EasyChair Preprint

№ 380

What Does This Notation Mean Anyway? BNF-style notation as it is actually used

David Feller, Joe Wells, Sébastien Carlier and Fairouz Kamareddine

July 26, 2018

What Does This Notation Mean Anyway?

BNF-Style Notation as it is Actually Used

D. A. Feller J. B. Wells S. Carlier F. Kamareddine

Following the introduction of BNF notation by Backus for the Algol 60 report and subsequent notational variants, a metalanguage involving formal "grammars" has developed for discussing structured objects in Computer Science and Mathematical Logic. We refer to this offspring of BNF as *Math-BNF* or *MBNF*, to the original BNF and its notational variants just as *BNF*, and to aspects common to both as *BNF-style*. MBNF is sometimes called abstract syntax, but we avoid that name because MBNF is in fact a concrete form and has a more abstract form. What all BNF-style notations share is the use of production rules roughly of this form:

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

Normally, such a rule says "every instance of $\circ_i$ for $i \in \{1, \ldots, n\}$ is also an instance of $\bullet$".

MBNF is distinct from BNF in the entities and operations it allows. Instead of strings, MBNF builds arrangements of symbols that we call *math-text*. Sometimes "syntax" is defined by interleaving MBNF production rules and other mathematical definitions that can contain chunks of math-text.

There is no clear definition of MBNF. Readers do not have a document which tells them how MBNF is to be read and must learn MBNF through a process of cultural initiation. To the extent that MBNF is defined, it is largely through examples scattered throughout the literature.

This paper gives MBNF examples illustrating some of the differences between MBNF and BNF. We propose a definition of *syntactic math text* (SMT) which handles many (but far from all) uses of math-text and MBNF in the wild. We aim to balance the goal of being accessible and not requiring too much prerequisite knowledge with the conflicting goal of providing a rich mathematical structure that already supports many uses and has possibilities to be extended to support more challenging cases.

1 Background and Motivation

MBNF is important to interpreting papers in theoretical computer science. Out of the 30 papers in the ESOP 2012 proceedings [23], 19 used MBNF, but not one used BNF.[1] We highlight some of the ways in which the notation we call MBNF differs from BNF to demonstrate the need for a definition.

Where BNF uses Strings, MBNF Uses Math-Text Parentheses for disambiguation are not needed in MBNF grammars and when an MBNF grammar specifies such parentheses they can often be omitted without any need to explain. When possible, MBNF takes advantage of the tree-like structure implicit in the layout of symbols on the page when features like superscripting and overbarring are used.

Instead of non-terminal symbols, MBNF uses *metavariables*[2], which appear in math-text and obey the conventions of mathematical variables. Metavariables are not distinguished from other symbols by annotating them as BNF does, but by font, spacing, or merely tradition.

[1] We chose ESOP 2012 because its book was the most recent conference proceedings that we had as a paper book. Because the first book we picked 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.

[2] We use metavariable to mean a variable at the meta-level which denotes something at an object-level.

Submitted to:
LFMTP 2018

In addition to arranging symbols from left to right on the page, math-text allows subscripting, 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 [14], or the Presentation MathML [11] and OpenDocument [12] standards. Here is a nonsense piece of Math-text to illustrate how it may be laid out:

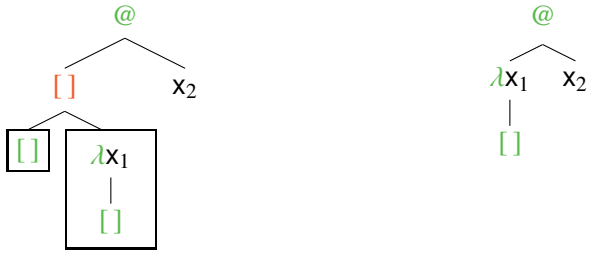
$$^{c}\!\downarrow a' = \check{p}\langle v''_x \odot a^{2+1}\rangle - \overline{f_x^n + \overline{y \cdot f\, j}} + \sum_{i=0}^{\infty} s_{i\in 1\ldots n} \xrightarrow{a,b,c} b\hat{a}$$

MBNF Is Aimed at Human Readers MBNF is meant to be interpreted by humans, not computers/parser generators. Authors may define a MBNF grammar in an article for humans and a separate grammar for use with a parser generator to build an implementation. MBNF defines entities not intended or expected to be serialized or parsed. MBNF grammars are typically missing features needed to disambiguate complex terms. Papers often put mathematical metalanguage inside MBNF notation.

MBNF Allows Powerful Operators Like Context Hole Filling (a.k.a. Tree Splicing) Chang and Felleisen [4, p 134] present an MBNF grammar defining the λ-term contexts with one hole where the spine[3] is a balanced segment[4] ending in a hole. We write $e@e$ instead of $e\,e$ and add parentheses. Concrete syntax and BNF-style notation are green. Metavariables are blue. Additional operators are red.

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

One can think of the context hole filling operation in this grammar ($[\,]$ in $(A[(\lambda x.A)]@e)$) as performing tree splicing operations within the syntax. Here are trees illustrating steps in building syntax trees for A:



These trees show the result of the second rule where each A is $[\,]$ and e is a variable. The tree on the left is the tree corresponding to $A[\lambda x.A]@e$ 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 $((\lambda x_1.[\,])@x_2)$. x_1 and x_2 are disambiguated instances of x. A metavariable assigned a value won't appear in the final tree. If it's 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. 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.

MBNF Mixes Math Stuff With BNF-Style Notation Germane and Might [7, pg 20] mix BNF-style notation freely with mathematical notation in such a way that the resulting grammar relies upon both sets

[3]The root node is on the spine. If A is applied to B by an application on the spine, 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.

[4]A balanced segment is one where each application has a matching abstraction and where each application/abstraction pair contains a balanced segment.

produced from the result of MBNF calculations and MBNF production rules which use metavariables defined using mathematical notation:

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

The results of math computations are interleaved with MBNF production rules, 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 (=)).

MBNF Has at Least the Power of Indexed Grammars Inoe and Taha [10, pg 361] use this MBNF:

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

This suggests that MBNF deals with the family of indexed grammars [9, p 389-390], which is yet another reason it's 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.

MBNF Allows Arbitrary Side Conditions on Production Rules An example of a production rule with a side condition can be found in Chang and Felleisen [4, p 134]:

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

It is possible to make side conditions that prevent MBNF rules from having a solution. A definition for MBNF can help in find conditions on side conditions that ensure MBNF rules actually define something.

MBNF "Syntax" Can Contain Very Large Infinite Sets Toronto and McCarthy [28, p 297] write:

$$e ::= \cdots \mid \langle t_{set}, \{e^{*\kappa}\} \rangle$$

We are told $\{e^{*\kappa}\}$ denotes "sets comprised of no more than κ terms from the language of e". It seems as though κ is also intended to be an inaccessible cardinal. This section of an MBNF for e is taken from a larger MBNF that contains a term which ranges over all the encodings of all the hereditarily accessible sets. BNF, by contrast, only deals with strings of finite length.

MBNF Allows Infinitary Operators Fdo, Díaz and Núñez [15, p 539] write an MBNF with the following operator, which the authors state is infinitary (i.e. we should regard I to be infinite):

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

The authors tell us the MBNF this is taken from is defined by regarding (M)BNF expressions as fixed point equations and a least fixed point can be found by bounding the size of the possible set of indices by some infinite cardinal. 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 deals with finite strings.

MBNF Allows Co-Inductive Definitions Eberhart, Hirschowitz and Seiller [6, p 94] intend the following MBNF to define infinite terms co-inductively:

$$\begin{aligned} P, Q &::= \Sigma_{i\in n} G_i \mid (P|Q) \\ G &::= \overline{a}\langle b\rangle.P \mid a(b).P \mid \nu a.P \mid \tau.P \mid \heartsuit.P \end{aligned}$$

We may think of co-inductive definitions as allowing us to define trees of infinite depth (i.e. trees in which paths may pass through infinitely many nodes), where BNF only deals with finite strings.

2 A Method to Allow Reading Some Uses of Mathematical "Syntax"

This section defines *syntactic math text* (SMT) which will allow reading some uses of math text as being "syntax" and standing for essentially themselves, e.g., 1 + 3 can continue to stand for 4 while $\lambda x.x$ can in some sense stand for itself. SMT plus a definition of the ::= notation allows us to interpret the more common uses of MBNF as they are written. It also provides some support for more complicated uses with a little extra machinery. We do not aim to cover every use of MBNF in the literature, but we hope to provide a good foundation which can be built upon.

As well as dealing with some of MBNF, SMT provides a more general notion of *objects* appearing within syntax that behave like equivalences over chunks of math-text representing syntax. This enables us to interpret working modulo equivalences on math-text representing syntax.

According to Kamareddine et al. [13], converting mathematical text to a form where it can be checked by a proof assistant is involves both human input and intermediary translations. Our proposal focuses on the translation, by the reader, of math-text used to define syntax, as it appears in a document, to a more formal structure, which is independent of any theorem prover format and could be easily translated to multiple provers. While we think that providing a semi-formal notion of this notation may be of interest to anyone creating and proof checking documents, we stress that this is not the main focus of our work. The gap we hope to fill is the lack of a good reference document for those yet to develop an intuition suited to working formally with SMT. Our proposal is intended to be descriptive rather than prescriptive. We aim to handle both historical documents and new works. For published uses of MBNF that our proposal fails to handle, this is a problem to be solved in future work. We do not aim at displacing the input languages of proof assistants or syntactic variants of BNF which already have definitions. The reader is invited to put themselves in the position of a grad student/mathematician interested in reading the theoretical computer science literature. Such a person can follow a complex definition, but may not have in depth knowledge of the background of a given text. If they were to search for resources relevant to understanding MBNF, they may have to extrapolate from a definition of BNF as context-free sets of strings, which could lead them to struggle with examples like those in Section 1. Alternatively, they may have to familiarise themselves with the language of some proof assistant/compiler generator and compare the MBNF grammars appearing in a number of documents with their formal implementations, or look for clarifications spread throughout the literature, which could take a long time and be quite challenging.

Even authors who do most of their work within a theorem prover quite often use SMT when writng about their results, so a semi-formal definition for SMT will still remain relevant as theorem provers become more popular. Furthermore, for as long as theorem provers remain somewhat cumbersome to use authors may appreciate a model which allows them to be confident their syntax definitions are realisable without fully implementing them in a theorem prover.

Our proposal relies as much as possible on the mathematical meta-level.[5] For example, we use

[5] We will give some loose heuristics for determining where text is meant to be on the syntax level or the meta-level in a

ellipses and related methods for abbreviating sequences from the mathematical meta-level. Incomplete definitions (relying on some choice of metavariable) cause the resulting grammar to be defined as the output of a function depending on this choice. Any otherwise pointless statement of the form $x \in S$ declares x and any *decorated* x (e.g., x_1, x_2, ..., x', x'', etc.) as a variable ranging over S. Metavariables appearing without a quantifier are bound by an implicit for all quantifier in the outermost position of whatever "context" they appear in.[6]

2.1 Objects, Arrangements, and Symbols

We now define the main notion of syntactic *objects* and the auxiliary notion of *arrangements*. In essence, syntactic objects are arrangements of *symbols*, numbers, and pointers to subobjects, where the arrangement can include left-to-right sequencing, superscripting, subscripting, overlining etc. To support α-conversion and operators that are associative, commutative, idempotent, etc., the objects are defined so that in effect they work modulo an equivalence relation on arrangements that is defined separately. We use pointers to subobjects inside objects rather than the subobjects themselves, because the sets within the model for objects would be too large otherwise and because we wanted to allow for objects to be nested within themselves, provided some syntax is added as part of this nesting. By using pointers we are able to include all equivalences over regular trees within our set of objects.

Let s range over the set Symbol containing syntactic *symbols* to be used in arrangements. We require that Symbol is disjoint from all other sets defined here. We also require that some symbols are *not* in Symbol, namely the square brackets ("[" and "]") and the special square symbol $\square$ (which represents a hole in which an object can be placed). The symbols can include letters, parentheses and other parenthesis-like symbols (e.g., $\langle$ and $\rangle$ and $[\![$ and $]\!]$), punctuation, and other symbols. Letters (Roman or Greek) used as syntactic symbols will be typeset using an upright sans-serif font to distinguish them from metavariables which are written in a slanted serif font (generally italics). For example, a, C, λ, and Γ could be syntactic symbols while a, C, λ, and Γ would be metavariables. We avoid using any particular letter both ways, except for symbols used in *names*, where for example x_i could be a syntactic name at the same time as x could be a metavariable ranging over names (see section 2.5).

The set Object of syntactic *objects* and the set Arrangement of syntactic *arrangements* are defined simultaneously. Let O range over Object and let A range over Arrangement. We represent each object by a member of the set Pointer. Let P_O be the pointer that indicates O. Let $\approx \subset \mathsf{Arrangement} \times \mathsf{Arrangement}$ be an equivalence relation that is reflexive on Arrangement. We require that if $A_1 \approx A_2$ and $A_1 \neq A_2$, then neither A_1 nor A_2 may have used the special object $\square$ in their construction, we do this to avoid any ambiguity in context hole filling which is defined in Section 2.3.

The sets Object and Arrangement are 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, numbers, and overlined arrangements. For any symbol s, pointer P_O, number $n \in \mathbb{N}$, and non-empty arrangement $A \neq \epsilon$, all of the following are in Arrangement: s, P_O, n, and $\overline{A}$. Furthermore, these are all *core arrangements*, which are ranged over by the metavariable $\hat{A}$.

longer version of this document. For this version, we require only that readers can use our definition provided they are able to guess at which pieces of syntax are on the meta-level.

[6] The notion of "context" employed here is deliberately vague. Not every context will span the whole paper and, similarly, contexts may be larger than a math-mode environment within the paper. We will provide heuristics for determining what the context for a metavariable is in a longer version of this document.

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 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}, $A^{A_1}_{A_2}$, ${}^{A_1}A$...

5. If $\mathcal{S} \subseteq$ Arrangement does not contain any arrangements consisting of a bare pointer to an object, $\mathcal{S}$ is non-empty, and $|\mathcal{S}| \leq \aleph_0$, then $\mathcal{S} \in$ Object.

   If $\mathcal{S} \subset$ Arrangement is not an equivalence class of $\approx$ or any members of $\mathcal{S}$ are ill-formed, then $\mathcal{S}$ is *ill-formed*. An arrangement is ill-formed iff any of its subcomponents is ill-formed. (Symbols and natural numbers are well formed.)

   (Thus, it is allowed to build an object from ill-formed arrangements, and the resulting object is ill-formed.)

6. There is a special symbol in Object indicating a *hole* $\square$ in which an object is to be placed.

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 math stuff in syntax, with math stuff 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.

We write $[A]_\approx$ for the object that contains all the arrangements equivalent to A by the equivalence relation $\approx$. Only objects of the form $[A]_\approx$ are *well formed*.

2.2 Syntax Shorthand: Arrangement Coercions

From example 2.13, the reader will observe that it's cumbersome to write P_O in so many places when all we're interested in is the identity for objects. We introduce the following convention:

Convention 2.1 (Coercing Objects to Pointers)**.** We allow O to be written instead of P_O in an arrangement. □

Example 2.2. The expression $[\lambda O_1.O_2]_\approx$, stands for $[\lambda P_{O_1}.P_{O_2}]_\approx$. □

We define *meta-level parentheses* to be those parentheses which surround a single object and which may optionally be omitted from some arrangements with a similar form.[7]

It is still cumbersome to write $[\,\cdot\,]_\approx$ in so many places. One of the ways we deal with this is to arrange for this to happen automatically at places where a piece of meta-level syntax requires an arrangement to be regarded as an object.

Convention 2.3 (Coercing Arrangements to Objects)**.** We require that when an arrangement A is written, but the surrounding context only makes sense if the value of the expression is an object, then the arrangement A is implicitly coerced to the object $[A]_\approx$, as though the latter had been written instead. As a special case of this, we require that an arrangement containing meta-level parentheses is to be read as though the parentheses were instead a use of $[\,\cdot\,]_\approx$. □

[7] We largely leave it up to the reader to determine which parentheses are meta-level. If a primitive constructor (Section 2.4) appears inside some arrangements with parentheses surrounding it and other arrangements without them, it usually indicates these parentheses are meta-level. Similarly, parentheses which only surround a single metavariable corresponding to an object are frequently meta-level. Parentheses surrounding syntax which is to be thought of as a sequence are normally not meta-level. To help with this ambiguity, from this point forward all parentheses appearing in arrangements inside this document are meta-level.

Convention 2.4 (Coercing Arrangements to Pointers)**.** We require that when an arrangement A is written, but the surrounding context only makes sense if the value of the expression is a pointer, then the arrangement A is implicitly coerced to the pointer to the object given by convention 2.3. □

Due to the combination of convention 2.3, convention 2.4 and the tight restrictions on where round parentheses can occur in proper arrangements, most uses of round parentheses will not be symbols that are part of syntactic arrangements but instead will be part of the meta-level mathematical reasoning.

Example 2.5. The expression $(O_1\ O_2)\ O_3$, which contains meta-level parentheses, stands for $[O_1\ O_2]_{\approx}\ O_3$. If we write $O = (O_1\ O_2)\ O_3$, then this stands for writing $O = [[O_1\ O_2]_{\approx}\ O_3]_{\approx}$, because the equation's left-hand side must be an object due to the declaration that the metavariable O ranges over Object. □

We have left $\approx$ mostly unspecified so far. The sets Object and Arrangement do not depend on $\approx$, but their subsets of well formed objects and arrangements do depend on $\approx$. The definition of $\approx$ may be adjusted by the authors of a paper at any point, and the set of well formed objects in scope will therefore change at the times these adjustments are made. The effect of convention 2.3 will similarly change; the same expression can denote different objects at different places if there is an intervening change to $\approx$.

2.3 Contexts and Hole Filling

A *context* is an object $O \in \mathsf{Object}$ with at least one use of the special hole object $\square$. The number of hole symbols in an object or arrangement is its *arity*. We now define *context-hole filling* for arbitrary objects and arrangements (although it will in general only do something useful for well formed objects and arrangements with the correct arity). Let the operations $O[O_1, \ldots, O_n]$, $P_O[O_1, \ldots, O_n]$ and $A[O_1, \ldots, O_n]$ which fill the holes reachable from O, P_O and A with the objects in the sequence $\vec{O} = [O_1, \ldots, O_n]$ be defined as follows:

1. $P_O\vec{O} = P_{O'}$ and $O\vec{O} = O'$ iff $\mathsf{fill}(O, \vec{O}) = (O', [])$. Similarly, $A\vec{O} = A'$ iff $\mathsf{fill}(A, \vec{O}) = (A', [])$. The results of $\mathsf{fill}(O, \vec{O})$ and $\mathsf{fill}(A, \vec{O})$ are undefined except where explicitly defined below. (The result is undefined unless all of the replacements are used, so the number of replacements must match the arity.)
2. $\mathsf{fill}(\square, [O] \cdot \vec{O}) = (O, \vec{O})$. (Each hole uses up one of the replacements.)
3. $\mathsf{fill}(\{A\}, \vec{O}) = ([A']_{\approx}, \vec{O'})$ if $\mathsf{fill}(A, \vec{O}) = (A', \vec{O'})$. (Context-hole filling in a well formed context can only descend inside an arrangement that is alone in its equivalence class, otherwise ambiguities may arise over whether free variables in an inserted object become bound. This is part of the motivation for our requirement that $\approx$ must not relate distinct arrangements containing holes.)
4. $\mathsf{fill}(O, \vec{O}) = (O, \vec{O})$ if O is not a context. (This is the only way context-hole filling can skip over embedded objects which are non-singleton equivalence classes of arrangements.)
5. $\mathsf{fill}(s, \vec{O}) = (s, \vec{O})$ and $\mathsf{fill}(n, \vec{O}) = (n, \vec{O})$.
6. Context-hole filling essentially traverses the arrangement tree in a left-to-right order filling in holes in the order it encounters them. Thus, for any arrangements A, A_1, and A_2, core arrangement $\hat{A}$, and object sequences $\vec{O}_1$, $\vec{O}_2$, $\vec{O}_3$, and $\vec{O}_4$, if it holds that

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

then all of these must follow:

$$\begin{aligned}
\mathsf{fill}(A\hat{A}, \vec{O}_1) &= (A'\hat{A}', \vec{O}_3) & \mathsf{fill}(A^{A_1}_{A_2}, \vec{O}_1) &= (A'^{A'_1}_{A'_2}, \vec{O}_4)\\
\mathsf{fill}(A^{A_1}, \vec{O}_1) &= (A'^{A'_1}, \vec{O}_3) & \mathsf{fill}(A_{A_1}, \vec{O}_1) &= (A'_{A'_1}, \vec{O}_3)\\
\mathsf{fill}(\overline{A}, \vec{O}_1) &= (\overline{A'}, \vec{O}_2) & \mathsf{fill}(\underline{A}, \vec{O}_1) &= (\underline{A'}, \vec{O}_2)
\end{aligned}$$

7. $\mathsf{fill}(P_O, \vec{O_1} \cdot \vec{O_2}) = \mathsf{fill}(P_{O_3}, \vec{O_2})$ if $\mathsf{fill}(O, \vec{O_1} \cdot \vec{O_2}) = \mathsf{fill}(O_3, \vec{O_2})$. (Context-hole filling descends object pointers until it encounters a hole)

Example 2.6. Here are some examples of context-hole filling:

$$(\Box\ \Box)[O, O] = O\ O \qquad \underline{\Box}[O] = \underline{O}$$
$$(\Box \to O_1)[O_2 \to O_2] = (O_2 \to O_2) \to O_1 \qquad (\Box \coloneqq \Box, \Box)[O_1, O_2, O_3] = (O_1 \coloneqq O_2, O_3)$$

□

We now will define $(\mathcal{S}_1, \mathcal{S}_2)$-Context to be the contexts which act as functions from $\mathcal{S}_1$ to $\mathcal{S}_2$, i.e., the set of every context O_c of arity 1 such that for all $O \in \mathcal{S}_1$ it holds that $O_\mathsf{c}[O] \in \mathcal{S}_2$. $\mathcal{S}$-Context can be thought of as the set of all the things that can be wrapped around an arbitrary member of $\mathcal{S}$ to give another member of $\mathcal{S}$.

Given a relation $\mathcal{R}$ such that $(\mathsf{domain}(\mathcal{R}) \cup \mathsf{range}(\mathcal{R})) \subseteq \mathcal{S} \subseteq \mathsf{Object}$, let $[\mathcal{R}]^\mathcal{S}$ denote the *$\mathcal{S}$-compatible closure of* $\mathcal{R}$, defined as follows: if $O_\mathsf{c} \in \mathcal{S}$-Context and $O_1 \xrightarrow{\mathcal{R}} O_2$,[8] then $O_\mathsf{c}[O_1] \xrightarrow{[\mathcal{R}]^\mathcal{S}} O_\mathsf{c}[O_2]$. Let $[\mathcal{R}]$ denote $[\mathcal{R}]^\mathcal{S}$ for some set $\mathcal{S}$ which the reader can infer from the context of discussion.

Let c range over *primitive constructors*, non-hole objects whose only immediate subobjects are $\Box$.

Example 2.7. Here are some examples of primitive constructors [4, p 134], [27, p386], [10, pg 360], [20]:

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

□

Every well formed non-hole object O can be decomposed into a primitive constructor and the subobjects to be placed in the primitive constructor's holes. A *primitive constructor decomposition* of O is a pair $(c, \vec{O})$ such that $O = c\vec{O}$. An object will have one primitive constructor decomposition for each of its arrangements. Furthermore, the subobjects in a decomposition can be recursively decomposed similarly. A recursive decomposition of an object into primitive constructors is very similar to the concept of an *abstract syntax tree* of a string in a language defined by a grammar. If any of the equivalence classes in an object are non-singletons, then the object will not have a unique recursive decomposition.

Example 2.8. Some examples of recursive decomposition of an object into primitive constructors can already be seen in example 2.6. Here are some additional examples:

$$\langle(!O)\rangle = \langle\Box\rangle\,[!\Box[O]] \qquad (O_1 + O_2) + O_3 = (\Box + \Box)\,[(\Box{+}\Box)[O_1, O_2], O_3]$$

□

2.4 Syntax Shorthand: Primitive Constuctor Decomposition

Convention 2.3 allows avoiding the need to write $[\,\cdot\,]_\approx$ by implicitly invoking $[\,\cdot\,]_\approx$ at obvious arrangement boundaries and also at most uses of $(\,\cdot\,)$ in arrangements. For example, convention 2.3 allows us to

[8] $O_1 \xrightarrow{\mathcal{R}} O_2$ is an alternative notation for $(O_1, O_2) \in \mathcal{R}$. See appendix A.4 for details.

know that the expression $(O_1 \mathbin{@} O_2) \mathbin{@} O_3$ stands for the object whose primitive constructor decomposition is given by $O' = c_@[c_@[O_1, O_2], O_3]$ where $c_@ = \Box \mathbin{@} \Box$.

But the shorthand notation provided by convention 2.3 is not enough. Additionally, we want to allow inferring uses of $[\,\cdot\,]_\approx$ in other places in the middle of what appear to be arrangements. As a concrete example, we want to allow inferring that the expression $O_1 \mathbin{@} O_2 \mathbin{@} O_3$ stands for the same object as the expression $(O_1 \mathbin{@} O_2) \mathbin{@} O_3$, namely the object O' mentioned in the previous paragraph. We want that the expression $O_1 \mathbin{@} O_2 \mathbin{@} O_3$ must *not* stand for the object whose primitive constructor decomposition is $c''[O_1, O_2, O_3]$ where $c'' = (\Box \mathbin{@} \Box \mathbin{@} \Box)$.

To provide the additional shorthand notation that is needed, we establish mechanisms for (1) declaring primitive constructors and (2) parsing arrangements. We build the parsing mechanism by adapting the notions of operator precedence and declared associativity from parsing of languages to our setting; this will allow splitting what appears to be a single primitive constructor into multiple primitive constructors.

As an auxiliary device, we define splicing of arrangements. Remember that every arrangement is, in effect, a sequence of core arrangements (symbols, objects, numbers, or overlined arrangements), possibly superscripted or subscripted. An arrangement A' can be *spliced* into another arrangement 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 2.9 (Declaring and Parsing Primitive Constructors)**.**

1. Unless prevented by part 2 of this convention, at the first use of a proper arrangement A, if there is a primitive constructor $c = \{A'\}$ and objects $O_1, \ldots, O_n$ such that $\{A\} = c[O_1, \ldots, O_n]$, then this use of A *declares* the primitive constructor c and the arrangement A'. Note that A' differs from A exactly in having $\Box$ in place of every non-$\Box$ object appearing in A.

2. At each place where we coerce an arrangement A into an object O using convention 2.3, the arrangement A is inspected to see if it can be built by splicing together already-declared arrangements. If A can be built entirely by splicing together already-declared arrangements, and then filling the holes in the splicing result with 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. (This choice will typically involve notions of operator precedence and declarations of associativity.) □

Example 2.10. Suppose we have written the expressions $\langle O_1 \rangle$ and $!O_2$. This declares the primitive constructors $\langle O_1 \rangle$ and $!O_2$. If we then write $O = \langle !O' \rangle$, then by convention 2.9 this produces the same result as writing $O = \langle (!O') \rangle$. This happens because the arrangement $\langle !O' \rangle$ can be built by splicing $!\Box$ into $\langle O_1 \rangle$ and then filling the hole with O'.

(If we wanted to avoid the interpretation of convention 2.9, we could do so by avoiding the implicit coercion of convention 2.3 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 $!\Box$.)

Suppose we write the expression $O_1 \mathbin{@} O_2$. This declares the primitive constructor $c_@ = \Box \mathbin{@} \Box$. If we then state that $c_@$ is left-associative, then writing $O = O_1 \mathbin{@} O_2 \mathbin{@} O_3$ produces the same result as writing $O = (O_1 \mathbin{@} O_2) \mathbin{@} O_3$. If we did not give the associativity of $c_@$, then writing $O = O_1 \mathbin{@} O_2 \mathbin{@} O_3$ would be an error, because there are multiple distinct ways the arrangement $\Box \mathbin{@} \Box \mathbin{@} \Box$ can be built by splicing the arrangement $\Box \mathbin{@} \Box$ into itself. □

2.5 Names, Binding, α-Conversion, and Substitution

The relation $\approx$ provides a mechanism for working with syntax considered modulo equivalences on arrangements. One of the most important equivalences is the notion of α-*conversion* which renames *bound names*.[9]

Some of the members of Object can be declared to be *names*. The names may be furthermore subdivided into groups. Formally, the concepts of names and groups of names are given by an equivalence relation $\sim \subset \mathsf{Object} \times \mathsf{Object}$ which relates names in the same group. An object O is a *name* iff $O \sim O$. Declaring a subset $\mathcal{S} \subset O$ to be a *name group* is the same as declaring that $\mathcal{S}$ is a $\sim$-equivalence class. The definition of $\sim$ will be extended incrementally with declarations of groups. Any objects that have not been declared to be related by $\sim$ are *not* related by $\sim$. To keep things simple we require that no name contains another name (of the same group or of a different group) as a subobject.

Specific primitive constructors can be declared to *bind* a name placed in one of the constructor's holes across some of the constructor's holes. We define the *free names* of an object O, written $\mathsf{FN}(O)$:

1. If O is a name, then $\mathsf{FN}(O) = \{O\}$.
2. Otherwise, if $\mathsf{FN}(O)$ is defined, it is as follows.

   First, we must define the free names of primitive constructor decompositions (p.c.d.'s) of O. Suppose $O = c[O_1, \ldots, O_n]$ gives one such p.c.d. Let $\mathcal{S}_i$ be the names bound by c in O_i for $1 \leq i \leq n$. Then $\mathsf{FN}(c, [O_1, \ldots, O_n]) = \bigcup_{i \in \{1,\ldots,n\}} \mathsf{FN}(O_i) \setminus \mathcal{S}_i$.

   If there exists a set $\mathcal{S}$ such that $\mathcal{S} = \mathsf{FN}(c, \vec{O})$ for every p.c.d. $(c, \vec{O})$ of O, then $\mathsf{FN}(O) = \mathcal{S}$.

The free names of an arrangement A are defined by $\mathsf{FN}(A) = \mathsf{FN}(\{A\})$. A name that is not free is *bound*.

Example 2.11. Consider $c_\lambda = \lambda\Box.\Box$ of arity 2. Suppose we declare that c_λ binds any name placed in its first hole in both of its holes. Suppose we declare that $\{\, \mathsf{x}_i \mid i \in \mathbb{N} \,\}$ is a name group. (We will in fact make both of these declarations later, so this example is not just hypothetical.) Suppose that we have not declared any bindings for the constructor $c_@ = \Box \mathbin{@} \Box$. Then $\mathsf{FN}((\lambda \mathsf{x}_1.(\mathsf{x}_1 \mathbin{@} \mathsf{x}_2)) \mathbin{@} \mathsf{x}_3) = \{\mathsf{x}_2, \mathsf{x}_3\}$.

Consider $c_{\mathsf{let}} = (\mathsf{let}\ \Box = \Box\ \mathsf{in}\ \Box)$ of arity 3. Suppose we declare that c_{let} binds any name placed in its 1st hole in its 1st and 3rd hole. Then $\mathsf{FN}(\mathsf{let}\ \mathsf{x}_1 = \mathsf{x}_3\ \mathsf{in}\ (\mathsf{x}_1 \mathbin{@} \mathsf{x}_2)) = \{\mathsf{x}_2, \mathsf{x}_3\}$. □

We now define the auxiliary notion of *name swapping*. Given two names O_x and O_y such that $O_\mathsf{x} \sim O_\mathsf{y}$, let $\mathsf{swap}(O_\mathsf{x}, O_\mathsf{y}, O)$ be the object O' that results from replacing every occurrence of O_x in O by O_y, and *vice versa*. Let $\mathsf{swap}(O_\mathsf{x}, O_\mathsf{y}, A)$ be defined similarly.

We now define α-*conversion*. Let $\equiv_\alpha$ be the smallest equivalence relation satisfying the following condition. 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 $A \equiv_\alpha \mathsf{swap}(O_\mathsf{x}, O_\mathsf{y}, A)$.

Definition 2.12 (α-Conversion as a Syntactic Equivalence)**.** 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 $A_1 \approx A_2$. □

Definition 2.12 implies that $\approx$ will change whenever adjustments are made to the declared bindings of primitive constructors or to the definition of $\sim$.

[9] We do not give an especially sophisticated 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. The notion of equivalence we provide is intended to be used in defining other syntactic equivalences in addition to α-equivalence. For example, suppose we wanted to declare that the primitive constructor $(\Box \mid \Box)$ 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 support the declaration of other equivalences in this fashion.

We now define the *substitution* operation, written as $O[O_{\mathsf{x}} := O']$. This expression will be defined to stand for the result of replacing all free occurrences of O_{x} in O by O'. This operation must be defined carefully. The result of $O[O_{\mathsf{x}} := O']$ must not allow names that are free in O' to be captured by bindings in O. Also, the operation must respect $\approx$ so that if both O and O' are well formed, then $O[O_{\mathsf{x}} := O']$ is also well formed. Given a name O_{x}, define $O[O_{\mathsf{x}} := O']$ formally 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 $O = c[O_1, \ldots, O_n]$, let $\mathcal{S}$ be the subset of $\{O_1, \ldots, O_n\}$ of names bound by this occurrence of c. If $\mathcal{S} \cap \mathsf{FN}(O') \neq \emptyset$, then let $(c, [O_1, \ldots, O_n])[O_{\mathsf{x}} := O']$ be undefined.[10] Otherwise, let $(c, [O_1, \ldots, O_n])[O_{\mathsf{x}} := O'] = c[O_1[O_{\mathsf{x}} := O'], \ldots, O_n[O_{\mathsf{x}} := O']]$.

   If there exists an O'' such that $O'' = (c, \vec{O})[O_{\mathsf{x}} := O']$ for every p.c.d. $(c, \vec{O})$ of O such that $(c, \vec{O})[O_{\mathsf{x}} := O']$ is defined, then $O[O_{\mathsf{x}} := O'] = O''$. Otherwise $O[O_{\mathsf{x}} := O']$ is undefined.[11]

The reader will note that without an infinite supply of names $O[O_{\mathsf{x}} := O']$ may not be defined. This is permissible, though equally, given a grammar with no more than countably many objects, we may extend $\sim$ for any name group appearing in the grammar by adding countably many objects distinct from those in the grammar. Also, since if a variable is declared and no further constraints are placed on it, we assume it to range over some countable set of names disjoint from any other object in the grammar, substitution is defined more often than not.

Example 2.13. Below, on the left are some example syntactic objects [4, p 134], [20], [27, p 386]. These objects may not be well formed, because the singleton sets may not be equivalence classes of $\approx$. The objects to the right of them are adjusted to be well formed (assuming the subobjects O_1, to O_4 are well formed):

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

□

2.6 Production Rules for Defining Syntactic Sets

We have already defined syntactic objects, but the set Object is too big. Carefully defined subsets of Object may be defined via *syntax production rules*, which we write in the form

$$\nu_1, \ldots, \nu_n \in \mathcal{S} ::= \mathcal{A}_1 \mid \cdots \mid \mathcal{A}_m$$

where $\nu_1, \ldots, \nu_n$ are metavariables, $\mathcal{S}$ is the name of the subset of Object being defined, and $\mathcal{A}_1, \ldots, \mathcal{A}_m$ are *alternatives*. Each alternative is either the special notation "$\cdots$" or an expression e,[12] 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

[10] 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.

[11] So the substitution must be defined for at least one of the primitive constructor decompositions to get a defined result.

[12] By expression we mean either an object, an object level variable or something like a primitive constructor which is allowed to have metavaribles in the place of holes. An expression can be thought of as either corresponding to a syntactic object, or else ranging over a set of objects corresponding to an object given by assigning values to the metavariables in the expression.

in e, provided both c holds of that choice of metavariables and the values supplied for metavariables occurring in e match any constraints placed on those metavariables by their production rules. One can omit the "$\in \mathcal{S}$", allowing the reader to fill in $\mathcal{S}$ whose name is distinct from the names of all other declared sets. One can omit the side condition in which case we can read it as if true. One can provide a global side condition if c' which we read as appending $\wedge c'$ to all $\mathcal{A}$. On either side of a production rule authors may choose to include an optional comment.[13]

Such a syntax production rule has the following effects:

1. It declares $\mathcal{S}$ to be a set of syntactic objects, in particular the smallest one that satisfies all other constraints placed on it not just by this rule but also by the rest of the document.
2. It declares the metavariables $\nu_1, \ldots, \nu_n$ to range over the set $\mathcal{S}$.
3. A global side condition if c' appends $\wedge c'$ to each $\mathcal{A}_1, \ldots \mathcal{A}_n$.
4. If each $\mathcal{A}_1, \ldots \mathcal{A}_n$ contains only undecorated instances of ν, then for any $\mathcal{A}$ containing multiple instances of ν and no side conditions containing ν that apply to $\mathcal{A}$, we can rewrite it with each ν given a different decoration. I.e., $m \in M ::= x \mid m\, m$ becomes $m, m_1, m_2 \in M ::= x \mid m_1\, m_2$.
5. For each alternative $\mathcal{A}$ in the rule which is not "$\cdots$", a constraint on the membership of $\mathcal{S}$ is added. The constraint is that for each legal choice[14] of values for the metavariables occurring in $\mathcal{A}$, if O is the result of evaluating the expression e in $\mathcal{A}$ using those metavariable assignments, then $O \in \mathcal{S}$.

   Metavariables occurring in an alternative $\mathcal{A}$ that are not yet declared to range over any set are presumed to range over a countable set of object-level variables disjoint from all the other sets of objects in the paper. This assumption is dropped if a value for a metavariable gets declared later in the paper and values for $\mathcal{A}$ are recalculated accordingly.
6. If the first alternative is not the special alternative "$\cdots$", then any constraints on the membership of $\mathcal{S}$ established by earlier rules are forgotten.
7. The rule triggers a recalculation of *all* of the sets declared by all syntax production rules. Such a recalculation is also triggered whenever the definition of $\approx$ is altered. Or when a definition of what metavariables range over is altered.

   This recalculation evaluates all of the constraint expressions for all syntactic sets using the *current* bindings for all metavariables, set names, the equivalence relation $\approx$, etc., and rebinds the set names to the recalculated values in the subsequent text.[15]

Multiple rules can be given for the same set $\mathcal{S}$. If a later rule for $\mathcal{S}$ begins with the special alternative "$\cdots$", then its alternatives are combined with the alternatives already in force for $\mathcal{S}$. 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 of his/her production rules, then they normally expect these to be combined as though they had used $\cdots$. The special alternative "$\cdots$" used as the final alternative of a rule has no mathematical consequence and is used only as a signal to the reader warning that there will be later rules for the same set.

[13] We won't deal with comments in depth as they have no effect on the resulting grammar. They are usually a short description of the metavariable being defined e.g. (label).

[14] By legal choice we mean a choice of metavariables matching the sets they are declared to range over and fulfilling any constraints added by any side conditions.

[15] It is an error if there is not a unique assignment of smallest values to the declared sets. Normally, the existence of a unique assignment will be provable using a fixed point theorem like the Knaster-Tarski theorem. However, the notation allows putting strange side conditions in the constraint expressions in alternatives, and this can cause a failure.

When a syntax alternative is intended to allow building terms from multiple subterms[16] of the same set, it is necessary to use distinct metavariables for each possible subterm to allow the subterms to differ. It is always possible to find distinct metavariables for the same set by using subscripts.

Example 2.14. We can define the usual *simple types* like this:

$$\begin{aligned} a, b \in \mathsf{Ty\text{-}Variable} &::= \mathsf{a}_i \\ T \in \mathsf{Simple\text{-}Type} &::= a \mid T_1 \to T_2 \end{aligned}$$

Given this definition, a possible example type is $T_0 = a \to (b \to a)$. In this example T_0, we leave unspecified which exact type variables are used. We could make T_0 concrete by specifying $a = \mathsf{a}_0$ 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 Simple-Type as $T \to T$, 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. □

Example 2.15. We can define the *lambda calculus* like this:

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

Each v ranges over a countable set of object-level variables disjoint from the objects produced by the other production rules. The production rule $e \in \mathsf{exp} ::= v$ can be read as giving us the constraint $\mathsf{var} \subseteq \mathsf{exp}$. The constraint $\{ [\lambda P_v.P_e]_\approx \mid \mathsf{ptr}(v) = P_v \wedge \mathsf{ptr}(e) = P_e \} \subseteq \mathsf{exp}$ is given by $e \in \mathsf{exp} ::= \lambda x.e$. The constraint $\{ [P_{e_1}\, P_{e_2}]_\approx \mid \mathsf{ptr}(e_1) = P_{e_1} \wedge \mathsf{ptr}(e_2) = P_{e_2} \} \subseteq \mathsf{exp}$ is given by $e \in \mathsf{exp} ::= e\, e$. We pick the least $\mathsf{exp} \subseteq \mathsf{Object}$ and $\mathsf{var} \subseteq \mathsf{Object}$ satisfying these constraints with an ordering given by the subset relation.

In addition to declaring e as ranging over exp this definition also declares e_1, e_2,..., e', e'' etc. to range over exp and similarly for $v \in \mathsf{var}$. The subset of Object picked out by these constraints depends on the choice of equivalence relation $\approx$, in the lambda calculus this is most likely α equivalence, although it may also be the identity relation on Arrangement

In order to be confident that this set can be picked out (e.g. for exp) we begin with exp^0 equal to the set ranged over by v and let exp^1 contain all the things exp must contain if exp is at least exp^0 and so on for each +1 case. For a limit point ε we let exp^ε be $\bigcup_{i=0}^{\varepsilon} \mathsf{exp}^i$. We take the least fixed point of the function $f \in \mathcal{P}(\mathsf{Object}) \to \mathcal{P}(\mathsf{Object})$ such that $f(\mathsf{exp}^i) = \mathsf{exp}^{i+1}$ over some appropriately large set of exp^i ordered by the subset relation.

We define the rewriting relation as the exp-compatible closure of the smallest β (in the ordering given by $\subseteq$) satisfying the constraint $(\lambda v.e_1)e_2 \xrightarrow{\beta} (e_1[v \coloneqq e_2])$. The notation $O_1[O_2 \coloneqq O_3]$ is defined in Section 2.5. We do the same for $\lambda v.e_1\, v \xrightarrow{\eta} e$. Note that because we have bracketed the term after substitution we are able to reapply equivalences that may have otherwise been lost in the process. □

Example 2.16. Given the definition of simple types in Example 2.14 we can define the *simply typed lambda calculus* as follows:

$$\hat{e} \in \mathsf{texp} ::= v \mid \hat{e}\, \hat{e} \mid \lambda v : T.\, \hat{e}$$

We add rewriting rules:

$$(\lambda v : T_1.\hat{e}_1)\hat{e}_2 \xrightarrow{\beta} (\hat{e}_1[v \coloneqq \hat{e}_2])$$

$$\lambda v : T_1.\hat{e}_1\, v \xrightarrow{\eta} \hat{e}$$

The rewriting relations are the texp-compatible closure of the least β and η satisfying this constraint. □

[16] When we say A is a term and B is a subterm of A what we mean is there is some context C such that filling the hole in C with B gives us A.

Example 2.17. We can extend example 2.14 with records in a similar way to Pierce [19, pg 129]. We can define *lambda calculus with records* like this:

$$\begin{array}{rrlrrll} l \in \mathsf{label} & ::= & \mathsf{y}_i & R \in \mathsf{Type\text{-}Records} & ::= & \epsilon \mid l : T, R & \text{where } l \notin \mathsf{lab}(R) \\ \hat{e} \in \mathsf{texp} & ::= & \cdots \mid \{r\} \mid \hat{e}.l & & & & \\ t \in \mathsf{Record\text{-}Type} & ::= & T \mid \{R\} & r \in \mathsf{Term\text{-}Records} & ::= & \epsilon \mid l = \hat{e}, r & \text{where } l \notin \mathsf{lab}(r) \end{array}$$

Where we define lab s.t. $\mathsf{lab}(\epsilon) = \emptyset$, $\mathsf{lab}(l : T, R) = \{l\} \cup \mathsf{lab}(R)$ and $\mathsf{lab}(l = \hat{e}, r) = \{l\} \cup \mathsf{lab}(r)$. Both r and R are equivalent up to reordering (i.e. $l : T, R \approx R, l : T$ and $l = \hat{e}, r \approx r, l = \hat{e}$). Here, $\approx$ is the smallest equivalence relation fulfilling these constraints. It is defined incrementally over each R and each r as a new one is added.

We add a rewriting rule:

$$\{l = v, r\}.l \xrightarrow{\mathsf{RCD}} v$$

For each $* \in \{ x \in \mathsf{Object} \times \mathsf{Object} \mid x = \beta \} \cup \{ x \in \mathsf{Object} \times \mathsf{Object} \mid x = \eta \} \cup \{\mathsf{RCD}\}$ we add additional constraints:

$$\frac{(\hat{e}_1 \xrightarrow{*} \hat{e}_2)}{(\hat{e}_1.l \xrightarrow{*} \hat{e}_2.l)}$$

$$\frac{(\hat{e}_1 \xrightarrow{*} \hat{e}_2)}{(\{r, l = \hat{e}_1\} \xrightarrow{*} \{r, l = \hat{e}_2\})}$$

(The horizontal line is read as the logical operator $\Rightarrow$). Our rewriting relations are the texp-compatible closure of the least relations, $\xrightarrow{\mathsf{RCD}}$, $\xrightarrow{\beta}$ and $\xrightarrow{\eta}$ satisfying all of these constraints. □

3 Model for Syntactic Math Text

In this section we show that there is a model for SMT. In order to do so, we choose sets to represent Symbol, Pos, $\mathsf{Pointer}$, $\square$, ϵ and $\overline{\mathsf{B}}$. Our invariant constraints are those that will hold of sets thought to approximate Object and $\mathsf{Arrangement}$ in the proof these are well defined. Our constraints on the final selection will only hold of the set we pick out from these approximations.

Definition 3.1 (Symbol, Pos, Pointer, $\{\square, \epsilon, \overline{\mathsf{B}}\}$)**.** We can create a countable set, D, representing symbols, marking/wrapping and positioning from the ordinals[17] following ω which are themselves smaller than 2ω. We pick a finite set of elements, Pos, from D to represent the positions subscript, superscript, pre-subscript, pre-superscript, text above, text below, etc. (at least as many as positions as detailed in the OpenDocument standard[12]). We pick out an element of D which we call $\overline{\mathsf{B}}$. We pick out an element of D to represent the context-hole $\square$, and one to represent the empty arrangement ϵ. We let the remainder of the elements in D represent Symbol (at least as many symbols as in Unicode). □

Definition 3.2 (Invariant Constraints)**.** Our invariant constraints on Object, $\mathsf{Arrangement}$, ptr, $\mathsf{Pointer}$, Symbol, Pos and Core hold if the following hold:
$\mathsf{ptr} \in \mathsf{Object} \to \mathsf{Pointer}$ and ptr is a bijection between Object and $\mathsf{Pointer}$.
$\overline{\mathsf{B}} \notin \mathsf{Object}$ and $\overline{\mathsf{B}} \notin \mathsf{Arrangement}$.
$\square \in \mathsf{Object}$ and $\square \notin \mathsf{Arrangement}$ and $\epsilon \in \mathsf{Arrangement}$
$\mathsf{Pos} \perp \mathsf{Object}$ and $\mathsf{Pos} \perp \mathsf{Arrangement}$.[18]
$\mathsf{Symbol} \subset \mathsf{Core}$.

[17] With the Von Neumann encoding.
[18] We use $X \perp Y$ to mean X and Y disjoint.

$\mathbb{N} \subset \mathsf{Core}$.
$\mathsf{Core} \subset \mathsf{Arrangement}$. □

Definition 3.3 (Constraints on Final Selection)**.** Our constants on the final selection of on Object, Arrangement, Pointer, Symbol, Pos and Core hold if the following hold:
$\mathsf{Pointer} \subset \mathsf{Core}$.
If $A \in \mathsf{Arrangement}$, $A \neq \epsilon$ and $x \in \mathsf{Symbol}$ then $(\overline{\mathsf{B}}, x, A) \in \mathsf{Core}$.
$\mathsf{Arrangement} \times \mathsf{Core} \subset \mathsf{Arrangement}$.
If $A \in \mathsf{Arrangement}$ and $x \in \mathsf{Pos} \to \mathsf{Arrangement} \setminus \{\epsilon\}$ and $x \neq \emptyset$ then
$(A, x) \in \mathsf{Arrangement}$.
If $\mathcal{S} \subset \mathsf{Arrangement}$ and $|\mathcal{S}| \leq \aleph_0$ and $S = \{p\}$ where $p \in \mathsf{Pointer}$, then $\mathcal{S} \in \mathsf{Object}$.[19] □

Theorem 3.4. *There exists some selection of* Object*,* Arrangement *and* ptr *such that our invariant constraints and our constraints on the final selection hold.*

Proof Sketch. We define a sequence of sets thought to contain closer approximations of Object and Arrangement until some member contains a model for Object and Arrangement themselves. The smallest set in our sequence contains all tuples of:

1. The set containing □ (approximating Object).
2. An injective function $p \in \{\square\} \to \mathsf{Pointer}$ (approximating ptr).
3. $\mathsf{Symbol} \cup \mathbb{N} \cup \{\epsilon\}$ (approximating Arrangement).
4. $\mathsf{Symbol} \cup \mathbb{N}$ (approximating Core).

Each subsequent set in our sequence contains those tuples of sets which would be added by applying our constraints as though Object were its approximation, Pointer were its approximation and Arrangement were its approximation. Where our sequence reaches a limit point each set in each tuple is calculated as though Arrangement was the union of arrangements up to that point (apart from the set approximating Arrangement which also gets the pointers to the approximation of Object at that limit).

These sets remain sufficiently small to pick mappings for Object. Further, there is a fixed point for the function mapping each member of this sequence to the member above it. From this fixed point we can select a model for Object and Arrangement.

(Full proof in Appendix B) □

4 How Can This Definition be Used?

Non-MBNF "Grammars" As well as covering some uses of MBNF to define syntax, SMT also provides us with a notion of what it means to use the structures of math-text together with syntactic equivalences, even in documents where MBNF does not feature, or where MBNF is mixed with other notation for picking out objects. Coverage of this sort may require users to select appropriate sets of objects that resolve ambiguities.

[19] We do not bother restricting objects to only include proper arrangements here as it does not particularly affect the logic of the proof. Provided one can pick out unique members from Symbol for left parenthesis, right parenthesis and comma, its not too hard to express what it means for an arrangement to be proper with a logical formula.

A Flexible Notion of Equivalence Not only is the notion of equivalence presented in SMT sufficient to deal with α-equivalence over finite terms, regardless of how binding may be represented in the syntax, it also deals with things like equality up to reordering of finitely many chunks of syntax and equality of finitely many compositions with zero, both of which appear in the π-calculus [3]. It deals with many of the equivalences an author might define using "=," provided they do not quantify over an uncountable set when using it. Furthermore it provides tools to consider equalities over sub-objects, not just the syntactic objects themselves, which can be vital when talking about the structure of a grammar.

Combining Objects in Math-Text SMT deals with most combinations of characters likely to appear in math text used to represent "syntax" in a fairly general manner (it does not deal with matrices/grids, numbers other than the naturals, or an use of sets that cannot be thought of in terms of equivalences up to reordering and repetition on finite lists of elements, but none of these is likely to appear in "syntax").

Automatic Bracketing Since SMT preserves the tree-like structure of syntax, it can be readily used for grammars where authors treat bracketing as optional. We also give authors the option of making this structure more explicit by primitive decomposition. Bracketing structures may often also be derived by noticing where objects appear in production rules.

Functionality Inherited From BNF Our definition extends the basic functions covered by BNF to MBNF and the richer syntactic structures that are represented by math-text. Substitution of non-terminals becomes assigning values to metavariables and choice of production rules remains supported.

Hole Filling The following chunk of the MBNF we took from Chang and Felleisen [4, p 134] defining A can be handled by our definition using convention 2.3:

$$e = \mathsf{x}_i \mid \lambda \mathsf{x}_i.e \mid e\, e$$
$$A = [\,] \mid A[\lambda \mathsf{x}_i.A]\, e$$

5 Related Work

Ott [24] provides a formal language for writing specifications like those written in MBNF. The process of moving from an Ott specification to an MBNF can be performed automatically. However, the focus of this article is on interpreting MBNF without requiring it to be specified in a theorem-prover friendly format. 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. 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 MBNF grammar as part of its definition. We handle more cases of context hole filling. We allow integration with mathematical text. It seems more feasible SMT could be extended to handle co-induction. There are also a variety of systems supporting HOAS, such as Hybrid [2], Twelf [22] and Beluga [18]. While HOAS bears some resemblance to MBNF and is widely used, it does not support all the same uses. For example, Dami [5], uses an MBNF to talk about dynamic binding.

Steele [25] covers many of the notational variants of BNF, including some MBNFs. However, Steele's focus is primarily on surface differences. He does not discuss how the underlying mathematical structure of MBNF differs wildly from BNF.

Grewe et al. [8] 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 checking this paper aims to help with.

Reynolds [21, 1-51] has the best attempt at a definition of MBNF which we could find after looking through the books in our collection, which he calls "abstract syntax"[20]. However, he only deals with context-free grammars and in many places he proceeds by example.

6 Future Work

While we do not deal with trees of infinite breadth or depth here, we hypothesise that the method outlined in this document could be used on trees with countably infinite breadth and depth. The main difference in doing so would be that Object and Arrangement would likely have to be of cardinality $\aleph_2$, rather than $\aleph_1$, but apart from that it seems likely a similar proof would work.

While we provide some powerful tools for writing syntax patterns more explicitly and dealing with numbers in the syntax, we do not provide procedures for generating countably many production rules. Guy Steele [25] has done work in this area, but doesn't address differences between MBNF and BNF.

A formalisation of SMT within a theorem prover would be an eventual avenue for future work, but our current goal is to develop a semi-formal definition that is sufficiently comprehensive to cover a good cross section of the literature. As such, implementation of the current definition in a theorem prover would be premature, as SMT is likely to be updated based on new examples and user feedback.

References

[1] Chelsea Battell & Amy Felty (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, doi:10.1145/2966268.2966271. Available at `http://doi.acm.org/10.1145/2966268.2966271`.

[2] Marco Carbone, Kohei Honda & Nobuko Yoshida (2007): *Structured Communication-Centred Programming for Web Services*. In Rocco De Nicola, editor: *Programming Languages and Systems*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 2–17.

[3] Stephen Chang & Matthias Felleisen (2012): *The Call-by-need Lambda Calculus, Revisited*. In Seidl [23], pp. 128–147.

[4] Laurent Dami (1998): *A lambda-calculus for dynamic binding*. *Theoretical Computer Science* 192(2), pp. 201 – 231, doi:https://doi.org/10.1016/S0304-3975(97)00150-3. Available at `http://www.sciencedirect.com/science/article/pii/S0304397597001503`.

[5] Clovis Eberhart, Tom Hirschowitz & Thomas Seiller (2015): *An Intensionally Fully-abstract Sheaf Model for π^**. In Lawrence S. Moss & Pawel Sobocinski, editors: *6th Conference on Algebra and Coalgebra in Computer Science (CALCO 2015)*, *Leibniz International Proceedings in Informatics (LIPIcs)* 35, Schloss Dagstuhl–Leibniz-Zentrum fuer Infor-

[20] We do not call MBNF "abstract syntax", because some of it is 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, $(\lambda\Box.\Box)$ would just be a name for a particular function taking two arguments of a certain type.

matik, Dagstuhl, Germany, pp. 86–100, doi:10.4230/LIPIcs.CALCO.2015.86. Available at http://drops.dagstuhl.de/opus/volltexte/2015/5528.

[6] Matthew Fluet, editor (2017): *POPL'17 :Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*. ACM, New York, NY, USA.

[7] Kimball Germane & Matthew Might (2017): *A Posteriori Environment Analysis with Pushdown Delta CFA*. In Fluet [1].

[8] Sylvia Grewe, Sebastian Erdweg, André Pacak, Michael Raulf & Mira Mezini (2018): *Exploration of language specifications by compilation to first-order logic*. *Sci. Comput. Program.* 155, pp. 146–172, doi:10.1016/j.scico.2017.08.001. Available at https://doi.org/10.1016/j.scico.2017.08.001.

[9] John E. Hopcroft, Rajeev Motwani & Jeffrey D. Ullman (2006): *Introduction to Automata Theory, Languages, and Computation (3rd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[10] Jun Inoue & Walid Taha (2012): *Reasoning About Multi-stage Programs*. In Seidl [23].

[11] Patrick D F Ion, Nico Poppelier, David Carlisle & Robert R Miner (2001): *Mathematical Markup Language (MathML) Version 2.0*. W3C Recommendation, W3C. Https://www.w3.org/TR/MathML2/chapter3.html.

[12] (2015): *Information technology – Open Document Format for Office Applications (OpenDocument) v1.2 – Part 1: OpenDocument Schema*. Standard, International Organization for Standardization, Geneva, CH.

[13] Fairouz Kamareddine, Joe Wells, Christoph Zengler & Henk Barendregt (2014): *Computerising Mathematical Text*. In Jörg H. Siekmann, editor: *Computational Logic*, *Handbook of the History of Logic* 9, North-Holland, pp. 343 – 396, doi:https://doi.org/10.1016/B978-0-444-51624-4.50008-3. Available at http://www.sciencedirect.com/science/article/pii/B9780444516244500083.

[14] Donald E. Knuth (1986): *The TeXbook*. Addison-Wesley Professional.

[15] Luis Fdo. Llana Díaz & Manuel Núñez (1997): *Testing semantics for unbounded nondeterminism*. In Christian Lengauer, Martin Griebl & Sergei Gorlatch, editors: *Euro-Par'97 Parallel Processing*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 538–545.

[16] Yiannis Moschovakis (1994): *Notes on Set Theory*, 1 edition. *0172-6056* 978-1-4757-4153-7, Springer-Verlag New York, doi:10.1007/978-1-4757-4153-7.

[17] John von Neumann (1923): *Zur Einführung der transfiniten Zahlen*. *Acta Scientiarum Mathematicarum (Szeged)* 1(4), pp. 199–208. Available at http://acta.fyx.hu/acta/showCustomerArticle.action?id=4981&dataObjectType=article. Auf Englisch nachgedruckt in GlossarWiki:Heijenoort:2002.

[18] Brigitte Pientka & Joshua Dunfield (2010): *Beluga: A Framework for Programming and Reasoning with Deductive Systems (System Description)*. In Jürgen Giesl & Reiner Hähnle, editors: *Automated Reasoning*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 15–21.

[19] Benjamin C. Pierce (2002): *Types and Programming Languages*, 1st edition. The MIT Press.

[20] V. Rahli, M. Bickford & R. L. Constable (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, doi:10.1109/LICS.2017.8005074.

[21] John C. Reynolds (2009): *Theories of Programming Languages*, 1st edition. Cambridge University Press, New York, NY, USA.

[22] Carsten Schürmann (2009): *The Twelf Proof Assistant*. In Stefan Berghofer, Tobias Nipkow, Christian Urban & Makarius Wenzel, editors: *Theorem Proving in Higher Order Logics*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 79–83.

[23] Helmut Seidl, editor (2012): *Programming Languages and Systems*. Springer.

[24] Peter Sewell, Francesco Zappa Nardelli, Scott Owens, Gilles Peskine, Thomas Ridge, Susmit Sarkar & Rok Strniša (2007): *Ott: Effective Tool Support for the Working Semanticist*. *SIGPLAN Not.* 42(9), pp. 1–12, doi:10.1145/1291220.1291155. Available at http://doi.acm.org/10.1145/1291220.1291155.

[25] Guy L. Steele, Jr. (2017): *It's Time for a New Old Language*. In: *Proceedings of the 22Nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '17, ACM, New York, NY, USA, pp. 1–1, doi:10.1145/3018743.3018773. Available at `http://doi.acm.org/10.1145/3018743.3018773`.

[26] Alfred Tarski (1955): *A lattice-theoretical fixpoint theorem and its applications*. *Pacific J. Math.* 5(2), pp. 285–309. Available at `https://projecteuclid.org:443/euclid.pjm/1103044538`.

[27] Kazunori Tobisawa (2015): *A Meta Lambda Calculus with Cross-Level Computation*. In: *POPL '15*, pp. 383–393.

[28] Neil Toronto & Jay McCarthy (2012): *Computing in Cantor's Paradise with λ ZFC*. In Tom Schrijvers & Peter Thiemann, editors: *Functional and Logic Programming*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 290–306.

[29] Norbert Wiener (1914): *A Simplification of the Logic of Relations*. *Proceedings of Cambridge Philosophical Society* 17, pp. 387–390. Nachgedruckt in GlossarWiki:Heijenoort:2002.

A Basic Logic and Mathematics

This appendix gives a brief overview of some concepts which are common enough in mathematics, but which are often represented in different ways, to say how they are used in this paper.

A.1 Metavariable Conventions

For this section, ν stands for an arbitrary metavariable (a meta-metavariable). Statements of the form "let ν range over C" declare and define ν as a metavariable that stands for some element of the class C.

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

Whenever we declare a metavariable ν as ranging over a class, this also defines as ranging over that class all variants of ν obtained by either (1) adding a subscript $i \in \mathbb{N}$ to ν to produce ν_i (e.g., ν_0, ν_1, ν_2, etc), (2) adding a single, double, or triple prime to ν, producing respectively in ν', ν'', and ν''', or (3) a combination of (1) and (2).

In contrast, we use superscripts (e.g., ν^1, ν^2) and accents (e.g., $\bar{\nu}$, $\tilde{\nu}$) 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 ν to range over the set $\mathcal{S}$, we might have ν^0 ranging over $\mathcal{S}^0$, ν^1 ranging over $\mathcal{S}^1$, and $\mathcal{S}^1 \subset \mathcal{S}^0 \subset \mathcal{S}$.

A.2 Sets

The mathematical foundation we use is set theory with choice. ZFC 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 $\{ X \in \mathcal{S} \mid P(X) \}$ for some set $\mathcal{S}$ which is left to the reader to infer from the context of discussion. Given some expression $f(X_1, \ldots, X_n)$ mentioning variables $X_1, \ldots, X_n$, 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.'

A.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.[21] Given two sets $\mathcal{S}$ and $\mathcal{T}$, the *product set* $\mathcal{S} \times \mathcal{T}$ is the set of pairs $\{ (X, Y) \mid X \in \mathcal{S} \text{ and } Y \in \mathcal{T} \}$. Let *tuple* notation be defined so that $(X_1, X_2, X_3, \ldots, X_n) = ((X_1, X_2, X_3, \ldots, X_{n-1}), X_n)$.

A.4 Relations

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

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

Let $\mathcal{R}^*$ be the reflexive and transitive closure of $\mathcal{R}$ and let $\mathcal{R}^=$ be the reflexive, symmetric, and transitive closure of $\mathcal{R}$; in both cases we use the above-mentioned convention that the reader must infer the set $\mathcal{S}$ w.r.t. which to take the reflexive closure. Let $X \xrightarrow{\mathcal{R}\!\!\!\!\rightarrow} Y$ mean $X \xrightarrow{\mathcal{R}^*} Y$, and let $X \xleftrightarrow{\mathcal{R}\!\!\!\!\rightarrow} Y$ mean $X \xrightarrow{\mathcal{R}^=} Y$.

A relation is an *equivalence* iff it is symmetric and transitive. Given an equivalence relation $\mathcal{R}$, let $[X]_{\mathcal{R}} = \{ Y \mid (X, Y) \in \mathcal{R} \}$ be the *equivalence class of X w.r.t.* $\mathcal{R}$ and let $[X]_{\mathcal{R}}$ be an equivalence class of $\mathcal{R}$.

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

A relation is *a partial order* on $\mathcal{S}$ iff it is transitive and antisymmetric. A partial order is *strict* iff it is irreflexive. A non-strict partial order, $\leq$, is a *total order* on $\mathcal{S}$ iff for all $X, Y \in \mathcal{S}$ either $X \leq Y$ or $Y \leq X$. A strict partial order, $<$, is a *strict total order* on $\mathcal{S}$ iff for all $X, Y \in \mathcal{S}$ s.t. $X \neq Y$ either $X < Y$ or $Y < X$.

A.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 $\mathcal{S} \to \mathcal{T} = \{ f \mid f \subseteq \mathcal{S} \times \mathcal{T} \text{ and } f \text{ is a function} \}$. Let f be *from* $\mathcal{S}$ *to* $\mathcal{T}$ iff $f \in \mathcal{S} \to \mathcal{T}$. A function f is *injective* iff 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 $\mathcal{S}$ iff $f(X)$ is defined for all $X \in \mathcal{S}$. Given a function f, let $f[X \mapsto Y] = (f \setminus \{ Z \in f \mid \mathsf{fst}(Z) = X \}) \cup \{(X, Y)\}$.

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 $f(X) \leq f(Y)$ iff $X \leq Y$.

A.6 Sequences

Given a set $\mathcal{S}$ which is not a relation (if $\mathcal{S}$ contains only pairs then instead the notation refers to the definition of $\mathcal{R}^*$ from section A.4, the reflexive and transitive closure of $\mathcal{R}$), let $\mathcal{S}^*$, the set of *finite*

[21] We therefore can not use Kuratowski's encoding of pairs 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 $0 = \varnothing$ and $i + 1 = i \cup \{i\}$ because $(0, 0) = \{0, \{0, 0\}\} = \{\varnothing, \{\varnothing, \varnothing\}\} = \{\varnothing, \{\varnothing\}\} = \{\varnothing\} \cup \{\{\varnothing\}\} = 1 \cup \{1\} = 2$. We can use Wiener's encoding of pairs where $(X, Y) = \{\{\{X\}, \varnothing\}, \{\{Y\}\}\}$, 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. We can also work in a set theory with a primitive pairing operator.

sequences of elements in $\mathcal{S}$, be the set of all finite functions f such that $\mathsf{range}(f) \subseteq \mathcal{S}$, and $\mathsf{domain}(f) \subseteq \mathbb{N}$, and $m < n \in \mathsf{domain}(f)$ implies $m \in \mathsf{domain}(f)$.

Convention A.1 (Metavariables over Sequences)**.** If ν is declared to range over $\mathcal{S}$, then $\vec{\nu}$ is automatically declared to range over $\mathcal{S}^*$. □

The notation $[\nu_0, \ldots, \nu_n]$ stands for the least-defined function $\vec{\nu}$ such that $\vec{\nu}(i) = \nu_i$ for all $i \in \{0, \ldots, n\}$. For example, the singleton sequence $[\nu]$ containing ν as its only element is $\{(0, \nu)\}$, and we have $[\nu_0, \nu_1, \nu_2] = \{(0, \nu_0), (1, \nu_1), (2, \nu_2)\}$. The *component of a sequence* $\vec{\nu}$ *at index* i is simply $\vec{\nu}(i)$. Note that the first component of a sequence is at index 0, and that the *empty sequence* [] is merely the empty set. The *length* of a sequence $\vec{\nu}$ is the smallest $n \in \mathbb{N}$ which is larger than all elements of $\mathsf{domain}(\vec{\nu})$. The *concatenation* of sequences $\vec{\nu}_1$ and $\vec{\nu}_2$ is $\vec{\nu}_1 \cdot \vec{\nu}_2 = \vec{\nu}_1 \cup \{ (|\vec{\nu}_1| + i, \nu) \,|\, (i, \nu) \in \vec{\nu}_2 \}$.

Note that $(\mathcal{S}^*, \cdot, [])$ forms a monoid, i.e., the following equalities hold:

$$[] \cdot \vec{\nu} = \vec{\nu} \qquad \vec{\nu} \cdot [] = \vec{\nu} \qquad (\vec{\nu}_1 \cdot \vec{\nu}_2) \cdot \vec{\nu}_3 = \vec{\nu}_1 \cdot (\vec{\nu}_2 \cdot \vec{\nu}_3)$$

B Proof of Key Results

Section 2 is sufficient for any reader who wants an outline of our definition in order to interpret those pieces of MBNF it defines. This appendix will be of interest either to those readers who are looking to extend our definition to define more uses of MBNF, or to those readers who want to reassure themselves that the sort of entities described in Section 2.1 can always be thought to exist. While it is our intention that our definition should be easy to work with, a deeper working knowledge of set theory is assumed for this appendix than the rest of this document and everything apart from Section 3 may be read without this section.

For this appendix we use Wiener's [29] encoding of pairs and von Neumann's encoding of ordinals, the natural numbers [17] and cardinal assignment [16]. We also make use of the axiom of choice.

Lemma B.1. *Given a countable set A and a set B of all trees C such that the interior nodes of C are elements of A and the leaf nodes of C are elements of* ω_1, $|B| = \aleph_1$.

Proof. Let D be the set of all trees C such that every element of C is an element of ω_1. Since we can make a bijection between members of A to a members of ω and the function $f \in \omega_1 \to \omega_1$ s.t. $f(x) = \omega + x$ is a bijection, $|C| = |D|$. For a finite subset, $\mathcal{S}$, of ω_1 the relation $<$ on $\mathcal{S}$ is a finite subset of $\omega_1 \times \omega_1$. Assuming choice, $|\omega_1 \times \omega_1| = \aleph_1$. The cardinality of the set of finite subsets of ω_1 is $\aleph_1$. So $|C| = |D| = \aleph_1$. □

We can now go on to show that Object and Arrangement are well defined. We do so by producing a model within set theory that fulfils most of the constraints in Section 2.1., which are written out formally in Appendix D.[22] This proof requires that the reader pick some appropriate values for Symbol, Pos, Pointer, $\square$, ϵ and $\overline{\mathsf{B}}$. The definition of these sets in Appendix D is adequate.

We for a given ordinal i we define a set of tuples OPAC_i which may be thought of as getting closer to the tuple (Object, ptr, Arrangement, Core).

Definition B.2 (OPAC)**.**
0 Case:

Let $\mathsf{Obj}_0 = \{\square\}$

[22] In order to simplify the proof this model allows the use of arrangements consisting of a single pointer in the formation of objects. It is not too difficult to rule this case out.

Let $\mathsf{ptrSpace}_0 = \{\, x \in \mathsf{Obj}_0 \to \mathsf{Pointer} \,|\, x \text{ is total on } \mathsf{Obj}_0 \wedge x \text{ is injective} \,\}$.
Let

$$\mathsf{OPAC}_0 = \{(\mathsf{Obj}_0, \mathsf{ptr}_0, \mathsf{Arr}_0, \mathsf{Core}_0) \quad | \quad \mathsf{ptr}_0 \in \mathsf{ptrSpace}_0 \wedge \mathsf{Core}_0 = \mathsf{Arr}_0 \setminus \{\epsilon\} \\ \wedge \mathsf{Arr}_0 = \mathbb{N} \cup \{\epsilon\} \cup \mathsf{Symbol} \cup \mathsf{ptr}_0(x)\}$$

+1 Case:
For $(\mathsf{Obj}_n^k, \mathsf{ptr}_n^k, \mathsf{Arr}_n^k, \mathsf{Core}_n^k) \in \mathsf{OPAC}_n$
Let $\mathsf{Accent}_{n+1}^k = (\{\overline{\mathsf{B}}\} \times \mathsf{Symbol}) \times (\mathsf{Arr}_n^k \setminus \{\epsilon\})$.
Let $\mathsf{Core}_{n+1}^k = \mathsf{Core}_n^k \cup \mathsf{Accent}_{n+1}^k$.
Let $\mathsf{Layout}_{n+1}^k = \mathsf{Arr}_n^k \times \{\, x \in \mathsf{Pos} \to \mathsf{Arr}_n^k \setminus \{\epsilon\} \,|\, x \neq \emptyset \,\}$.
Let $\mathsf{Seq}_{n+1}^k = \mathsf{Arr}_n^k \times \mathsf{CoreArr}_{n+1}^k$.
Let $\mathsf{Obj}_{n+1}^k = \mathsf{Obj}_n^k \cup \{\, x \in \mathcal{P}(\mathsf{Arr}_n^k) \,|\, |x| \leq \aleph_0 \wedge (y \in \mathsf{Pointer} \Rightarrow x \neq \{y\}) \,\}$
Let

$$\mathsf{ptrSpace}_{n+1} = \{x \in \mathsf{Obj}_{n+1}^k \to \mathsf{Pointer} \quad | \quad (\mathsf{Obj}_n, \mathsf{ptr}_n, \mathsf{Arr}_n, \mathsf{Core}_n) \in \mathsf{OPAC}_n \wedge \\ x \text{ is total on } \mathsf{Obj}_{n+1}^k \wedge x \text{ is injective}\}$$

Let $\mathsf{ptr}_{n+1}^{k,i}(x) \in \mathsf{ptrSpace}_{n+1}$ such that $\mathsf{ptr}_{n+1}^{k,i}(x) \subseteq \mathsf{ptr}_{n+1}^k$ if such a set exists. If no such set exists let $\mathsf{ptr}_{n+1}^{k,0} = \emptyset$
Let $\mathsf{Arr}_{n+1}^{k,i} = \mathsf{Arr}_n^k \cup \mathsf{CoreArr}_{n+1}^k \cup \mathsf{Layout}_{n+1}^k \cup \mathsf{Seq}_{n+1}^k \cup \{\, \mathsf{ptr}_{n+1}^{k,i}(x) \,|\, x \in \mathsf{Obj}_{n+1} \,\}$ if $\mathsf{ptr}_{n+1}^{k,i}(x)$ is defined and $\emptyset$ otherwise.
Let

$$\mathsf{OPAC}_{n+1} = \{(\mathsf{Obj}_{n+1}^k, \mathsf{ptr}_{n+1}^{k,i}, \mathsf{Arr}_{n+1}^{k,i}, \mathsf{Core}_{n+1}^k) \quad | \quad (\mathsf{Obj}_n, \mathsf{ptr}_n, \mathsf{Arr}_n, \mathsf{Core}_n) \in \mathsf{OPAC}_n \\ \wedge \mathsf{Arr}_{n+1}^{k,i} \neq \emptyset\}$$

Limit Case:
We now define the above functions for a limit point ε.
Let

$$\mathsf{stack} = \{\mathcal{S} \subseteq \bigcup_{i<\varepsilon} \mathsf{OPAC}_i \quad | \quad ((\mathsf{Obj}_i^k, \mathsf{ptr}_i^k, \mathsf{Arr}_i^k, \mathsf{Core}_i^k) \in \mathcal{S} \wedge (\mathsf{Obj}_j^l, \mathsf{ptr}_j^l, \mathsf{Arr}_j^l, \mathsf{Core}_j^l) \in \mathcal{S}) \\ \Rightarrow \quad ((j < i \Rightarrow (\mathsf{Obj}_j^l \subseteq \mathsf{Obj}_i^k \wedge \mathsf{Arr}_j^l \subseteq \mathsf{Arr}_i^k \wedge \mathsf{ptr}_j^l \subseteq \mathsf{ptr}_i^k)) \\ \wedge \quad (j < i \vee i < j \\ \vee \quad (\mathsf{Obj}_i^k, \mathsf{ptr}_i^k, \mathsf{Arr}_i^k, \mathsf{Core}_i^k) = (\mathsf{Obj}_j^l, \mathsf{ptr}_j^l, \mathsf{Arr}_j^l, \mathsf{Core}_j^l)) \\ \wedge \quad \forall n < \varepsilon, (\mathsf{Obj}_n^m, \mathsf{ptr}_n^m, \mathsf{Arr}_n^m, \mathsf{Core}_n^m) \in \mathcal{S})\}$$

For $\mathcal{S} \in \mathsf{stack}$
Let

$$Obj_\varepsilon^{\mathcal{S}} = (\bigcup\{\, \mathsf{Obj}_i \,|\, (\mathsf{Obj}_i^k, x, y, z) \in \mathcal{S} \,\}) \cup \\ \{\, x \in \mathcal{P}(\bigcup\{\, \mathsf{Arr}_i^k \,|\, (a, b, \mathsf{Arr}_i^k, c) \in \mathcal{S} \,\}) \,|\, |x| \leq \aleph_0 \wedge (y \in \mathsf{Pointer} \Rightarrow x \neq \{y\}) \,\}$$

Let $\mathsf{ptr}_\varepsilon^{\mathcal{S},k}$ be a bijection between $\mathcal{S}_\varepsilon \subseteq \mathsf{Pointer}$ and Obj_ε such that for all $(\mathsf{Obj}_i, \mathsf{ptr}_i, \mathsf{Arr}_i, \mathsf{Core}_i) \in \mathcal{S}$, $\mathsf{ptr}_i \subseteq \mathsf{ptr}_\varepsilon^{\mathcal{S},k}$. If no such bijection exists, let $\mathsf{ptr}_\varepsilon^{\mathcal{S},0} = \emptyset$.
Let $\mathsf{Arr}_\varepsilon^{\mathcal{S},k} = \{\, \mathsf{ptr}_\varepsilon^{\mathcal{S},k}(x) \,|\, x \in \mathsf{Obj}_\varepsilon^{\mathcal{S}} \,\} \cup \bigcup\{\, \mathsf{Arr}_i \,|\, (a, b, \mathsf{Arr}_i^k, c) \in \mathcal{S} \,\}$ if $\mathsf{ptr}_\varepsilon^{\mathcal{S},k}(x)$ is defined and $\emptyset$ otherwise.
Let $\mathsf{Core}_\varepsilon^{\mathcal{S}} = \bigcup\{\, \mathsf{CoreArr}_i \,|\, (x, y, z, \mathsf{Core}_i^k) \in \mathcal{S} \,\}$.
Let

$$\mathsf{OPAC}_\varepsilon = \left\{ (\mathsf{Obj}_\varepsilon^{\mathcal{S}}, \mathsf{ptr}_\varepsilon^{\mathcal{S},k}, \mathsf{Arr}_\varepsilon^{\mathcal{S},k}, \mathsf{Core}_\varepsilon^{\mathcal{S}}) \,\middle|\, \mathcal{S} \in \mathsf{stack} \wedge \mathsf{Arr}_{n+1}^{k,i} \neq \emptyset \right\}$$

□

Lemma B.3. OPAC_i *is Non-Empty for all Ordinals* i.

Proof. The only way OPAC_i may be empty for some i is if $|\mathsf{Obj}_i^k| > \mathsf{Pointer}$ for some Obj_i^k such that $(\mathsf{Obj}_i^k, a, b, c) \in \mathsf{OPAC}_i$. We prove by induction on the size of Obj_n^k and Arr_n^k such that $(\mathsf{Obj}_n^k, x, \mathsf{Arr}_n^k, y) \in \mathsf{OPAC}_n$ that this cannot be the case.

0 Case:

$|\mathsf{Obj}_0| = 1 \leq \aleph_1$ and for all $\mathsf{Arr}_0 \in \mathsf{ArrSpace}_0$, $|\mathsf{Arr}_0| = \aleph_0 \leq \aleph_1$.

+1 Case:

If, for all $(\mathsf{Obj}_n^k, x, \mathsf{Arr}_n^k, y) \in \mathsf{OPAC}_n$, $|\mathsf{Obj}_n^k| \leq \aleph_1$ and $|\mathsf{Arr}_n^k| \leq \aleph_1$, then, for all Obj_{n+1}^k, $|\mathsf{Obj}_{n+1}^k| \leq \aleph_1$, provided we have some way of ordering Obj_i^k and Arr_i^k. With choice this follows quite easily from the fact that the cardinality of the set of subsets of $\aleph_1$ which are of cardinality less than or equal to $\aleph_0$ is $(2^{\aleph_0})^{\aleph_0} = 2^{\aleph_0 \cdot \aleph_0} = 2^{\aleph_0}$. As, for all Obj_{n+1}^k there exists some Obj_n^j s.t. $\mathsf{Obj}_{n+1}^k \subseteq \mathsf{Obj}_n^j$, there exists some ptr which assigns pointers for Obj_{n+1}^k and which may also be used to assign pointers for Obj_n^j.

It is easy to observe that, if $|\mathsf{Obj}_n^k| \leq \aleph_1$ and $|\mathsf{Arr}_n^k| \leq \aleph_1$, then $|\mathsf{Arr}_{n+1}^{k,i}| \leq \aleph_1$ since neither Accent_{n+1}^k, nor Layout_{n+1}^k nor Seq_{n+1}^k can add cardinality greater than $\aleph_1$.

Limit Case:

We show $\exists \varepsilon; \forall i < \varepsilon; (|\mathsf{Obj}_i| \leq \aleph_1 \wedge |\mathsf{Arr}_i| \leq \aleph_1) \Rightarrow (|\mathsf{Obj}_\varepsilon| \leq \aleph_1 \wedge |\mathsf{Arr}_\varepsilon| \leq \aleph_1)$. We note that no Arr_i has $\aleph_0$ sub-arrangements and all such Arr_i can be readily identified with some finite tree whose interior nodes are labelled corresponding the operations marking/wrapping, concatenation and the finite number of possible combinations of Subscript, superscript etc. and whose leaf nodes are labelled with members of the set ω_1. So, by lemma B.1, $|\bigcup_{i=0}^{\varepsilon} \mathsf{Arr}_i| \leq \aleph_1$. Similarly we may readily identify each set in Obj_i apart from the $\square$ with some countable subset of the set of trees we used to define each Arr_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 $|\bigcup_{i=0}^{\varepsilon} \mathsf{Obj}_i| \leq \aleph_1$. The desired result follows easily. As $\mathsf{Obj}_\varepsilon \subseteq \mathsf{Obj}_i$, for each $i \leq \varepsilon$ there exists some ptr which assigns pointers for Obj_ε and which may also be used to assign pointers for Obj_i. □

Definition B.4 (fun)**.** Let $Z = \{ \mathsf{OPAc}_i \mid i < \kappa \}$ for some $\kappa < \omega_2$. We define a function $\mathsf{fun} \in Z \to Z$ such that $\mathsf{fun}(\mathsf{OPAC}_i) = \mathsf{OPAC}_{i+1}$ □

Lemma B.5. fun *has a least fixed point.*

Proof. The Knaster–Tarski theorem [26] tells us that any any order-preserving function on a complete lattice has a least fixed point. For $\mathsf{OPAC}_a, \mathsf{OPAC}_b \in Z$ we define $\leq$ such that $\mathsf{OPAC}_a \leq \mathsf{OPAC}_b$ iff (either, for all $(\mathsf{Obj}_b, p, q, r) \in \mathsf{OPAC}_b$, there exists $(\mathsf{Obj}_a, b, c, d) \in \mathsf{OPAC}_a$ s.t. $\mathsf{Obj}_a \subset \mathsf{Obj}_b$, or, for all $(\mathsf{Obj}_b, p, \mathsf{Arr}_b, r) \in \mathsf{OPAC}_b$, there exists $(\mathsf{Obj}_a, b, \mathsf{Arr}_a, d) \in \mathsf{OPAC}_a$ s.t. ($\mathsf{Obj}_a = \mathsf{Obj}_b$ and $\mathsf{Arr}_a \subset \mathsf{Arr}_b$)). Z is a complete lattice ordered by $\leq$ and fun is an order preserving function on Z.[23] □

Theorem B.6. Object *and* $\mathsf{Arrangement}$ *are well defined.*

Proof. For some tuple a in OPAC_i there exists some tuple b in OPAC_i such that:

1. The first member of b contains all the objects our rules say must exist if $\mathsf{Arrangement}$ is at least the third member of a,

[23] Note that the way in which we have defined Obj and Arr is such that $j \leq i$ implies ($\mathsf{Obj}_j \subseteq \mathsf{Obj}_i$ and $\mathsf{Obj}_j \subseteq \mathsf{Obj}_i$). Note also that, for all limit ordinals $\varepsilon \leq \kappa$; $(\mathsf{Obj}_\varepsilon, \mathsf{Arr}_\varepsilon) \in Z$. Finally note that ω_2 is large enough that it has a larger cardinality than any Obj_i or Arr_i, so we can select some κ larger than the partition of Obj_i and Arr_i into the extra elements added at each stage.

2. The third member of b contains all the arrangements that our rules say must exist if Arrangement is at least the third member of a, Object is at least the first member of a and ptr is at least the second member of a.

If OPAC_i is the least fixed point of fun then $\mathsf{OPAC}_{i+1} = \mathsf{OPAC}_i$.

We now take the least fixed point, lfp(fun), of $\mathsf{fun} \in Z \to Z$ and select some tuple $(\mathsf{Obj}, \mathsf{ptr}, \mathsf{Arr}, \mathsf{Core}) \in \mathsf{lfp}(\mathsf{fun})$. The first member of the tuple gives us a model for Object and the third member, Arrangement. □

C Examples of our Definition in Action

C.1 Call by Need

The following example is derived from Chang and Felleisen [4, p 134]:

$$
\begin{array}{ll}
e \in se ::= x \mid \lambda x.e \mid e\,e & \hat{A} \in s\hat{A} ::= \Box \mid A[\hat{A}]\,e \\
v \in sv ::= \lambda x.e & \check{A} \in s\check{A} ::= \Box \mid A[\lambda x.\check{A}]\,e \\
a \in sa ::= A[v] & E \in sE ::= \Box \mid E\,e \mid A[E] \mid \hat{A}[A[\lambda x.\check{A}[E[x]]]E] \\
A \in sA ::= \Box \mid A[\lambda x.A]\,e & \qquad\qquad \text{where} \hat{A}[\check{A}] \in A
\end{array}
$$

Each constraint is added sequentially and the least set of objects satisfying them is recalculated. Where the value of a set a metavariable can range over is recalculated and it is referenced in another rule, the set that rule applies to is recalculated with a new value. For example, initially $a \in sa = \emptyset$, but when $A ::= \Box$ is read it triggers a recalculation of $A[v]$ so $sa = sv$. Then when $A ::= A[\lambda x.A]\,e$ is read, first it triggers a recalculation of A so $sA = \{\Box\} \cup \{\, [P_{\lambda A}\, P_e]_\approx \mid \mathsf{ptr}(e) = P_e \wedge \mathsf{ptr}([\lambda x.\Box]_\approx) = P_{\lambda A} \,\}$ then it triggers a recalculation of a so $sa = v \cup \{\, [P_{\lambda A[v]}\, P_e]_\approx \mid \mathsf{ptr}(e) = P_e \wedge \mathsf{ptr}([\lambda x.P_v]_\approx) = P_{\lambda A[v]} \wedge \mathsf{ptr}(v) = P_v \,\}$ then a won't trigger recalculations, but we have a recalculation on A waiting. Let $f(se, sv, sa, sA, s\hat{A}, s\check{A}, sE)$ take $(se, sv, sa, sA, s\hat{A}, s\check{A}, sE)$ when a recalculation is triggered to their values after a recalculation is performed. Let $<$ be an relation on $(se, sv, sa, sA, s\hat{A}, s\check{A}, sE)$ such that $(se^1, sv^1, sa^1, sA^1, s\hat{A}^1, s\check{A}^1, sE^1) < (se^2, sv^2, sa^2, sA^2, s\hat{A}^2, s\check{A}^2, sE^2)$ iff $se^1 \subset se^2$ or $sv^1 \subset sv^2$ or $sa^1 \subset sa^2$ or $sA^1 \subset sA^2$ or $s\hat{A}^1 \subset s\hat{A}^2$ or $s\check{A}^1 \subset s\check{A}^2$ or $sE^1 \subset sE^2$. We observe that each set out of $(se, sv, sa, sA, s\hat{A}, s\check{A}, sE)$ either gets new elements added to it or remains the same every time a recalculation is triggered and is bounded above by Object. We can therefore take the least fixed point on f satisfying all of these constraints.

If the side condition on E were to re-trigger the calculation on A, $\hat{A}$ or $\check{A}$ we would have to be able to check that the new side condition produced by this recalculation could not effect any E previously added. This grammar relies on an assignation of values to x, otherwise all of its sets are either $\emptyset$ or $\Box$. For a given equivalence (e.g. $\equiv_\alpha$ or $\equiv_A$) this grammar may define a different collection of $(se, sv, sa, sA, s\hat{A}, s\check{A}, sE)$.

We add the reduction rule for this Grammar which is the least $\mathcal{R}$ satisfying:

$$
\hat{A}[A_1[\lambda x.\check{A}[E[x]]]A_2[v]] \xrightarrow{\mathcal{R}} (\hat{A}[A_1[A_2[(\check{A}[E[x]])[x := v]]]]) \qquad \text{where } \hat{A}[\check{A}] \in A
$$

C.2 Lambda Calculus (Missing Steps)

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

When v is read, we take it to range over some 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 $f : \mathsf{Object} \to \mathsf{Object}$ such that $\forall \mathsf{exp}^{i}$, $f(\mathsf{exp}^{i}) = \mathsf{exp}^{i+1}$ (which exists by the Knaster-Tarski theorem). Instead, if k is the first k such that $\mathsf{exp}^{k} = \mathsf{exp}^{k} \cup \{\, \{\lambda P_v.P_e\} \,|\, v \in V \wedge e \in \mathsf{exp}^{k} \,\}$, 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 $\mathsf{exp}^{n+1} = \mathsf{exp}^{n} \cup \{\, \{\lambda P_v.P_e\} \,|\, v \in V \wedge e \in \mathsf{exp}^{n} \,\}$, otherwise $\mathsf{exp}^{n+1} = \mathsf{exp}^{n} \cup \{\, \{\lambda P_v.P_e\} \,|\, v \in V \wedge e \in \mathsf{exp}^{n} \,\} \cup \{\, \{P_e\, P_e\} \,|\, e \in \mathsf{exp}^{n} \,\}$. Since there are no further constraints, the least fixed point of f gives us the desired result.

This induction may be carried out with terms identified up to α-equivalence. So exp^{0} would remain the same. For the $n + 1$ case (for $n < k$) we have:

$$\mathsf{exp}^{n+1} = \mathsf{exp}^{n} \cup \{\, x \,|\, (a \in x \wedge b \in x) \Leftrightarrow (a, b \in \{\, \lambda P_v.P_e \,|\, v \in V \wedge e \in \mathsf{exp}^{n} \,\} \wedge a \equiv_{\alpha} b) \,\}$$

For $n > k$ we have:

$$\mathsf{exp}^{n+1} = \mathsf{exp}^{n} \cup \{\, x \,|\, (a \in x \wedge b \in x) \Leftrightarrow (a, b \in \{\, \lambda P_v.P_e \,|\, v \in V \wedge e \in \mathsf{exp}^{n} \,\} \wedge a \equiv_{\alpha} b) \,\} \cup \{\, \{P_e\, P_e\} \,|\, e \in \mathsf{exp}^{n} \,\}$$

For a limit point ε we have:

$$\mathsf{exp}^{\varepsilon} = \left\{ x \;\middle|\; (a \in x \wedge b \in x) \Leftrightarrow (a, b \in \bigcup_{i<\varepsilon} \mathsf{exp}^{i} \wedge a \equiv_{\alpha} b) \right\}$$