1, \ldots, x_n$ where each x_i is either an object or a metavariable and $A[x_1, \ldots, x_n]$ appears somewhere in the rule set.

Example 5.36. $\mathsf{isub}(\Box\,\Box) = \mathsf{isub}(P_\Box\, P_\Box) = \mathsf{isub}(\{P_\Box\, P_\Box\}) = \{\Box\}$.
$\mathsf{isub}(\Box = \Box \in \Box) = \mathsf{isub}(P_\Box = P_\Box \in P_\Box) = \mathsf{isub}(\{P_\Box = P_\Box \in P_\Box\}) = \{\Box\}$.
$\mathsf{isub}(\lambda\Box.\Box) = \mathsf{isub}(\lambda P_\Box.P_\Box) = \mathsf{isub}(\{\lambda P_\Box.P_\Box\}) = \{\Box\}$.
$\mathsf{isub}(\langle(\lambda O_1.O_2)\rangle) = \mathsf{isub}(\langle(\lambda P_{O_1}.P_{O_2})\rangle) = \mathsf{isub}(\langle P_{[\lambda P_{O_1}.P_{O_2}]_\approx}\rangle) = \{[\lambda P_{O_1}.P_{O_2}]_\approx\} = \{[\lambda O_1.O_2]_\approx\}$.
$\mathsf{isub}(\langle O_1\, O_2\rangle) = \mathsf{isub}(\langle P_{O_1}\, P_{O_2}\rangle) = \{O_1, O_2\}$.
$\mathsf{isub}((O_1 + O_2) + O_3) = \mathsf{isub}(P_{[P_{O_1}+P_{O_2}]_\approx} + P_{O_3}) = \{O_1 + O_2, O_3\}$.
$\mathsf{ln}(\Box\,\Box) = \mathsf{ln}(\Box = \Box \in \Box) = \mathsf{ln}(\langle\Box\rangle) = 1$.
$\mathsf{ln}([\langle(\lambda O_1.O_2)\rangle]_\approx) = \mathsf{ln}([\langle P_{[\lambda P_{O_1}.P_{O_2}]_\approx}\rangle]_\approx) = |[\langle P_{[\lambda P_{O_1}.P_{O_2}]_\approx}\rangle]_\approx| = |[\langle(\lambda O_1.O_2)\rangle]_\approx|$.
$\mathsf{ln}([\langle O_1\, O_2\rangle]_\approx) = \mathsf{ln}([\langle P_{O_1}\, P_{O_2}\rangle]_\approx) = |[\langle P_{O_1}\, P_{O_2}\rangle]_\approx|$.
If $[O_1 + O_2]_\approx = \{O_1 + O_2, O_2 + O_1\}$ then $\mathsf{ln}([O_1 + O_2]_\approx) = 2$.

We now define the notion of decomposition into constructors. We may think of constructors as the hole filling operations that would yield an object.

Example 5.37. A simple example of a single decomposition is as follows:

$$\overbrace{P_\Box P_{[\lambda \mathsf{x}.\mathsf{x}]_\approx}}^{\mathsf{dc}} = \{(P_\Box P_\Box, [\Box, [\lambda\mathsf{x}.\mathsf{x}]_\approx])\}$$

Definition 5.38 (Decomposition $\overbrace{\cdot}^{\mathsf{dc}}$)**.** Let $\approx$ be an $\mathsf{Arrangement}$-Equivalence. We define the decompostion class $\overbrace{\heartsuit}^{\mathsf{dc}}$ of objects and arrangements $\heartsuit$ as follows:

[23]We pick a subset of object where arrangements referenced in the text act like production rules for the grammar and objects are considered up to $\approx$-equivalence.

- If $\heartsuit \in \{\square, \epsilon, s, n\}$ then $\overbrace{\heartsuit}^{\mathsf{dc}} = \{(\heartsuit, [])\}$.[24]

- If $(O', \vec{O}) \in \overbrace{O}^{\mathsf{dc}}$ then $(P_\square, [O'\vec{O}^{\approx}]) \in \overbrace{P_O}^{\mathsf{dc}}$

- If $A \neq \epsilon$ and $(A', \vec{O}) \in \overbrace{A}^{\mathsf{dc}}$ then $(\overline{A'}, \vec{O}) \in \overbrace{\overline{A}}^{\mathsf{dc}}$.

- For non-empty arrangements A_1, A_2, A_3, and core arrangement $\hat{A}$, if $(\hat{A}', \vec{O}) \in \overbrace{\hat{A}}^{\mathsf{dc}}$ and for all $1 \leq i \leq 3$, $(A'_i, \vec{O}_i) \in \overbrace{A_i}^{\mathsf{dc}}$ and A'_i is non-empty then:

$$(A'_1\hat{A}', \vec{O}_1 \cdot \vec{O}) \in \overbrace{A_1\hat{A}}^{\mathsf{dc}} \qquad (A'^{\,A'_2}_{1\,A'_3}, \vec{O}_1 \cdot \vec{O}_2 \cdot \vec{O}_3) \in \overbrace{A_1{}^{A_2}_{A_3}}^{\mathsf{dc}}$$

$$\overbrace{A_{1A_2}}^{\mathsf{dc}} \in \{(A'_{1A'_2}, \vec{O}_1 \cdot \vec{O}_2)\} \qquad \overbrace{A_1{}^{A_2}}^{\mathsf{dc}} \in \{(A'_1{}^{A'_2}, \vec{O}_1 \cdot \vec{O}_2)\}$$

- If $O \neq \square$ then $\overbrace{O}^{\mathsf{dc}} = \{([A']_{\approx}, \vec{O}) \mid A \in O \text{ and } (A', \vec{O}) \in \overbrace{A}^{\mathsf{dc}}\}$.

□

We show that decomposition breaks objects (resp. arrangements) into objects (resp. arrangements) whose arity is equal to their immediate sub-objects and a sequence of these objects.

Lemma 5.39. *Let* $\approx$ *be an* Arrangement-*Equivalence.*

1. *If* $\heartsuit$ *is an object (resp. arrangement) and* $(\diamondsuit, \vec{O}) \in \overbrace{\heartsuit}^{\mathsf{dc}}$ *then* $\diamondsuit$ *is an object (resp. arrangement),* $\vec{O}$ *is a sequence of objects,* $\diamondsuit\vec{O}^{\approx}$ *is well defined, and if* $\diamondsuit \neq \square$ *then the arity of* $\diamondsuit = |\vec{O}|$.

Proof. 1. By induction on the structure of $\heartsuit$. We do the following cases:

- Case P_O where $(P_\square, [O'\vec{O}^{\approx}]) \in \overbrace{P_O}^{\mathsf{dc}}$ and $(O', \vec{O}) \in \overbrace{O}^{\mathsf{dc}}$ then by IH, O' is an object, $\vec{O}$ is a sequence of objects and $O'\vec{O}^{\approx}$ is well defined. Now, $\mathsf{fill}_{\approx}(P_\square, [O'\vec{O}^{\approx}]) =$

[24]The set of tuples we build here may be thought of as each hole filling operation that constructs an object from its immediate sub-objects.

$(P_{O'\vec{O}^{\approx}}, [])$ and hence $P_\Box[O'\vec{O}^{\approx}]^{\approx}$ is well defined. Finally, the arity of $P_\Box$ = $|[O'\vec{O}^{\approx}]|$.

- Let non-empty arrangements A_1, and core arrangement $\hat{A}$, where $(\hat{A}', \vec{O}) \in \overbrace{\hat{A}}^{\mathsf{dc}}$ and $(A_1', \vec{O}_1) \in \overbrace{A_1}^{\mathsf{dc}}$ and A_1' is non-empty. Then, $(A_1'\hat{A}', \vec{O}_1 \cdot \vec{O}) \in \overbrace{A_1\hat{A}}^{\mathsf{dc}}$. Use IH to show that $A_1'\hat{A}'$ is an arrangement and $\vec{O}_1 \cdot \vec{O}$ is a sequence of objects. Also by IH, $A_1'[\vec{O}_1]^{\approx}$ and $\hat{A}'[\vec{O}]^{\approx}$ are well defined (i.e., for some arrangements A_1'' and $\hat{A}''$ we have $\mathsf{fill}_{\approx}(A_1', \vec{O}_1) = (A_1'', [\,])$ and $\mathsf{fill}_{\approx}(\hat{A}', \vec{O}) = (\hat{A}'', [\,])$) and the arity of $A_1' = |\vec{O}_1|$ and the arity of $\hat{A}' = |\vec{O}|$. By Lemma 5.27, $\mathsf{fill}_{\approx}(A_1', \vec{O}_1 \cdot \vec{O}) = (A_1'', \vec{O})$ and by definition, $\mathsf{fill}_{\approx}(A_1'\hat{A}', \vec{O}_1 \cdot \vec{O}) = (A_1''\hat{A}'', [\,])$. Hence $(A_1'\hat{A}')[\vec{O}_1 \cdot \vec{O}]^{\approx}$ is well defined. Arity is easy by IH.

- Let $([A']_{\approx}, \vec{O}) \in \overbrace{O}^{\mathsf{dc}}$ where $O \neq \Box$, and for some $A \in O$, $(A', \vec{O}) \in \overbrace{A}^{\mathsf{dc}}$. By IH, A' is an arrangement, $\vec{O}$ is a sequence of objects, $A'\vec{O}^{\approx}$ is well defined (hence $\mathsf{fill}_{\approx}(A', \vec{O}) = (A'', [\,])$ for some A''), and the arity of $A' = |\vec{O}|$. By Lemma 5.16, $[A']_{\approx}$ is an object. By Lemma 5.27.5, $\mathsf{fill}_{\approx}([A']_{\approx}, \vec{O}) = ([A'']_{\approx}, [\,])$. This means that $[A']_{\approx}\vec{O}^{\approx}$ is well defined. Finally, if the arity of $A' = |\vec{O}| = 0$, then the arity of $[A']_{\approx} = 0 = |\vec{O}|$; else, if the arity of $A' = |\vec{O}| \neq 0$, then by Lemma 5.16, $[A']_{\approx} = \{A'\}$ and hence the arity of $[A']_{\approx}$ = the arity of $A' = |\vec{O}|$.

$\Box$

If we want to prove that for all $1 \leq i \leq n$, $([A_1']_{\approx}, \vec{O}_1) = ([A_i']_{\approx}, \vec{O}_i)$ (i.e., unicity of decomposition), then we would need to assume $\approx$ to be closed under hole filling and in this case,

- If the arity of $O \geq 1$ then by Lemma 5.20, $n = 1$ and we are done.

- If the arity of $O = 0$ then since for all $1 \leq i \leq n$, $A_i = A_i'\vec{O}_i^{\approx}$, and $A_i \approx A_1$, we get for all $1 \leq i \leq n$, $A_i'\vec{O}_i^{\approx} \approx A_1'\vec{O}_1^{\approx}$. Since $\approx$ is closed under hole filling, we get for all $1 \leq i \leq n$, $A_i' \approx A_1'$ and $\vec{O}_i = \vec{O}_1$. Hence, for all $1 \leq i \leq n$, $([A_1']_{\approx}, \vec{O}_1) = ([A_i']_{\approx}, \vec{O}_i)$.

We show that decomposition preserves $\approx$-well formedness.

Lemma 5.40. *Let $\approx$ be an* Arrangement*-Equivalence which preserves $\approx$-well-formedness. $\heartsuit, \heartsuit'$ be $\approx$-well-formed objects/arrangements and let A be $\approx$-well-formed.*

1. *If $(\diamondsuit, \vec{O}) \in \overbrace{\heartsuit}^{\mathsf{dc}}$ then $\diamondsuit$ is $\approx$-well-formed, $\vec{O}$ is a sequence of $\approx$-well-formed objects, and $\mathsf{fill}_{\approx}(\diamondsuit, \vec{O}) = (\heartsuit, [\,])$ (hence $\heartsuit = \diamondsuit\vec{O}^{\approx}$). Furthermore, if either $\heartsuit = \Box$ or $\diamondsuit = \Box$ then both $\heartsuit = \Box$ and $\diamondsuit = \Box$ and $\vec{O} = []$.*

2. *There is a $\approx$-well-formed $\diamondsuit$ and a sequence of $\approx$-well-formed objects $\vec{O}$ such that $(\diamondsuit, \vec{O}) \in \overbrace{\heartsuit}^{\mathsf{dc}}$.*

3. *If $(A', \vec{O}) \in \overbrace{A}^{\mathsf{dc}}$ then:*

   (a) If $\Box \in \mathsf{isub}(A')$ then $\mathsf{isub}(A') = \{\Box\}$ and A' differs from A exactly in having $\Box$ in place of every non-$\Box$ object appearing in A.

   (b) If $\Box \notin \mathsf{isub}(A')$ then $\mathsf{isub}(A') = \emptyset$, $A = A'$ and $\vec{O} = []$.

4. *If $\overbrace{\heartsuit}^{\mathsf{dc}} \cap \overbrace{\heartsuit'}^{\mathsf{dc}} \neq \emptyset$ then $\heartsuit = \heartsuit'$.*

5. *If $(\diamondsuit, \vec{O}) \in \overbrace{\heartsuit}^{\mathsf{dc}}$ and $(\diamondsuit', \vec{O}') \in \overbrace{\heartsuit}^{\mathsf{dc}}$ then $\diamondsuit\vec{O}^{\approx} = \diamondsuit'\vec{O}'^{\approx}$.*

6. *If $\heartsuit$ is an arrangement or $\heartsuit$ is a context, then $|\overbrace{\heartsuit}^{\mathsf{dc}}| = 1$, else $|\overbrace{\heartsuit}^{\mathsf{dc}}| = \mathsf{ln}(\heartsuit)$.*

Proof. 1. By induction on the structure of $\heartsuit$. We do the following cases:

- Case P_O is $\approx$-well-formed where $(P_{\Box}, [O'\vec{O}^{\approx}]) \in \overbrace{P_O}^{\mathsf{dc}}$ and $(O', \vec{O}) \in \overbrace{O}^{\mathsf{dc}}$ then by IH, O' is an $\approx$-well-formed object, $\vec{O}$ is a sequence of $\approx$-well-formed objects, and $\mathsf{fill}_{\approx}(O', \vec{O}) = (O, [\,])$, and hence $O = O'\vec{O}^{\approx}$. Now, $\mathsf{fill}_{\approx}(P_{\Box}, [O'\vec{O}^{\approx}]) = (P_{O'\vec{O}^{\approx}}, []) = (P_O, [])$ (hence $P_O = P_{\Box}[O'\vec{O}^{\approx}]^{\approx}$).

- Let non-empty arrangements A_1, and core arrangement $\hat{A}$, where $(\hat{A}', \vec{O}) \in \overbrace{\hat{A}}^{\mathsf{dc}}$ and $(A_1', \vec{O}_1) \in \overbrace{A_1}^{\mathsf{dc}}$ and A_1' is non-empty. Then, $(A_1'\hat{A}', \vec{O}_1 \cdot \vec{O}) \in \overbrace{A_1\hat{A}}^{\mathsf{dc}}$. Use

IH to show that $A'_1\hat{A}'$ is an $\approx$-well-formed arrangement and $\vec{O}_1 \cdot \vec{O}$ is a sequence of $\approx$-well-formed objects. Also by IH, $\mathsf{fill}_\approx(A'_1, \vec{O}_1) = (A_1, [\,])$ and $\mathsf{fill}_\approx(\hat{A}', \vec{O}) = (\hat{A}, [\,])$. But, by Lemma 5.39, the arity of $A'_1 = |\vec{O}_1|$ and hence, by Lemma 5.27, $\mathsf{fill}_\approx(A'_1, \vec{O}_1 \cdot \vec{O}) = (A_1, \vec{O})$ and by definition, $\mathsf{fill}_\approx(A'_1\hat{A}', \vec{O}_1 \cdot \vec{O}) = (A_1\hat{A}, [\,])$ and $A_1\hat{A} = (A'_1\hat{A}')(\vec{O}_1 \cdot \vec{O})^\approx$.

- Let $([A']_\approx, \vec{O}) \in \overbrace{O}^{\mathsf{dc}}$ where $O \neq \square$, O is $\approx$-well-formed and for some $A \in O$, $(A', \vec{O}) \in \overbrace{A}^{\mathsf{dc}}$. Since O is $\approx$-well-formed and $A \in O$ then A is $\approx$-well-formed and $[A]_\approx = O$. By IH, A' is an $\approx$-well-formed arrangement, $\vec{O}$ is a sequence of $\approx$-well-formed objects, and $\mathsf{fill}_\approx(A', \vec{O}) = (A, [\,])$ (hence $A = A'\vec{O}^\approx$). By Lemma 5.27.5, $\mathsf{fill}_\approx([A']_\approx, \vec{O}) = ([A]_\approx, [\,]) =$[25]$(O, [\,])$. This means that $O = [A']_\approx\vec{O}^\approx$ where $[A']_\approx$ is an $\approx$-well-formed object and $\vec{O}$ is a sequence of $\approx$-well-formed objects.

2. By induction on the structure of $\heartsuit$.
3. By induction on the derivation $(A', \vec{O}) \in \overbrace{A}^{\mathsf{dc}}$ where A is $\approx$-well-formed.
4. By 1. above, $\heartsuit = \diamondsuit\vec{O}^\approx$ and $\heartsuit' = \diamondsuit\vec{O}^\approx$. Hence, $\heartsuit = \heartsuit'$.
5. By 1. above, $\heartsuit = \diamondsuit\vec{O}^\approx$ and $\heartsuit = \diamondsuit'\vec{O}'^\approx$. Hence $\diamondsuit\vec{O}^\approx = \diamondsuit'\vec{O}'^\approx$.
6. By induction on the structure of $\heartsuit$. $\square$

Example 5.41. The following are examples of decomposition (we write $\heartsuit$ instead of $\heartsuit[]^\approx]$):

- $\overbrace{\lambda \mathsf{x}.\mathsf{x}}^{\mathsf{dc}} = \{(\lambda \mathsf{x}.\mathsf{x}, [])\}$.

- $\overbrace{[\lambda \mathsf{x}.\mathsf{x}]_\approx}^{\mathsf{dc}} = \{([\lambda \mathsf{x}.\mathsf{x}]_\approx, [])\}$.

- $\overbrace{\lambda P_{[\mathsf{x}]_\approx}.P_{[\mathsf{x}]_\approx}}^{\mathsf{dc}} = \{(\lambda P_\square.P_\square, [[\mathsf{x}]_\approx, [\mathsf{x}]_\approx])\}$.

- $\overbrace{P_{[\lambda \mathsf{x}.\mathsf{x}]_\approx}}^{\mathsf{dc}} = \{(P_\square, [[\lambda \mathsf{x}.\mathsf{x}]_\approx])\}$.

- $\overbrace{\square}^{\mathsf{dc}} = \{(\square, [])\}$.

[25]This is the only place we needed $\approx$-well-formedness.

- $\overbrace{P_\Box}^{\mathsf{dc}} = \{(P_\Box, [\Box])\}$.

- $\overbrace{P_\Box^{P_\Box}}^{\mathsf{dc}} = \{(P_\Box^{P_\Box}, [\Box, \Box])\}$.

- $\overbrace{P_\Box P_{[\lambda x.x]_\approx}}^{\mathsf{dc}} = \{(P_\Box P_\Box, [\Box, [\lambda x.x]_\approx])\}$.

- Assume $\overbrace{O}^{\mathsf{dc}} = \{(O', \vec{O})\}$ then
  - $\overbrace{[!O]_\approx}^{\mathsf{dc}} = \{([!\Box]_\approx, [O'\vec{O}^\approx])\} = \{([!\Box]_\approx, [O])\}$.
  - Using our Coercion Convention 5.46, $\overbrace{!O}^{\mathsf{dc}} = \{(!\Box, [O])\}$.
  - Since $(!O) = P_{[!O]_\approx}$ then $\overbrace{(!O)}^{\mathsf{dc}} = \{(P_\Box, [[!\Box]_\approx [O'\vec{O}^\approx]^\approx])\} = \{(P_\Box, [[!\Box]_\approx [O]^\approx])\} = \{(P_\Box, [[!O]_\approx])\}$.
  - $\overbrace{\langle (!O) \rangle}^{\mathsf{dc}} = \{(\langle P_\Box \rangle, [[!O]_\approx])\}$.
  - Using our Coercion Convention 5.46, $\overbrace{\langle (!O) \rangle}^{\mathsf{dc}} = \{(\langle \Box \rangle, [!O])\}$.

- Assume $\overbrace{O}^{\mathsf{dc}} = \{(O', \vec{O})\}$ and $O_1 = \{!P_O, \langle P_O P_O \rangle\}$ where the arity of $O = 0$. Then $\overbrace{O_1}^{\mathsf{dc}} = \{(!P_\Box, [O'\vec{O}^\approx]), (\langle P_\Box P_\Box \rangle, [O'\vec{O}^\approx, O'\vec{O}^\approx])\}$

$$= \{(!P_\Box, [O]), (\langle P_\Box P_\Box \rangle, [O, O])\}.$$

Note that in all the cases of the above example except for the first 2 lines, if $\overbrace{\heartsuit}^{\mathsf{dc}} = \{(\diamondsuit, \vec{O})\}$ then $\mathsf{isub}(\diamondsuit) = \{\Box\}$. For the first 2 lines, $\mathsf{isub}(\diamondsuit) = \emptyset$.

From our notion of decomposition we move into a notion of decomposition into primitive constructors.

Design Decision 5.42. Sub-objects in a decomposition can similarly be recursively decomposed. A recursive decomposition of an object into primitive constructors is similar to an *abstract syntax tree* of a string in a language defined by a grammar. If an object is a non-singleton equivalence class, then it will not have a unique recursive decomposition.

Definition 5.43 (Primitive Constructors, p.c.d.)**.** Let $\approx$ be an $\mathsf{Arrangement}$-Equivalence.

- A *primitive constructor* is an object O such that $\mathsf{isub}(O) = \{\Box\}$. We use c to range over primitive constructors.
- A*primitive constructor decomposition* (p.c.d.) of O is a pair $(c, \vec{O})$ such that c is a primitive constructor, $\vec{O}$ is a sequence of objects and $O = c\vec{O}^{\approx}$. A p.c.d. is related to a decomposition as follows: each p.c.d, of O is an element of the set $\overbrace{O}^{\mathsf{dc}}$, but not every element of the $\overbrace{O}^{\mathsf{dc}}$ is a p.c.d. (it may start with an arrangement in which one of the holes are filled).
- We give the following syntax for the set of primitive constructor decompositions: Let p.c.d.$(O) = \{(c, \vec{O}) \mid c \in \mathsf{Object}, \mathsf{isub}(c) = \{\Box\}$ and $O = c\vec{O}^{\approx}\}$. [26]

$\Box$

The next lemma shows that p.c.d.s exist for $\approx$-well-formed objects.

Lemma 5.44. *Let* $\approx$ *be an* $\mathsf{Arrangement}$*-Equivalence which preserves* $\approx$*-well-formedness and let* O *be* $\approx$*-well-formed.*

1. *If there is an* $A \in O$ *such that* $(A', \vec{O}) \in \overbrace{A}^{\mathsf{dc}}$ *and* $\mathsf{isub}(A') = \{\Box\}$*, then* $([A']_{\approx}, \vec{O})$ *is a primitive constructor decomposition (* p.c.d.*) of object* O*.*
2. p.c.d.$(O) = \overbrace{O}^{\mathsf{dc}} \cap \{(\{A\}, \vec{O}) \mid \Box \in \mathsf{isub}(A)\}$.

Proof. 1. By Lemma 5.34, $\mathsf{isub}(A') = \{\Box\}$. Since the arity of $A' \geq 1$ then $[A']_{\approx} = \{A'\}$ and $\mathsf{isub}([A']_{\approx}) = \{\Box\}$. By definition 5.43, $([A']_{\approx}, \vec{O}) \in \overbrace{O}^{\mathsf{dc}}$ and hence by Lemma 5.39,

[26]Every well formed non-hole object $O = \{A_1, \ldots, A_n\}$ where $n \geq 1$ which is an equivalence class of $\approx$ can be decomposed into a primitive constructor and the sub-objects to be placed in its holes, as long as for each $1 \leq i \leq n$, $\overbrace{A_i}^{\mathsf{dc}} = (A'_i, \vec{O})$ where $\mathsf{isub}(A'_i) = \{\Box\}$. This depends crucially on O being an equivalence class of $\approx$, so that any one of the object's arrangements can be chosen and the whole object can be recovered by filling the holes.

$O = [A']_\approx \vec{O}^\approx$ and $([A']_\approx, \vec{O})$ is a p.c.d. of O.

2. Assume $(\{A\}, \vec{O}) \in \overbrace{O}^{\mathsf{dc}} \cap \{(\{A\}, \vec{O}) \mid \Box \in \mathsf{isub}(A)\}$. By Lemma 5.39, $\{A\}$ is an object and by Lemma 5.39. $O = \{A\}_\approx \vec{O}^\approx$. Since $(\{A\}, \vec{O}) \in \overbrace{O}^{\mathsf{dc}}$, then $(A, \vec{O}) \in \overbrace{A'}^{\mathsf{dc}}$ for some $A' \in \overbrace{A}^{\mathsf{dc}}$. Since $\Box \in \mathsf{isub}(A)$ then $\Box \in \mathsf{isub}(\{A\})$ and by Lemma 5.34, $\mathsf{isub}(A) = \{\Box\}$. Hence, $(\{A\}, \vec{O}) \in \text{p.c.d.}(O)$.

Conversely, assume $(c, \vec{O}) \in \text{p.c.d.}(O)$ $\Box$

Example 5.45. • Primitive constructors (derived from (Chang & Felleisen 2012, p 134), (Tobisawa 2015, p. 386), (Inoue & Taha 2012, p. 360), Rahli et al. (2017)) include:

$$\boxed{(\Box\,\Box)} \qquad \boxed{{}^{\Box}\!\downarrow \Box \cdot \Box} \qquad \boxed{!\Box} \qquad \boxed{\langle \Box \rangle} \qquad \boxed{\Box = \Box \in \Box}.$$

• Some examples of recursive decomposition of an object into primitive constructors can already be seen in Examples 5.26 and 5.41.

• Note that $(O_1 + O_2) + O_3$ is a shorthand for $[P_{[P_{O_1} + P_{O_2}]_\approx} + P_{O_3}]_\approx$ and can be decomposed as $[P_\Box + P_\Box]_\approx [[P_\Box + P_\Box]_\approx [O_1, O_2]^\approx, O_3]^\approx$ which can be written as
$(\Box + \Box)[(\Box{+}\Box)[O_1, O_2], O_3]^\approx$. That is,
$(O_1 + O_2) + O_3 = (\Box + \Box)[(\Box{+}\Box)[O_1, O_2]^\approx, O_3]^\approx$.

Convention 5.46 avoids the need to write $[\,\cdot\,]_\approx$ by implicitly invoking $[\,\cdot\,]_\approx$ at obvious arrangement boundaries and at most uses of $(\cdot)$ in arrangements. E.g., $\overbrace{(O_1 @ O_2) @ O_3}^{\mathsf{dc}} = \{(c_@, [c_@[O_1, O_2]^\approx, O_3])\}$ where $c_@ = \Box @ \Box$. Due to the restriction on the allowed positions for parentheses in proper arrangements, this expression does *not* stand for a (hypothetical) object with p.c.d. $c'[O_1, O_2, O_3]$ where $c' = (\Box @ \Box) @ \Box$, because $(\Box @ \Box) @ \Box$ is an improper arrangement, so by Convention 5.46 it forms a context that is not a primitive constructor.

But Convention 5.46 is not enough. We also want to infer uses of $[\,\cdot\,]_\approx$ in other places in the middle of what appear to be arrangements. E.g., if we have already used the arrangement $O @ O'$ somewhere in the text, then we want to infer that $O_1 @ O_2 @ O_3$ stands for the same object as $(O_1 @ O_2) @ O_3$. We want $[\,\cdot\,]_\approx$ in other places in the middle of what appear to be arrangements. We want $O_1 @ O_2 @ O_3$ to *not* stand for the object whose p.c.d. is $c''[O_1, O_2, O_3]$ where $c'' = (\Box \mathbin{@} \Box \mathbin{@} \Box)$. This is so that arrangements declared in the text can help divide objects into complex parse trees, rather than assuming a flat structure every time. For this, we build parsing mechanisms for declaring primitive constructors and parsing arrangements. We adapt operator precedence and declared associativity to allow splitting what appears as a single primitive constructor into multiple primitive constructors. Remember that every arrangement consists of core arrangements (symbols, objects, numbers, or overlined arrangements), arranged in a finite number of positions.

An arrangement A' can be *spliced* into A'' by inserting the main core arrangement sequence of A' into one of the core arrangement sequences of A'' in place of an occurrence of $\Box$.

Convention 5.46 (Declaring and Parsing Primitive Constructors)**. 1.** Unless prevented by part 2 below, at the first use of an A, if there are A', $\vec{O}$ such that $\Box \in \mathsf{isub}(A')$ and $\overbrace{A}^{\mathsf{dc}} = \{(A', \vec{O})\}$ then this use of A *declares* the primitive constructor $c = \{A'\} = [A']_{\approx}$ and the arrangement A'.[27]

2. We define what it means to coerce an arrangement A into an object O. Whenever we coerce an arrangement A into an object O using Convention 5.46, the arrangement A is inspected to see if it can be built by splicing already declared arrangements. If A can be built entirely by splicing together already-declared arrangements, and filling the holes in the splicing result in objects, and there is no explicit indication forbidding the use of this convention, then A is to be interpreted as though it had been written with uses of $[\,\cdot\,]_{\approx}$ around each splice point. If there is more than one way A can be built by splicing already-declared arrangements, then it must be specified somewhere which one to choose.[28]

Example 5.47. • $\overbrace{\langle P_{O_1}\rangle}^{\mathsf{dc}} = \{(\langle P_{\Box}\rangle, [O_1])\}$. Using Convention 5.46, $\overbrace{\langle O_1\rangle}^{\mathsf{dc}} = \{(\langle\Box\rangle, [O_1])\}$.

By Convention 5.46, arrangement $\langle O_1\rangle$ declares arrangement $\langle\Box\rangle$ and primitive constructor $[\langle\Box\rangle]_{\approx}$.[29]

[27]Recall Lemma 5.44 which states that A' differs from A exactly in having $\Box$ in place of every non-$\Box$ object appearing in A.

[28]This choice typically involves notions of operator precedence and declarations of associativity.

[29]The notation $\langle\cdot\rangle$ here is just an arbitrary choice of syntactic function.

- $\overbrace{!P_{O_2}}^{\mathsf{dc}} = \{(!P_\Box, [O_2])\}$. Using Convention 5.46, $\overbrace{!O_2}^{\mathsf{dc}} = \{(!\Box, [O_2])\}$. By Convention 5.46, arrangement $!O_2$ declares arrangement $!\Box$ and primitive constructor $[!\Box]_\approx$.

- By Convention 5.46, arragement $\langle !O' \rangle$ can be coerced into object $[\langle !O' \rangle]_\approx$. Inspecting $\langle !O' \rangle$, we see that it can be built by splicing already declared arrangements $!\Box$ and $\langle \Box \rangle$. More specifically,[30] $\langle !O' \rangle$ can be built by splicing $!\Box$ into $\langle \Box \rangle$ and filling the hole with O'.[31]

- Take $O_1 @ O_2$ which declares primitive constructor $c_@ = \Box @ \Box$. If we state that $c_@$ is left-associative, then writing $O = O_1 @ O_2 @ O_3$ produces the same result as writing $O = (O_1 @ O_2) @ O_3$. Without associativity of $c_@$, writing $O = O_1 @ O_2 @ O_3$ would be an error, because there are multiple distinct ways the arrangement $\Box @ \Box @ \Box$ can be built by splicing the arrangement $\Box @ \Box$ into itself.

Convention 5.46 allows syntactic equivalences and syntactic boundaries to be inferred from uses of constructors. Note that parsing can happen in clauses of syntax production rules and that occurrences of arrangements in syntax production rules are not necessarily declaring occurrences.

Example 5.48. The top line gives objects derived from (Chang & Felleisen 2012, p 134), Rahli et al. (2017), (Tobisawa 2015, p 386). They may not be well formed, as the singleton sets may not be $\approx$-equivalence classes, as they are singleton sets. The objects under them are adjusted to be well formed (when $O_1 \cdots O_4$ are also adjusted):

$$\begin{array}{ccc} \boxed{\{\lambda P_{O_1}.P_{O_2}\}} & \boxed{\{\Pi P_{O_1} : P_{O_2}.P_{O_3}\}} & \boxed{\{{}^{P_{O_1}} \downarrow \{P_{O_2}\, P_{O_3}\} \cdot P_{O_4}\}} \\ [\lambda P_{O_1}.P_{O_2}]_\approx & [\Pi P_{O_1} : P_{O_2}.P_{O_3}]_\approx & [{}^{P_{O_1}} \downarrow [P_{O_2}\, P_{O_3}]_\approx \cdot P_{O_4}]_\approx \end{array}$$

5.1.7 α-Equivalence, Names and Binding

The relation $\approx$ gives a mechanism for working with syntax modulo equivalences on arrangements. An important equivalence is *α-conversion* which renames *bound names.*

[30]By Convention 5.46 this gives the same result as writing $O = \langle (!O') \rangle$.

[31]If we wanted to avoid the interpretation of Convention 5.46, we could do so by avoiding the implicit coercion of Convention 5.46 and writing instead $O = [\langle !O' \rangle]_\approx$, which would use the primitive constructor $\langle !\Box \rangle$ instead of the two smaller primitive constructors $\langle \Box \rangle$ and $!\Box$.

We do not give a complex notion of binding here. We are only interested in providing a concept of binding that can be readily grasped and is sufficiently general for wide use in a variety of grammars. [32] The notion of α-*conversion* is the basis of *higher-order* syntax, such as in the λ-calculus.

To define α-equivalence we need some notion of swapping objects inside a binder for objects of a similar kind. In MathSyn, we use *names.* Any countable number of names may belong to the same group.

Definition 5.49 (Free/Bound/Swapping Names/Closed under Swapping)**.**

- *Names/groups of names* are given by an equivalence relation $\sim \subset \mathsf{Object} \times \mathsf{Object}$ relating names in the same group.
  - Object O is a *name* iff $O \sim O$. Let $\mathsf{Name} = \{O \in \mathsf{Object} \mid O \sim O\}$. We use O_{x}, O_{y}, $\cdots$ to range over Name.
  - If $S \subset \mathsf{Object}$ is an $\sim$-equivalence class i.e., $O_1 \sim O_2$ for all $O_1, O_2 \in S$, and no other names are related by $\sim$ to members of S, we call S a *name group.*
  - The relation $\sim$ is extended incrementally with declarations of groups.
  - Objects not declared related by $\sim$ are *not* related by $\sim$.
  - We require that $\square \notin \mathsf{Name}$.
  - We require that no name occurs as a sub-object inside another name. I.e., if $O \in \mathsf{Name}$, $O' \in \mathsf{Name}$ and $A \in O'$, then O does not occur anywhere in A.
- For names O_{x} and O_{y} where $O_{\mathsf{x}} {\sim} O_{\mathsf{y}}$, $\mathsf{swap}(O_{\mathsf{x}}, O_{\mathsf{y}}, O)$ (resp. $\mathsf{swap}(O_{\mathsf{x}}, O_{\mathsf{y}}, A)$) is the object (resp. arrangement) obtained by simultaneously changing all occurrences of O_{x} resp. O_{y} in O (resp. *Arrangement*) by O_{y} resp. O_{x}.

  An $\mathsf{Arrangement}$-Equivalence $\approx$ is closed under swapping if whenever $\heartsuit \approx \heartsuit'$ then $\mathsf{swap}(O_{\mathsf{x}}, O_{\mathsf{y}}, \mathsf{FN}(\heartsuit)) \approx (O_{\mathsf{x}}, O_{\mathsf{y}}, \heartsuit')$.

[32] The notion of equivalence we provide is intended to be used in defining other syntactic equivalences in addition to α-equivalence. E.g., suppose we wanted to declare that the primitive constructor $(\square \mid \square)$ is equivalent up to reordering when the holes are filled with $u \in U$, we can write $(u_1 \mid u_2) \approx (u_2 \mid u_1)$. We allow other equivalences to be declared in this way.

- A primitive constructor c can be declared to *bind* a name placed in one of its constructor holes across some of its constructor's holes.
- We define $\mathsf{FN}(O)$, the *free names* of object O by:
  - If O is a name, then $\mathsf{FN}(O) = \{O\}$.
  - If $O = \Box$, then $\mathsf{FN}(O) = \emptyset$.
  - Otherwise if O is not a name,
    - define the free names of primitive constructor decompositions (p.c.d.'s) of O: if $(c, \vec{O})$ is one such p.c.d., and for every $O' \in \vec{O}$, $B_{cO'}$ are the names bound by c in O' then $\mathsf{FV}(c, \vec{O}) = \bigcup_{O' \in \vec{O}} \mathsf{FN}(O') \setminus B_{cO'}$;[33]
    - if there exists a set S s.t. $S = \mathsf{FV}(c, \vec{O})$ for every p.c.d. $(c, \vec{O})$ of O, then $\mathsf{FN}(O) = S$ else $\mathsf{FN}(O) = \emptyset$.
- The free names of an arrangement A, are defined as follows:
  - If $A = P_O$ then $\mathsf{FV}(A) = \mathsf{FN}(O)$ else if there is A', $\vec{O}$ such that $\overbrace{A}^{\mathsf{dc}} = \{(A', \vec{O})\}$ and $\Box \in \mathsf{isub}(A')$ then $\mathsf{FV}(A) = \mathsf{FV}(\{A\})$.
  - Else, $\mathsf{FV}(A) = \emptyset$.
- A name that is not free is *bound*.

$\Box$

For names O_{x} and O_{y} where $O_{\mathsf{x}} \sim O_{\mathsf{y}}$, and object O (resp. arrangement A) we have:

- $\mathsf{swap}(O_{\mathsf{x}}, O_{\mathsf{y}}, O_{\mathsf{x}}) = O_{\mathsf{y}}$
- $\mathsf{swap}(O_{\mathsf{x}}, O_{\mathsf{y}}, O_{\mathsf{y}}) = O_{\mathsf{x}}$
- If $A \neq \epsilon$ then $\mathsf{swap}(O_{\mathsf{x}}, O_{\mathsf{y}}, \overline{A}) = \overline{\mathsf{swap}(O_{\mathsf{x}}, O_{\mathsf{y}}, A)}$.

[33]Note that FV refers only to one p.c.d. of O. It is equivalent to FN only in the case where for any $(c_1, \vec{O}_1), (c_2, \vec{O}_2) \in$ p.c.d.(O), $\mathsf{FV}(c_1, \vec{O}_1) = \mathsf{FV}(c_2, \vec{O}_2)$

- $\mathsf{swap}(O_\mathsf{x}, O_\mathsf{y}, P_O) = P_{\mathsf{swap}(O_\mathsf{x}, O_\mathsf{y}, O)}$.
- $\mathsf{swap}(O_\mathsf{x}, O_\mathsf{y}, \{A\}) = \{\mathsf{swap}(O_\mathsf{x}, O_\mathsf{y}, A)\}$.
- $\mathsf{swap}(O_\mathsf{x}, O_\mathsf{y}, \mathsf{FN}(\heartsuit)) = \heartsuit$ if $\heartsuit \in \{\epsilon, \Box, s, n\}$.
- $\mathsf{swap}(O_\mathsf{x}, O_\mathsf{y}, \hat{A}) \in \mathsf{C} - \mathsf{Arrangement}$.
- For core arrangement $\hat{A}$ and non-empty arrangements A, A_1 and A_2,
  - $\mathsf{swap}(O_\mathsf{x}, O_\mathsf{y}, A\hat{A}) = \mathsf{swap}(O_\mathsf{x}, O_\mathsf{y}, A)\mathsf{swap}(O_\mathsf{x}, O_\mathsf{y}, \hat{A})$.
  - $\mathsf{swap}(O_\mathsf{x}, O_\mathsf{y}, A^{A_1}) = \mathsf{swap}(O_\mathsf{x}, O_\mathsf{y}, A)^{\mathsf{swap}(O_\mathsf{x}, O_\mathsf{y}, A_1)}$.
  - $\mathsf{swap}(O_\mathsf{x}, O_\mathsf{y}, A_{A_1}) = \mathsf{swap}(O_\mathsf{x}, O_\mathsf{y}, A)_{\mathsf{swap}(O_\mathsf{x}, O_\mathsf{y}, A_1)}$.
  - $\mathsf{swap}(O_\mathsf{x}, O_\mathsf{y}, A^{A_1}_{A_2}) = \mathsf{swap}(O_\mathsf{x}, O_\mathsf{y}, A)^{\mathsf{swap}(O_\mathsf{x}, O_\mathsf{y}, A_1)}_{\mathsf{swap}(O_\mathsf{x}, O_\mathsf{y}, A_2)}$.
- $\mathsf{swap}(O_\mathsf{x}, O_\mathsf{y}, O) = \{\mathsf{swap}(O_\mathsf{x}, O_\mathsf{y}, A) \mid A \in O\}$.

Note $\mathsf{swap}(O_\mathsf{x}, O_\mathsf{y}, A)$ is an arrangement and $\mathsf{swap}(O_\mathsf{x}, O_\mathsf{y}, O)$ is an object.

We prove some facts about when free variables or free names are preserved and that they are well defined for $\approx$-well formed objects or arrangements.

Lemma 5.50. *1. If $\heartsuit \in \{\Box, \epsilon, s, n\}$ then $\mathsf{FN}(\heartsuit) = \emptyset$.*

2. If $A = P_O$ then $\mathsf{FV}(A) = \mathsf{FN}(O)$.

3. If $A \neq \epsilon$ then $\mathsf{FV}(\overline{A}) = \mathsf{FV}(A)$, provided that, for $\overbrace{A}^{\mathsf{dc}} = \{(A', \vec{O})\}$ we have $\mathsf{FV}(\mathsf{fill}_\approx(\overline{A'}, \vec{O})) = \mathsf{FV}(\mathsf{fill}_\approx(A', \vec{O}))$[34].

4. If O (resp. A) is $\approx$-well-formed then $\mathsf{FN}(O)$ (resp. $\mathsf{FV}(A)$) is well defined.

Proof. 1. If $\heartsuit \in \{\epsilon, s, n\}$ then $\heartsuit \neq P_O$ and there are no A', $\vec{O}$ such that $\overbrace{\mathsf{FN}(\heartsuit)}^{\mathsf{dc}} = \{(A', \vec{O})\}$ and $\Box \in \mathsf{isub}(A')$. Hence $\mathsf{FN}(\heartsuit) = \emptyset$. By definition, $\mathsf{FV}(\Box) = \emptyset$.
2. By definition, $\mathsf{FV}(P_O) = \mathsf{FN}(O)$.

[34] I.e. $\overline{A'}$ does not bind anything which A' does not and vice versa.

3. $\mathsf{FV}(\overline{A}) = \mathsf{FV}(\{\overline{A}\}) = \mathsf{FV}(\mathsf{fill}_\approx(\overline{A'}, \vec{O})) = \mathsf{FV}(\mathsf{fill}_\approx(A', \vec{O})) = \mathsf{FV}(\{A\}) = \mathsf{FV}(A)$.

4. If O is either a name or $\Box$ then this is trivial. If there is no A', $\vec{O}$ such that $\overbrace{A}^{\mathsf{dc}} = \{(A', \vec{O})\}$ and $\Box \in \mathsf{isub}(A')$ then this is likewise trivial. If O or A do not contain names anywhere in their construction then likewise this is trivial. Otherwise we prove it inductively on p.c.d.s of O (or $\{A\}$). Suppose O (or $\{A\}$) is the first such that FN (or FV) is well defined on its immediate sub-objects, but $\mathsf{FN}(O)$ (or $\mathsf{FV}(A)$) is not well-defined. Consider each $(c, \vec{O})$ such that $(c, \vec{O})$ is a p.c.d. of O (or $\{A\}$). For each of these $\mathsf{FV}(c, \vec{O}) = \bigcup_{O' \in \vec{O}} \mathsf{FN}(O') \setminus B_{cO'}$ exists. So $\mathsf{FN}(O)$ (or $\mathsf{FV}(A)$) exists. $\Box$

Example 5.51. Assume that O_{x}, O_{y} and O_{z} are names.

- If $O_{\mathsf{x}} \sim O_{\mathsf{y}}$ then
  - $\mathsf{swap}(O_{\mathsf{x}}, O_{\mathsf{y}}, [\lambda P_{O_{\mathsf{x}}}.P_{O_{\mathsf{y}}}]_\approx) = [\lambda P_{O_{\mathsf{y}}}.P_{O_{\mathsf{x}}}]_\approx$.
  - $\mathsf{swap}(O_{\mathsf{x}}, O_{\mathsf{y}}, [\lambda P_{O_{\mathsf{x}}}.(P_{O_{\mathsf{y}}}@P_{O_{\mathsf{x}}})]_\approx) = [\lambda P_{O_{\mathsf{y}}}.(P_{O_{\mathsf{x}}}@P_{O_{\mathsf{y}}})]_\approx$.
- Consider $c_\lambda = [\lambda P_\Box.P_\Box]_\approx$ of arity 2. Suppose we declare that c_λ binds any name placed in its first hole in both of its holes. Since $\overbrace{[\lambda P_{O_{\mathsf{x}}}.P_{O_{\mathsf{x}}}]_\approx}^{\mathsf{dc}} = \{(c_\lambda, [O_{\mathsf{x}}, O_{\mathsf{x}}])\}$, we have $\mathsf{FV}([\lambda P_{O_{\mathsf{x}}}.P_{O_{\mathsf{x}}}]_\approx) = \emptyset$.
- Since $\overbrace{[\lambda P_{O_{\mathsf{x}}}.P_{O_{\mathsf{y}}}]_\approx}^{\mathsf{dc}} = \{(c_\lambda, [O_{\mathsf{x}}, O_{\mathsf{y}}])\}$, we have $\mathsf{FN}([\lambda P_{O_{\mathsf{x}}}.P_{O_{\mathsf{y}}}]_\approx) = \{O_{\mathsf{y}}\}$.
- Suppose that we have not declared any bindings for the constructor $c_@ = [P_\Box @ P_\Box]_\approx$. Then since $\overbrace{[P_{O_{\mathsf{x}}}@P_{O_{\mathsf{y}}}]_\approx}^{\mathsf{dc}} = \{(c_@, [O_{\mathsf{x}}, O_{\mathsf{y}}])\}$, we have $\mathsf{FN}(P_{O_{\mathsf{x}}}@P_{O_{\mathsf{y}}}]_\approx) = \{O_{\mathsf{x}}, O_{\mathsf{y}}\}$. Since $\overbrace{[\lambda P_{O_{\mathsf{x}}}.(P_{O_{\mathsf{x}}}@P_{O_{\mathsf{y}}})]_\approx}^{\mathsf{dc}} = \{([\lambda P_\Box.P_\Box]_\approx, [O_{\mathsf{x}}, [P_{O_{\mathsf{x}}}@P_{O_{\mathsf{y}}}]_\approx])\}$, we have $\mathsf{FN}([\lambda P_{O_{\mathsf{x}}}.(P_{O_{\mathsf{x}}}@P_{O_{\mathsf{y}}})]_\approx) = \{O_{\mathsf{y}}\}$.
- Since $\overbrace{[(\lambda P_{O_{\mathsf{x}}}.(P_{O_{\mathsf{x}}}@P_{O_{\mathsf{y}}}))@O_{\mathsf{z}}]_\approx}^{\mathsf{dc}} = \{(c_@, [[\lambda P_{O_{\mathsf{x}}}.(P_{O_{\mathsf{x}}}@P_{O_{\mathsf{y}}})]_\approx, O_{\mathsf{z}}])\}$, we have:

  $\mathsf{FN}[(\lambda P_{O_{\mathsf{x}}}.(P_{O_{\mathsf{x}}}@P_{O_{\mathsf{y}}}))@O_{\mathsf{z}}]_\approx) = \{O_{\mathsf{y}}, O_{\mathsf{z}}\}$.
- Consider $c_{\mathsf{let}} = [\mathsf{let} P_\Box = P_\Box \mathsf{in} \Box]_\approx$ of arity 3. If we declare c_{let} binds any name placed in its 1st hole in its 1st and 3rd hole, then since $[\mathsf{let} P_{O_{\mathsf{x}}} = P_{O_{\mathsf{z}}} \mathsf{in} P_{O_{\mathsf{x}}}@P_{O_{\mathsf{y}}}]_\approx =$

$(c_{\mathsf{let}}, [O_{\mathsf{x}}, O_{\mathsf{z}}, [P_{O_{\mathsf{x}}}@P_{O_{\mathsf{y}}}]_{\approx}])$ we have:

$$\mathsf{FN}([\mathsf{let}O_{\mathsf{x}} = P_{O_{\mathsf{z}}}\mathsf{in}P_{O_{\mathsf{x}}}@P_{O_{\mathsf{y}}}]_{\approx}) = \{O_{\mathsf{y}}, O_{\mathsf{z}}\}.$$

A primitive constructor c of the form $\{A\Box\}$ that binds a name in its rightmost hole is a *prefix binder*.

Convention 5.52 (Parsing of Prefix Binders)**.** Unless otherwise specified, for a prefix binder $c = \{A\Box\}$, the connection between the hole $\Box$ and the rest of the arrangement A is parsed with lower precedence than all non-binding constructors (usually, the scope extends as far to the right as allowed by parentheses). I.e., when we recursively decompose an object into a tree of primitive constructors, prefix binders appear as high up the resulting tree as permissible by parentheses.

Definition 5.53 implies that $\approx$ will change whenever adjustments are made to the declared bindings of primitive constructors or to the definition of $\sim$.

Definition 5.53 (α-Conversion as a Syntactic Equivalence)**.** • Let $\equiv_\alpha$ be the smallest equivalence relation such that: for all O_{x}, O_{y}, O, and A, if $O_{\mathsf{x}} \sim O_{\mathsf{y}}$ and $\{O_{\mathsf{x}}, O_{\mathsf{y}}\} \cap \mathsf{FN}(O) = \{O_{\mathsf{x}}, O_{\mathsf{y}}\} \cap \mathsf{FN}(A) = \emptyset$, then $O \equiv_\alpha \mathsf{swap}(O_{\mathsf{x}}, O_{\mathsf{y}}, O)$ and $A \equiv_\alpha \mathsf{swap}(O_{\mathsf{x}}, O_{\mathsf{y}}, A)$. • If a paper says that it is "working modulo α" or "identifying α-equivalent terms" that means $\equiv_\alpha$ restricted to arrangements is a subset of $\approx$, i.e., if $A_1 \equiv_\alpha A_2$ then $A_1 \approx A_2$. $\Box$

If we consider an object, O up to α-equivalence, then $\forall A \in O$, if n is bound in A, no occurrence of n is above $\Box$, and n' is in the same group as n, then, if A' is the result of replacing each occurrence of n in A by n', $A' \approx A$ and $A' \in O$ and all sub-objects of A are also considered up to α-equivalence. If a name occurring in a piece of syntax is not bound, then it is free.

Convention 5.54 (Interpreting unassigned variables)**.** If the set a variable ranges over has no constraints telling us what it contains (such as the set v ranges over above), it is assumed to range over a countable set of names, which is also a group of names and disjoint from any other such set.

We may redefine $\approx$ to be α-equivalence[35] for the grammar in example 5.23 as follows:

Let $\lambda\square.\square$ bind the object placed in the first of its holes in both of its holes. Let $e \in \mathsf{exp}$ be identified up to α-equivalence.

This text is sufficient for MathSyn to be able to recognise an author is working modulo α-equivalence and construct the relevant objects, but we will show how it is interpreted using the hidden machinery of MathSyn.

Names are arrangements consisting only of symbols (not pointers).

In the example above, MathSyn would define α-equivalence as follows: if A is a syntactic arrangement of the form $\lambda n_1.O$ whose first hole is filled with the name n_1 and whose second hole is filled by some object O, then each occurence of n_1 in A is bound. Also, if A is a syntactic arrangement of the form $\lambda n_1.O$ whose first hole is filled with the name n_1 and whose second hole is filled by some object O such that O does not contain $\square$ and such that, $\forall A_1, A_2 \in O$, we have $A_1 \approx A_2$, then let n_2 be a name in the same group as n_1, which is not free in A, but which may be free in O, and let A_1 be the arrangement produced by replacing every instance of n_1 in A with n_2, then $A_1 \approx A$ (i.e. $A_1, A \in [A]_\approx$).

Assuming $\approx$ is the α equivalence relation given above and x, y, and z are in the same group of names, then the object $[\lambda x.x]_\approx$ denotes the syntactic object:

$\{\lambda P_{\{\mathsf{x}\}}.P_{\{\mathsf{x}\}}, \lambda P_{\{\mathsf{y}\}}.P_{\{\mathsf{y}\}}, \lambda P_{\{\mathsf{z}\}}.P_{\{\mathsf{z}\}}, \cdots\}$.[36] Normally, provided they have written the text given in the grey boxes above, notational conventions would allow authors to write variously $\lambda\mathsf{x}.\mathsf{x}$, $\lambda\mathsf{y}.\mathsf{y}$, and $\lambda\mathsf{z}.\mathsf{z}$ and these would be understood as standing for the same syntactic object, unless the author tells us, explicitly or by convention, that x, y, and z are not in the set v ranges over.

[35]Recall from lemma 4.28 equivalences are calculated simultaneously with production rules. They must also be calculated simultaneously with hole filling.

[36]Modelling equivalences as sets of arrangements at syntax boundaries may seem odd to those focused on building parsers that reduce equivalences to a single term. However, MathSyn is interested in a model for human readers looking to reason about articles using MBNF. These include many more equivalences than just α-equivalence. They often define equivalences using standard math notation and do not limit themselves to equivalences that have been translated to a parser friendly form.

We show that name swapping preserves $\approx$-well formedness under any equivalence that is closed under swapping (e.g. one that has α-equivalence amongst its equivalences).

Lemma 5.55. *Let $\approx$ be an* $\mathsf{Arrangement}$*-Equivalence which preserves $\approx$-well-formedness and is closed under swapping. If $\heartsuit$ is $\approx$-well-formed, O_{x} and O_{y} are names such that $O_{\mathsf{x}}{\sim}O_{\mathsf{y}}$ then:*

1. $\mathsf{swap}(O_{\mathsf{x}}, O_{\mathsf{y}}, \heartsuit)$ *is $\approx$-well-formed.*
2. *If $\{O_{\mathsf{x}}, O_{\mathsf{y}}\} \cap \mathsf{FN}(\heartsuit) = \emptyset$ then we have* $\mathsf{FN}(\heartsuit) = \mathsf{FN}(\mathsf{swap}(O_{\mathsf{x}}, O_{\mathsf{y}}, \heartsuit))$.

Proof. 1. By induction on the structure of $\heartsuit$.
2. For any arrangement A, since $\approx$ is closed under name swapping then we know that $[A]_{\approx} = \mathsf{swap}(O_{\mathsf{x}}, O_{\mathsf{y}}, [A]_{\approx})$. Since $\heartsuit$ is $\approx$-well formed then, for some A, $[A]_{\approx} = \heartsuit$, then $\mathsf{FN}(\heartsuit) = \mathsf{FN}(\mathsf{swap}(O_{\mathsf{x}}, O_{\mathsf{y}}, \heartsuit))$. ☐

Example 5.56. If $O_{\mathsf{y}}{\sim}O_{\mathsf{x}}$, then

- $[\lambda P_{O_{\mathsf{x}}}.P_{O_{\mathsf{x}}}]_{\approx} \equiv_{\alpha} [\lambda P_{O_{\mathsf{y}}}.P_{O_{\mathsf{y}}}]_{\approx}$.
- $[\lambda P_{O_{\mathsf{x}}}.P_{O_{\mathsf{z}}}]_{\approx} \equiv_{\alpha} [\lambda P_{O_{\mathsf{y}}}.P_{O_{\mathsf{z}}}]_{\approx}$.
- $[\lambda P_{O_{\mathsf{x}}}.P_{O_{\mathsf{z}}}@P_{O_{\mathsf{x}}}]_{\approx} \equiv_{\alpha} [\lambda P_{O_{\mathsf{y}}}.P_{O_{\mathsf{z}}}@P_{O_{\mathsf{y}}}]_{\approx}$.
- $[\lambda P_{O_{\mathsf{y}}}.(\mathsf{let}O_{\mathsf{x}} = O_{\mathsf{z}}\mathsf{in}O_{\mathsf{x}}@O_{\mathsf{y}})]_{\approx} \equiv_{\alpha} [\lambda P_{O_{\mathsf{x}}}.(\mathsf{Let}O_{\mathsf{y}} = O_{\mathsf{z}}\mathsf{in}[O_{\mathsf{y}}@O_{\mathsf{x}})]_{\approx}$.

5.1.8 Substitution, Sub-Object, Sub-Arrangement

We now show how MathSyn defines $O[O_{\mathsf{x}} := O']$ the *substitution* of all free occurrences of O_{x} in O by O'.[37]

Design Decision 5.57. Here, $O[O_{\mathsf{x}} := O']$ should a) not allow free names in O' to be captured by bindings in O; and b) respect $\approx$: if O and O' are equivalence classes of $\approx$, then so is $O[O_{\mathsf{x}} := O']$.

[37]This is sometimes called capture avoiding substitution to distinguish it from the swapping of objects without avoiding capture by bindings.

Definition 5.58 (Substitution). • For name O_{x}, define $O[O_{\mathsf{x}} := O']$ as follows:

1. If $O = O_{\mathsf{x}}$, then $O[O_{\mathsf{x}} := O'] = O'$.

2. Otherwise, $O[O_{\mathsf{x}} := O']$ is defined as follows.

   First, we must define substitution for primitive constructor decompositions (p.c.d.'s). Given $O = c[O_1, \dots, O_n]$, let S be the subset of $\{O_1, \dots, O_n\}$ of names bound by this occurrence of c. If $S \cap \mathsf{FN}(O') \neq \emptyset$ or ($O_{\mathsf{x}} \in S$, and $O' \in \mathsf{FN}(O)$), then let $(c, [O_1, \dots, O_n])[O_{\mathsf{x}} := O']$ be undefined[38] (This prevents us from substituting names that are free in $O_1, ..., O_n$, but would be bound by c. This prevents us, e.g., from replacing x with y in $\lambda x.\lambda y.xy[x := y]$). Otherwise, let $(c, [O_1, \dots, O_n])[O_{\mathsf{x}} := O'] = c[O_1[O_{\mathsf{x}} := O'], \dots, O_n[O_{\mathsf{x}} := O']]$.

   If $\exists O''$ such that $O'' = (c, \vec{O})[O_{\mathsf{x}} := O']$ for all $c, \vec{O}$, s.t. both $(c, \vec{O})$ is a p.c.d. of O and $(c, \vec{O})[O_{\mathsf{x}} := O']$ is defined, then $O[O_{\mathsf{x}} := O'] = O''$. Otherwise $O[O_{\mathsf{x}} := O']$ is undefined.[39]

$\square$

Definition 5.59 (Sub-object, Sub-arrangement). • O' is a *sub-object* of an object O iff ($O' = [O']_{\approx}$ and either (($\exists O_1$, O_2, O_3, O_4 such that ($O_1 \neq O$ and $O_3 = O_1[O_2 := O_4]$ and ($\forall A \in O$, $\forall A' \in O_3$, $A' \approx A$) and ($\forall A \in O'$, $\forall A' \in O_4$, $A' \approx A$))) or (O' is bound in O and any instances of O' appearing in the same $\{O\}-\mathsf{Context}$ are written using a "literal" arrangement[40] which is coerced into O'))).

• O' is a sub-object of an arrangement A if O' is a sub-object of $\{A\}$

• A' is a *sub-arrangement* of an object O if $\exists O'$ such that O' is a sub-object of O and $A' \in O'$.

[38] For simplicity, we do not check whether the substitution needs only to proceed into holes of c which are not subject to its bindings. This will behave well enough for our uses provided each group of names is big enough that fresh names can be found.

[39] So the substitution must be defined for at least one of the p.c.d.s to get a defined result.

[40] By *literal arrangement* we mean one that is equal to one which has been coerced up to arrangement equivalence. We use "equal to" here, rather than saying it is been coerced that way, as literal arrangements may be used in other syntax coercions.

• A' is a sub-arrangement of an arrangement A if $\exists O'$ such that O' is a sub-object of A and $A' \in O'$. □

Design Decision 5.60. We use substitution to pick out sub objects rather than extending recursively the notion of immediate sub-object for three reasons. The first is that the notation is more compact this way. The second is that this way works for chains of syntax that are self-referential whereas an extension of immediate sub-object would loop. The third is that this way allows us to treat differently objects appearing as bound variables.

Note that, without an infinite supply of names, $O[O_{\mathsf{x}} := O']$ may not be defined. This is permissible but, given a set S, with no more than countably many objects, we may extend $\sim$ for any name group, G appearing in S. We choose objects disjoint from any other set of objects occurring within the same "context" as G and also disjoint from the set of any sub-objects of these, provided these sub-objects were not in the same position as a "bottom-level" metavariable (i.e. a sub-object ranged over by a declared metavariable with no other sub-objects ranged over by declared metavariables beneath it). By convention 5.54, if a variable is used without its range being declared, we assume it ranges over an appropriate such set of names. Therefore, this is not normally an issue.

5.2 The Model for Object and Arrangement

It may not be obvious to the reader that Object and $\mathsf{Arrangement}$ exist, or how they are modelled. We provide one such model in this section.

Recall we chose to have arrangements contain pointers to objects. This leads to some tricky technical details in the proof Object and $\mathsf{Arrangement}$ exist, since they depend on some choice of function mapping pointers to an uncountable set and this can only be made after the fixed point containing them is constructed. Therefore, each stage of our construction contains every candidate for this function.

We give a sketch illustrating roughly how it works below. Let Symbol be some countable set and $\mathsf{Pointer}$ be some uncountable set. The proof relies on building an ordinal indexed sequence of sets. Readers more well versed in computer science literature than set theory may associate this idea with transfinite step indexing Spies et al. (2021) (our proof does not rely on the example cited).

Theorem 5.61. (Object, $\mathsf{Arrangement}$ and ptr Have a Model). *There exists some selection of* Object, $\mathsf{Arrangement}$ *and* ptr *such that:*

$\forall S \subseteq \mathsf{Arrangement}$ *s.t., S is countable* $S \in \mathsf{Object}$. *Every way of arranging a finite number of symbols and pointers that is allowed in math text is in* $\mathsf{Arrangement}$. *There is a bijection* $\mathsf{ptr} : \mathsf{Pointer} \to \mathsf{Object}$.

Proof. (sketch) We define a sequence of sets, OPA_i, containing closer approximations of Object and $\mathsf{Arrangement}$ one contains a model for Object and $\mathsf{Arrangement}$ themselves. The smallest set in our sequence, OPA_0, contains all triples of:

• The set containing $Ob_0 = \{\Box\}$ (approximating Object).

• An injective function $Pt_0 \in \{\Box\} \to \mathsf{Pointer}$ (approximating ptr).

• $Ar_0 = \mathsf{Symbol}$ (approximating $\mathsf{Arrangement}$).

For $(Ob_i, Pt_i, Ar_i) \in \mathsf{OPA}_i$, let OPA_{i+1} contain all triples of:

• $Ob_{i+1} = Ob_i \cup \{S \mid S \text{ is countable and } S \subseteq Ar_i\}$

• $\{Pt_{i+1} \in Ob_{i+1} \to \mathsf{Pointer} \mid Pt_{i+1} \text{ is injective and } Pt_i \subseteq Pt_{i+1}\}$[41]

• $Ar_i \cup \mathsf{Pt}_i(Ob_i) \cup S = Ar_{i+1}$ where S is the set of every way of positioning one element of $\mathsf{Pt}_i(Ob_i) \cup \mathsf{Ar}_i$ around another in math text.[42]

Before we give the limit case we define what it is to be a "chain" of triples $(Ob_i, Pt_i, Ar_i) \in \mathsf{OPA}_i$. If, for some $(Ob_1, Pt_1, Ar_1) \in \mathsf{OPA}_1$, $(Ob_2, Pt_2, Ar_2) \in \mathsf{OPA}_2$,...,etc., we have each of $Ob_1 \subseteq Ob_2 \subseteq, \ldots,$ etc., $Pt_1 \subseteq Pt_2 \subseteq, \ldots,$ etc., and $Ar_1 \subseteq Ar_2 \subseteq, \ldots,$ etc. then we call these a chain of $(Ob_i, Pt_i Ar_i) \in \mathsf{OPA}_i$.

[41]We prove in lemma 5.67 if Pt_i has an injective function then Pt_{i+1} does.

[42]We can represent this as a triple (x, y, z) where x and y are members of $\mathsf{Pt}_i(Ob_i) \cup \mathsf{Ar}_i$ and z is a member of some countable index set representing positions

For a limit, ε, and a chain, C, of $(Ob_i, Pt_i Ar_i) \in \mathsf{OPA}_i$, s.t. $\forall i < \varepsilon$, we have $(Ob_i, Pt_i Ar_i) \in C$ let OPA_ε contain all triples of:

- $Ob_\varepsilon = \bigcup\{Ob_i \mid (Ob_i, x, y) \in C\}$
- $\{Pt_\varepsilon \in Ob_\varepsilon \to \mathsf{Pointer} \mid Pt_\varepsilon \text{ is injective and } \forall (x, Pt_i, z) \in C, Pt_i \subseteq Pt_\varepsilon\}$[43]
- $(\bigcup\{Ar_i \mid (x, y, \mathsf{Ar}_i) \in C\}) \cup \mathsf{Pt}_\varepsilon(Ob_\varepsilon) \cup S = Ar_\varepsilon$ and S is the set of every way of positioning one element of $\mathsf{Pt}_\varepsilon(Ob_\varepsilon) \cup \bigcup\{Ar_i \mid (x, y, \mathsf{Ar}_i) \in C\}$ around another in math text.

These sets remain sufficiently small to pick mappings for each Ob_i. Further, there is a fixed point for the function mapping each member of this sequence to the member above it. We pick a bijection from this fixed point which itself will give a model for Object and Arrangement as well as ptr. $\square$

Both this definition and its proof sketch are very informal. We include them only to make following the proofs in this section easier. A formal definition is given by definition 5.65. A formal proof is given by theorem 5.68. The particular encoding of positioning, symbols and pointers to objects is arbitrary. We have chosen one that keeps the proof simple. In practice, the specifics of this model need not be used when reading MBNF grammars and we can stick to a general, semi-formal definition like definition 5.4.

Definition 5.62 (Symbol, Pos, Pointer, $\square$, ϵ)**.** The ordinal interval $\mathsf{D} = [\omega \mathinner{..} \omega \cdot 2)$ represents symbols, marking/wrapping and positioning. We use ω to indicate the first countably infinite ordinal. We start at ω to free the natural numbers to stand for themselves. Let $\mathsf{Pos} \subset \mathsf{D}$ and $|\mathsf{Pos}| < \aleph_0$ such that Pos has enough members to represent positions subscript, superscript, text above, text below, etc. This will also be used to represent placing items in a sequence and wrapping an accent symbol around an arrangement in the version of our model used to construct the proof. Let $\square$ and ϵ be two distinct elements of $\mathsf{D} \setminus \mathsf{Pos}$, representing the context hole and the empty arrangement, and let $\mathsf{Symbol} = \mathsf{D} \setminus (\mathsf{Pos} \cup \{\square, \epsilon\})$, representing any symbol one might

[43]We prove in lemma 5.67 that Pt_ε remains of cardinality less than or equal to Pointer and thus choice can be relied on to give us an injection.

draw. Let the interval of ordinals from $\omega \cdot 2$ which are less than the first uncountable ordinal, $\mathsf{Pointer} = [\omega \cdot 2 \mathinner{..} \omega_1)$, represent pointers to objects. $\square$

Definition 5.63 (Constraints)**.** • **Invariant Constraints.** $\mathsf{IC}(Ob, Pt, Ar)$ is true if the following hold:

1. $Pt \in Ob \to \mathsf{Pointer}$ and Pt is injective and total on Ob.
2. $Pt(Ob) \in Ar$.
3. $\square \in Ob \setminus Ar$ and $\epsilon \in Ar$.
4. $\mathsf{Pos} \cap (Ob \cup Ar) = \emptyset$.
5. $x \subset Ar$ for all $x \in Ob$ such that $x \neq \square$.
6. $\mathsf{Symbol} \cup \mathbb{N} \subset Ar$.

• **Constraints at each step.** $\mathsf{STEP}(Ob_0, .., Ob_\varepsilon, Pt_0, ..Pt_\varepsilon, Ar_0, ..Ar_\varepsilon)$ holds if:
$\forall i < \varepsilon$, ($Pt_i \subseteq Pt_\varepsilon$ and $Ar_i \subseteq Ar_\varepsilon$); and
(if $A \in Ar_i$ and $x \in \mathsf{Pos} \to Ar_i \setminus \{\epsilon\}$ and $x \neq \emptyset$ then $(Ar_i, x) \in Ar_\varepsilon$);
and if $S \subset \bigcup_{j<\varepsilon} Ar_j$ and $|S| \leq \aleph_0$ and $S \neq \{p\}$ where $p \in \mathsf{Pointer}$, then $S \in Ob_\varepsilon$.

Some constraints only hold of the sets we build if they are in the fixed point.

• **Final Selection Constraints.** $\mathsf{FS}(Ob, Pt, Ar)$ is true if the following hold:

1. If $A \in Ar$ and $x \in \mathsf{Pos} \to A \setminus \{\epsilon\}$ and $x \neq \emptyset$ then $(A, x) \in Ar$.

2. If $S \subset Ar$ and $|S| \leq \aleph_0$ and $S \neq \{p\}$ where $p \in \mathsf{Pointer}$, then $S \in Ob$.

$\square$

We will show that Object, $\mathsf{Arrangement}$ and ptr such that $\mathsf{FS}(\mathsf{Object}, \mathsf{ptr}, \mathsf{Arrangement})$ and

$\mathsf{IC}(\mathsf{Object}, \mathsf{ptr}, \mathsf{Arrangement})$ exist. First, we establish a lemma.

Lemma 5.64. **1.** *There is a bijection $b \in \alpha \to \alpha \times \alpha$ for each infinite ordinal α.*
2. *For cardinals $|a|$ and $|b|$ where a or b is infinite, $|a \times b| = \mathsf{max}(a, b)$.*
3. *If A is a countable set and B, a set of all trees C s. t. the interior nodes of C are elements of A and the leaf nodes of C are elements of ω_1. We have $|B| \leq \aleph_1$.*

Proof. **1.** We take our sketch of this standard result from a longer proof in Lévy (1979). For ordinals On let P be the function on $\mathsf{On} \times \mathsf{On}$ which is the isomorphism of $(\mathsf{On} \times \mathsf{On}, R)$ on $(\mathsf{On}, <)$, where R is the canonical ordering on $\mathsf{On} \times \mathsf{On}$. This function exists. All that remains to show is $|P(\alpha \times \alpha)| = |\alpha|$. We proceed by induction on the size of $|\alpha|$. If $|P(\alpha \times \alpha)| < |\alpha|$ then $|\alpha \times \alpha| < |\alpha|$ so $|P(\alpha \times \alpha)| \geq |\alpha|$. Suppose for a contradiction $|P(\alpha \times \alpha)| > |\alpha|$ then $|P(\gamma \times \delta)| = |\alpha|$ for some $\gamma, \delta < |\alpha|$. Let η be the successor of $\mathsf{max}(\gamma, \delta)$. Since $|\alpha|$ is a limit ordinal, $\eta < |\alpha|$. We have $|\alpha| \leq |\eta \times \eta|$. If η is finite this cannot hold. If η is infinite let $|\eta| = \xi$. We have $\xi < |\alpha|$. By the induction hypothesis $|P(\eta \times \eta)| = |\eta|$. Hence $|\alpha| \leq \xi$ giving us our contradiction. It follows that b exists.
2. By choice, there is a well ordering on a and b. There is a bijection $b \in \alpha \to \alpha \times \alpha$ for each infinite α. Let $m = \mathsf{max}(a, b)$ then $|m| \leq |a + b| \leq |a \times b| \leq |m \times m|$.
3. $|B| \geq \aleph_1$ as B contains all trees consisting of a single node in ω_1. Let D be the set of finite trees E where $|E| \subset \omega_1$. The injection $A \to \omega_1$ gives an injection $B \to D$, so $|B| \leq |D|$. For $S \subset \omega_1 \wedge S$ finite, $<$ on S is a finite subset of $\omega_1 \times \omega_1$. By 2., $|\omega_1 \times \omega_1| = \aleph_1$ The cardinality of the set of finite subsets of ω_1 is $\aleph_1$ ($|\omega_1 + \omega_1 \times \omega_1 + ...| = |\omega_1 + \omega_1 ...| = |\omega_1 \times \omega| = |\omega_1|$). So $|B| = |D| = \aleph_1$. ☐

For ordinal i we define OPA_i, a step towards the tuple $(\mathsf{Object}, \mathsf{ptr}, \mathsf{Arrangement})$. Each element of OPA_i, $(Ob_i, Pt_i, Ar_i) \in \mathsf{OPA}_i$ is such that $\mathsf{IC}(Ob_i, Pt_i, Ar_i)$ holds. Let f be a function that takes OPA_i to OPA_{i+1} and $\mathsf{OPA}_{\mathsf{fix}}$ be the least fixed point of this function (we will demonstrate that both this function and its least fixed point exists). Then each element of $\mathsf{OPA}_{\mathsf{fix}}$, $(Ob_{\mathsf{fix}}, Pt_{\mathsf{fix}}, Ar_{\mathsf{fix}}) \in \mathsf{OPA}_{\mathsf{fix}}$ is such that $\mathsf{FS}(Ob_{\mathsf{fix}}, Pt_{\mathsf{fix}}, Ar_{\mathsf{fix}})$ holds. We choose this construction rather than defining a single triple, (Ob_i, Pt_i, Ar_i), which is intended to approximate $(\mathsf{Object}, \mathsf{ptr}, \mathsf{Arrangement})$, because otherwise our assignment of pointers to objects throughout would have to depend on the choice of mapping for the fixed point of Object and $\mathsf{Arrangement}$ and we wanted to reassure this readers that relying on a choice of function made at the end was permissible.

Definition 5.65 (OPA$_i$, fun). **•** We define OPA$_i$ by cases:[44]

0 Case: $Ob_0 = \{\square\}$,

[44]The i of Ob_i, Pt_i, Ar_i and Cr_i is just decorative to show which OPA_i they are in. I.e., we do not define these independently of the OPA_i and they may be defined with any tuple from this function.

$ptrSpace_0 = \{x \in Ob_0 \to \mathsf{Pointer} \mid x \text{ is total injective on } Ob_0\}$,

$$\mathsf{OPA}_0 = \left\{ (Ob_0, Pt_0, Ar_0) \left| \begin{array}{l} Pt_0 \in \mathsf{ptrSpace}_0 \\ \wedge Ar_0 = \mathbb{N} \cup \{\epsilon\} \cup \mathsf{Symbol} \cup \{Pt_0(O) \mid O \in Ob_0\} \end{array} \right. \right\}$$

+1 Case: Define $\mathsf{buildOpSC}((Ob_n, Pt_n, Ar_n))$ and $\mathsf{buildA}((Ob_n, Pt_n, Ar_n), Pt_{n+1})$ by:

For $(Ob_n, Pt_n, Ar_n) \in \mathsf{OPA}_n$

Let $\mathsf{Layout}_{n+1} = Ar_n \times \{x \in \mathsf{Pos} \to Ar_n \setminus \{\epsilon\} \mid x \neq \emptyset\}$.

Let $Ob_{n+1} = Ob_n \cup \{x \in \mathcal{P}(Ar_n) \mid |x| \leq \aleph_0 \wedge \forall y \in \mathsf{Pointer},\ x \neq \{y\}\}$.

(note that $\forall i \{Pt_i(O) \mid O \in Ob_i\} \subseteq Ar_i$)

Let $ptrSpace_{n+1} = \{x \in Ob_{n+1} \to \mathsf{Pointer} \mid x \text{ is total on } Ob_{n+1} \wedge x \text{ is injective}\}$. [45]

Let $ptS_{n+1} = \{p \in ptrSpace_{n+1} \mid \forall x \in Ob_n,\ (x, Pt_n(x)) \in p\}$.

For $Pt_{n+1} \in ptS_{n+1}$:

let $Ar_{n+1} = Ar_n \cup \mathsf{Layout}_{n+1} \cup \{Pt_{n+1}(O) \mid O \in Ob_{n+1}\}$.

Then $\mathsf{buildOpSC}((Ob_n, Pt_n, Ar_n)) = (Ob_{n+1}, ptS_{n+1})$.

And $\mathsf{buildA}((Ob_n, Pt_n, Ar_n), Pt_{n+1}) = Ar_{n+1}$.

$$\mathsf{OPA}_{n+1} = \left\{ (Ob_{n+1}, Pt_{n+1}, Ar_{n+1}) \left| \begin{array}{c} (Ob_n, Pt_n, Ar_n) \in \mathsf{OPA}_n \wedge Pt_{n+1} \in ptS_{n+1} \wedge \\ \mathsf{buildOpSC}((Ob_n, Pt_n, Ar_n)) = (Ob_{n+1}, ptS_{n+1}) \\ \wedge \mathsf{buildA}((Ob_n, Pt_n, Ar_n), Pt_{n+1}) = Ar_{n+1} \end{array} \right. \right\}.$$

Limit Case: We now define OPA_i for a limit point ε.

$$\mathsf{stack}_\varepsilon = \left\{ S \subseteq \bigcup_{i<\varepsilon} \mathsf{OPA}_i \left| \begin{array}{l} S \neq \emptyset \wedge \forall (Ob, Pt, Ar) \in \bigcup_{i<\varepsilon} \mathsf{OPA}, \\ \forall (Ob', Pt', Ar') \in S, \\ ((Ob, Pt, Ar) \in S) \Leftrightarrow \begin{array}{l} ((Ob' \subseteq Ob \wedge Ar' \subseteq Ar \wedge Pt' \subseteq Pt) \\ \vee (Ob \subseteq Ob' \wedge Ar \subseteq Ar' \wedge Pt \subseteq Pt')) \end{array} \end{array} \right. \right\}.$$

Note $\mathsf{stack}_\varepsilon$ is the union of the subset of the OPA_i such that pointers are consistently assigned the same objects.

[45]We will prove in the next lemma this is always non-empty.

Let $\mathsf{SbuildOpSC}(S_\varepsilon)$ and $\mathsf{SbuildA}(S_\varepsilon, Pt_\varepsilon)$ be defined as follows:

For $S_\varepsilon \in \mathsf{stack}_\varepsilon$:

$$\begin{aligned}\text{Let } Ob_\varepsilon \;=\;& (\bigcup\{Ob \mid (Ob, a, b) \in S_\varepsilon\}) \cup \\ & \left\{ x \in \mathcal{P}(\bigcup\{Ar \mid (a, b, Ar) \in S_\varepsilon\}) \;\middle|\; \begin{array}{l} |x\,| \leq \aleph_0 \wedge \\ (y \in \mathsf{Pointer} \Rightarrow x \neq \{y\}) \end{array} \right\}.\end{aligned}$$

Let $ptrSpace_\varepsilon = \{x \in Ob_\varepsilon \to \mathsf{Pointer} \mid x \text{ is total on } Ob_\varepsilon \wedge x \text{ is injective}\}$.

Let ptS_ε be the $\{p \in ptrSpace_\varepsilon \mid \forall (Ob, Pt, Ar) \in S_\varepsilon,\ Pt \subseteq p\}$.

For $Pt_\varepsilon \in ptS_\varepsilon$ let $Ar_\varepsilon = \{Pt_\varepsilon(O) \mid O \in Ob_\varepsilon\} \cup \bigcup\{Ar_i \mid (a, b, Ar_i, c) \in S_\varepsilon\}$.

Then $\mathsf{SbuildOpSC}(S_\varepsilon) = (Ob_\varepsilon, ptS_\varepsilon)$ and $\mathsf{SbuildA}(S_\varepsilon, Pt_\varepsilon) = Ar_\varepsilon$.

Now let

$$\mathsf{OPA}_\varepsilon = \left\{ (Ob_\varepsilon, Pt_\varepsilon, Ar_\varepsilon) \;\middle|\; \begin{array}{l} S_\varepsilon \in \mathsf{stack}_\varepsilon \wedge \mathsf{SbuildOpSC}(S_\varepsilon) = (Ob_\varepsilon, ptS_\varepsilon) \\ \wedge Pt_\varepsilon \in ptS_\varepsilon \wedge \mathsf{SbuildA}(S_\varepsilon, Pt_\varepsilon) = Ar_\varepsilon \end{array} \right\}.$$

• Let $Z = \{\mathsf{OPA}_i \mid i < \kappa\}$ for $\kappa < \omega_2$ and $\mathsf{fun} \in Z \to Z$ s.t. $\mathsf{fun}(\mathsf{OPA}_i) = \mathsf{OPA}_{i+1}$ □

Remark 5.66. If we can show each OPA_i is Non-Empty for all Ordinals i, then $\mathsf{IC}(Ob_i, Pt_i, Ar_i)$ and $\mathsf{STEP}(Ob_0, ..., Ob_i, Pt_0, ..., Pt_i, Ar_0, ...Ar_i)$ hold for some for some $(Ob_0, Pt_0, Ar_0) \in \mathsf{OPA}_0$, ..., $(Ob_i, Pt_i, Ar_i) \in \mathsf{OPA}_i$ by construction.

The next result rests on the Knaster–Tarski theorem Tarski (1955) which states: Let $(L\ \leq)$ be a complete lattice and let $f : L \to L$ be an monotonic function (w.r.t. $\leq$). Then the set of fixed points of f in L also forms a complete lattice under $\leq$ (notably there is a least fixed point).

Lemma 5.67. 1. $\mathsf{OPA}_i \neq \emptyset$ *for all Ordinals* i. 2. fun *has a least fixed point.*

Proof. 1. $\mathsf{OPA}_i = \emptyset$ only if $|Ob_i| > |\mathsf{Pointer}|$ for Ob_i s.t. $(Ob_i, a, b) \in \mathsf{OPA}_i$. By induction on the $|Ob_n|$ and $|Ar_n|$ s.t. $(Ob_n, x, Ar_n) \in \mathsf{OPA}_n$ this cannot hold.

0 Case: $|Ob_0| = 1 \leq \aleph_1$ and for all $Ar_0 \in \mathsf{ArrSpace}_0$, $|Ar_0| = \aleph_0 \leq \aleph_1$.

+1 Case: If, $\forall (Ob_n, x, Ar_n) \in \mathsf{OPA}_n$, $|Ob_n| \leq \aleph_1$ and $|Ar_n| \leq \aleph_1$, then $\forall Ob_{n+1}$,

$|Ob_{n+1}| \leq \aleph_1$ (since Ob_i and Ar_i can be ordered as by choice, the cardinality of the set of subsets of $\aleph_1$ which are of cardinality $\leq \aleph_0$ is $(2^{\aleph_0})^{\aleph_0} = 2^{\aleph_0 \cdot \aleph_0} = 2^{\aleph_0}$). Suppose it is possible to choose a Pt_n that assigns a injection between Ob_n and $\mathsf{Pointer}$ we can also choose an injection from $\mathsf{Pointer}$ to a subset S of pointer such that $|\mathsf{Pointer} \setminus S| = |\mathsf{Pointer}|$. Say, for example, given a bijective mapping $b \in \mathsf{Pointer} \rightarrow \kappa$ from $\mathsf{Pointer}$ to an ordinal κ, for each limit ordinal $\gamma \leq \kappa$, we could have $S = \{q \mid b(q) = \gamma + 2 \cdot n\}$. As such there exists Pt_n that can assign a injection from some Ob_n to S. We can define this recursively taking first a composition of the mappings from Ob_n to $\mathsf{Pointer}$ and $\mathsf{Pointer}$ to S and then dropping those Ob_n containing pointers not in this mapping. Since Ob_{n+1} can contain at most $\aleph_1$ objects Ob_n doesn't, there is always a injection between Ob_{n+1} and $\mathsf{Pointer}$, if there is one between Ob_n and $\mathsf{Pointer}$. For Pt_n that's an injective mapping from Ob_n to S we can extend it to Pt_{n+1} by taking an injective mapping $p \in Ob_{n+1} \setminus Ob_n \rightarrow \mathsf{Pointer} \setminus S$ and letting Pt_{n+1} equal $Pt_n \cup p$. Since Layout_{n+1} cannot add cardinality greater than $\aleph_1$, if $|Ob_n| \leq \aleph_1$ and $|Ar_n| \leq \aleph_1$, then $|Ar_{n+1}| \leq \aleph_1$.

Limit Case: Show $\exists \varepsilon, \forall i < \varepsilon$, $(|Ob_i| \leq \aleph_1 \wedge |Ar_i| \leq \aleph_1) \Rightarrow (|Ob_\varepsilon| \leq \aleph_1 \wedge |Ar_\varepsilon| \leq \aleph_1)$. No Ar_i has $\aleph_0$ sub-arrangements and all Ar_i can be identified with a finite tree whose interior nodes are labelled corresponding to the finite number of possible combinations of positions and whose leaf nodes are labelled with members of ω_1. By lemma 5.64, $|\bigcup_{i=0}^{\varepsilon} Ar_i| \leq \aleph_1$. Similarly we identify each set in Ob_i apart from $\Box$ with a countable subset of the set of trees used to define each Ar_i. The cardinality of the countable subsets of a set of size $\aleph_1$ is $(2^{\aleph_0})^{\aleph_0} = 2^{\aleph_0 \cdot \aleph_0} = 2^{\aleph_0}$. So $|\bigcup_{i=0}^{\varepsilon} Ob_i| \leq \aleph_1$. As $Ob_\varepsilon \subseteq Ob_i$, for $i \leq \varepsilon$ there is a Pt which assigns pointers for Ob_ε and can be used to assign pointers for Ob_i.

2. For $\mathsf{OPA}_a, \mathsf{OPA}_b \in Z$ define $\leq$ such $\mathsf{OPA}_a \leq \mathsf{OPA}_b$ iff (either, $\forall (Ob_b, p, q) \in \mathsf{OPA}_b$, $\exists (Ob_a, b, c) \in \mathsf{OPA}_a$ s.t. $Ob_a \subset Ob_b$, or,$\forall (Ob_b, p, Ar_b) \in \mathsf{OPA}_b$,

$\exists (Ob_a, b, Ar_a) \in \mathsf{OPA}_a$ s.t. $(Ob_a = Ob_b$ and $Ar_a \subset Ar_b))$. Z is a complete lattice ordered by $\leq$ and fun is an order preserving function on Z. By Knaster–Tarski theorem Tarski (1955), we are done. The way in which we have defined Ob and Ar is such that $j \leq i$ implies $(Ob_j \subseteq Ob_i$ and $Ar_j \subseteq Ar_i)$. For all limit ordinals $\varepsilon \leq \kappa$; $(Ob_\varepsilon, Ar_\varepsilon) \in Z$. Finally ω_2 is large enough that it has a larger cardinality than any Ob_i or Ar_i, so we can select some κ larger than the partition of Ob_i and Ar_i into the extra elements added at each stage. $\square$

Theorem 5.68. **(Object, $\mathsf{Arrangement}$ and ptr Have a Model).** *There exist* Object*,* $\mathsf{Arrangement}$*, and* ptr *such that* $\mathsf{FS}(\mathsf{Object}, \mathsf{ptr}, \mathsf{Arrangement})$ *and* $\mathsf{IC}(\mathsf{Object}, \mathsf{ptr}, \mathsf{Arrangement})$*.*

Proof. For OPA_i least fixed point of fun, $\mathsf{OPA}_{i+1} = \mathsf{OPA}_i$. Take the least fixed point $\mathsf{lfp}(\mathsf{fun})$ of $\mathsf{fun} \in Z \to Z$ and $(Ob, Pt, Ar) \in \mathsf{lfp}(\mathsf{fun})$. Let $\mathsf{Object} = Ob$, $\mathsf{Arrangement} = Ar$, $\mathsf{ptr} = Pt$. Then $\mathsf{FS}(\mathsf{Object}, \mathsf{ptr}, \mathsf{Arrangement})$ and likewise $\mathsf{IC}(\mathsf{Object}, \mathsf{ptr}, \mathsf{Arrangement})$. $\square$

We have chosen Zermelo Frankel Set theory with the axiom of choice (ZFC) as the metatheory for MathSyn, although it is not preferred by people whose primary focus is parsing and theorem provers, as most mathematicians and computer scientists are familiar with it and is likely to be used in fields adjacent to theoretical computer science where people do not use MBNF, but may have an interest in articles using it. Set theory notation is also heavily used as part of MBNF grammars, which made it a natural choice.

5.3 What Was Covered by This Chapter

In this chapter we defined the concept of syntactic objects and arrangements as they appear in various uses of Computer Science Metalanguage, including, notably for us, MBNF production rules. These provide enough power for authors to consider syntax up to arbitrary countable equivalences, while preserving the idea of a syntax tree and enabling authors to use them on only parts of this tree. We proved that these have a model within set theory. We showed that the universe of syntactic objects can be readily extended to include any other mathematical object the author may wish to define, provided they do not do so for more than $\aleph_1$ such objects.

We also covered what it meant to coerce syntactic arrangements into an equivalence $\approx$ provided by the author. We defined the hole filling operation on syntactic objects and arrangements, showed that it preserves the well-formedness of objects arrangements under $\approx$-equivalence and used it to provide the notion of a context and the compatible closure of a set of objects with respect to a set S. This allows us to define relations that descend into the syntactic objects within a context, such as rewriting relations. We will introduce a solution to a grammar which relies on hole filling as part of the grammar rules in Lemma 7.6.

We introduced the idea of an immediate sub-object and introduced the notion of a primitive constructor. We covered what it means to decompose objects into their primitive constructors. This provides a notation for writing syntactic objects as a number of functions applied to symbols and names. It also provides something comparable to a parse tree for syntactic objects.

We provided an account of names, binding and what it means for an object to be an α-equivalence class. We showed that α-equivalence classes can be well-formed. We used this to give an account of substitution and give the notion of a sub-object and sub-arrangement. This is helpful as authors often take various notions of binding and substitution for granted.

In the next chapter we will show how the notions of object and arrangement given here can be included alongside an equivalence relation and a set of MBNF production rules to pick out a set of syntactic objects.

Chapter 6

Interpreting Production Rules in MathSyn

In this chapter we will describe how MBNF production rules are to pick out objects from the set Object. Due to how many ways production rules are employed we do not define them formally from the outset. Instead we start with an informal definition and look at one way in which the interpretation of production rules may be formalised to cover most use cases. In Section 6.1 we give an account of what may be included in an MBNF production rule and give an informal account of how a set of MBNF production rules is read that does not include an account of how to tell if a set of production rules defines something. We also cover some of the conventions that generally operate while reading production rules and discuss the effect different conventions of indexing may have on how production rules are read. In Section 6.2 we give one general approach for determining whether a set of MBNF production rules has a solution.

6.1 What is an MBNF Production Rule

Expressions appear in MBNF production rules. We give an account of what an expression is and what it means to evaluate one. Expressions may be pieces of syntax with metavariables which are replaced in the process of evaluating them or they may be other operations taken from maths, although we will consider them only as syntax which may evaluate to different syntax.

Definition 6.1 ((Evaluating) and (Part-Evaluating) Expressions)**.** • Expressions are objects (i.e., elements of Object), which need not be well formed. Instead of adjusting expressions to be equivalence classes of $\approx$, we are allowed to use any object they evaluate to in their place.

• Expressions may "Evaluate" to other expressions or to objects. In order to define what it means for an expression to evaluate to another expression or object, we first define what it means for an expression to *part-evaluate* to another. Expression X part-evaluates to expression Y iff either Y is X with one instance of a metavariable replaced by an object it ranges over; or Y is X after a mathematical operation defined outside of this document has been computed;[1] or X part-evaluates to Z and Z part-evaluates to Y.

• Expression X evaluates to expression Y iff either (X part-evaluates to Y and Y doesn't part-evaluate to anything) or ($X = Y$ and X doesn't part-evaluate to anything). □

The *default range constraint* for a metavariable v states that v ranges over S if $v \in S$ is given in the text, or if v occurs in the left hand side of a production rule whose alternatives give S. A metavariable v occurring on its own is assumed to have a default range constraint saying that it ranges over some set of names distinct from other metavariables occurring in the grammar.

The set Object given in Definition 5.4 whose model in set theory you can find in Section 5.2 is uncountable (cardinality $\aleph_1$). Subsets of Object may be given via *syntax production rules* like:

$$v_1, \ldots, v_n \in S ::= \mathcal{A}_1 \mid \cdots \mid \mathcal{A}_m$$

where v_1, …, v_n are metavariables, S is the subset of Object being defined, and $\mathcal{A}_1$, …, $\mathcal{A}_m$ are *alternatives*. Each alternative is either the special notation $\boxed{\cdots}$ (in which case it appears as the first alternative in a list and after some alternatives for S have

[1]In principle we allow for any well defined computation as long as it is expressed mathematically and it takes one object to another. Typically such computations are basic arithmetic or set theoretic constructions.

been given, or at the end of a list indicating that it will be extended. The notation $\boxed{S ::= \cdots \mid \mathcal{A}}$ is like ABNF's incremental alternatives.) or an expression e, together with an optional side condition c (written "e if c", where c is a formula with expressions in the place of variables), which evaluates to a member of Object when values are supplied for metavariables occurring in e, provided:

1. c holds of that choice of metavariables.

2. The values supplied for metavariables occurring in e obey any default range constraints for them (E.g. those introduced by production rules with those metavariables on the left-hand side).

Sometimes the same side condition is applied to multiple alternatives. One can omit the $\boxed{\in S}$, allowing the reader to fill in S whose name is distinct from the names of all other declared sets. One can omit the side condition in which case we read it as if true. One can provide a "global" side condition $\boxed{\text{if } c'}$ which we read as appending if c' to all $\mathcal{A}$.

A context is a chunk of math text that is sufficient to define the default range constraints of any metavariables appearing in that context and the $\approx$-equivalence of any arrangements appearing in that context. Given a context, $ctxt$, we write $\mathsf{Constraint}(ctxt)$ to be the set of all constraints derived from $ctxt$. Each member of $\mathsf{Constraint}(ctxt)$ is a statement that is held to be true in that context. Constraints of the form $v_1, ..., v_n \in S$ say each of $v_1, ..., v_n$ range over S. Constraints of the form $O \in S$ for some object O say that $S = S \cup \{O\}$ (alternatives in production rules also give constraints of this form for multiple objects). Constraints of the form $A_1 \approx A_2$ for alternatives A_1 and A_2 give the $\approx$-equivalence classes arrangements A_1 and A_2 are coerced into when they are treated as well-formed objects. Contexts can be extended throughout the document. Multiple rules can be given for the same S. If a later rule for S begins with the alternative $\boxed{\cdots}$, then its alternatives are combined with the alternatives already in force for S. Usually the alternatives of the later rule replace the previous alternatives if this is not the case. However, if the author uses a single alternative in each production rule, then they normally expect these to be combined as though they had used $\boxed{\cdots}$. If $\boxed{\cdots}$ is used as the final alternative of a rule, it has no consequence and is used only as a signal to the reader warning that there will be later rules for the same set. Such a syntax production rule is intended to communicate the following constraints:

1. S is the smallest set of objects that satisfies all constraints placed on it by this rule and the rest of the document. The metavariables $v_1, \ldots, v_n$ range over S.

2. A "global" side condition $\boxed{\text{if } c'}$ appends $\wedge c'$ to the constraints added by each $\mathcal{A}_1, \ldots \mathcal{A}_n$ (i.e. any assignment of objects to metavariables such that $\mathcal{A}_i \in S$ must also satisfy c' in order for this constraint to hold). So for example, supposing $\boxed{\text{if } e \neq 0}$ was a global side condition for S then each e_i appearing in any A_i of S would have to obey the constraint $e_i \neq 0$.

3. If each $\mathcal{A}_1, \ldots \mathcal{A}_n$ contains only undecorated instances of v, then for any $\mathcal{A}_i$ containing multiple instances of v and no side conditions containing v that apply to $\mathcal{A}_i$, we usually rewrite it with each v given a different decoration. As an example, $\boxed{m \in M ::= x \mid m\,m}$ is read as though it were $m, m_1, m_2 \in M ::= x \mid m_1\,m_2$.

4. For each alternative $\mathcal{A}_i$ which is not $\boxed{\cdots}$, a constraint on the membership of S is added to $\mathsf{Constraint}(ctxt)$, where $ctxt$ is the context in which $\mathcal{A}_i$ is read. The constraint is that for each legal choice[2] of values for the metavariables in $\mathcal{A}$, if $O \in \mathsf{Object}$ is the result of evaluating an expression e in $\mathcal{A}$ using those metavariable assignments, then $O \in S$. So for example if we have $\boxed{m \in M ::= \lambda v.m}$ then, say the object $O_1 \in \mathsf{Object}$ was one of the objects v ranges over and the object $O_2 \in \mathsf{Object}$ was a member of M (and so an object M ranges over) and letting P_{O_i} signify a pointer to the object O_i, then the object given by $[\lambda P_{O_1}.P_{O_2}]_{\approx}$ is a member of M.

5. By default, metavariables occurring in $\mathcal{A}_i$, that are not yet declared to range over any set, are presumed to range over a countable set of names containing objects disjoint from any other undeclared sets of metavariables and which do not use symbols occurring elsewhere in the grammar. This assumption is dropped if a

[2]By legal choice we mean a choice of metavariables satisfying the constraints of their default range and fulfilling any side conditions.

metavariable is declared later than $\mathcal{A}_i$ and values are recalculated. So for example given the production rule $\boxed{m \in M ::= v\,r \mid \lambda v.m \mid \heartsuit}$, v and r would be expected to range over disjoin sets of names that do not include $\boxed{\lambda}$, $\boxed{.}$ or $\boxed{\heartsuit}$. If we added the production rule $\boxed{r \in R ::= \spadesuit}$, then r would range over $\{[\spadesuit]_{\approx}\}$ and the values for S would be recalculated accordingly.

6. If the first alternative is not the special alternative $\boxed{\cdots}$, then any constraints on the membership of S appearing in $\mathsf{Constraint}(ctxt)$ are usually removed. So given production rules $\boxed{m \in M ::= \heartsuit}$ and $\boxed{m \in M ::= \cdots \mid \spadesuit}$ we would write $M = \{[\heartsuit]_{\approx}, [\spadesuit]_{\approx}\}$, but if instead we had the 2 rules given as $\boxed{m \in M ::= \heartsuit}$ and $\boxed{m \in M ::= \spadesuit}$, we might sometimes read the second rule as overwriting the first and redefining M. We are unable to provide a convention for when authors expect us to do this. As such we avoid it in this thesis.

7. A recalculation of *all* sets appearing in $\mathsf{Constraint}(ctxt)$, to a "least" solution that makes all the constraints in $\mathsf{Constraint}(ctxt)$ true. This also happens when the definition of $\approx$ or of what metavariables range over is altered.

Note, if we do not rewrite it as in the convention for 3. above, that an alternative $\mathcal{A}_i$ written as $O_{\mathsf{c}}[v, v]$ behaves quite differently than an alternative $\mathcal{A}_i$ written as $O_{\mathsf{c}}[v_1, v_2]$. By the rules we give for interpreting alternatives, the former requires using whatever value is assigned to v twice, while the latter allows using two different values.

Note also that the recalculation of all sets appearing in $\mathsf{Constraint}(ctxt)$ evaluates all the constraint expressions for all syntactic sets using the *current* bindings for all metavariables, set names, $\approx$, etc., and rebinds the set names to the recalculated values in the subsequent text. This is important as it allows parts of documents to be checked in the context they appear in and grammars to be amended throughout a paper. Such a recalculation cannot (and should not) be performed if there is no unique assignment of smallest values to the declared sets after each undeclared metavariable has been chosen to range over a countable set of names. Normally, the existence of a unique assignment will be provable using a fixed point theorem like the Knaster-Tarski theorem Tarski (1955). However, the notation doesn't automatically rule out putting side conditions in the constraint expressions in alternatives that prevent sets from being defined in a monotonic increasing way, which can cause a failure. We have discussed some such difficulties caused by side conditions (and other MBNF features) in Section 4.3

Outermost parentheses on an alternative are not required as they would be to get unique parse trees if we were defining our syntax as a set of strings like many BNF variants do. Declared metavariables take a similar role to non-terminals in BNF, while syntactic symbols correspond to terminals. The font is used to distinguish them. When an alternative allows building terms from multiple subterms[3] of the same set, we need to use distinct metavariables for each possible subterm to allow the subterms to differ. We find distinct metavariables for the same set by using subscripts. The following example illustrates how different use of subscripts can be used to make terms that are read differently:

Example 6.2 (Simple types). We can use the following MBNF to define the syntax of *simple types*:

$$a, b \in \mathsf{Ty-Variable} ::= \mathsf{a}_i \qquad T \in \mathsf{Simple-Type} ::= a \mid T_1 \to T_2$$

Here a stands for a variable whereas a is a symbol. A possible type is $T_0 = a \to (b \to a)$ where exact type variables are left unspecified in T_0. We could make T_0 concrete by stating $a = \mathsf{a}_0$ and $b = \mathsf{a}_1$ yielding $T_0 = \mathsf{a}_0 \to (\mathsf{a}_1 \to \mathsf{a}_0)$. If we had written the second alternative in the production rule for $\mathsf{Simple-Type}$ as $T \to T$, or less ambiguously as $T_1 \to T_1$ then we would expect it to be translated to something like the above, but if we didn't then the type T_0 would not be allowed and we could only write types like $a \to a$ and $(a \to a) \to (a \to a)$ where both arguments of each $\to$ are equal. This would yield a set like $\{a, a \to a, (a \to a) \to (a \to a), \ldots\}$. We would not usually default to assuming this is the set intended, but for some cases 2 instances of the same metavariable are meant to stand for the same object. We rely on readers to know when to treat them as such and when to follow our convention of treating them as though they were labelled differently.

[3] B is a subterm of A if there is a context C such that filling the hole in C with B gives us A.

6.2 One Way in Which Production Rules are to be Safely Defined

So far we have talked about how production rules in MBNF are interpreted but we have deliberately left out the detail of how the sets of objects satisfying these constraints are supposed to be calculated. This is largely because we want readers and authors to be able to use the full power of set theory in determining whether such a set of constraints has a solution. However, we sketch one method of doing so which is generally adequate.

With this method each set, S, is built inductively in a monotonic increasing fashion. Since Object gives an upper bound on the possible syntactic objects we can take a least fixed point of this set building operation. We define this operation so that it exists in most cases (it only fails where sets are defined coinductively or where a side condition means an appropriate increasing function cannot be found).

Methodology 6.3 (A Safe Process for Defining Production Rules)**.** Let S be the name of each of the sets in our grammar and i be an ordinal index (of size less than $\aleph_2$). Assuming no notational conflicts occur, we write S^i for the set whose name is given by the pair (S, i). We proceed by induction on i. We define S^0 as follows:

1. If the first alternative of any set S corresponds to a piece of concrete syntax $O' \in \mathsf{Object}$ (i.e. syntax which does not contain any metavariables) then we write $S^0 = \{O'\}$.

2. If the first alternative of object corresponds to a set of names N then we write $S^0 = N$.

3. If some set T of objects and metavariable $t \in T$ are already defined, no value given to any set not yet calculated would trigger a recalculation of T and the first alternative of S is t, then we might choose to write $S^0 = T$. This is just an optional choice for notational convenience, to cut down on pointless steps and make presentation a little easier to digest.

4. Otherwise $S^0 = \emptyset$.

We write S^{i+1} as follows:

1. For S consisting of a single alternative of the form $O \in \mathsf{Object}$ or a set of names N, $S^{i+1} = S^i$.

2. If S consists of a single alternative of the form $O_c[v_1, \cdots, v_n]$ if c. Suppose also that c is defined such that it is a boolean for some choice of some $v_1 \in M_1, \cdots, v_n \in M_n$. Let c' be the result of replacing each instance of M_j in c with M_j^i, where M_j^i is the set defined as having a name that corresponds to the pair (M_j, i). There are a few such cases to consider:

   (a) In the case where each of $M_1 \cdots M_n$ are already defined at this point of our reading of the document, then we have
   $$S^{i+1} = \left\{ O \in \mathsf{Object} \,\middle|\, \begin{array}{c} \exists v_1 \in M_1, \cdots, v_j \in M_n \\ \text{s.t.}\, O = O_c[v_1, \cdots, v_n] \wedge c \end{array} \right\}$$

   (b) In the case where M_j^i is already defined and that it would be a well defined statement to use in the right hand side of a set comprehension in this case. In this case, if

   $S^i \subset \{O \in \mathsf{Object} \mid \exists v_1 \in M_1^i, \cdots, v_n \in M_n^i \text{ s.t.}\, O = O_c[v_1, \cdots, v_n] \wedge c'\}$, then

   $S^{i+1} = \{O \in \mathsf{Object} \mid \exists v_1 \in M_1^i, \cdots, v_j \in M_n^i \text{ s.t.}\, O = O_c[v_1, \cdots, v_n] \wedge c'\}$.

   (c) We have ommitted the cases where the first alternative is O' if c for $O' \in \mathsf{Object}$ or where it is N if c for a set of names as these are not generally used, but if they were a similar principle would apply.

3. Production rules with more than one alternative are "read" as follows:

   (a) The first alternative of every production rule in the present context is considered to need to be "read."

   (b) When only the first alternative $\mathcal{A}$ has been "read" for S^i we define S^{i+1} as it would be defined if $S ::= A$ provided $S^i \subset S^{i+1}$.

(c) Supposing we have already "read" alternatives $\mathcal{A}_1, \cdots, \mathcal{A}_n$ for S^i and let X^i and Y^i be names not already in use by the grammar.

   i. Let $X ::= \mathcal{A}_1, \cdots, \mathcal{A}_n$ and suppose $X^i \subset X^{i+1}$. Then we continue to read the first n alternatives in which case $S^{i+1} = X^{i+1}$.

   ii. Let M be some set in our construction that is defined simultaneously to S. If $X^i ::= \mathcal{A}_1, \cdots, \mathcal{A}_n$ and no $\mathcal{A}_{n+1}$ exists amongst the alternatives for S and X^i would equal X^{i+1}, then we delay reading S_{i+1} until one or other of the following is true:

      A. We have tried to read every alternative to any M such that $v_i \in M$ appears amongst the alternatives for S and for each of them M^i would equal M^{i+1} in which case we consider every alternative to S_i to have been "read" and write $S^i = S^{i+1} = S$.

      B. There is some M such that $v_i \in M$ appears amongst the alternatives for S such that $M^i \subset M^{i+1}$, in which case we consider S^i to have been "read" after M^i and calculate S^{i+1} as though every such v_i were taken from M^{i+1}. In this case we say that reading M^i "triggered a recalculation" of S^{i+1}.

   iii. If there exists some $\mathcal{A}_{n+1}$ amongst the alternatives of S_i and nothing would trigger a recalculation of $X ::= \mathcal{A}_1, \cdots, \mathcal{A}_n$, then we read $\mathcal{A}_{n+1}$ in which case:

      A. If $\mathcal{A}_{n+1}$ corresponds to a piece of concrete syntax $O' \in \mathsf{Object}$ then we write $S^{i+1} = X_i \cup \{O'\}$.

      B. If $\mathcal{A}_{n+1}$ corresponds to a set of names N then we write $S^{i+1} = X^i \cup N$.

      C. If S consists of a single alternative, of the form $O_c[v_1, \cdots, v_n]$ if c, where c is defined such that it is a boolean for some choice of some $v_1 \in M_1, \cdots, v_n \in M_n$ and supposing M_j^i is already defined and that it would be a well defined statement to use in the right hand side

of a set comprehension in this case. Let c' be the result of replacing each instance of M_j in c with M_j^i. In this case,
$S^{i+1} = X_i \cup \left\{ O \in \mathsf{Object} \,\middle|\, \begin{array}{l} \exists v_1 \in M_1^i, \cdots, v_j \in M_n^i \\ \text{s.t.}\, O = O_c[v_1, \cdots, v_n] \wedge c' \end{array} \right\}$.
Note that, by 3 (c) i., for as long as S^{i+1} would continue to be larger than S^i we would continue to read this alternative and any previous alternatives before moving on to $\mathcal{A}_{n+2}$ if it exists.

For a limit case S^ϵ we write

$$S^\varepsilon = \left\{ x \,\middle|\, \begin{array}{l} (a \in x \wedge b \in x) \Leftrightarrow \\ (a, b \in \bigcup \bigcup_{i<\varepsilon} S^i \wedge a \approx b) \end{array} \right\}$$

□

Example 6.4 (Non-positive inductive types).
For $t \in T ::= f$(where f is in $(T \to T) \to T$). We'd first begin with $T^0 = \emptyset$. We have $T^0 \to T^0 = \{\emptyset\}$. There is no function with a nonempty domain and an empty range. So $t \in T^1 = \emptyset$ by our methodology. So $T = \emptyset$ and $t \in T$ doesn't exist.

There are some functions this approach would rule out which we might like to allow. For example if we had an existing type S such that $t \in T ::= f$(where f is in $(S \to T) \to T$) where $S \neq T$ the default assumption that $T^0 = \emptyset$ would similarly rule it out. Readers could not rely on our methodology to determine it is permitted, but would instead have to defer to type theory.

To convince readers that we do indeed rule them out we therefore give another example of a non-positive inductive type:

$$\begin{array}{rcl} p \in P & ::= & \bullet \mid \prod_{j \in I} p_j \\ i \in I & ::= & \circ \mid p \end{array}$$

In this case we can show that as I reaches its limit point p exceeds the size of $O \in \mathsf{Object}$. As such there is no lattice on which P is monotonic increasing.

Example 6.5 (Conflicting Side Conditions)**.** Consider the following MBNF grammar:

$$\begin{aligned} a \in A &::= \circ \mid \heartsuit a \mid b && \text{where } \heartsuit b \notin C \\ b \in B &::= \circ \mid a \mid \spadesuit b \\ c \in C &::= a \mid b \end{aligned}$$

For $b \in B^i$, suppose $b \in A^i$ and every rule in the grammar has been read, then $\heartsuit b \in A^i$ and $\heartsuit b \in C^i$, then $b \notin A^{i+1}$. Suppose by contrast $b \in B^j$ and $b \notin A^j$ then $\heartsuit b \notin C^j$ then $b \in A^{j+1}$. Then A is not monotonic under $\subseteq$. Then A is undefined.

The following grammar is simple enough to be expressed with a BNF but should provide a picture of how this process should work. A more comprehensive study of an example which cannot be expressed in MBNF appears in the next chapter.

Example 6.6.

$$\begin{aligned} p \in \mathsf{Letter} &::= \mathsf{A} \mid \mathsf{B} \mid \mathsf{C} \mid \mathsf{D} \mid \mathsf{E} \mid \mathsf{F} \mid \mathsf{G} \mid \mathsf{H} \mid \mathsf{I} \mid \mathsf{J} \mid \mathsf{K} \mid \mathsf{L} \mid \mathsf{N} \mid \mathsf{O} \mid \mathsf{P} \mid \mathsf{Q} \mid \mathsf{R} \\ &\quad \mid \mathsf{S} \mid \mathsf{T} \mid \mathsf{U} \mid \mathsf{V} \mid \mathsf{W} \mid \mathsf{X} \mid \mathsf{Y} \mid \mathsf{Z} \\ w \in \mathsf{Word} &::= p \mid wp \end{aligned}$$

Here the first alternative of Letter is a piece of concrete syntax so we write $\mathsf{Letter}^0 = \{\{\mathsf{A}\}\}$ (here the object corresponding to the arrangement A is the set containing A as no other arrangements have been placed in the same equivalence class. The first alternative of Word is neither a piece of concrete syntax or a set of names so we write $\mathsf{Word}^0 = \emptyset$. For Letter^1, we check whether "reading" A again would give a larger result. It would not so we "read" B by 3(c)iiiA in our set of rules. In this case $\mathsf{Letter}^1 = \mathsf{Letter}^0 \cup \{\{\mathsf{B}\}\} = \{\{\mathsf{A}\}, \{\mathsf{B}\}\}$. Reading $\{\mathsf{A}\}$ triggers a recalculation of p in Word so, by 3. (b) and 2 (b) we have $\mathsf{Word}^1 = \{\{\mathsf{A}\}\}$. Similarly we have $\mathsf{Letter}^2 = \mathsf{Letter}^1 \cup \{\{\mathsf{C}\}\} = \{\{\mathsf{A}\}, \{\mathsf{B}\}, \{\mathsf{c}\}\}$, ..., $\mathsf{Letter}^{26} = \mathsf{Letter}^{25} \cup \{\{\mathsf{Z}\}\}$ and $\mathsf{Word}^2 = \mathsf{Letter}^1$, ..., $\mathsf{Word}^{27} = \mathsf{Letter}^{26}$. When we get to Letter^{27} there are no further alternatives and nothing which might trigger a recalculation if another alternative is "read" so by 3. (c) ii. A. $\mathsf{Letter}^{27} = \mathsf{Letter}^{26} = \mathsf{Letter}$. When we get to Word^{28} then by 3(c)iiiC. we have the following:

$$\mathsf{Word}^{28} = \mathsf{Word}^{27} \cup \left\{ O \in \mathsf{Object} \;\middle|\; \begin{array}{l} \exists w \in \mathsf{Word}^{27}, p \in \mathsf{Letter} \\ \text{s.t.}\, O = \{wp\} \end{array} \right\}$$

This new value of Word^{i} means we reread this alternative by 3. (c) i., so for ordinals $27 < i < \omega$ we have the following:

$$\mathsf{Word}^{i+1} = \mathsf{Word}^{i} \cup \left\{ O \in \mathsf{Object} \;\middle|\; \begin{array}{l} \exists w \in \mathsf{Word}^{i}, p \in \mathsf{Letter} \\ \text{s.t.}\, O = \{wp\} \end{array} \right\}$$

We apply our rule for the limit case where $i = \omega$:

$$\mathsf{Word}^{\omega} = \left\{ x \;\middle|\; \begin{array}{l} (a \in x \wedge b \in x) \Leftrightarrow \\ (a, b \in \bigcup \bigcup_{i<\omega} \mathsf{Word}^{i} \wedge a \approx b) \end{array} \right\}$$

Note that in this case $a \approx b$ if and only if they are the same arrangement because we haven't applied any arrangement equivalences. There are no further Alternatives to read for Word and

$$\mathsf{Word}^{\omega} = \mathsf{Word}^{\omega} \cup \left\{ O \in \mathsf{Object} \;\middle|\; \begin{array}{l} \exists w \in \mathsf{Word}^{\omega}, p \in \mathsf{Letter} \\ \text{s.t.}\, O = \{wp\} \end{array} \right\}$$

By 3 (c) ii. B. $\mathsf{Word}^{\omega} = \mathsf{Word}^{\omega+1} = \mathsf{Word}$.

Since no value given to Word could trigger a recalculation of Letter we could have elected to calculate Word in a context where the value for Letter was already given. in this case we could write $\mathsf{Word}^{0} = \mathsf{Letter}$, the rest of the calculation would precede similarly with the 1st 26 steps skipped.

The above example is a simple one which can be dealt with using BNF and this may leave readers wondering what the extra machinery is for. However we may have chosen any equivalence over Object for $\approx$ and any number of n-ary functions $f : \mathsf{Object}^{*} \rightarrow \mathsf{Object}$ for our alternatives and, provided these preserve $\approx$-well-formedness and our method for building sets form production rules is monotonic, these would still yield a value. This allows for the full power of mathematics to be used alongside production rules. We will discuss a real world example which showcases more of the power of this method for reading MBNF in the next chapter.

Lemma 6.7. *Let P be a set of production rules whose definition can be given by the process in Methodology 6.3. Then P has a solution.*

Proof. For each set S given by the above process and for each $\mathcal{A}_i$ amongst the alternatives for S. S_i is monotonic increasing and bounded above by Object. So S_i has a least fixed point. $S^0 \subseteq S$ by construction (it only contains values the first alternative tells us are members of S). If $S^i \subseteq S$ and for each $A, B, C, ...$ defined by P $A_i \subseteq A, B_i \subseteq B, C_i \subseteq C, ...$, then $S^{i+1} \subseteq S$ by construction (objects are only members of S_{i+1} if some alternative would tell us to add them to S, given values known to be in S and $A, B, C, ...$). For a limit case S^ε, if $S_i \subseteq S$ for all $i < \varepsilon$ then $S^\varepsilon \subseteq S$ (by construction since S^ε has no members which are not mebers of those S_i). We can now see that, if Z is a limit point of S_i, $Z \subseteq S$. In fact, if some alternative $\mathcal{A}_x$ would tell us that an additional object O were a member of S, if $Z \subseteq S$, then Z would not be a fixed point of S_i by construction, as reading $\mathcal{A}_x$ would trigger a recalculation of Z so in fact $Z = S$. As such the above process gives us the solution to P as the fixed point of each set S which P provides constraints for. ☐

6.3 Notes on the Coinductive Case

One might well wonder reading Definition 6.3, whether a similar process might be used to find the greatest rather than least fixed point of the sets involved. We believe such a process does exist and provides something analogous to the coinductive definition of a production rule alluded to in Example 4.25. The set Object provides a good starting point for a monotonic decreasing function to give these fixed points. The reason we do not provide a more thorough account of this process is that the only way we could think of as doing this relied on a coinductive definition which relied on both something akin to a coinductive model for equivalence and something akin to a coinductive definition of a primitive constructor decomposition, the latter of which might in turn rely on something like a coinductive notion of hole-filling. Furthermore each of a coinductive notion of primitive constructor decomposition and a coinductive notion of equivalence could be much better understood within the context of a coinductive grammar. In addition, most of the above would be set theoretic analogues for coinduction which resist definition in the normal way with a head and a tail because they would be dealing with objects modulo equivalences as opposed to streams so much of the standard notation and conventions for dealing with coinduction would need to be rethought. As such we had something of a tangle of definitions with no one thread to pull to disentangle them and we thought the resulting thesis would not be able to build its definitions and proofs so clearly and simply as we have done here. Disentangling this Gordian knot could be a fruitful avenue for future work.

6.4 What Was Covered by This Chapter

In section 6.1. we covered what it means to evaluate an expression and we gave an idea of how production rules could be read as a set of constraints. We also cover some of the conventions that generally operate while reading production rules and discuss the effect different conventions of indexing may have on how production rules are read. In section 6.2. we defined a family of production rules whose alternatives could be read as contributing syntactic objects to a series of sets in a monotonic increasing manner. We showed that these are well defined and gave an example of how the proof of this may be applied to a specific set of production rules. In section 6.3 we suggested a greatest fixed point computation to find subsets of Object may be comparable to production rules said to work coinductively and we discuss barriers that still exist to providing this definition in a clear and easy to follow manner.

Chapter 7

The λ-Calculus and Extensions in MathSyn v BNF

In this section we showcase some features of MathSyn, which BNF lacks. First we examine the λ-calculus without records or generalised β-reduction in BNF and MathSyn, an example which is widely popular in the computer science literature. We discuss small adjustments that need to be made to the BNF-like syntax authors may write in order to use BNF here. Finally, we show how this syntax may be extended with records and generalised β-reduction in MathSyn and discuss why a similar extension cannot be made in BNF.

7.1 The Lambda Calculus in BNF

In BNF we can define the strings used in the λ-*calculus* like this:

$$\langle v \rangle ::= v \mid \langle v \rangle' \qquad\qquad \langle e \rangle ::= \langle v \rangle \mid (\lambda \langle v \rangle . \langle e \rangle) \mid (\langle e \rangle\ \langle e \rangle)$$

Our raw syntax is the language of e. BNF needs extensions to define the λ-calculus that MathSyn does not and also requires careful use of syntax. These adjustments are trivial and well documented, but we discuss them briefly below.

7.1.1 BNF Alone Only Permits String Equality

The only thing the above grammar gives us is a set of strings (consisting only of concatenated terminal symbols). There is no sense of syntactic equivalence up to renaming. MathSyn supports syntactic equivalence at syntactic boundaries which allow the identification of multiple arrangements of text with one another. The grammar above would give $\boxed{\lambda v.v}$ and $\boxed{\lambda v'.v'}$ as two separate strings with nothing to indicate their syntactic equivalence up to renaming. To provide α-equivalence, we would have to partition this language into α-equivalent terms after it is generated, and build the rest of our semantics for sets of strings in these partitions, or we would have to provide some semantics for reducing α-equivalent terms to the same string. BNF lacks this semantics. Normally this is handled with De-Bruijn indices, or similar. De-Bruijn indices are numbers that relate term positions back to the λ they are bound with. A De-Bruijn index of 1 binds to the previous lambda A De-Bruijn index of 2 binds one further λ back in scope and so on. Details can be found in de Bruijn (1972).

7.1.2 BNF Must Provide "Arbitrary" Non-Terminals as Sets of Strings

At a trivial syntax level, we must include brackets around each non terminal, although many authors omit these. Moreover, we cannot just declare $\langle v \rangle$ to range over a countable set. We need to define the set of strings $\langle v \rangle$ ranges over, otherwise the BNF is meaningless. If two authors use different $\langle v \rangle$, BNF regards these as generating unrelated sets of strings, unless we provide semantics for translating one to the other. Many authors view the choice of set of variables as irrelevant to the structure of the λ-calculus and offer no method for selecting one. Moreover, our choice of $\langle v \rangle$ must be a string, countable sets of objects of different data types (e.g. $\mathbb{N}$) cannot be used for $\langle v \rangle$ unless they are first translated into sets of strings consisting of terminal symbols. BNF has no default way of performing these translations, or choosing a language for $\langle v \rangle$.

7.1.3 BNF Might Not Produce a Unique Abstract Syntax Tree

We need brackets around terms if we want their range to be clear. These brackets must always appear in the literal syntax. For example, if our syntax was:

$$\langle v \rangle ::= v \mid \langle v \rangle' \qquad\qquad \langle e \rangle ::= \langle v \rangle \mid \lambda\langle v \rangle.\langle e \rangle \mid \langle e \rangle\,\langle e \rangle$$

Then $\lambda v.v\,v'$ could be read either as an instance of $\langle e \rangle\,\langle e \rangle$ where we choose the first $\langle e \rangle$ to be $\lambda\langle v \rangle.\langle e \rangle$ or an instance of $\lambda\langle v \rangle.\langle e \rangle$ where we choose $\langle e \rangle$ to be $\langle e \rangle\,\langle e \rangle$. If we wanted to read λ-calculus syntax where some of the parentheses are omitted, we would need to provide semantics for reducing these to a string where all the brackets are present. This semantics is not part of BNF. Suppose we wanted a separate semantics for dealing with optional parentheses. First we need a syntax for the λ-calculus where parentheses were not mandatory:

$$\langle v \rangle ::= v \mid \langle v \rangle' \qquad\qquad \langle e \rangle ::= \langle v \rangle \mid \lambda\langle v \rangle.\langle e \rangle \mid \langle e \rangle\,\langle e \rangle \mid (\langle e \rangle)$$

Then we would need a parser to read the strings in which parentheses are ommitted. Here is how that might look: Starting with the leftmost, outermost instances of $\lambda\langle v \rangle.\langle e \rangle$ in a piece of syntax, if the λ is not preceded by (, replace with $(\lambda\langle v \rangle.\langle e \rangle)$. Starting with the rightmost, innermost instances of $\langle e \rangle\,\langle e \rangle$, if the $\langle e \rangle\,\langle e \rangle$ is not surrounded by brackets, replace with $(\langle e \rangle\,\langle e \rangle)$. Build the abstract syntax tree as though for the fully parenthesised BNF:

$$\langle v \rangle ::= v \mid \langle v \rangle' \qquad\qquad \langle e \rangle ::= \langle v \rangle \mid (\lambda\langle v \rangle.\langle e \rangle) \mid (\langle e \rangle\,\langle e \rangle)$$

7.2 The Lambda Calculus in MathSyn

Because of the automatic interpretation of parentheses amd identification of metavariables, in MathSyn we can define the λ-*calculus* using a standard MBNF grammar:

Example 7.1.

$$e \in \mathsf{exp} ::= v \mid \lambda v.e \mid e\, e$$

By convention 5.54, in the grammar given by Example 7.1 each v ranges over a countable set of object-level variables which are not sub-objects of any e containing a pointer. The production rule $e \in \mathsf{exp} ::= v$ can be read as giving us the constraint $\mathsf{var} \subseteq \mathsf{exp}$. The rule $e \in \mathsf{exp} ::= \lambda x.e$ gives $\{[\lambda P_v.P_e]_{\approx} \mid \mathsf{ptr}(v) = P_v \wedge \mathsf{ptr}(e) = P_e\} \subseteq \mathsf{exp}$. The rule $e \in \mathsf{exp} ::= e\, e$ gives $\{[P_{e_1}\, P_{e_2}]_{\approx} \mid \mathsf{ptr}(e_1) = P_{e_1} \wedge \mathsf{ptr}(e_2) = P_{e_2}\} \subseteq \mathsf{exp}$. We pick the least $\mathsf{exp} \subseteq \mathsf{Object}$ and $\mathsf{var} \subseteq \mathsf{Object}$ satisfying these constraints with an ordering given by $\subseteq$.

As well as declaring e as ranging over exp, MathSyn also declares e_1, e_2,..., e', e'' etc. ranging over exp, likewise for $v \in \mathsf{var}$. The subset of Object chosen by these constraints depends on the choice of equivalence relation $\approx$, in the λ-calculus this is likely α-equivalence, but it may also be the identity relation on $\mathsf{Arrangement}$. It also depends on the choice of set for v to range over.

In order to be confident that this set can be chosen (e.g. for exp) MathSyn begins with exp^0 equal to the set ranged over by v and lets exp^1 contain the objects exp must contain if exp is at least exp^0 and so on as an i-step approximation of the limit point. For a limit point[1], ε, we let $\mathsf{exp}^{\varepsilon}$ be $\bigcup_{i=0}^{\varepsilon} \mathsf{exp}^i$. We take the least fixed point of the function $f : \mathcal{P}(\mathsf{Object}) \mapsto \mathcal{P}(\mathsf{Object})$ such that $f(\mathsf{exp}^i) = \mathsf{exp}^{i+1}$ over some appropriately large set of exp^i ordered by $\subseteq$. The same method of proof is employed in each of our demonstrations a grammar has a model in MathSyn.

[1]We are only interested in demonstrating the existence of a set of objects here. Limit points needn't be computed when parsing. To "parse" an object, O, defined using MathSyn we require a proof that a particular use of BNF-style notation defines a set, S, and a proof that, if $\exists S$, then $O \in S$. This notion is deliberately kept general, as we are not yet prescribing methods of implementation.

Lemma 7.2 (The BNF Syntax of the λ-Calculus up to Identity has a Model in MathSyn)**.** *The grammar for the λ-calculus where $\approx$ is the identity relation on* $\mathsf{Arrangement}$*, given by Example 7.1 (i.e. loosely the same syntax BNF generates for the λ-calculus) has a model in MathSyn.*

Proof. First we read v. It ranges over a set of countable object-level variables, V. If this were the only constraint we'd have $\mathsf{exp} = V$. Call this exp^0. When the $\lambda v.e$ is read it triggers a recalculation of exp^0. We define $\mathsf{exp}^{n+1} = \mathsf{exp}^n \cup \{\{\lambda P_v.P_e\} \mid v \in V \wedge e \in \mathsf{exp}^n\}$ (we increment the index on exp each time a recalculation is performed). For a limit point ε we define $\mathsf{exp}^\epsilon = \bigcup_{i<\varepsilon} \mathsf{exp}^i$ (we can think of this as the limit of each of these recalculations as $\mathsf{exp} \subseteq \mathsf{exp}^{n+1}$). If v and $\lambda v.e$ were the only constraints then exp would be the least fixed point of the least function $f : \mathsf{Object} \mapsto \mathsf{Object}$ such that $\forall \mathsf{exp}^i$, $f(\mathsf{exp}^i) = \mathsf{exp}^{i+1}$ (which exists by the Knaster-Tarski theorem Tarski (1955)). Instead, if k is the first k such that $\mathsf{exp}^k = \mathsf{exp}^k \cup \{\{\lambda P_v.P_e\} \mid v \in V \wedge e \in \mathsf{exp}^k\}$, then we look to see if there are any other constraints which would trigger a recalculation of e. In this case there is the constraint $e\,e$. We update our definition of exp^i like so: If $n < k$, then $\mathsf{exp}^{n+1} = \mathsf{exp}^n \cup \{\{\lambda P_v.P_e\} \mid v \in V \wedge e \in \mathsf{exp}^n\}$, otherwise $\mathsf{exp}^{n+1} = \mathsf{exp}^n \cup \{\{\lambda P_v.P_e\} \mid v \in V \wedge e \in \mathsf{exp}^n\} \cup \{\{P_e\,P_e\} \mid e \in \mathsf{exp}^n\}$. Since there are no further constraints, the least fixed point of f gives us the desired result. ☐

7.2.1 Adding α-Equivalence

If we wanted an α-equivalence, we could extend the grammar in Example 7.1:

$$e \in \mathsf{exp} ::= v \mid \lambda v.e \mid e\,e$$

Let $\lambda\square.\square$ bind the object placed in the first of its holes in both of its holes. Let $e \in \mathsf{exp}$ be identified up to α-equivalence.

Lemma 7.3 (MathSyn Fully Supports the Syntax of the λ-Calculus with α-Equivalence)**.** *The λ-calculus with all syntax trees and sub-trees quotiented by α-equivalence has a model in MathSyn.*

Proof. exp^0 would remain the same. For the $n+1$ case (for $n < k$ where k is the first k such that $\mathsf{exp}^k = \mathsf{exp}^k \cup \{[\lambda P_v.P_e]_{\approx} \mid v \in V \wedge e \in \mathsf{exp}^k\}$) we have:

$$\mathsf{exp}^{n+1} = \left\{ x \,\middle|\, \begin{array}{l} (a \in x \wedge b \in x) \Leftrightarrow \\ (a, b \in (\bigcup \mathsf{exp}^n) \cup \{\lambda P_v.P_e \mid v \in V \wedge e \in \mathsf{exp}^n\} \wedge a \approx_\alpha b) \end{array} \right\}$$

This is the same as we had for the λ-calculus up to identity with the added requirement that any two terms in the object x are identified up to α-equivalence.

For $n > k$ we have:

$$\mathsf{exp}^{n+1} = \left\{ x \,\middle|\, \begin{array}{l} (a \in x \wedge b \in x) \Leftrightarrow \\ \left(a \approx_\alpha b \wedge a, b \in \begin{array}{l} (\bigcup \mathsf{exp}^n) \cup \{\lambda P_v.P_e \mid v \in V \wedge e \in \mathsf{exp}^n\} \\ \cup \{P_{e_1}\, P_{e_2} \mid e_1, e_2 \in \mathsf{exp}^n\} \end{array} \right) \end{array} \right\}$$

For a limit point ε we have $\mathsf{exp}^\varepsilon = \left\{ x \,\middle|\, \begin{array}{l} (a \in x \wedge b \in x) \Leftrightarrow \\ (a, b \in \bigcup \bigcup_{i<\varepsilon} \mathsf{exp}^i \wedge a \approx_\alpha b) \end{array} \right\}$

The least fixed point of the least function $f : P(\mathsf{Object}) \mapsto P(\mathsf{Object})$ such that $\forall \mathsf{exp}^i$, $f(\mathsf{exp}^i) = \mathsf{exp}^{i+1}$ gives us the result. $\square$

This syntax also let us consider α-equivalent sub-trees, which quotienting after calculating BNF does not let us do and which some authors may find useful. This is particularly helpful if we want calculations on the tree that depend on equivalence to descend inside of sub-trees. We will see examples of this in Section 7.3. This feature is not used much here, but MBNF grammars may not feature the same pieces of literal syntax after quotienting, and may feature holes in syntax, in which case some sub-trees may be quotiented by some equivalence, but parts of the tree occurring above the hole are literal syntax (e.g. for $\lambda x.\Box\lambda y.yy$, $\lambda y.yy$ may be quotiented up to α-equivalence, but everything else is literal). For Chang & Felleisen (2012) this quotienting cannot wait until afterwards and is instead done on the fly.

7.2.2 Adding β and η rewriting

If we wanted β and η rewriting, we could add:

...Our rewriting rules are: $(\lambda v.e_1)e_2 \xrightarrow{\beta} (e_1[v := e_2])$ $\quad\quad$ $\lambda v.e_1 v \xrightarrow{\eta} e_1$

then MathSyn gives β rewriting as the exp-compatible closure of the smallest β (ordering given by $\subseteq$) satisfying $(\lambda v.e_1)e_2 \xrightarrow{\beta} (e_1[v := e_2])$. Similarly for η. This is relevant as with BNF we often see rewriting rules being mentioned without covering how they descend through syntax, a question answered more explicitly in MathSyn.

We address the idiosyncrasies of our approach. In order to arrive at exp, we pass multiple ordinal limits. However, all we are interested in is that, for each ordinal i, j, such that $j < i$, we have $\mathsf{exp}_j \subseteq \mathsf{exp}_i \subseteq \mathsf{Object}$. Remember, BNF also cannot construct the set of strings used in the λ-calculus in finite time, although we can use it to prove such a language exists. We extend the syntax building relation as production rules are read, instead of calculating them all from the start. This is because there are documents where production rules are slowly extended and proofs about a grammar are expanded throughout. Reading each fresh rule separately keeps track of what is true at a given point.

7.3 The λ-Calculus in MathSyn With Records and Generalised β-Reduction

In this section we will provide an example grammar which is handled by MathSyn that deals with two features often added to the λ-calculus: generalised β-reduction (introduced by de Bruijn's Automath de Bruijn (1970) then used in Nederpelt's strong normalisation Nederpelt (1994) and subsequently appearing in several other papers Kamareddine & Nederpelt (1995), Bloo et al. (1996), Ariola et al. (1995), Groote (1993), Kfoury & Wells (1995)) and records Cardelli (1988), Cardelli & Mitchell (1990). These features cannot be handled by BNF. We suggest MathSyn as a useful way of interpreting many uses of MBNF that deals with inclusion of mathematical objects (E.g.

function sets) and mathematical operators (E.g. hole filling) better than other BNF formalisms, such as EBNF ISO (1996), parsing expression grammars Ford (2004), LBNF Forsberg & Ranta (2005) etc.

Records are given as finite functions from labels to terms. They are used to deal with typing for hierarchically organised data such as objects.

We give the following as an informal remark meant to aid intuition about what generalised β-reduction is and how it works. Generalised β-reduction is a form of β-reduction which descends through balanced segments.[2] In $((\lambda x.\lambda y.N)P)Q$, the function starting with λx and the argument P result in the redex $(\lambda x.\lambda y.N)P$ which when contracted will turn the function starting with λy and Q into a redex. Generalised β reduction allows us to exploit this fact by giving the future redex based on matching λy with Q the same priority as the redex based on matching λx and P.

Example 7.4. We can extend the λ-Calculus in MathSyn with records, like this:

$$\ell \in \mathsf{Label} \qquad\qquad t \in \mathsf{Term} ::= v \mid \lambda v.t \mid t\,t \mid t.\ell \mid r$$

$$r \in \mathsf{Rec} = \mathsf{Label} \overset{\text{fin}}{\to} \mathsf{Term} \qquad\qquad S \overset{\text{fin}}{\to} T \;=\; \left\{ \begin{array}{r} f \in S \times T : f \text{ is a function} \\ \wedge f \text{ is finite} \end{array} \right\}$$

Let $\lambda\square.\square$ bind the object placed in the first of its holes in both of its holes. Let $e \in \mathsf{exp}$ be identified up to α-equivalence.

We have used math to define $r \in \mathsf{Rec}$ rather than define it using a production rule. The "=" in $r \in \mathsf{Rec} = \mathsf{Label} \overset{\text{fin}}{\to} \mathsf{Term}$ is a math = and $\overset{\text{fin}}{\to}$ is a function defined using maths that builds a set from two other sets. However, MBNF and MathSyn both allow math to be mixed freely with production rules. By construction $r \in \mathsf{Rec}$ is no larger than Object (in fact there is a direct mapping onto those objects containing finitely many arrangements consisting of a pair of pointers to Label and Term), so we can support its inclusion as some other kind of mathematical entity to an object, using the construction

[2] A balanced segment is one where each application has a matching abstraction and where each application/abstraction pair contains a balanced segment.

Object_M. Since there is at least one way of encoding Rec as syntactic objects, this would clearly be the case. This is permitted in MathSyn where it clearly would not be in BNF as the only mathematical entities BNF allows inside a grammar are strings. For a BNF grammar each of these records would have to be given an encoding as a finite string.

Lemma 7.5 (MathSyn Supports the Syntax for the λ-Calculus with Records)**.** *The MBNF grammar for* Term *given in example 7.4 has a model in MathSyn.*

Proof. Here, the first 3 constraints for Term are the same as those for exp. So, for each n such that $\mathsf{exp}^{n+1} \neq \mathsf{exp}^n$, we have $\mathsf{Term}^n = \mathsf{exp}^n$. Let j be the 1st ordinal such that $\mathsf{exp}^j = \mathsf{exp}$. Then $\mathsf{Term}^j = \mathsf{exp}$. When we get to Term^j, we look to any constraints that would trigger a recalculation of Term. In this case, there is the constraint $t.\ell$. For $n > j$ we define a function f as follows:

$$f(\mathsf{Term}^n) = \left\{ x \;\middle|\; \begin{array}{l} (a \in x \wedge b \in x) \Leftrightarrow \\ \left(a \approx_\alpha b \wedge a, b \in \begin{array}{l} (\bigcup \mathsf{Term}^n) \cup \{\lambda P_v.P_t \mid v \in V \wedge t \in \mathsf{Term}^n\} \\ \cup \{P_{t_1}\, P_{t_2} \mid t_1, t_2 \in \mathsf{Term}^n\} \\ \cup \{P_t.P_\ell \mid t \in \mathsf{Term}^n \wedge \ell \in \mathsf{Label}\} \end{array} \right) \end{array} \right\}$$

For $n > j$ such that n is less than the least fixed point of f on Term^i (with the ordering given by $\subseteq$), we have $\mathsf{Term}^{n+1} = f(\mathsf{Term}^n)$.

For a limit point ε we have $\mathsf{Term}^\varepsilon = \left\{ x \;\middle|\; \begin{array}{l} (a \in x \wedge b \in x) \Leftrightarrow \\ (a, b \in \bigcup \bigcup_{i<\varepsilon} \mathsf{Term}^i \wedge a \approx_\alpha b) \end{array} \right\}$

At m equal to the least fixed point of f, we look for any constraints which may trigger recalculation. We see the constraint r. Each recalculation of Term causes a recalculation of Rec, so, for each ordinal i, $r \in \mathsf{Rec}_i = \mathsf{Label} \overset{\text{fin}}{\to} \mathsf{Term}$.

We define a function g as follows: For $n <$ the least fixed point of f on Term^i with the ordering given by $\subseteq$, we have $g(\mathsf{Term}^n) = \mathsf{Term}^{n+1}$. In a sense we use g to "extend" the step function f beyond the ordinal giving us fixed point.

For $n >$ the least fixed point of f (on Term^i with the ordering given by $\subseteq$:

$$g(\mathsf{Term}^n) = \left\{ x \;\middle|\; \begin{array}{l} (a \in x \wedge b \in x) \Leftrightarrow \\ \left(a \approx_\alpha b \wedge a, b \in \begin{array}{l} (\bigcup \mathsf{Term}^n) \cup \{\lambda P_v . P_t \mid v \in V \wedge t \in \mathsf{Term}^n\} \\ \cup \{P_{t_1}\, P_{t_2} \mid t_1, t_2 \in \mathsf{Term}^n\} \\ \cup \{P_t . P_\ell \mid t \in \mathsf{Term}^n \wedge \ell \in \mathsf{Label}\} \cup \mathsf{Rec}^n \end{array} \right) \end{array} \right\}$$

As there are no other constraints which trigger a recalculation, Term is the least fixed point of g on Term^i. Since, for $j < i$, $\mathsf{Rec}^j \subseteq \mathsf{Rec}^i$, this fixed point exists. $\square$

Before we add rewriting rules, we need extra machinery for generalised β-reduction. For terms rewritten in item notation Nederpelt (1992) generalised beta reduction applies to any couple. Item notation writes $[x]Y$ for the abstraction $\lambda x.y$ and $(B)A$ for the application AB. Generalised beta reduction skips over application abstraction pairs. E.g. if β_{gen} is the name of the generalised β-reduction relation, then $(\lambda z.(\lambda x.\lambda y.N)P)RQ \overset{\beta_{gen}}{\to} (\lambda z.(\lambda x.N[y := Q])P)R$ may be written in item notation $(Q)(R)[z]P[x][y]N\beta_{gen}{\to}(R)[z]P[x]\{N[y := Q]\}$. In each case it skips over abstractions which are already paired off with an application. In item notation this is very clear. $(R)[z]$ and $(P)[x]$ are couples and so $(Q)[y]$ is a couple around them. To extend our grammar for Term with generalised β-reduction, we define the balanced segments ending with a hole. We add to our grammar the following rule based on one used by Chang & Felleisen (2012):

$$b \in \mathsf{bseg} ::= \square \mid b[\lambda x.b]\, t$$

We note that while the grammar above can be expressed easily with MathSyn (as we will show later) it cannot possibly be expressed using BNF. We can show that, unlike BNF, bseg is not context-free (This result has been proven already using the pumping lemma in Lemma 4.9 “Hole filling is not context-free”). Not only is bseg not context-free, but even BNF variants which allow for the generation of non-context-free grammars do not usually allow hole filling as the syntax is generated (we were unable to find any in our literature search). Even if we were to work with another BNF variant with far more expressive power than BNF, we would have to rewrite the above grammar.

Lemma 7.6 (MathSyn Supports Balanced Segments with a Hole)**.** *The set* bseg *has a model in MathSyn.*

Proof. When $\Box$ is read, we define $\mathsf{bseg}_0 = \{\Box\}$. Since no further calculations are invited by this constraint, we move on to the constraint $b[\lambda x.b]\,t$. Since, in MathSyn, $\approx$ is always literal equivalence on an object containing a hole (although crucially subterms without holes can still be quotiented by their own equivalences), and $b[\lambda x.b]\,t$ doesn't trigger a recalculation of Term we can write:

$$\mathsf{bseg}_{n+1} = \mathsf{bseg}_n \cup \{\{b[\lambda x.b]\,t\} \mid b \in \mathsf{bseg}_n, t \in \mathsf{Term}\}$$

And, for a limit point ε, we have $\mathsf{bseg}_\varepsilon = \bigcup\limits_{i<\varepsilon} \mathsf{bseg}_i$

Since there are no further constraints, bseg is the least fixed point of f on bseg_i, such that $f(\mathsf{bseg}_n) = \mathsf{bseg}_{n+1}$. $\Box$

Notably, while the parts of each $a \in \mathsf{bseg}$ above $\Box$ are alone in their equivalence class, each record term $t \in \mathsf{Term}$, remains quotiented by α-equivalence.

We can now add rewriting rules, like so:

...Our rewriting rules are

$$r.l \overset{RCD}{\rightarrow} t \quad \text{if}\,(l,t) \in r \qquad\qquad b[\lambda v.t_1]\,t_2 \overset{\beta_{gen}}{\rightarrow} b[t_1[v := t_2]] \qquad\qquad \lambda v.t_1 v \overset{\eta}{\rightarrow} t_1$$

We also add the following rule, where $*$ can be either β_{gen}, RCD or η:

$$r_1 \overset{*}{\rightarrow} r_2 \qquad\qquad \text{if}\,t_1 \overset{*}{\rightarrow} t_2\,\text{and}\,(l_1,t_1) \in r_1\,\text{and}\,(l_1,t_2) \in r_2$$

Then MathSyn gives generalised β-reduction, η-reduction and the rewriting rule for records, RCD, as the Term-compatible closures of the smallest β_{gen}, η and RCD satisfying all of the functions above.

Chapter 8

Related Work, Conclusions and Future Work

8.1 Advance Over State of the Art

There has already been some work done on CSM that uses BNF-style notation, however, to our knowledge, no other authors have highlighted all the issues we have, or presented it as a significant departure from BNF. We take a look at some of the existing work in this area.

Ott Sewell et al. (2007) provides a language for writing specifications like those written with CSM. An Ott specification can be written automatically into BNF-style notation. However, the focus of this PhD is moving the other way, i.e. it is on translating BNF-style notation into a mathematical formalism. Ott does not offer support for interpreting CSM without requiring it to be specified in a theorem-prover friendly format. As a key design principle, we wish to provide a general mathematical intuition suitable for translation to multiple theorem provers, whereas Ott focuses on translating to Coq 8.3, HOL 4 and Isabelle directly, but offers less support for those seeking a general mathematical intuition. Our method should be reasonable to translate to any theorem prover which can model ZFC (as this is the metatheory of our model). Ott only allows contexts with a single hole, does not allow for hole-filling operations to appear in the clause of a production rule and currently does not support rules being used coinductively. Ott also does not handle the common practice of using mathematical text outside of the production rules as part of its definition. We handle more cases of context hole filling. We allow integration with mathematical text. It seems more feasible MathSyn could be extended to handle co-induction as the kinds of trees one could feasibly define coinductively are already part of our universe of Objects and can be picked out by means of set comprehension, whereas Ott contains no such structures, though this remains future work.

There are also a variety of systems supporting higher order abstract syntax (HOAS), some examples are Hybrid Battell & Felty (2016), Twelf Schürmann (2009) and Beluga Pientka & Dunfield (2010). HOAS is a technique for the representation of abstract syntax trees for languages with variable binders. There are a number of reasons why this technique is useful. First, it makes the binding structure of a program explicit: just as there is no need to explain operator precedence in a first order abstract syntax representation, there is no need to have the rules of binding and scope at hand to interpret a HOAS representation. Second, programs that are alpha-equivalent (differing only in the names of bound variables) have identical representations in HOAS, which can make equivalence checking more efficient. While HOAS bears some resemblance to CSM as it is used with BNF-style notation and is widely used, it does not support all the same uses. For example, Dami (1998), uses an BNF-style notation to talk about dynamic binding. It also is not as flexible as our approach in providing syntactic equivalences beyond equivalences on binding.

Steele (2017) began documenting the grammars which MathSyn deals with as part of CSM. He covers many of the notational variants of BNF, including some MBNF grammars. However, Steele's interest is primarily with making an initial attempt at documenting computer science meta-notation. He is interested in the differences between CSM and earlier versions, such as BNF, only insofar as they help this goal and remind us of alternative choices that might have been better than the ones we ended up with. While the grammars which MathSyn deals with are examples of CSM, he does not discuss how the underlying mathematical structure of some of these differs wildly from grammars based around rewriting relations on strings and the parse trees derived from them, such as BNF and many of its formal variants. He admitted his exploration of CSM was only a start and called for rigorous exploration and documentation of this notation. Our work is an attempt to fit this call.

Grewe et al. (2018) discuss the exploration of language specifications with first-order theorem provers. However, they still require the reader to be able to intuitively translate language specifications to a sufficiently formal language first. This is the part of language specification checking this PhD aims to help with.

(Reynolds 2009, 1-51) gives an attempt at a definition of CSM used with BNF-style notation. This is the closest to the basic groundwork for a human readable definition of this notation which we could find after looking through the books in our collection, which he calls "abstract syntax"[1]. However, he only deals with context-free grammars and in many places he proceeds by example.

None of the above examples deal with sets generated by BNF-style syntax which have very large infinite things in them (i.e., infinite sets or trees with infinite depth) or with the mixing of BNF and set theory. We have explored these in our introduction to MBNF and they are supported insofar as MathSyn provides a system for defining sets of objects and the objects in the set Object may be infinitely "deep" (i.e. they may be represented only by an infinite tree) furthermore they may contain any countable equivalence over arrangements. All that is necessary to make use of this power is a little extra text describing how these are picked out of Object as some of the more complex functionality of MathSyn has been defined for the inductive case only.

8.2 Conclusions

Following on from Steele, we explore a previously unexplored area of computer Science metanotation we call MBNF. One concrete contribution arising from this is a series

[1]We do not use the term "abstract syntax", because some of the things we are interested in are concrete syntax. For example, if we were to write $\lambda x.e$ in the form of an abstract syntax tree, we would not be interested that the x and the e are arranged with a dot between them and a λ in front of them. Rather, ($\lambda\square.\square$) would just be a name for a particular function taking two arguments of a certain type.

of examples and proofs showing MBNF to be strictly more expressive than BNF and some of its variants and showing it to deal with syntactic entities other than strings which may be infinite rather than finite and which are defined modulo some notion of equivalence, where it is largely irrelevant to understanding the syntax whether or not members of that equivalence class have a canonical order. Further contributions are the proof that some constraints provided by an MBNF ruleset may not be solvable and the establishment of an incompleteness result for MBNF. This motivates the development of a system for defining some subset of MBNF grammars safely.

We outlined one such system, which we call MathSyn. MathSyn offers tools to handle some non MBNF "Grammars" which appear in CSM and gives a notion of what it means to handle the strucures of math text together with syntactic equivalences and operations for manipulating syntax, even outside of MBNF. It allows a flexible notion of equivalence (including over sub-objects), which is sufficient to deal with most notions of binding (i.e., both those expressed as countable equalities on syntax trees as well as their subtrees up to name swapping, including for typed groups of names and the use of arithmetic in syntax for De Bruijn indices while permitting any syntax to be used for the binder itself), equality up to reordering of finitely many chunks of syntax, etc. as well as any countable author-defined equivalence. MathSyn allows syntax to be combined in a mathematical layout, encodes invisible bracketing structure and allows use of the inherited structure of objects. While MathSyn retains functionality inherited from BNF, it also allows hole filling and inherits more functionality from set theory. MathSyn offers facilities which help deal with semantics in addition to syntax. MathSyn also explicitly allows extension of its syntax with sets of mathematical objects, which needn't be serialisable. We have proven that the syntactic entities MathSyn uses have a model in ZFC. We have also proven that some of the more advanced machinery of MathSyn is well defined in the inductive case.

We outline how MBNF production rules function and provide one way of ensuring production rules select a subset of Object. We show that production rules plus MathSyn allow for the definition of multiple grammars including one for the λ-calculus with β-reduction and and records that makes use of hole filling within the syntax, nested equivalences and function sets.

8.3 Future Work

8.3.1 Coinduction Analogues in MathSyn

Some of the lemmas in this work are provided only for the inductive case. It is the case that Object contains all countable equivalences over Arrangement and most cases of nesting objects that one might typically define with coinduction. While there are several ways of writing coinductive definitions, coinduction in the area of parsing syntax traditionally works with streams of symbols with a head and a tail. These do not closely resemble syntactic objects. We can sketch certain intuitions about selecting objects one might traditionally define using coinduction.

We give the following as a guideline to authors hoping to expand our system to coinductive readings for production rules: for given a set of constraints we may look only at those subsets of Object which are well-formed $\approx$-equivalence classes and whose primitive constructors match at the top level and then at the next level down and so on (similarly to how we defined the inductive reading but working the other way). For a limit point we may look at the intersect of each of these steps. We may take the greatest fixed point of a function which takes each step in this process to the subsequent step and this should correspond to something like a coinductive reading of a set of production rules.

One issue for this is that it might require a definition of primitive constructor decomposition and well-formed $\approx$-equivalence classes which is not inductive. Ultimately coinductive uses would require a reworking of several lemmas and definitions which could no longer be thought of as necessarily being independently defined (at least for the general case). This was simply too cumbersome and redundant for an introduction to interpreting MBNF and related parts of CSM, it would require that much of the material in chapters 5 and 6 were rewritten slightly and treated simultaneously. Producing a coinductive version, or, given that we do not define objects as streams, some version that dealt with entities traditionally expressed via coinduction, is an avenue for authors interested in expanding on MathSyn.

8.3.2 Parsing Analogues in MathSyn

We have not dealt with parsing in this thesis as usually the parser for an MBNF will be presented separately using a different grammar altogether. However, there is a close analogue to parsing in MathSyn. Parsing a syntactic object may be thought of as similar to proving it belongs to a set of syntactic objects given by an MBNF rule set by means of primitive constructor decomposition. Authors may wish to apply this intuition with respect to individual cases, but also proving this more generally given a framework written in MBNF might be a fruitful area for future research.

Bibliography

Aggarwal, A. (n.d.), *Ashtadhyayi of Panini Complete*, Lulu.com.
URL: *https://books.google.co.uk/books?id=FMB-DwAAQBAJ*

Aho, A. V. (1968), 'Indexed grammars—an extension of context-free grammars', *J. ACM* **15**(4), 647–671.
URL: *https://doi.org/10.1145/321479.321488*

Ariola, Z. M. & Felleisen, M. (1994), The call-by-need lambda calculus, Technical Report CIS-TR-94-23, Department of computer science, University of Oregon.

Ariola, Z. M., Maraist, J., Odersky, M., Felleisen, M. & Wadler, P. (1995), A call-by-need lambda calculus, *in* 'Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages', p. 233–246.

Attenborough, M. (2003), *Mathematics for Electrical Engineering and Computing*, Elsevier Science.
URL: *https://books.google.co.uk/books?id=CcwBLa1G8BUC*

Backus, J. W. (1959), The syntax and semantics of the proposed international algebraic language of the zurich acm-gamm conference., *in* 'IFIP Congress', pp. 125–131.

Backus, J. W., Bauer, F. L., Green, J., Katz, C., McCarthy, J., Perlis, A. J., Rutishauser, H., Samelson, K., Vauquois, B., Wegstein, J. H., van Wijngaarden, A. & Woodger, M. (1963), 'Revised report on the algorithm language algol 60', *Commun. ACM* **6**(1), 1–17.

Barnes, J. G. P. (1980), ‘An overview of ada’, *Software Practice and Experience* **10**, 851–887.

Battell, C. & Felty, A. (2016), The logic of hereditary harrop formulas as a specification logic for hybrid, *in* ‘Proceedings of the Eleventh Workshop on Logical Frameworks and Meta-Languages: Theory and Practice’, LFMTP ’16, ACM, New York, NY, USA, pp. 3:1–3:10.
URL: *http://doi.acm.org/10.1145/2966268.2966271*

Berstel, J., Lauve, A., Reutenauer, C. & Saliola, F. V. (2009), *Combinatorics on words. Christoffel words and repetitions in words.*, AMS.

Bhate, S. & Kak, S. (1993), ‘Pāṇini’s grammar and computer science’, *Annals of the Bhandarkar Oriental Research Institute* **72**, 79–94.

Bloo, R., Kamareddine, F. & Nederpelt, R. (1996), ‘The barendregt cube with definitions and generalised reduction’, *Information and Computation* **126**(2), 123 – 143.

Boute, R. T. (1980), ‘Simplifying ada by removing limitations’, *SIGPLAN Not.* **15**(2), 17–29.
URL: *http://doi.acm.org/10.1145/947586.947588*

Cardelli, L. (1988), ‘A semantics of multiple inheritance’, *Information and Computation* **76**(2), 138 – 164.

Cardelli, L. & Mitchell, J. C. (1990), Operations on records, *in* M. Main, A. Melton, M. Mislove & D. Schmidt, eds, ‘Mathematical Foundations of Programming Semantics’, Springer New York.

Castagna, G., Gesbert, N. & Padovani, L. (2009), ‘A theory of contracts for web services’, *ACM Trans. Program. Lang. Syst.* **31**(5), 19:1–19:61.
URL: *http://doi.acm.org/10.1145/1538917.1538920*

Chang, S. & Felleisen, M. (2012), The call-by-need lambda calculus, revisited, *in* Seidl (2012), pp. 128–147.

Chomsky, N. (1956), 'Three models for the description of language', *IRE Transactions on Information Theory* **2**, 113–124.

Crocker, D., Ed. & Overell, P. (2008), 'Augmented bnf for syntax specifications: Abnf', Internet Requests for Comments.
URL: *http://www.rfc-editor.org/info/rfc5234*

Crocker, D. & Overell, P. (1997), Augmented bnf for syntax specifications: Abnf, RFC 2234, RFC Editor.
URL: *https://www.rfc-editor.org/info/rfc2234*

Dami, L. (1998), 'A lambda-calculus for dynamic binding', *Theoretical Computer Science* **192**(2), 201 – 231.
URL: *http://www.sciencedirect.com/science/article/pii/S0304397597001503*

Davey, B. A. & Priestley, H. A. (2002), *Introduction to Lattices and Order*, 2 edn, Cambridge University Press, Cambridge.

de Bruijn, N. (1972), '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.
URL: *https://www.sciencedirect.com/science/article/pii/1385725872900340*

de Bruijn, N. G. (1970), The mathematical language automath, its usage, and some of its extensions, *in* M. Laudet, D. Lacombe, L. Nolin & M. Schützenberger, eds, 'Symposium on Automatic Demonstration', Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 29–61.

Dolan, S. & Mycroft, A. (2017), Polymorphism, subtyping, and type inference in mlsub, *in* Fluet (2017).

Dybjer, P. & Setzer, A. (2003), 'Induction-recursion and initial algebras', *Ann. Pure Appl. Log.* **124**, 1–47.

Eberhart, C., Hirschowitz, T. & Seiller, T. (2015), An Intensionally Fully-abstract Sheaf Model for π^*, *in* L. S. Moss & P. Sobocinski, eds, '6th Conference on Algebra and

Coalgebra in Computer Science (CALCO 2015)', Vol. 35 of *Leibniz International Proceedings in Informatics (LIPIcs)*, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, pp. 86–100.

Farrel, A. (2009), Routing backus-naur form (rbnf): A syntax used to form encoding rules in various routing protocol specifications, RFC 5511, RFC Editor.
URL: *https://tools.ietf.org/html/rfc5511*

Fluet, M., ed. (2017), *POPL'17 :Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, ACM, New York, NY, USA.

Ford, B. (2004), Parsing expression grammars: A recognition-based syntactic foundation, *in* 'Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages', ACM, pp. 111–122.

Forsberg, M. & Ranta, A. (2005), 'The labelled bnf grammar formalism'.

Forster, Y., Kirst, D. & Wehr, D. (2020), Completeness theorems for first-order logic analysed in constructive type theory, *in* S. Artemov & A. Nerode, eds, 'Logical Foundations of Computer Science', Springer International Publishing, Cham, pp. 47–74.

Frenkel, A., Bar-Hittel, Y. & Levy, A. (1973), *Foundations of Set Theory*, Elsevier, Customer Service Department P.O. Box 211 Amsterdam.

Frumin, D., Gondelman, L. & Krebbers, R. (2019), Semi-automated reasoning about non-determinism in c expressions, *in* L. Caires, ed., 'Programming Languages and Systems', Springer International Publishing, Cham, pp. 60–87.

Garnock-Jones, T. & Felleisen, M. (2016), Coordinated concurrent programming in syndicate, *in* P. Thiemann, ed., 'Programming Languages and Systems', Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 310–336.

Germane, K. & Might, M. (2017), A posteriori environment analysis with pushdown delta cfa, *in* Fluet (2017).

Grewe, S., Erdweg, S., Pacak, A., Raulf, M. & Mezini, M. (2018), 'Exploration of language specifications by compilation to first-order logic', *Sci. Comput. Program.* **155**, 146–172.

Grigore, R. (2017), Java generics are turing complete, *in* Fluet (2017).

Groote, P. (1993), The conservation theorem revisited, *in* M. Bezem & J. F. Groote, eds, 'Typed Lambda Calculi and Applications', Springer Berlin, pp. 163–178.

Harbison, S. P. (1984), *C - A Reference Manual*, Prentice-Hall.

Hausmann, D. & Schröder, L. (2019), Optimal satisfiability checking for arithmetic μ-calculi, *in* M. Bojańczyk & A. Simpson, eds, 'Foundations of Software Science and Computation Structures', Springer International Publishing, Cham, pp. 277–294.

Hopcroft, J. E., Motwani, R. & Ullman, J. D. (2006), *Introduction to Automata Theory, Languages, and Computation (3rd Edition)*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Hopcroft, J. E. & Ullman, J. D. (1979), *Introduction to Automata Theory, Languages, and Computation*, Addison-Wesley Publishing Company.

Inoue, J. & Taha, W. (2012), Reasoning about multi-stage programs, *in* Seidl (2012).

Ion, P. D. F., Poppelier, N., Carlisle, D. & Miner, R. R. (2001), Mathematical markup language (MathML) version 2.0, W3C recommendation, W3C. https://www.w3.org/TR/MathML2/chapter3.html.

ISO (1996), Information technology – Syntactic metalanguage – Extended BNF, Standard, International Organization for Standardization, Geneva, CH.

ISO (2015), Information technology – Open Document Format for Office Applications (OpenDocument) v1.2 – Part 1: OpenDocument Schema, Standard, International Organization for Standardization, Geneva, CH.

Jones, S. P., Augustsson, L., Barton, D., Boutel, B., Burton, W., Fasel, J., Hammond, K., Hinze, R., Hudak, P., Hughes, J., Johnsson, T., Jones, M., Launchbury, J., Meijer, E., Peterson, J., Reid, A., Runciman, C. & Wadler, P. (1999), 'Haskell 98: A non-

strict, purely functional language'.
URL: *https://www.haskell.org/definition/*

Kadvany, J. (2007), 'Positional value and linguistic recursion', *Journal of Indian Philosophy* **35**(5/6), 487–520.
URL: *http://www.jstor.org/stable/23497283*

Kamareddine, F. & Nederpelt, R. (1995), 'Refining reduction in the lambda calculus', *Journal of Functional Programming* **5**(4), 637–651.

Kamareddine, F., Wells, J., Zengler, C. & Barendregt, H. (2014), Computerising mathematical text, *in* J. H. Siekmann, ed., 'Computational Logic', Vol. 9 of *Handbook of the History of Logic*, North-Holland, pp. 343 – 396.
URL: *http://www.sciencedirect.com/science/article/pii/B9780444516244500083*

Kfoury, A. J. & Wells, J. B. (1995), New notions of reduction and non-semantic proofs of strong /spl beta/-normalization in typed /spl lambda/-calculi, *in* 'Proceedings of Tenth Annual IEEE Symposium on Logic in Computer Science', pp. 311–321.

Knuth, D. E. (1986), *The TeXbook*, Addison-Wesley Professional.

Kuratowski, C. (1921), Sur la notion de l'ordre dans la théorie des ensembles, *in* 'Fundamenta Mathematicae'.

Lévy, A. (1979), *Basic set theory*, Springer-Verlag, Berlin.

Llana Díaz, L. F. & Núñez, M. (1997), Testing semantics for unbounded nondeterminism, *in* C. Lengauer, M. Griebl & S. Gorlatch, eds, 'Euro-Par'97 Parallel Processing', Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 538–545.

Mailloux, B. J., Peck, J. E. L., Koster, C. H. A. & van Wijngaarden, A. (1969), *Report on the Algorithmic Language ALGOL 68*, Springer Berlin Heidelberg, Berlin, Heidelberg.

Mann, P. B. (2006), 'A translational bnf grammar notation (tbnf)', *SIGPLAN Not.* **41**(4), 16–23.

Markov, A. (1947), 'Impossibility of certain algorithms in the theory of associative systems', *Proceedings of the USSR Academy of Sciences* **58**, 353–356.

Matsumoto, Y. (1988), *Ruby Language Reference Manual*, 1.4.6 edn.
URL: *http://docs.huihoo.com/ruby/ruby-man-1.4/yacc.html*

McCarthy, J. (1964), A formal description of a subset of algol, Technical report, Stanford Artificial Intelligence Project.

Milner, R. (1999), *Communicating and Mobile Systems: The &Pgr;-calculus*, Cambridge University Press, New York, NY, USA.

Milner, R., Parrow, J. & Walker, D. (1992), 'A calculus of mobile processes, i', *Inf. Comput.* **100**(1), 1–40.
URL: *http://dx.doi.org/10.1016/0890-5401(92)90008-4*

Mislove, M. W. (1995), 'Denotational models for unbounded nondeterminism', *Electronic Notes in Theoretical Computer Science* **1**, 393 – 410. MFPS XI, Mathematical Foundations of Programming Semantics, Eleventh Annual Conference.
URL: *http://www.sciencedirect.com/science/article/pii/S1571066104000234*

Morris, Jr., J. H. (1968), Lambda-calculus models of programming languages, PhD thesis, Massachusets Institute of Technology.

Mosses, P. D. (1975), Mathematical semantics and compiler generation, PhD thesis, University of Oxford.

Nederpelt, R. (1992), *The fine-structure of lambda calculus*, Computing science notes, Technische Universiteit Eindhoven.

Nederpelt, R. (1994), Strong normalization in a typed lambda calculus with lambda structured types, *in* R. Nederpelt, J. Geuvers & R. de Vrijer, eds, 'Selected Papers on Automath', Elsevier, pp. 389 – 468.

Pfenning, F. & Elliott, C. (1988), Higher-order abstract syntax, *in* 'Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implemen-

tation', PLDI '88, ACM, New York, NY, USA, pp. 199–208.
URL: *http://doi.acm.org/10.1145/53990.54010*

Pientka, B. & Dunfield, J. (2010), Beluga: A framework for programming and reasoning with deductive systems (system description), *in* J. Giesl & R. Hähnle, eds, 'Automated Reasoning', Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 15–21.

Pierce, B. C. (2002), *Types and Programming Languages*, 1st edn, The MIT Press.

Post, E. L. (1943), 'Formal reductions of the general combinatorial decision problem', *American Journal of Mathematics* **65**(2), 197–215.
URL: *http://www.jstor.org/stable/2371809*

Post, E. L. (1947), 'Recursive unsolvability of a problem of thue', *J. Symbolic Logic* **12**(1), 1–11.
URL: *https://projecteuclid.org:443/euclid.jsl/1183395203*

Radin, G. & Rogoway, H. P. (1965), 'Npl: Highlights of a new programming language', *Commun. ACM* **8**(1), 9–17.
URL: *http://doi.acm.org/10.1145/363707.363708*

Rahli, V., Bickford, M. & Constable, R. L. (2017), Bar induction: The good, the bad, and the ugly, *in* '2017 32nd Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)', pp. 1–12.

Ranta, A. (2004), 'Grammatical framework', *J. Funct. Program.* **14**(2), 145–189.
URL: *http://dx.doi.org/10.1017/S0956796803004738*

Redziejowski, R. R. (2013), 'From ebnf to peg', *Fundam. Inf.* **128**(1-2), 177–191.
URL: *http://dx.doi.org/10.3233/FI-2013-940*

Reiser, J. F. (1976), Sail (aim-289), Technical report, Stanford Artificial Intelligence Laboratory.

Reynolds, J. C. (2009), *Theories of Programming Languages*, 1st edn, Cambridge University Press, New York, NY, USA.

Rossum, G. (1995), Python reference manual, Technical report, PythonLabs, Amsterdam, The Netherlands, The Netherlands.

Schürmann, C. (2009), The twelf proof assistant, *in* S. Berghofer, T. Nipkow, C. Urban & M. Wenzel, eds, 'Theorem Proving in Higher Order Logics', Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 79–83.

Scowan, R. S. (1981), Method of defining syntactic metalanguage (bs 6154), Technical report, British Standards Institution.

Seidl, H., ed. (2012), *Programming Languages and Systems*, Springer.

Sewell, P., Nardelli, F. Z., Owens, S., Peskine, G., Ridge, T., Sarkar, S. & Strniša, R. (2007), 'Ott: Effective tool support for the working semanticist', *SIGPLAN Not.* **42**(9), 1–12.

Shaw, M. (1981), *Alphard: Form and Content*.

Shaw, M., Wulf, W. A. & London, R. L. (1977), 'Abstraction and verification in alphard: Defining and specifying iteration and generators', *Commun. ACM* **20**(8), 553–564.
URL: *http://doi.acm.org/10.1145/359763.359782*

Spies, S., Krishnaswami, N. & Dreyer, D. (2021), 'Transfinite step-indexing for termination', *Proc. ACM Program. Lang.* **5**(POPL).
URL: *https://doi.org/10.1145/3434294*

Steele, Jr., G. L. (1990), *Common LISP: The Language (2Nd Ed.)*, Digital Press, Newton, MA, USA.

Steele, Jr., G. L. (2017), It's time for a new old language, PPoPP '17, ACM, New York, NY, USA, pp. 1–1.
URL: *https://www.youtube.com/watch?v=7HKbjYqqPPQ*

Tarski, A. (1955), 'A lattice-theoretical fixpoint theorem and its applications.', *Pacific J. Math.* **5**(2), 285–309.

Thue, A. (1914), 'Probleme über veränderungen von zeichenreihen nach gegebenen regeln.', *Skrifter udgivne af Videnskabsselskabet i Christiania. I, Mathematisk-naturvidenskabelig klasse.* **10**.

Tobisawa, K. (2015), A meta lambda calculus with cross-level computation, *in* 'POPL '15', pp. 383–393.

Toronto, N. & McCarthy, J. (2012), Computing in cantor's paradise with λ zfc, *in* T. Schrijvers & P. Thiemann, eds, 'Functional and Logic Programming', Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 290–306.

Vijay-Shanker, K. & Weir, D. J. (1994), 'The equivalence of four extensions of context-free grammars', *Mathematical systems theory* **27**(6), 511–546.
URL: *https://doi.org/10.1007/BF01191624*

von Neumann, J. (1923), 'Zur Einführung der transfiniten Zahlen', *Acta Scientiarum Mathematicarum (Szeged)* **1**(4), 199–208.

Wiener, N. (1967), A simplification of the logic of relations, *in* J. van Heijenoort, ed., 'From Frege to Gödel: A Source Book in Mathematical Logic', Harvard University Press, Cambridge MA.

Wirth, N. (1968), 'Pl360, a programming language for the 360 computers', *J. ACM* **15**(1), 37–74.
URL: *http://doi.acm.org/10.1145/321439.321442*

Wulf, W. A., Russell, D. B. & Habermann, A. N. (1971), 'Bliss: A language for systems programming', *Commun. ACM* **14**(12), 780–790.
URL: *http://doi.acm.org/10.1145/362919.362936*

Zaytsev, V. (2012), 'The grammar hammer of 2012', *CoRR* **abs/1212.4446**.
URL: *http://arxiv.org/abs/1212.4446*

Index

$R^{=}$, 13

$(S_1, S_2)-\mathsf{Context}$, $S-\mathsf{Context}$, 96

$A\vec{O}^{\approx}$, 92

$O\vec{O}^{\approx}$, 92

$S \rightarrow T$, 14

$S \times T$, 13

S^*, 15

$[A]_{\approx}$, 88, 90–95, 99, 100, 102–106, 111–115, 131, 143

$[v_0, \ldots, v_n]$, 15

$\aleph_0$, $\aleph_1$, 20

α-Equivalence, 112

$\approx$, 88

$\approx$-context-hole filling, 91

$\approx$-well-/ill-formed, 88

p.c.d.(O), 104

$\mathsf{Arrangement}$, 81

OPA_i, 123

Object, 81

Object_M, 86

$\mathsf{Pointer}$, 81

Symbol, 81

fun, 123

isub, 98

ln, 98

ptr_M, 86

$\mathsf{swap}(O_{\mathsf{x}}, O_{\mathsf{y}}, O)$, 108

$\overbrace{\cdot}^{\mathsf{dc}}$, 99–104, 106, 107, 110, 111

ε, 15

$\vec{v}$, 15

$\{X \mid P(X)\}$, 12

n-Tuple, 13

$R(X, Y)$, $X\ R\ Y$, $X \overset{R}{\rightarrow} Y$, 13

ABNF, 36

Abstract syntax tree, 1, 21, 38, 50, 105

Arity of an arrangement, 82

Arrangement equivalence, 88

Balanced segment, 51, 53

BNF, 20

Bound names, 109

Cardinal, 17, 61

Compatible closure, 96

Computer science metanotation, 4

Concatenation, 15

Concrete syntax, 47, 51

Context, 91

Context free, 22, 30, 41

CSM, 4

Decomposition, 99

Direct derivation, 56

EBNF, 27
Empty sequence, 15, 117
Equivalence, 13
Equivalence class, 14
Evaluating an expression, 128
Evaluating expressions, 128
Expression, 128

Finite tree, 20
Fixed point, 14
Free names, 109, 114
Function, 14

Group of names, 108

Immediate sub-objects, 98
Indexed grammar, 55–57
Induction on a set, 14
Infinitary operator, 62, 63, 71, 73
Injective function, 14
Interior node, 20

Language of a metavariable, 23, 24
Language of a non-terminal, 21
Lattice, 16
LBNF, 38
Leaf node, 20
Length of a sequence, 15
Limit ordinal, 16

Math text, 2, 47–49, 53, 81
MathSyn, 7, 78–80, 87, 90, 95, 112, 126, 141, 143, 144, 152
MBNF, 22
MBNF alternative, 23
MBNF production rule, 23
MBNF rule set, 22
MBNF syntax, 24
Metavariable, 12

Names, 108, 112
Non-terminal, 21

Order preserving, 14
Ordered pairs, 13
Ordinal, 16

Parsing expression grammars, 32, 155
Partial order, 14
Pointer to an object, 81, 86, 89
Primitive constructor, 104
Primitive constructor decomposition, 104
Product set, 13
Projection operator, 13
Pumping lemma, 30

RBNF, 37
Reflexive, 13
Relation, 13
Rewriting relation, 22, 96
Root node, 20

Sequences, 15
Serialisation, serialisable, 15
Solution to a grammar, 24
Spine, 51
Strict partial order, 14
Sub-arrangement, 115

Sub-object, 115

Substitution, 114

Substring, 30

Swapping names, 108

Syntactic arrangement, 81, 113

Syntactic object, 81, 84, 86, 98

TBNF, 38

Terminals, 21

Terminating relation, 14

Total order, 14

Tree, 20

Undecideable, 50

Word problem for semigroups, 40