SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Constructing Linear Types in Isabelle/HOL

Felix Krayer

SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Constructing Linear Types in Isabelle/HOL

Konstruktion linearer Typen in Isabelle/HOL

Author:	Felix Krayer
Examiner:	Florian Bruse
Supervisor:	Dmitriy Traytel, Tobias Nipkow
Submission Date:	13-11-2025

I confirm that this master's thesis is my own work and I have documented all sources and material used.

Munich, 13-11-2025 Felix Krayer

Abstract

In this thesis, we address the problem of constructing linear Map-Restricted Bounded Natural Functors (MRBNFs) in the Isabelle/HOL proof assistant. We formalize the process of linearization as taking a linear subtype from a non-linear MRBNF and show that the resulting type is also a MRBNF. Furthermore, we automate this process in a new **linearize_mrbnf** command, which generates the linearized type, derives the typical MRBNF constants, and proves the necessary properties.

We demonstrate the utility of our approach with an example based on the POPLmark Challenge, showing how a manual linearization can be streamlined with our new command.

Our work extends Isabelle/HOL to simplify the generation of complex, non-repetitive datatypes, supporting further reasoning about syntaxes with bindings.

Contents

1 Introduction

Isabelle/HOL provides a fleshed-out system for defining datatypes and codatatypes built on the theory of Bounded Natural Functors (BNFs) developed by Traytel et al. [1]. The **datatype** command developed by Blanchette et al. [2], based on this theory, allows Isabelle users to construct possibly recursive datatypes from primitive, built-in datatypes like the list, product, or function type. The resulting datatype is equipped with several predicates that allow for reasoning about the datatype, for example, through case distinction and structural induction.

However, these regular datatypes are not well-suited to express and reason about syntaxes with bindings, like, for example, lambda calculus terms. As an example, we consider lambda terms with parallel let bindings, where a term is either a variable, an application, a lambda abstraction, or a parallel let binding. Concretely, we want to represent the following syntax, where `x` are variable names.

$$\texttt{t} := \texttt{x} \mid \texttt{t}_1\ \texttt{t}_2 \mid (\lambda x.\ t) \mid \texttt{let } \texttt{x}_1 = \texttt{t}_1 \texttt{ and } \dots \texttt{ and } \texttt{x}_n = \texttt{t}_n \texttt{ in } \texttt{t}_{n+1}$$

We can express the structure of these terms with a regular Isabelle datatype as follows, where α is the type of variable names:

$$\begin{aligned}&\textbf{datatype}\ \alpha\ \mathsf{ltrm} = \mathsf{Var}\ \alpha \mid \mathsf{App}\ \text{"}\alpha\ \mathsf{ltrm}\text{"}\ \text{"}\alpha\ \mathsf{ltrm}\text{"} \mid \mathsf{Abs}\ \alpha\ \text{"}\alpha\ \mathsf{ltrm}\text{"}\\ &\quad \mid \mathsf{Let}\ \text{"}(\alpha \times \alpha\ \mathsf{ltrm})\ \mathsf{list}\text{"}\ \text{"}\alpha\ \mathsf{ltrm}\text{"}\end{aligned}$$

However, this representation is not ideal. For example, it does not consider α-equivalence of lambda terms, i.e., the notion that $(\lambda x.\ x\ z)$ and $(\lambda y.\ y\ z)$ are semantically equivalent terms. This would be desirable, since it facilitates reasoning about the semantics of terms. We could try to quotient the type by α-equivalence, but this is a manual process and requires effort.

Secondly, it would be convenient to have functions for renaming and obtaining free variables in a term. While the map and set functions that are provided by the datatype seem like suitable candidates, they do not fulfill these tasks. For example, the mapper does not support capture-avoiding substitution, i.e., it might rename a bound variable to the name of a free one, which changes the term semantically.

Furthermore, this datatype represents invalid terms. This makes working with the type tedious, since it calls for a separate definition and checks for validity. For example,

α ltrm contains "Let $[(\text{Var } x, \text{Var } y), (\text{Var } x, \text{Var } z)]$ $(\text{Var } x)$" as a valid term. This would be the representation of "`let x = y and x = z in x`" which is not a valid type, since a variable cannot be bound to different terms in the same parallel let binding. This can, in theory, be solved by defining a subtype of $(\alpha \times \beta)$ list that only contains lists of pairs, where the first elements of each pair are unique in the list. However, this subtype is not a BNF right after that, and thus it cannot be used in the recursive definition.

An important step towards solving these issues has been taken by van Brügge et al. [3] in their implementation of a definitional package for binding-aware datatypes in Isabelle/HOL. They provide a new **binder_datatype** command that allows for the definition of datatypes that do not distinguish between α-equivalent terms and provide functions for renaming and obtaining the free variables of a term. Furthermore, these binding-aware datatypes (or short *binder datatypes*) provide propositions similar to those provided by a regular datatype, e.g., a principle for structural induction. This package is built on the theory of Map-Restricted Bounded Natural Functors (MRBNFs), a generalization of BNFs developed by Blanchette et al. [4].

Declaring ltrm as a binder datatype fixes the first two issues regarding α-equivalence and the obtaining and renaming of free variables in a capture-avoiding fashion. Additionally, the explicit notion of *bound* variables in MRBNFs allows us to define a linearized version of $(\alpha \times \beta)$ list that ensures the pairwise distinctness of α atoms in each element of the type. This is possible, since declaring a variable as bound allows us to restrict the map function to bijections on a certain type variable — α in this case.

In general, we call a type *linear* or *non-repetitive* on a type variable α, when all its elements only contain pairwise distinct α atoms. As described by Blanchette et al. [4, §4], complex linear MRBNFs cannot be built recursively from simpler ones. This is because non-repetitiveness is not closed under fixpoints. Alternatively, we can construct the desired type without taking non-repetitiveness into account, and as a last step, take the subset of non-repetitive elements. This produces a type constructor that has the desired properties. However, it is not registered as an MRBNF. To use the type in the construction of other binder datatypes, it is necessary to prove certain MRBNF axioms for it. Conducting these proofs has been a manual process that had to be undertaken for each linear type separately.

We aim to automate this process of linearization. For this, we implement a new command in Isabelle/HOL: **linearize_mrbnf**. This command produces a linearized type constructor equipped with the typical MRBNF constants and propositions that is non-repetitive on the specified type variables. All proofs are internally conducted in an automated fashion after the user shows two axioms necessary for these proofs — namely, the existence of a non-repetitive element of the type and strong pullback preservation.

Our new command, together with the **binder_datatype** command from van Brügge et al. [3], allows us to easily construct a binding-aware datatype that represents the lambda terms with parallel let bindings with all the desired properties. This is accomplished through the following two commands, where the goals generated by our new command are proven through an auto tactic "by (auto . . .)":

```
linearize_mrbnf (keys: α :: var, vals: β) alist = "(α :: var × β) list" on α
  by (auto simp: list_eq_iff_nth_eq map_prod_def split_beta prod_eq_iff)

binder_datatype α ltrm = Var α | App "α ltrm" "α ltrm"
  | Abs x::α t::"α ltrm" binds x in t
  | Let "(fs::α, t::α ltrm) alist" "u::α ltrm" binds fs in t u
```

Structure
The thesis is structured as follows: First, we introduce BNFs and their importance for defining datatypes in Isabelle/HOL in Section 2.1. We continue with describing their generalization to MRBNFs in Section 2.2. In the following Chapter 3, we first introduce the theory behind linearization and give sketches of the associated proofs in Section 3.1, before we describe the implementation of our new **linearize_mrbnf** command in Section 3.2. The next Chapter 4 presents an application of our command in an existing formalization of the POPLmark Challenge, where we automate the manual linearization of an important type. We conclude in Chapter 5 with a summary and two suggestions for future work that would help integrate our new command more closely with the existing datatypes.

2 Background

This Chapter serves to introduce Bounded Natural Functors (BNFs) and their generalization to Map-Restricted Bounded Natural Functors (MRBNFs). Note that when we use the notion "element" of an n-ary type constructor F, we are always talking about a term of type $(\alpha^n)\ F$. In contrast to that, we call the α_i elements that make up an F element "atoms". The structure of the F element dictates how and where the atoms occur in it.

2.1 Bounded Natural Functors (BNFs)

As described in Chapter 1, BNFs are essential for constructing datatypes and codatatypes in Isabelle/HOL. Especially for defining a datatype with recursion, it is required that the type constructor used in that recursion is registered as a BNF, i.e., it fulfills the BNF axioms. For example the following **datatype** command only succeeds if α list and $(\alpha \times \beta)$, are BNFs.

$$\textbf{datatype}\ \alpha\ \mathsf{ex} = \mathrm{A}\ "((\alpha \times \alpha)\ \mathsf{ex})\ \mathsf{list}"$$

Since BNFs are closed under composition and fixpoints, the resulting datatype (here α ex) can be automatically registered as a BNF as well.

We write type variables as Greek letters $(\alpha, \beta, \dots)$ in this thesis. However, in the Isabelle proof assistant, type variables are written with a "′" in front of a name, e.g., ′a list. To copy our examples to Isabelle, one has to replace these Greek letters with "′" variables. Alternatively, a "′" can be prepended to the Greek letters, since, for example, $'\alpha$ is a valid type variable in Isabelle.

The type variables of a BNF are divided into two groups: *live* and *dead* variables or *lives* and *deads*. Recursive occurrences may appear only in positions corresponding to live variables when defining a new datatype. Dead variables do not allow for this. We take the function type $\alpha \Rightarrow \beta$ as an example. Its first type argument α is dead, while the second one β is live. Thus, of the following, the first command succeeds while the second one fails:

$$\textbf{datatype}\ \alpha\ \mathsf{success} = \mathrm{S1} \mid \mathrm{S2}\ "\alpha \Rightarrow \alpha\ \mathsf{success}"$$
$$\textbf{datatype}\ \alpha\ \mathsf{fail} = \mathrm{F1} \mid \mathrm{F2}\ "\alpha\ \mathsf{fail} \Rightarrow \alpha"$$

The reason for this failure is that certain properties have to hold for a BNF with regard to its live variables. These properties are necessary for the internal construction of a newly specified type. A variable is dead when it has to be omitted such that these axioms hold, which is the case for the first type variable of the function type.

These properties — or "BNF axioms" as we call them — are formalized in terms of constants that characterize a BNF. For a BNF F with l live variables, these are one $l+1$-ary map function, l set functions, a bound, and an $l+2$-ary relator. In the following Subsections 2.1.1 to 2.1.4 we define these constants and motivate the BNF axioms that we formalize in Figure 2.1. We use the notation $f^l = f_1 \ldots f_l$ for the arguments of the mapper and similarly $R^l = R_1 \ldots R_l$ for the relator. Note that we extend this notation to binary (infix and prefix) operations op on functions and relations as follows: $(R^l \operatorname{op} Q^l) = (R_1 \operatorname{op} Q_1) \ldots (R_l \operatorname{op} Q_l)$ and $(\operatorname{op}\ R^l\ Q^l) = (\operatorname{op}\ R_1\ Q_1) \ldots (\operatorname{op}\ R_l\ Q_l)$.

(MAP_ID)	$\mathsf{map}_F\ \mathrm{id}^l\ x = x$
(MAP_COMP)	$\mathsf{map}_F\ g^l\ (\mathsf{map}_F\ f^l\ x) = \mathsf{map}_F\ (g \circ f)^l\ x$
(MAP_CONG)	$(\forall i.\ \forall z \in \mathsf{set}_{F,i}\ x.\ f_i\ z = g_i\ z) \Longrightarrow \mathsf{map}_F\ f^l\ x = \mathsf{map}_F\ g^l\ x$
(SET_MAP)	$\forall i.\ \mathsf{set}_{F,i}\ (\mathsf{map}_F\ f^l\ x) = f_i\ \grave{}\ \mathsf{set}_{F,i}\ x$
(BD)	$\mathsf{infinite}\ \mathsf{bd}_F \wedge \mathsf{regular}\ \mathsf{bd}_F \wedge \mathsf{cardinal\_order}\ \mathsf{bd}_F$
(SET_BD)	$\forall i.\ \lvert \mathsf{set}_{F,i}\ x \rvert <_o \mathsf{bd}_F$
(REL_COMPP)	$(\mathsf{rel}_F\ R^l\ \bullet\ \mathsf{rel}_F\ Q^l)\ x\ y \Longrightarrow \mathsf{rel}_F\ (R\ \bullet\ Q)^l\ x\ y$
(IN_REL)	$\mathsf{rel}_F\ R^l\ x\ y =$
	$\exists z.\ (\forall i.\ \mathsf{set}_{F,i}\ z \subseteq \{(a,b).\ R_i\ a\ b\}) \wedge \mathsf{map}_F\ \mathrm{fst}^l\ z = x \wedge \mathsf{map}_F\ \mathrm{snd}^l\ z = y$

where $\grave{}$ is the image function on sets, $\bullet$ is the composition of relations and $<_o$ is the less than relation on cardinals

Figure 2.1: The BNF axioms

2.1.1 Map function and functors

The map function or *mapper* takes one function for each live of F as arguments, as well as one F element. The domain types of these functions are the lives of F. These functions are applied to the atoms of an element. The result is a new element of type F, where the original type variables are replaced by the range types of the mapped functions. Taking the α list type as an example, a BNF with one live α, the mapper has the type $\mathsf{map}_{\mathsf{list}}\ ::\ (\alpha \Rightarrow \alpha') \Rightarrow \alpha\ \mathsf{list} \Rightarrow \alpha'\ \mathsf{list}$.

To make F with its mapper a *functor* on the universe of all types, the mapper has to fulfill two axioms [1]. First, mapping the id function on all lives over an element should leave it unchanged, which is formalized in MAP_ID (Fig. 2.1). The second property MAP_COMP (Fig. 2.1) is concerned with mapping composed functions and reads as follows: Mapping two lists of functions over an element, e.g., first $f_1 \dots f_l$ and then $g_1 \dots g_l$, should produce the same result as mapping the pair-wise composed functions $(g_1 \circ f_1) \dots (g_l \circ f_l)$ over it once. A type constructor F with a map function map_F fulfilling these two properties is considered a functor.

2.1.2 Set functions and naturality

A set function or *setter* is defined for each of the l live variables. Applied to an F-element, the i-th setter returns the set of all atoms that are part of the element and correspond to the i-th live. For example, the setter of the list type returns the set of elements in the list. We note here that when we write i as an index, we assume it to be in the range $1 \leq i \leq l$.

The set functions together with the mapper give rise to another property. We want the setters $\mathsf{set}_{F,i}$ to be natural transformations from F and map_F to the set and image function. Thus, they should fulfill the SET_MAP axiom (Fig. 2.1). It states that taking the i-th set of an F after mapping $f_1 \dots f_l$ to it, results in the same set as if the i-th set was taken from the original F before the image of f_i was applied to it. Figure 2.2 shows a visualization of this axiom and reads as follows: Starting from an F element, first applying the setter and then mapping a function (path through the top right) results in the same as first mapping the function and then applying the setter (path through the bottom left).

$$\begin{tikzcd}
(\alpha_1, \dots \alpha_l)\ F \arrow[r, "\mathsf{set}_{F,i}"] \arrow[d, "\mathsf{map}_F\ f_1\ \dots\ f_l"'] & \alpha_i\ \mathsf{set} \arrow[d, "\text{image}\ f_i"] \\
(\beta_1, \dots \beta_l)\ F \arrow[r, "\mathsf{set}_{F,i}"] & \beta_i\ \mathsf{set}
\end{tikzcd}$$

for all i where α_i is a live variable of F

Figure 2.2: $\mathsf{set}_{F,i}$ as a natural transformation

This axiom alone would be fulfilled by declaring every setter as the constant function returning the empty set. To solve this, the MAP_CONG axiom (Fig. 2.1) acts as a completeness property on the setter. It states that if two (lists of) functions f^l and g^l are equal when applied to the corresponding sets of all atoms of an F element (obtained through the setters), then mapping these two lists of functions over the F element each produces the same result. When this property holds, we can be sure that the mapper only depends on how the functions f^l behave on the atoms of the F element. At the same time, this axiom ensures that the setters actually return the complete set of atoms of the F element.

2.1.3 Bound and boundedness

Lastly, the BNF must be equipped with an infinite cardinal as a bound. This bound may depend on the cardinalities of the dead variables, but not on the cardinalities of the live variables. In Isabelle/HOL, cardinals are implemented as minimal wellorders with respect to isomorphisms [5]. For example, *natLeq*, the cardinal that originates from the $\leq$ order on natural numbers, is equivalent to the smallest infinite cardinal $\aleph_0$. While details about this implementation are certainly interesting, we will not focus on these details in this thesis and refer to cardinals in terms of their $\aleph$-notation.

Besides being a cardinal order, the bound is required to be infinite, i.e., at least $\aleph_0$ with respect to the cardinal order $\leq_o$, and regular. Regularity means that an infinite cardinal κ is stable under union, i.e., the union of any set of sets that are of smaller cardinality than κ also has smaller cardinality than κ. Note that the set of sets must also be of smaller cardinality than κ:

$$(\bigwedge_{i \in I} |S_i| <_o \kappa) \wedge |I| <_o \kappa \Longrightarrow |\bigcup_{i \in I} S_i| <_o \kappa$$

The bound is used in the SET_BD axiom (Fig. 2.1) to ensure that the sets obtained by the setters are bounded. This ensures that the branching of a recursively defined datatype is also bounded, and so is the resulting type F as well.

2.1.4 Relator and shapes

The relator is used to build a relation on F by relating the atoms of an F element. It takes one relation for each live, that relates the corresponding type variables of the two Fs that are to be related. As an example, we give the type and definition of the relator for the product and list type as follows, where $x!i$ refers to the atom of list x at index i:

$$\begin{aligned}
&\mathsf{rel}_{\mathsf{prod}} :: \ (\alpha \Rightarrow \alpha' \Rightarrow \mathsf{bool}) \Rightarrow (\beta \Rightarrow \beta' \Rightarrow \mathsf{bool}) \Rightarrow (\alpha \times \beta) \Rightarrow (\alpha' \times \beta') \Rightarrow \mathsf{bool}\\
&\mathsf{rel}_{\mathsf{prod}}\ R\ Q\ x\ y := \ R\ (\mathsf{fst}\ x)\ (\mathsf{fst}\ y) \wedge Q\ (\mathsf{snd}\ x)\ (\mathsf{snd}\ y)\\
&\mathsf{rel}_{\mathsf{list}} :: \ (\alpha \Rightarrow \alpha' \Rightarrow \mathsf{bool}) \Rightarrow \alpha\ \mathsf{list} \Rightarrow \alpha'\ \mathsf{list} \Rightarrow \mathsf{bool}\\
&\mathsf{rel}_{\mathsf{list}}\ R\ x\ y := \ \mathsf{length}\ x = \mathsf{length}\ y \wedge (\forall i \leq \mathsf{length}\ x.\ R\ (x!i)\ (y!i))
\end{aligned}$$

Considering the list type again, we make an interesting observation: There are some α lists x and y that the relator cannot relate, no matter which α relation is chosen. The relator on lists is defined index-wise, i.e., the α relation must relate the elements of both lists for each index. Consequently, lists of different lengths cannot be positively related. We think of the length of a list as its *shape*. We can generalize this idea of shape to an arbitrary type constructor F. The shape of an F element is defined by the way it is constructed, and the relator can only ever relate those that have the same or equivalent shape, i.e., it will always evaluate to *false* when two elements of different shape are given, regardless of the relations given to the relator. We can think of an element of type F as a container that has a certain *shape* with slots for *atoms*. These atoms are elements of the type constructor's type arguments.

2.1.5 Additional BNF axioms

Additionally to the axioms we already motivated in the previous Subsections (MAP_ID and MAP_COMP for the functoriality of F, SET_MAP and MAP_CONG to ensure that the setters are natural transformations and the boundedness of the setters SET_BD), we have three additional ones.

The axiom BD (Fig. 2.1) just ensures that the bound bd_F is a suitable cardinal, i.e., a regular and infinite one.

The relator is required to be distributive, i.e., it should fulfill $(\mathsf{rel}_F\ R^l\ \bullet\ \mathsf{rel}_F\ Q^l)\ x\ y = \mathsf{rel}_F\ (R\ \bullet\ Q)^l\ x\ y$. We note here that for showing that a type constructor is a BNF, it is only necessary to prove the implication stated in REL_COMPP (Fig. 2.1). The other direction of the implication follows automatically.

Lastly, IN_REL (Fig. 2.1) characterizes the relator. It is the most abstract and complex axiom, but we want to give an intuition about it here: The idea is that two elements x and y of the type $(\alpha^l)\ F$ are related through a relation R iff there exists a z that acts as a "zipped" version of x and y. The atoms of this z are R_i-related pairs of the atoms of x and y, where the first position in the pair corresponds to x and the second one to y. This axiom (IN_REL) together with the previous one (IN_REL) amounts to *weak pullback preservation* of the BNF.

2.1.6 Non-emptiness witnesses

BNF carry non-emptiness witnesses as proof that the type contains at least one element. Witnesses may depend on a subset of the BNF's live variables. For example a witness of $(\alpha_1,\ \ldots\ \alpha_l)\ F$ that depends on the first and last type variable of F, this witness has the type $\mathsf{wit}_F :: \alpha_1 \Rightarrow \alpha_l \Rightarrow (\alpha_1,\ \ldots\ \alpha_l)\ F$. It denotes that given witnesses for the types α_1 and α_l, a witness for F can be constructed. A deeper insight into the theory and usage of witnesses for datatypes in Isabelle/HOL is given by Blanchette et al. [6].

Witnesses have to fulfill the following properties: For all type variables α_i the witness depends on, the witness may only contain the α_i elements w_i, that were given to the witness as arguments, i.e., $\textsc{set}_{F,i}$ applied to the witness evaluates to the singleton $\{w_i\}$. Furthermore, the witness must not contain any elements of the live type variables α_j, the witness does not depend on, i.e., $\textsc{set}_{F,j}$ must be empty. We formalize these properties in the following, where $\bar{w}$ denotes the arguments that the witness depends on.

$$(\textsc{wits}) \qquad \forall i.\ \mathsf{set}_{F,i}\ (\mathsf{wit}_F\ \bar{w}) = (\texttt{if}\ \mathsf{wit}_F \text{ depends on } \alpha_i\ \texttt{then}\ \{w_i\}\ \texttt{else}\ \emptyset)$$

Multiple witnesses can exist for a given type constructor. For example, we can give two witnesses for α list: $[\,]$ of type α list and $(\lambda a.\ [a])$ of type $\alpha \Rightarrow \alpha$ list. The first of these witnesses does not depend on a type variable, while the second one depends on α. The first witness $[\,]$ is the more general of the two, since it allows us to give an element of the list type without the need for an element of α. In general, we say a witness $\mathsf{wit}_{F,1}$ *subsumes* $\mathsf{wit}_{F,2}$, when $\mathsf{wit}_{F,1}$ depends on a proper subset of the arguments of $\mathsf{wit}_{F,2}$. Subsuming witnesses are more versatile than subsumed ones, and thus we ignore subsumed ones when registering them with a BNF. However, when two witnesses have overlapping dependencies but neither depends on a subset of the other, we are interested in both, since neither fully subsumes the other.

2.1.7 BNF examples

A Further example of a simple BNFs is the type of finite sets α fset. This type is interesting for the reason that it is a subtype of the set type, which is not a BNF. By enforcing finiteness for the elements of the type, it is possible to give a bound for the set function, fulfilling the SET_BD axiom, which is not possible for the unrestricted set type. Since unboundedness is the only reason that the set type is not a BNF, α fset can be shown to be a BNF.

To demonstrate how BNFs can be combined to create new ones, we consider the type constructor (α, β) plist = $(\alpha \times \beta)$ list. We define for it a map function ($\mathsf{map}_{\mathsf{plist}}$) and two set functions ($\mathsf{set1}_{\mathsf{plist}}$ and $\mathsf{set2}_{\mathsf{plist}}$) as well as a relator $\mathsf{rel}_{\mathsf{plist}}\ R\ Q$. We give the exact definitions in terms of the standard map, set, and relator functions of the list and product type as follows:

$$\begin{aligned}
\mathsf{map}_{\mathsf{plist}}\ f\ g\ &= \mathsf{map}_{\mathsf{list}}\ (\mathsf{map}_{\mathsf{prod}}\ f\ g)\\
\mathsf{set1}_{\mathsf{plist}}\ xs &= \mathsf{set}_{\mathsf{list}}\ (\mathsf{map}_{\mathsf{list}}\ \mathsf{set1}_{\mathsf{prod}}\ xs)\\
\mathsf{set2}_{\mathsf{plist}}\ xs &= \mathsf{set}_{\mathsf{list}}\ (\mathsf{map}_{\mathsf{list}}\ \mathsf{set2}_{\mathsf{prod}}\ xs)\\
\mathsf{rel}_{\mathsf{plist}}\ R\ Q &= \mathsf{rel}_{\mathsf{list}}(\mathsf{rel}_{\mathsf{prod}}\ R\ Q)
\end{aligned}$$

To show that (α, β) plist is a BNF, we have to prove the BNF axioms for it. Besides the definitions above, we give $\aleph_0$ as the bound $\mathsf{bd}_{\mathsf{plist}}$.

2.2 Map-Restricted Bounded Natural Functors (MRBNFs)

Type constructors that involve names or bindings often violate the requirements of BNFs. An example of this is the type of key-value lists $\alpha\ \beta$ alist with α as the type of keys. This is a subtype of the previously defined $\alpha\ \beta$ plist that describes only lists of pairs that are pairwise distinct on the first elements, i.e., the α atoms. We call it a key-value list since one can think of the first atom of each pair as a distinct key that identifies a value, i.e., the second atom.

The main issue with this type is that the map function we defined in Subsection 2.1.7 for plist cannot be used for alist right away. The reason is that it cannot guarantee that the list that results from the map is still distinct on α, i.e., that it is still a member of the type. Thus, in BNF terms, the type variable α of (α, β) alist is dead, which is a huge drawback for the versatility of the type. For example, using (α, β) alist in a **datatype** command would kill α also in the resulting datatype, effectively disabling any map functions on that type variable. However, by enforcing that only *bijections* are mapped over α, we can ensure that the result of a map on an alist still fulfills the distinctness property of the type. We note here that there might exist a map function with which alist could be proven to be a BNF with two live variables. However, it is not obvious how this map function should be defined.

MRBNFs are a generalization of BNFs. Restricting the map function of a functor to *small-support* functions or *small-support bijections* for certain type variables allows us to reason about type constructors in terms of BNF properties, even in cases where this would not be possible otherwise. Concretely, it allows us to include type variables in the mapper that would be considered *dead* from a BNF point of view. We can reason about these variables in terms of the BNF axioms (Fig. 2.1) with a few restrictions and consequently use them in recursive definitions of binder-datatypes under certain conditions. This is explained in more detail in Subsection 2.2.2.

We call type variables that are restricted to small-support functions *free* variables or *frees* and those restricted to small-support bijections *bound* variables or *bounds*. This allows us to define MRBNFs with four types of variables (lives, frees, bounds, and deads) as opposed to BNFs, which only distinguish between lives and deads. Our example from the beginning of this section, the distinct list α dlist is an MRBNF with α as a bound variable.

A small-support function leaves most arguments unchanged, meaning it acts as the identity function on them. Concretely defined, the cardinality of the set of arguments the function changes must be smaller than the cardinality of the argument type itself:

$$\mathsf{small\_supp}\ f = |\{x :: \alpha.\ f\ x \neq x\}| <_o |\Omega_\alpha|$$

where Ω_α is the universe of type α.

To motivate the restrictions to small-support functions for frees and small-support bijections for bounds, we need to think about lambda calculus again. Consider as an example, that we want to apply a capture avoiding substitution of y for x in the term $(\lambda y.\ (y\ x)\ \mathtt{t})$, where $\mathtt{t}$ is an arbitrary term that can contain further occurrences of x or y. This substitution is only capture-free if we rename the bound variable y to a fresh variable (e.g., y' if it does not occur as a free variable in $\mathtt{t}$) before applying the substitution. Mapping a function v over the free variables is like applying multiple capture-free substitutions at the same time. The restriction to small-support is necessary to ensure that when we encounter such a situation, where we need to rename a bound variable first, we can always find a fresh variable that is not in the support of v, i.e., a variable that is not changed by v. The restriction to bijections for bounds comes into play when renaming bounds. Here, we additionally need to avoid collapsing two bound variables to the same, which could alter the semantics of a term.

For an MRBNF F with l lives, fr frees, and b bounds, we define $\varpi = l + fr + b$ as the number of all non-dead type variables. With this, the mapper and setters are expanded to work for the frees and bounds just as they do for lives. Thus, F has ϖ setters and a mapper with arity $\varpi + 1$. We note here that any small-support function acts as the identity function at least for "some" (actually for "most") inputs and thus its domain and co-domain are the same, i.e., it is a *endofunction*. This means that the type variables for frees and bounds are the same for the F argument of the mapper and the result, while the live type variables may be changed through a map.

As before, we use the notation $f^l = f_1 \ldots f_l$ for functions and $R^l = R_1 \ldots R_l$ for relations on the live variables. Analogously, we write v^{fr} and u^b for functions on frees and bounds. Furthermore, we write the arguments of the map function as $f^l\ v^{fr}\ u^b$. As an example, the mapper of the type $(\alpha,\ \beta,\ \gamma)\ F$ where α and β are free and γ is bound has the following type:

$$\mathsf{map}_F :: (\alpha \Rightarrow \alpha) \Rightarrow (\beta \Rightarrow \beta) \Rightarrow (\gamma \Rightarrow \gamma) \Rightarrow (\alpha,\ \beta,\ \gamma)\ F \Rightarrow (\alpha,\ \beta,\ \gamma)\ F$$

From now on, we assume that the type variables of any MRBNF are ordered *lives* first, followed by *frees* and *bounds*, and *deads* at the end. This simplifies many definitions and arguments we make about MRBNFs. However, these are all easily generalized to an arbitrary ordering of type variables. For example, the argument order of the mapper might be different, as the lives, bounds, and frees do not have to be separated, but can be interlaced in some order. In concrete examples, we may use MRBNFs that are explicitly defined in terms of primitive types like list or prod. For these examples, we may define a different order for live, free, and bound type variables.

We keep the original relator that only relates live variables with given relations and relates the free and bound variables with equality. Thinking in our model of F elements being shapes with atoms in slots, the regular relator rel_F requires the frees and bounds in each slot to be the same for both elements that are compared. To relate F elements that are not equal in all frees and bounds, we introduce a new map-restricted relator $\mathsf{mr\_rel}_F$. It takes a function for each free and bound — with the appropriate restrictions to small-support and bijectivity — in addition to the relations for the lives. It then uses the graphs Grp of these functions as relations for the respective free or bound variable. Transferring the ideas of bijectivity and small-support to these graph relations, the graph of a bijective function relates each atom to exactly one other atom, while the graph of a small-support function acts as equality on all the arguments that are not in its support. The new arguments of the map-restricted relator are placed in front of the relations for the live variables. It is then defined in terms of the relator as shown below. Note that relating two elements with the graph of a function v is equivalent to mapping v over the first element and relating that to the second one by equality. Thus, we define it as follows:

$$\mathsf{mr\_rel}_F\ u^b\ v^{fr}\ R^l\ x\ y = \mathsf{rel}_F\ R^l\ (\mathsf{map}_F\ \mathrm{id}^l\ v^{fr}\ u^b\ x)\ y$$

2.2.1 MRBNF axioms

MRBNFs require the same axioms as BNFs with slight modifications. We take the formalized axioms from Figure 2.1 as a base and explain the differences.

For the MAP_COMP, MAP_CONG, and SET_MAP axioms, we add the assumptions that the functions that correspond to frees and bounds are small-support functions and that the ones corresponding to bounds are additionally bijections. It means that $\mathsf{small\_supp}\ v^{fr} \wedge \mathsf{small\_supp}\ u^b \wedge \mathsf{bijective}\ u^b$ is added as assumptions to these axioms.

Furthermore, while REL_COMPP stays unchanged, using the original relator, IN_REL is changed to be defined in terms of the map-restricted relator $\mathsf{mr\_rel}_F$ as follows:

$$\begin{aligned}\mathsf{mr\_rel}_F\ u^b\ v^{fr}\ R^l\ x\ y = \exists z.\ (\forall i.\ \mathsf{set}_{F,i}\ z \subseteq \{(a,b).\ R_i\ a\ b\})\wedge \\ \mathsf{map}_F\ \mathrm{fst}^l\ \mathrm{id}^{fr}\ \mathrm{id}^b\ z = x \wedge \mathsf{map}_F\ \mathrm{snd}^l\ v^{fr}\ u^b\ z = y\end{aligned}$$

The type variables of an MRBNF are required to be *large* and *regular*. Largeness is necessary to ensure that there are always fresh atoms available for renaming. It is defined as the cardinality of the type being at least the bound of F bd_F for datatypes or the cardinal successor $\mathsf{cardSucbd}_F$ for codatatypes. In Isabelle, the requirements of largeness and regularity are combined in dedicated type classes. For the cases where $\mathsf{bd}_F = \aleph_0$, predefined `var` and `covar` implement the appropriate requirements.

2.2.2 Datatypes with bindings

MRBNFs can be used in a **binder_datatype** command to produce a binding aware datatype. We consider the example from the introduction (Chapter 1) again, where we defined a binder datatype to represent lambda terms with parallel let bindings:

```
binder_datatype α ltrm = Var α | App "α ltrm" "α ltrm"
  | Abs x::α t::"α ltrm" binds x in t
  | Let "(fs::α, t::α ltrm) alist" "u::α ltrm" binds fs in t u
```

The core differences we notice compared to a regular datatype definition are the names given to certain components (e.g., x::α) and the usage of these names to declare bindings (e.g., **binds** x **in** t). For the abstraction case Abs, this declaration means that for a term Abs x t, the variable x is considered bound in t.

In general, any name occurring between **binds** and **in** corresponds to a type variable that is to be bound. In contrast to that, a name that occurs after the **in** corresponds to a recursive occurrence of the datatype in which the variables are being bound. These *binds-in* clauses define a *binding dispatcher* between the inputs of binding variables and the term inputs. Most importantly, the **binder_datatype** command allows us to use an MRBNF with bound variables in the definition of a datatype, if the bound variables correspond to type variables that are to be bound.

The result of this command is a binding-aware datatype. This datatype considers α-equivalent terms equal, i.e., it is quotiented by alpha equivalence, supports capture-avoiding variable substitution, and other features relevant for reasoning about lambda terms or other syntaxes with bindings. More details about the details on how these are given by Blanchette et al. [4, §5].

3 Linearizing MRBNFs

This chapter explains the theoretical definitions and steps for linearizing an arbitrary MRBNF. We state necessary conditions for the linearization and sketch out proofs for a few intermediate lemmas that help us to construct the new linearized MRBNF. We note here that the theory we present in Section 3.1 is a generalization of an example by Blanchette et al. [4]. This example is the linearization of an MRBNF in Isabelle/HOL. After the theory, we describe the syntax of our new **linearize_mrbnf** command and details about its implementation in Section 3.2.

3.1 Theory of linearization

In this section, we define the linearization of an MRBNF F on a subset of its *live* variables. Linearization means that the resulting type only contains elements for which all atoms of the linearized variables are distinct. We say F is *non-repetitive* on these variables. This type is also an MRBNF with the same variable types (*live*, *free*, *bound*, *dead*), except for the linearized variables that change their type from *live* to *bound*. This means that the new map function is restricted to only allow bijective and small-support functions on these variables.

We formalize the idea of distinctness of atoms as *non-repetitiveness* on the linearized variables. In our notation, we use $\mathit{lin} \leq l$ to refer to the number of linearized variables. Furthermore, we assume the variables that we linearize on to be the *last* lin of the live variables. Consequently, the first $l' = l - \mathit{lin}$ lives of F are not linearized.

3.1.1 Non-repetitiveness

At the core of linearization lies the notion of *non-repetitiveness*. An element x of a type is considered to be non-repetitive with respect to a type variable α if it does not contain repeating α-atoms. For example, an α list is non-repetitive if all of its α-elements it contains are pairwise distinct. To define non-repetitiveness for an arbitrary MRBNF, we express this property in terms of its map, set, and relator functions. Considering α lists again, we can show a list xs to be distinct, iff for each other list ys of the same length, we can find a function f such that $ys = \mathsf{map}_{\mathsf{list}}\ f\ xs$. If xs were not distinct, there must exist two indices with the same α element in xs. Furthermore, there exists a ys that has different elements at these two indices and thus a function mapping *xs* to *ys* cannot exist, since it would have to map two same elements in *xs* to two differing ones in *ys*.

In Subsection 2.1.4, we proposed the idea to think about elements of a BNF (or MRBNF) F as containers with a certain shape with atoms in slots specified by the shape. Using this model, we can generalize the notion of lists having the same length to F elements having the same shape. We can express this through the relator by using the $\top$ relation that relates every pair of arguments to each other. Thus, we give the definition of equivalent shape and non-repetitiveness for a list:

$$\begin{aligned}
&\mathsf{eq\_shape}_{\mathsf{list}}\ x\ y = \mathsf{rel}_{\mathsf{list}}\ (\top)\ x\ y \\
&\mathsf{nonrep}_{\mathsf{list}}\ x = \forall y.\ \mathsf{eq\_shape}_{\mathsf{list}}\ x\ y \Longrightarrow (\exists f.\ y = \mathsf{map}_{\mathsf{list}}\ f\ x)
\end{aligned}$$

Note that we use the regular relator that only relates live variables with given relations, while it requires equality for all frees and bounds.

Based on this, x is a non-repetitive element if for all other elements y with equal shape, a function exists through which x can be mapped to y. In our example of list, this holds for all lists with distinct elements (given a second list y of the same length, one can easily define a function mapping the distinct atoms of x to those of y). It does not hold for lists with repeating elements, because no f exists that could map two equal elements at different positions in this list to distinct elements in an arbitrary second list.

More interesting is the case of $(\alpha,\ \beta)$ alist which we only want to be non-repetitive on α. For our purpose of defining non-repetitiveness on a subset of the live variables, we fix the other live variables to be equal when defining equivalent shape. For MRBNFs with more than one live variable, we can give a definition of *non-repetitiveness* and having *equal shape* on the last lin live variables. In that case, we consider x and y of type F to have equal shape with respect to the variables $\alpha_{l'+1} \ldots \alpha_{\mathit{lin}}$, iff they are *equal* in the atoms corresponding to the non-linearized lives and are related with $\top$ in the linearized variables. Consequently, for the map in the nonrep definition, the id function is applied to the non-linearized lives, since they are already required to be equal.

$$\begin{aligned}
&(\textsc{eq\_shape})\quad \mathsf{eq\_shape}_F^{\mathit{lin}}\ x\ y = \mathsf{rel}_F\ \langle (=)^{l'}\ (\top)^{\mathit{lin}} \rangle\ x\ y \\
&(\textsc{nonrep})\quad \mathsf{nonrep}_F^{\mathit{lin}}\ x = \forall y.\ \mathsf{eq\_shape}_F^{\mathit{lin}}\ x\ y \Longrightarrow (\exists f^{\mathit{lin}}.\ y = \mathsf{map}_F\ \langle \mathrm{id}^{l'}\ f^{\mathit{lin}} \rangle\ \mathrm{id}^{\mathit{fr}}\ \mathrm{id}^{b}\ x)
\end{aligned}$$

Note that we use $\langle \ldots \rangle$ to indicate arguments that belong together, e.g., that they are both related to lives in this case. They are just inserted to improve readability. Once again, our arguments hold for any ordering of type variables. The assumption that we linearize on the last lin variables only serves readability and is not a limitation in the actual implementation, where we allow an arbitrary subset of lives to be chosen for linearization.

3.1.2 Prerequisites for linearization

Two properties are necessary to linearize a MRBNFs. First, to ensure that the resulting type constructor is non-empty, it is required that there exists a non-repetitive element (with respect to the linearized variables): $\exists x.\ \mathsf{nonrep}_F^{lin}\ x$.

Furthermore, even though MRBNFs are already required to preserve weak pullbacks, as described in Subsection 2.1.5, for the linearization, it is required that they preserve *all* pullbacks. Formalized, this means that the existence of z in the IN_REL axiom (Fig. 2.1) has to be fulfilled uniquely, i.e., for each R^l-related x and y, there exists *exactly one* z fulfilling this property. For example, strong pullback preservation is fulfilled by the α list and $\alpha\ \beta$ prod functor but not by α fset, the type constructor for finite sets of αs. We refer to this property as "strong pullback preservation" even though strictly speaking, REL_COMPP

(IN_REL_STRONG) $\mathsf{rel}_F\ R^l\ x\ y = \exists! z.\ (\forall i.\ \mathsf{set}_{F,i}\ z \subseteq \{(a,b).\ R_i\ a\ b\}) \wedge$
$\mathsf{map}_F\ \mathrm{fst}^l\ \mathrm{id}^{fr}\ \mathrm{id}^b\ z = x \wedge \mathsf{map}_F\ \mathrm{snd}^l\ \mathrm{id}^{fr}\ \mathrm{id}^b\ z = y$

We note here that the requirement of strong pullback preservation can be omitted when the MRBNF is linearized on all its live variables, i.e., when the linearized MRBNF has no live variables. This is because in this case, the REL_EXCHANGE lemma explained in Subsection 3.1.3 becomes trivial. In all other cases, that lemma is the sole reason, strong pullback preservation is required.

3.1.3 Intermediate lemmas

We want to prove the MRBNF axioms for the linearized MRBNF. For this, we utilize a few intermediate lemmas, which we present in this section.

F is strong

From the pullback preservation with uniqueness, we can prove the following lemma. In fact, this notion of strength is equivalent to pullback preservation:

(F_STRONG) $\mathsf{rel}_F\ R^l\ x\ y\ \wedge\ \mathsf{rel}_F\ Q^l\ x\ y \Longrightarrow \mathsf{rel}_F\ (\inf\ R\ Q)^l\ x\ y$

Here, the infimum inf of two relations R_i and Q_i relates exactly those elements that are related by both R_i and Q_i. To prove this lemma, we first conclude that since x and y are related through some relations, they are also related with the $\top$-relation on all lives. This is the case since the relator or an MRBNF is monotonic and $\top$ relates all atoms with each other. Unfolding IN_REL_STRONG on that, we can eliminate the subset conditions for the setters, since the right sides of the subsets are just the universe of pairs with the appropriate type. The reason for this is the $\top$-relation that is fulfilled by every pair in this set. This now gives us the knowledge that there exists exactly one z that is the "zipped" version of x and y, or in other words, any two z and z' fulfilling this condition must be equal.

$$\begin{aligned}
& \mathsf{rel}_F\ R^l\ x\ y \\
\Longrightarrow\ & \mathsf{rel}_F\ (\top)^l\ x\ y && \mathsf{rel}_F \text{ mono} \\
\Longrightarrow\ & \exists! z.\ (\forall i.\ \mathsf{set}_{F,i}\ z \subseteq \{(a,b).\ (\top)\ a\ b\}) \land && \textsc{in\_rel\_strong} \\
& \quad \mathsf{map}_F\ \mathrm{fst}^l\ \mathrm{id}^{fr}\ \mathrm{id}^b\ z = x \land \mathsf{map}_F\ \mathrm{snd}^l\ \mathrm{id}^{fr}\ \mathrm{id}^b\ z = y \\
\Longrightarrow\ & \exists! z.\ \mathsf{map}_F\ \mathrm{fst}^l\ \mathrm{id}^{fr}\ \mathrm{id}^b\ z = x \land \mathsf{map}_F\ \mathrm{snd}^l\ \mathrm{id}^{fr}\ \mathrm{id}^b\ z = y && (\top) \equiv \mathrm{True} \\
\Longrightarrow\ & \forall z\ z'.\ (\mathsf{map}_F\ \mathrm{fst}^l\ \mathrm{id}^{fr}\ \mathrm{id}^b\ z = x \land \mathsf{map}_F\ \mathrm{snd}^l\ \mathrm{id}^{fr}\ \mathrm{id}^b\ z = y \land \\
& \quad \mathsf{map}_F\ \mathrm{fst}^l\ \mathrm{id}^{fr}\ \mathrm{id}^b\ z' = x \land \mathsf{map}_F\ \mathrm{snd}^l\ \mathrm{id}^{fr}\ \mathrm{id}^b\ z' = y \Longrightarrow && \exists! \text{ alternative} \\
& \quad z = z')
\end{aligned}$$

When we unfold the MRBNF axiom IN_REL on the original formulation of the lemma, we obtain a construct with two existential quantifiers $\exists z$ in the assumptions and one in the goal. With the knowledge we gained above, we can conclude that they must all be equal.

$$\begin{aligned}
& \mathsf{rel}_F\ R^l\ x\ y\ \land\ \mathsf{rel}_F\ Q^l\ x\ y \Longrightarrow \mathsf{rel}_F\ (\inf\ R\ Q)^l\ x\ y \\
\equiv\ & \exists z_R.\ (\forall i.\ \mathsf{set}_{F,i}\ z_R \subseteq \{(a,b).\ R_i\ a\ b\}) \land && \textsc{in\_rel} \\
& \quad \mathsf{map}_F\ \mathrm{fst}^l\ \mathrm{id}^{fr}\ \mathrm{id}^b\ z_R = x \land \mathsf{map}_F\ \mathrm{snd}^l\ \mathrm{id}^{fr}\ \mathrm{id}^b\ z_R = y \land \\
& \exists z_Q.\ (\forall i.\ \mathsf{set}_{F,i}\ z_Q \subseteq \{(a,b).\ Q_i\ a\ b\}) \land \\
& \quad \mathsf{map}_F\ \mathrm{fst}^l\ \mathrm{id}^{fr}\ \mathrm{id}^b\ z_Q = x \land \mathsf{map}_F\ \mathrm{snd}^l\ \mathrm{id}^{fr}\ \mathrm{id}^b\ z_Q = y\ \Longrightarrow \\
& \exists z_{\inf}.\ (\forall i.\ \mathsf{set}_{F,i}\ z_{\inf} \subseteq \{(a,b).\ (\inf R_i\ Q_i)\ a\ b\}) \land \\
& \quad \mathsf{map}_F\ \mathrm{fst}^l\ \mathrm{id}^{fr}\ \mathrm{id}^b\ z_{\inf} = x \land \mathsf{map}_F\ \mathrm{snd}^l\ \mathrm{id}^{fr}\ \mathrm{id}^b\ z_{\inf} = y \\
\equiv\ & (\forall i.\ \mathsf{set}_{F,i}\ z \subseteq \{(a,b).\ R_i\ a\ b\}) \land && z = z' \text{ (see above)} \\
& (\forall i.\ \mathsf{set}_{F,i}\ z \subseteq \{(a,b).\ Q_i\ a\ b\}) \Longrightarrow \\
& (\forall i.\ \mathsf{set}_{F,i}\ z \subseteq \{(a,b).\ (\inf R_i\ Q_i)\ a\ b\})
\end{aligned}$$

The last step is proven by applying common rules on sets of pairs, subsets, and conjunction after $(\inf R_i\ Q_i)\ a\ b$ is unfolded to $R_i\ a\ b \land Q_i\ a\ b$.

Relation exchange

The *exchange of relations* is a consequence of the previous property, F_STRONG: If two elements x and y are related through the relator with two different lists $R^l = R_1 \ldots R_l$

and $Q^l = Q_1 \ldots Q_l$ of atom-level relations, then x and y are also related with any index-wise combination of R^l or Q^l. For each index i, either the relation R_i or Q_i is selected.

For our purpose of linearization, we are specifically interested in the case where for all live variables that we linearize on the relation from R^l is chosen and for all others the relation from Q^l relation, i.e., $\langle Q^{l'}\ R^{lin} \rangle$. This results in the following lemma for an MRBNF F:

(REL_EXCHANGE) $$\mathsf{rel}_F\ R^l\ x\ y\ \wedge\ \mathsf{rel}_F\ Q^l\ x\ y \Longrightarrow \mathsf{rel}_F\ \langle Q^{l'}\ R^{lin} \rangle\ x\ y$$

The lemma F_STRONG states that for each type variable, the atoms are related with the infimum of R_i and Q_i. This means that the atoms of each type variable can be related with R_i and Q_i at the same time. To prove this lemma, we choose the appropriate one of the two relations from each of the l infima. This informal idea is the core of this lemma's formal proof.

In the specific case, that the MRBNF is linearized on *all* of its live variables, $l' = 0$ and $lin = l$ resulting in R^l as the combination that is chosen. Then the lemma becomes trivial, since its goal is equal to its first assumption in this case.

As a consequence of this, the previous lemma F_STRONG is not needed to prove this lemma. Furthermore, this lemma is the sole reason why F_STRONG and strong pullback preservation are needed for the linearization. Thus, the requirement of pullback preservation can be lifted in the case that the linearization is applied to all live variables at the same time.

Mapper peresrves non-repetitiveness

An important lemma used frequently in the following proofs is that the mapper preserves non-repetitiveness. It means that given a non-repetitive element x, the result of mapping functions over it is also non-repetitive. These functions must fulfill the appropriate restrictions of bijectivity and small-support for the bounds and lives. Additionally, the functions f^{lin} on the linearized lives need to be bijective.

(NONREP_MAP) $$\begin{aligned}&\mathsf{small\_supp}\ v^{fr}\ \wedge\ \mathsf{small\_supp}\ u^b\ \wedge\ \mathsf{bijective}\ u^b\ \wedge\\ &\mathsf{bijective}\ f^{lin}\ \wedge\ \mathsf{nonrep}_F^{lin}\ x\ \Longrightarrow \mathsf{nonrep}_F^{lin}\ (\mathsf{map}_F\ \langle g^{l'}\ f^{lin} \rangle\ v^{fr}\ u^b\ x)\end{aligned}$$

To give a proof sketch for this lemma, we split it into two parts. First, we argue that mapping bijective f^{lin} over the linearized variables and id over all others preserves non-repetitiveness. $\mathsf{eq\_shape}_F^{lin}$ is transitive (both $=$ and $\top$ are transitive) and x and $\mathsf{map}_F\ \langle \mathrm{id}^{l'}\ f^{lin} \rangle\ \mathrm{id}^{fr}\ \mathrm{id}^b\ x$ have the same shape. Thus, we have to think about the same $\forall y$ in the NONREP definition to show non-repetitiveness for the mapped x. This means we can fix the y and show the following, where we rename the f^{lin} in the existential quantifier to f^{lin}_{E1} and f^{lin}_{E2} to avoid naming clashes and for clarity.

$$\exists f^{lin}_{E1}.\ y = \mathsf{map}_F\ \langle \mathrm{id}^{l'}\ f^{lin}_{E1}\rangle\ \mathrm{id}^{fr}\ \mathrm{id}^{b}\ x \Longrightarrow$$
$$\exists f^{lin}_{E2}.\ y = \mathsf{map}_F\ \langle \mathrm{id}^{l'}\ f^{lin}_{E2}\rangle\ \mathrm{id}^{fr}\ \mathrm{id}^{b}\ (\mathsf{map}_F\ \langle \mathrm{id}^{l'}\ f^{lin}\rangle\ \mathrm{id}^{fr}\ \mathrm{id}^{b}\ x)$$

From this, we obtain and fix h^{lin} from the existential quantifier $\exists f^{lin}_{E1}$ in the assumption and instantiate the existential quantifier in the goal with $f^{lin}_{E2} = (h \circ (\mathrm{inv}\ f))^{lin}$. Since all f^{lin} are bijective, we know that the inverse inv for each of them exists. Using MAP_COMP, we can transform the instantiated goal to the assumption with the fixed h^{lin} as follows, which proves this part of the lemma:

$$\begin{aligned}
& y = \mathsf{map}_F\ \langle \mathrm{id}^{l'} (h \circ (\mathrm{inv}\ f))^{lin}\rangle\ \mathrm{id}^{fr}\ \mathrm{id}^{b}\ (\mathsf{map}_F\ \langle \mathrm{id}^{l'}\ f^{lin}\rangle\ \mathrm{id}^{fr}\ \mathrm{id}^{b}\ x) && \\
\equiv\ & y = \mathsf{map}_F\ \langle \mathrm{id}^{l'} ((h \circ (\mathrm{inv}\ f)) \circ f)^{lin}\rangle\ \mathrm{id}^{fr}\ \mathrm{id}^{b}\ x && \text{MAP\_COMP} \\
\equiv\ & y = \mathsf{map}_F\ \langle \mathrm{id}^{l'} (h \circ ((\mathrm{inv}\ f) \circ f))^{lin}\rangle\ \mathrm{id}^{fr}\ \mathrm{id}^{b}\ x && \circ\ \text{assoc} \\
\equiv\ & y = \mathsf{map}_F\ \langle \mathrm{id}^{l'} (h \circ \mathrm{id})^{lin}\rangle\ \mathrm{id}^{fr}\ \mathrm{id}^{b}\ x && \text{inv} \circ \\
\equiv\ & y = \mathsf{map}_F\ \langle \mathrm{id}^{l'}\ h^{lin}\rangle\ \mathrm{id}^{fr}\ \mathrm{id}^{b}\ x && \circ\ \text{id}
\end{aligned}$$

$$\begin{tikzcd}
\mathbf{x} \arrow[r, "\exists \mathbf{f^{lin}}"] \arrow[d, "\mathsf{map}_F\ \langle \mathbf{g^{l'}}\ \mathrm{id}^{\mathbf{lin}}\rangle\ \mathbf{v^{fr}}\ \mathbf{u^{b}}"'] & \forall \mathbf{y}.\ \mathsf{eq\_shape}^{lin}_F\ \mathbf{x}\ \mathbf{y} \arrow[d, dashed, "\mathsf{map}_F\ \langle g^{l'}\ \mathrm{id}^{lin}\rangle\ v^{fr}\ u^{b}"] \\
\mathbf{x'} \arrow[r, "\exists f^{lin}"] & \forall y'.\ \mathsf{eq\_shape}^{lin}_F\ x'\ y'
\end{tikzcd}$$

Figure 3.1: Illustration of lemma NONREP_MAP

It remains to show that mapping appropriately restricted functions over the other, not linearized type variables, also preserves non-repetitiveness. From the definition NONREP we know that for any element y of equivalent shape, we can find functions f^{lin} acting only on the linearized variables to map to this other element. Next, we consider the mapped $x' = \mathsf{map}_F\ \langle g^{l'}\ \mathrm{id}^{lin}\rangle\ v^{fr}\ u^{b}\ x$ and all y' of equivalent shape to it. To proceed we show, that for every fixed y' there exists a y of equivalent shape to the original x, such that y' can be expressed as $y' = \mathsf{map}_F\ \langle g^{l'}\ \mathrm{id}^{lin}\rangle\ v^{fr}\ u^{b}\ y$. Knowing this, we deduce that the same f^{lin} that maps x to y from the assumption also maps x' to y'.

Using MAP_COMP, this proves non-repetitiveness for x'. Figure 3.1 shows an illustration of this lemma. The thick or bolded elements are what we know from the assumption. The dashed arrow stands for the existence of a y with equivalent shape to x for each y' on which this proof depends. This existence can be shown using IN_REL and other theorems about the relator.

Mapper reflects non-repetitiveness

Lastly, we need the reflection of non-repetitiveness through the map function. This lemma can be seen as a partial reverse of NONREP_MAP, but only on the non-linearized lives.

(NONREP_MAP_REV) $$\mathsf{nonrep}_F^{lin}\ (\mathsf{map}_F\ \langle g^{l'}\ \mathrm{id}^{lin}\rangle\ \mathrm{id}^{fr}\ \mathrm{id}^{b}\ x) \Longrightarrow \mathsf{nonrep}_F^{lin}\ x$$

We start of the proof by defining $x' = \mathsf{map}_F\ \langle g^{l'}\ \mathrm{id}^{lin}\rangle\ \mathrm{id}^{fr}\ \mathrm{id}^{b}\ x$ and fixing a y with equivalent shape to x: $\mathsf{eq\_shape}_F^{lin}\ x\ y$. This time we can easily obtain and fix $y' = \mathsf{map}_F\ \langle g^{l'}\ \mathrm{id}^{lin}\rangle\ \mathrm{id}^{fr}\ \mathrm{id}^{b}\ y$ with $\mathsf{eq\_shape}_F^{lin}\ x'\ y'$ and $y' = \mathsf{map}_F\ \langle \mathrm{id}^{l'}\ f^{lin}\rangle\ \mathrm{id}^{fr}\ \mathrm{id}^{b}\ y$ from the assumption. We can express these maps in terms of the relator using graphs Grp of functions and *converse* graphs Grp^{-1}. For this, we use the following properties obtained from the MRBNF F:

(REL_MAP_1) $$\mathsf{rel}_F\ (Grp\ f)^{l}\ x\ y \equiv \mathsf{map}_F\ f^{l}\ \mathrm{id}^{fr}\ \mathrm{id}^{b}\ x = y$$

(REL_MAP_2) $$\mathsf{rel}_F\ (Grp^{-1}\ f)^{l}\ x\ y \equiv x = \mathsf{map}_F\ f^{l}\ \mathrm{id}^{fr}\ \mathrm{id}^{b}\ y$$

With this, we can express the current proof state as shown in Figure 3.2. The arrows denote a relation from the element at their base to the element at their tip through the given relator. The top relation is obtained from $\mathsf{eq\_shape}_F^{lin}\ x\ y$, the bottom arrow with f^{lin} from the non-repetitiveness in the assumption, while the vertical arrows denote the definitions of $'x$ and $'y$ in terms of the relator.

$$\begin{tikzcd}
x \arrow[r, "\mathsf{rel}_F\ \langle(=)^{l'}\ (\top)^{lin}\rangle"] \arrow[d, "\mathsf{rel}_F\ \langle(\mathrm{Grp}\ g)^{l'}\ (=)^{lin}\rangle"'] & y \\
x' \arrow[r, "\mathsf{rel}_F\ \langle(=)^{l'}\ (\mathrm{Grp}\ f)^{lin}\rangle"] & y' \arrow[u, "\mathsf{rel}_F\ \langle(\mathrm{Grp}^{-1}\ g)^{l'}\ (=)^{lin}\rangle"']
\end{tikzcd}$$

Figure 3.2: Illustration of lemma NONREP_MAP_REV

Following the arrows and applying relational composition with the associated axiom REL_COMPP we can express the relationship path between x and y through x' and y' as $\mathsf{rel}_F\ \langle(\text{Grp}\ g\ \bullet\ Grp^{-}1\ g)^{l'}\ (\text{Grp}\ f)^{lin}\rangle\ x\ y$. Together with the top relation from the definition of equivalent shape, we can apply the REL_EXCHANGE lemma to receive $\mathsf{rel}_F\ \langle(=)^{l'}\ (\text{Grp}\ f)^{lin}\rangle\ x\ y$. Unfolding the REL_MAP_1 equality, we can show the non-repetitiveness of x:

$$\begin{aligned}
&\mathsf{rel}_F\ \langle(\text{Grp}\ g\ \bullet\ Grp^{-}1\ g)^{l'}\ (\text{Grp}\ f)^{lin}\rangle\ x\ y \wedge \mathsf{rel}_F\ \langle(=)^{l'}\ (\top)^{lin}\rangle\ x\ y \\
\equiv\ &\mathsf{rel}_F\ \langle(=)^{l'}\ (\text{Grp}\ f)^{lin}\rangle\ x\ y && \text{REL\_EXCHANGE} \\
\equiv\ &y = \mathsf{map}_F\ \langle \text{id}^{l'}\ f^{lin}\rangle\ id^{fr}\ \text{id}^{b}\ x && \text{REL\_MAP\_1}
\end{aligned}$$

3.1.4 Defining the subtype and its constants

Using our definition of non-repetitiveness, we carve out a subtype of F using Isabelle's **typedef** command. This subtype F' contains exactly those elements from F that are non-repetitive on the linearized variables $\alpha_{l'+1} \ldots \alpha_{lin}$. It furthermore provides us with the morphisms $\mathsf{rep}_{F'}$ to convert F' elements to the type F and $\mathsf{abs}_{F'}$ to convert F elements to F' — provided that they are non-repetitive.

In the following, we specify the MRBNF constants, i.e., the mapper, setters, bound, and relator for F'. We define these in terms of the base type's constants and apply the morphisms to match the types: For the setters, $\mathsf{rep}_{F'}$ is applied to the argument before applying the appropriate setter of F to it. Since this does not change the set that is outputted, we can use the bound of F for F'. For the relator, the relations for the linearized lives are fixed to the equality relation, since in the new MRBNF these will be bounds. Lastly, for the mapper, we only allow it to map bijective functions on the linearized variables in addition to the restrictions for the existing frees and bounds. This restriction is necessary to ensure that applying the map function to an F' element preserves its non-repetitiveness. If a function that violates any of the restrictions is given to the mapper, it is ignored and not applied.

As for the morphisms, concretely, we apply $\mathsf{rep}_{F'}$ to the F' arguments of the new mapper, setters, and relator, and $\mathsf{abs}_{F'}$ to the result of the mapper. This leads us to the following definitions:

$$\begin{aligned}
\mathsf{bd}_{F'} &= \mathsf{bd}_F \\
\mathsf{set}_{F',i} &= \mathsf{set}_{F,i} \circ \mathsf{rep}_{F'} \\
\mathsf{map}_{F'}\ \langle f^{l'}\ g^{lin}\rangle\ u^{fr}\ v^{b} &= \mathsf{abs}_{F'} \circ (\mathsf{map}_F\ \langle f^{l'}\ (\text{asBij}\ g)^{lin}\rangle\ (\text{asSS}\ v)^{fr}\ (\text{asBij}\ (\text{asSS}\ u))^{b}) \circ \mathsf{rep}_{F'} \\
\mathsf{rel}_{F'}\ R^{l'}\ x\ y &= \mathsf{rel}_F\ \langle R^{l'}\ (=)^{lin}\rangle\ (\mathsf{rep}_{F'}\ x)\ (\mathsf{rep}_{F'}\ y)
\end{aligned}$$

where we enforce bijectivity of the g^{lin} and u^b using asBij $f = \texttt{if}$ bijective f $\texttt{then}$ f $\texttt{else}$ id. Analogously, both v^{fr} and u^b are enforced to be small-support functions using an analogously defined asSS.

3.1.5 Proving MRBNF axioms

To show that F' is an MRBNF, we have to prove the axioms from Figure 2.1 for it. For most of the axioms, this is straightforward, as they only require unfolding the definitions of the new F' constants, applying the axioms of the original F, and a few simple transformations. The axioms MAP_ID, MAP_CONG and SET_BD are proven this way, while MAP_COMP and SET_MAP require just a little more effort. Both contain the composition of $\mathsf{map}_{F'}$ or $\mathsf{set}_{F'}$ with $\mathsf{map}_{F'}$, respectively.

As an example, we show SET_MAP for F' below. Note that we assume i to be in the range $1 \leq i \leq \varpi$ where ϖ is the number of all non-dead type variables, i.e., $\varpi = l + fr + b$. The proof works the same for setters of frees and bounds. Furthermore, we assume all functions f^{ϖ} fulfilling their respective requirements (bijectivity and small-support) and thus all asBij and asSS evaluating to the then case.

$$\begin{aligned}
& \mathsf{set}_{F',i}\ (\mathsf{map}_{F'}\ f^{\varpi}\ x) && \\
\equiv\ & \mathsf{set}_{F,i} \circ \mathsf{rep}_{F'}\ ((\mathsf{abs}_{F'} \circ (\mathsf{map}_F\ f^{\varpi}) \circ \mathsf{rep}_{F'})\ x) && \text{unfold defs} \\
\equiv\ & \mathsf{set}_{F,i}\ (\mathsf{rep}_{F'}\ (\mathsf{abs}_{F'}\ (\mathsf{map}_F\ f^{\varpi}\ (\mathsf{rep}_{F'}\ x)))) && \circ \text{ application} \\
\equiv\ & \mathsf{nonrep}_F^{lin}\ (\mathsf{map}_F\ f^{\varpi}\ (\mathsf{rep}_{F'}\ x)) \Longrightarrow \mathsf{set}_{F,i}\ (\mathsf{map}_F\ f^{\varpi}\ (\mathsf{rep}_{F'}\ x)) && \mathsf{abs\ inverse} \\
\equiv\ & \mathsf{nonrep}_F^{lin}\ (\mathsf{rep}_{F'}\ x) \Longrightarrow \mathsf{set}_{F,i}\ (\mathsf{map}_F\ f^{\varpi}\ (\mathsf{rep}_{F'}\ x)) && \text{NONREP\_MAP} \\
\equiv\ & \mathsf{set}_{F,i}\ (\mathsf{map}_F\ f^{\varpi}\ (\mathsf{rep}_{F'}\ x)) && \mathsf{nonrep\ rep}_{F'} \\
\equiv\ & f_i\ `\ \mathsf{set}_{F,i}\ (\mathsf{rep}_{F'}\ x) && \text{SET\_MAP of } F \\
\equiv\ & f_i\ `\ \mathsf{set}_{F',i}\ x && \text{fold defs, } \circ
\end{aligned}$$

where "$\mathsf{abs\ inverse}$" denotes the theorem that $\mathsf{rep}_{F'}$ is the inverse of $\mathsf{abs}_{F'}$ for arguments that are non-repetitive. Furthermore "$\mathsf{nonrep\ rep}_{F'}$" states that converting a F' element to F inherently means that the F element is non-repetitive.

The validity of the bound BD is trivially proven, since the bound is copied from F.

It remains to show REL_COMPP and IN_REL for F'. While the former is easily proven using the corresponding axiom of F and some simple properties of relational composition, the latter is certainly the most interesting axiom to show.

We do not show a full proof of this property here, but investigate an interesting step. In the proof we reach a state, where we need to show that $\mathsf{nonrep}_F^{lin}\ (\mathsf{map}_F\ \mathrm{fst}^l\ \mathrm{id}^{fr}\ \mathrm{id}^b\ z) \Longrightarrow \mathsf{nonrep}_F^{lin}\ (\mathsf{map}_F\ \langle \mathrm{id}^{l'}\ \mathrm{fst}^{lin} \rangle\ \mathrm{id}^{fr}\ \mathrm{id}^b\ z)$. To give an intuition for why this is necessary, we obtain the left side of the implication from the IN_REL axiom of F and need to show the right side to eliminate a composition $\mathsf{abs}_{F'} \circ \mathsf{rep}_{F'}$ in the goal state.

The step is proven as follows:

$$\begin{aligned}
&\mathsf{nonrep}_F^{lin}\ (\mathsf{map}_F\ \mathrm{fst}^{l}\ \mathrm{id}^{fr}\ \mathrm{id}^{b}\ z) \Longrightarrow\\
&\mathsf{nonrep}_F^{lin}\ (\mathsf{map}_F\ \langle \mathrm{fst}^{l'}\ \mathrm{id}^{lin}\rangle\ \mathrm{id}^{fr}\ \mathrm{id}^{b}\ (\mathsf{map}_F\ \langle \mathrm{id}^{l'}\ \mathrm{fst}^{lin}\rangle\ \mathrm{id}^{fr}\ \mathrm{id}^{b}\ z)) \Longrightarrow\\
&\mathsf{nonrep}_F^{lin}\ (\mathsf{map}_F\ \langle \mathrm{id}^{l'}\ \mathrm{fst}^{lin}\rangle\ \mathrm{id}^{fr}\ \mathrm{id}^{b}\ z)
\end{aligned}$$

The first step is reached through MAP_COMP of F, while the second one needs the NONREP_MAP_REV lemma. This is the final place where strong pullback preservation is used, and the reason why it is required.

3.1.6 Lifting Witnesses

Existing witnesses of the original MRBNF that do not depend on any of the linearized variables can be lifted to be witnesses of the linearized MRBNF.

For this, it is necessary to show that they are non-repetitive on the linearized elements, i.e., that they are part of the new type. From WITS (Subsection 2.1.6) we know that any witness not depending on the linearized lives does not contain atoms from these lives. Thus, we can show that these witnesses are non-repetitive, since an element with no α atoms is trivially non-repetitive on α.

Other witnesses that depend on the linearized variables cannot be lifted and have to be discarded. Even if they are non-repetitive, witnesses of an MRBNF may only depend on lives and not on bounds, which the linearized lives turn into.

Additionally, new witnesses may be specified for the resulting MRBNF. For these, the property WITS defined in Subsection 2.1.6 has to be proven, i.e., that they only consist of the atoms given to them as arguments. Furthermore, it has to be shown that they are part of the type, i.e., that they are non-repetitive.

When a liftable witness of the original MRBNF exists or a new witness fulfilling WITS is specified, the existence of a non-repetitive element we motivated in Subsection 3.1.2 is trivially proven.

3.1.7 Preservation of strength

Linearizing an MRBNF preserves its strength property. This means, that for the new F' the following axiom holds, provided that F fulfills F_STRONG:

$$\mathsf{rel}_{F'}\ R^{l'}\ x\ y\ \wedge\ \mathsf{rel}_{F'}\ Q^{l'}\ x\ y \Longrightarrow \mathsf{rel}_{F'}\ (\mathrm{inf}\ \ R\ Q)^{l'}\ x\ ys$$

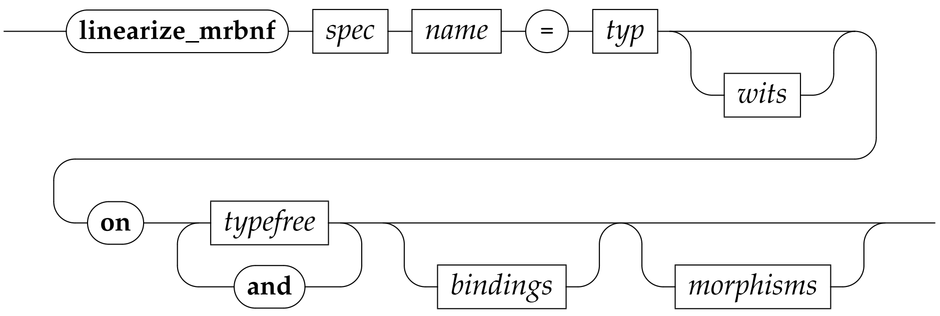
This is easily proven by unfolding the definition of $\mathsf{rel}_{F'}$, applying F_STRONG and unfolding (inf (=) (=)) $\equiv$ (=) for the linearized variables. Strength of an MRBNF is a property that often comes in useful. However, in Isabelle, it is not tracked in the MRBNF construct at the moment. At the very least, this proof allows us to easily linearize a linearized MRBNF again on further live variables.

3.2 Implementation of the linearize_mrbnf command

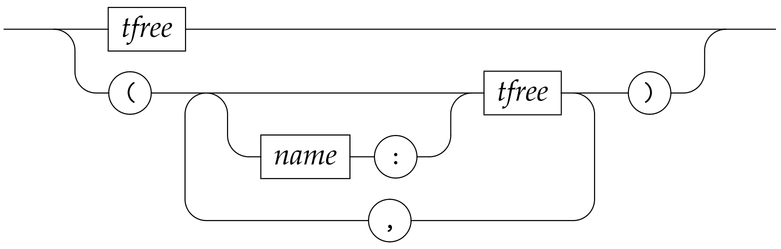
In this section, we implement a new **linearize_mrbnf** command that allows the user to linearize an existing MRBNF or BNF on one or multiple of its live variables. Our implementation is written as `ML` code.

3.2.1 Command syntax

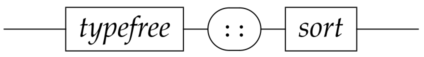
We give the syntax of the command in the following rail diagram:



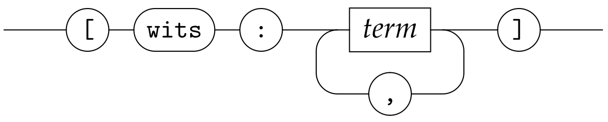
spec



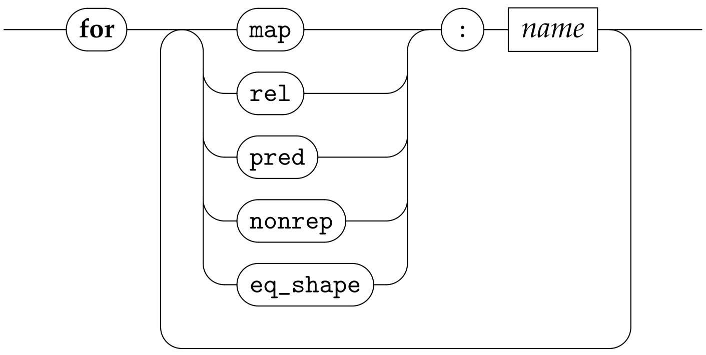
tfree



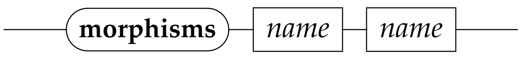
wits



bindings



morphisms



After the name of the command, the new MRBNF is specified by a *name* and a specification. The specification *spec* is a list of type variables (*typefree*s), where it is possible (but optional) to give a *name* for the setter associated with a certain *typefree*. We note here that it is necessary to annotate the *typefree*s corresponding to free and bound variables with the appropriate type class or *sort* as it is shown in the definition of *tfree*. This also applies to the linearized variables that turn into bound variables.

The specification is followed by the type (*typ* in the diagram) that is to be linearized. This type must a registered BNF, MRBNF or a construct made of (MR)BNFs, e.g., $(\alpha \times \beta)$ list. Furthermore, the type needs to contain the same type variables that are listed in the specification *spec*. However, the same type variable may occur in multiple positions. Type variables must be annotated with the respective type class as in *spec*.

Next, the user can optionally add witnesses *wits* in a list of correctly typed *term*s before specifying the type variables the type should be linearized on in a list separated by **and**s.

Lastly, custom names for the mapper, relator, and predicator can optionally be specified, as well as for the definitions of nonrep and eq_shape and the morphisms abs and rep. We did not explain the predicator in this thesis, as it is not relevant in this context. It is used to lift unary predicates on atoms of a type to a predicate on the type itself in a similar fashion to how the relator lifts binary predicates.

If no names are given, the default names are the base name of the constant (map, rel, pred, nonrep, eq_shape, Abs, Rel) suffixed with "_<F'>", where <F'> is replaced by the name of the linearized type. Setters without a specified name are named "set<i>_<F'>", where <i> is replaced with the 1-based index of the corresponding type variable in the specification *spec*.

By writing the following syntactically correct command in Isabelle, we can linearize our example of the list of pairs on the first type variable:

$$\textbf{linearize\_mrbnf}\ (\text{keys: } \alpha :: \text{var},\ \text{vals: } \beta)\ \mathsf{alist} = \text{"}(\alpha :: \text{var} \times \beta)\ \mathsf{list}\text{"}\ \textbf{on}\ \alpha$$

This generates the setters with the names "keys" and "vals" once the linearization is completed, while the other constants, definitions, and morphisms follow the default naming scheme, e.g., "map_alist" for the mapper.

3.2.2 Proof obligations

After the user has written the command in the syntax we introduced above, a number of goals to be proven or "proof obligations" are presented to them. These are the Conditions for linearization we presented in Subsection 3.1.2, i.e., the non-emptiness of the linear type, strong pullback preservation IN_REL_STRONG, and the witness axioms WITS if applicable.

These goals are internally constructed as terms and given dynamically to the user. For example, it is only necessary to show strong pullback preservation (i.e., show IN_REL_STRONG) when the resulting MRBNF has live variables remaining. We explained this in Subsections 3.1.2 and 3.1.3. Furthermore, as mentioned in Subsection 3.1.6, the non-emptiness of the non-repetitive type is easily proven when the user specified a non-emptiness witness, or a liftable witness of the original type exists. The user is only asked to prove the goals that are actually necessary for a specific linearization.

Furthermore, we simplified the goal that the user has to prove. Since the original MRBNF already fulfills weak pullback preservation, we extract the uniqueness property from strong pullback preservation. Strong pullback preservation IN_REL_STRONG can be proven from the IN_REL axiom together with the uniqueness property we specify as follows:

(PB_UNIQUE)
$$\begin{aligned}\forall x\ y.\ (&\mathsf{map}_F\ \text{fst}^l\ \text{id}^{fr}\ \text{id}^b\ x = \mathsf{map}_F\ \text{fst}^l\ \text{id}^{fr}\ \text{id}^b\ y \wedge \\ &\mathsf{map}_F\ \text{snd}^l\ \text{id}^{fr}\ \text{id}^b\ x = \mathsf{map}_F\ \text{snd}^l\ \text{id}^{fr}\ \text{id}^b\ y) \Longrightarrow \\ &x = y\end{aligned}$$

For $(\alpha \times \beta)$ list in our example, both type variables are live. Since we only linearize on α, it is necessary to prove the uniqueness of for preservation for this MRBNF, i.e., PB_UNIQUE. However, it is not necessary to prove the existence of a non-repetitive element, since the empty list $[\,]$ is a witness of the list type constructor that does not depend on any type variable.

3.2.3 Automatization of the proofs

When the user has proven all goals successfully (or, if there were no goals to be proven), the construction of the linearized MRBNF begins. The linear subtype is created through a `ML` version of the **typedef** command, where the non-emptiness of the resulting type is shown either through the proof of the user or using a witness — either a lifted one or one specified by the user (see Subsection 3.1.6).

Thereafter, the definitions of equivalent shape and non-repetitiveness, as well as the constants from Subsection 3.1.4, are constructed and added to the local context. These definitions and constants — as well as the theorems describing how they are defined — are visible to the user after the command completes.

Next, the premises and goals of the intermediate lemmas from Subsection 3.1.3 are constructed as terms and are consequently proven through `ML` tactics. These intermediate lemmas are not visible to the user afterwards.

As a last step, the linearized MRBNF is registered according to the given specifications and the defined constants using an `ML` version of the **mrbnf** command. We give a record of `ML` tactics to this command so that the MRBNF axioms can be shown.

The `ML` tactics automate the proofs we described in Subsections 3.1.3 and 3.1.5. To construct these tactics, we converted the existing high-level apply-style and Isar proofs to single-step apply proofs. These proofs avoid using the automatic proof tactics of Isabelle like `metis`, `auto`, `fastforce`, and even `simp`. Instead, they rely on explicit rule applications, substitutions, and deterministic repetitions. In certain cases, it is necessary to instantiate free variables in existing theorems and lemmas, for example, when an explicit term is introduced to replace an existential quantifier in the goal. This can be challenging in certain cases because the term has to be constructed in a type-correct manner.

3.2.4 Implementation challenges

In the theoretical part of this thesis, we always assumed that the type variables are ordered according to their variable type (lives, frees, bounds, deads). However, for our implementation, we have to consider an arbitrary ordering. Furthermore, the subset of lives that are linearized can be an arbitrary one, i.e., they are not necessarily the last *lin* of the lives. For constructing lemmas and `ML` tactics, it is easier to think of the variables grouped by their variable type, e.g., when defining the map-function, the functions for the frees need to be wrapped by the asSS function. This is made even more complicated when considering that the order of type variables in the goal type (defined by the specification *spec*) does not necessarily agree with the ordering in the base type. Thus, careful interlacing is required at every step.

Another challenge is related to the possible complexity of the base type. For the standard substitution tactic in Isabelle, it is required to specify the number of occurrences to be substituted. However, for a more complex base type, the sub-term that we want to substitute might occur more often than expected for a simple type. For example, the sub-term $\mathtt{rep}_{F'} \circ \mathtt{abs}_{F'}$ can occur more often than expected if the base type has a complex map function. This could be solved by repeatedly substituting a single occurrence. However, this is inconvenient since it generates multiple versions of the same subgoal, and we want to avoid unbounded repetitions. Thus, we employ a custom tactic that replaces all occurrences of a subgoal. Internally, it collects a list of all occurrences of the sub-term, applies a single substitution for each of them, and then removes duplicate subgoals. This allows us to conveniently substitute all occurrences of a sub-term without knowing its exact count.

4 Example application: POPLmark Challenge - Pattern

The POPLmark Challenge [7] presents a selection of problems to benchmark the progress in formalizing programming language metatheory. The challenges are built around formalizing aspects of *System* $F_{<:}$ calculus, a polymorphic typed lambda calculus with subtyping. We are interested in part 2B of this Challenge, which has the goal to formalize and prove *type soundness* for terms with pattern matching over records. Type soundness is considered in terms of *preservation* (evaluating a term preserves its type) and *progress* (a term is either a value or can be evaluated).

The challenge defines the syntax of terms $\texttt{t}$ and patterns $\texttt{p}$ as follows:

(*variable*)	$\texttt{t} ::= \texttt{x}$
(*abstraction*)	$\mid \lambda\texttt{x} : \texttt{T}.\, \texttt{t}$
(*application*)	$\mid \texttt{t}_1\ \texttt{t}_2$
(*type abstraction*)	$\mid \lambda\texttt{X} <: \texttt{T}.\, \texttt{t}$
(*type application*)	$\mid \texttt{t}\ [\texttt{T}]$
(*record*)	$\mid \{j \in 1 \ldots n :\ \texttt{l}_j = \texttt{t}_j\}$
(*projection*)	$\mid \texttt{t.l}$
(*pattern-let*)	$\mid \texttt{let}\ \texttt{p} = \texttt{t}_1\ \texttt{in}\ \texttt{t}_2$

(*variable pattern*)	$\texttt{p} ::= \texttt{x} : \texttt{T}$
(*record pattern*)	$\mid \{j \in 1 \ldots n :\ \texttt{l}_j = \texttt{p}_j\}$

In this syntax, types `T` are defined similarly. We omit this here as it is not important for this example. Furthermore, `x` stands for variables, `X` for type variables, and `l` for labels.

We focus on the *record* and *pattern-let* terms together with the *variable* and *record patterns*. A record is a term defined as a finite set of assignments of n terms to one label each. The labels l within a record must be pairwise distinct. A pattern is defined as either a typed variable or a finite set of n assignments of patterns to labels. Again, the labels must be pairwise distinct.

Van Brügge et al. present a formalization of part 2B of the POPLmark Challenge in Isabelle/HOL in their ITP '25 contribution [3]. Datatypes with bindings play an important role in their formalization. They represent types `T` as α typ and terms `t` as (α, β) trm. These two are defined with the **binder_datatype** command, where α is the type representing type variables `X` and β represents variable names `x`. Labels are implemented as strings. However, any infinite type could be used in their place.

A central notion in this formalization is the *labeled finite set* (α, β) lfset that is used in the representation of records and record patterns. This type constructor is a subtype of $(\alpha \times \beta)$ fset that only includes elements that are non-repetitive on α. This restriction is necessary because for both records and patterns the label α must be mutually distinct, i.e., the set representing them has to be non-repetitive.

While by construction $(\alpha \times \beta)$ fset is a BNF (and an MRBNF since all BNFs are also MRBNFs) with both variables being live, (α, β) lfset is a MRBNFs with α as a bound variable, since it is non-repetitive on α. While this is a linearization, the finite set of pairs does not fulfill strong pullback preservation. Thus, the approach and command we presented in Chapter 3 cannot be used here. Because of an alternate, equivalent description of non-repetitiveness specific to this type, it is still possible to manually linearize this MRBNF. The linearized lfset is used in the **binder_datatype** definition of term trm as one the alternative representing records:

| Rec "(string, (α, β) trm) lfset"

For the pattern, a new type is used. It is constructed by linearizing an intermediate datatype prepat that is defined using the **datatype** command:

datatype (α, β) prepat = PPVar "β" "α typ" | PPRec "(string, (α, β) prepat) lfset"

The prepat datatype is an MRBNF with α as a free and β as a live variable. The binder-datatype typ is a unary MRBNF with its only type variable being free, prepat inherits this variable type for α.

This datatype can represent all possible patterns just fine. However, it also represents many patterns that are not allowed. In the definition of Challenge 2B, it is stated that "[...] the variable patterns appearing in a pattern are assumed to bind pairwise distinct variables" [7, §3]. Thus, we linearize (α, β) prepat on the type variables for β

This MRBNF can be linearized using the **linearize_mrbnf** command from Chapter 3 as follows:

linearize_mrbnf (PTVars: α :: var, PVars: β :: var) pat = "(α :: var, β :: var) prepat"
  [wits: "PPRec lfempty :: (α :: var, β :: var) prepat"] **on** β

We specified the non-emptiness witness "PPRec lfempty" that is constructed by applying the PPRec constructor of prepat to the empty lfset lfempty. This witness is independent of any type variable and thus it trivially fulfills the witness axiom WITS.

The command generates the conditions that need to be proven. Since β is the only live of prepat, strong pullback preservation IN_REL_STRONG (or PB_UNIQUE) does not have to be proven as explained in Subsection 3.1.3. Furthermore, since we specified a non-emptiness witness, we do not have to prove that a non-repetitive element of the type exists, but show that the witness "PPRec lfempty" is non-repetitive instead. This is easily shown with the MRBNF axioms after noticing that "PPRec lfempty" is only equivalent in shape to itself.

The linearized MRBNF is then used for the definition of trm in the alternative that represents pattern-lets:

```
| Let "(α, p::β) pat" "(α, β) trm" t::"(α, β) trm" binds p in t
```

We note here, that van Brügge et al. already used the (α, β) pat type in this manner for their ITP '25 contribution [3], however, they had to manually linearize it from prepat, i.e., define the subtype, its constants and prove the MRBNF axioms. We simplified this process using our new **linearize_mrbnf** command.

5 Conclusions and future Work

In this thesis, we introduced theoretical steps for *linearizing* Map-Restricted Bounded Natural Functors (MRBNFs) in Isabelle/HOL. Based on the existing theory of BNF and MRBNF, we formalized the notion of linearization as defining the subtype of a MRBNF that contains only *non-repetitive* elements, i.e., elements without repeating atoms. Furthermore, we proved that the resulting type again satisfies the MRBNF axioms. We implemented this process in the new **linearize_mrbnf** command, which automates the creation of linear types and the associated proofs. Furthermore, we demonstrated an application of the command to the POPLmark Challenge. Thereby, we showed how the command simplifies the construction of a linear type and how it integrates with the **binder_datatype** package.

Future work
We want to motivate two additions to the datatype and binder datatype packages that will streamline the integration of our new **linearize_mrbnf** command in the definition of new binding-aware datatypes.

For our first proposal, we consider the preliminary, non-linearized type (α, β) prepat from Chapter 4 again. While we talk about this type only in terms of an MRBNF, it is not registered as one right away in the current state of the Isabelle/HOL packages. We define prepat with the **datatype** command that only considers BNFs. As we stated in Chapter 4, α typ is an MRBNF with α as a free variable. However, in the view of BNFs, free and bound variables are considered dead. Thus, α is dead for (α, β) prepat as well, since it uses α typ in its definition. Thus, for now it is necessary to register (α, β) prepat with appropriately defined map and set functions as an MRBNF. This requires calling the **mrbnf** command and proving the MRBNF axioms manually.

We propose an integration of MRBNFs into the **datatype** command to allow users to recursively build complex MRBNFs from simpler ones without killing free and bound variables in the process. This would solve the issue mentioned above, since prepat could be constructed from typ without having to manually prove the MRBNF axioms for it. Alternatively, we can think of a modification to the **binder_datatype** command that already works on MRBNFs. The modified command would allow the construction of MRBNFs without the need to introduce a new binding in the process.

Secondly, we propose the implementation of a way to mark (MR)BNFs as *strong* (MR)BNFs when they fulfill strong pullback preservation, i.e., the IN_REL_STRONG property defined in Subsection 3.1.2. This would remove the necessity to prove the strength of a strong (MR)BNF when it is linearized on a proper subset of its live variables.

Furthermore, if the strength of BNFs is closed under composition and fixpoints, complex strong BNFs can be constructed using the **datatype** command from simpler strong BNFs. A strong BNF constructed this, was could be linearized without the need to manually prove its strength. Although we have not proven this closure, it appears plausible based on our intuition. However, we have shown in Subsection 3.1.7 that strength is closed under the linearization of MRBNFs and thus a linearized MRBNF can be linearized again — provided it still has live variables to be linearized.

Abbreviations

BNF	Bounded Natural Functor
MRBNF	Map-Restricted Bounded Natural Functor

List of Figures

Bibliography

[1] D. Traytel, A. Popescu, and J. C. Blanchette, "Foundational, compositional (co)datatypes for higher-order logic: Category theory applied to theorem proving," in *2012 27th Annual IEEE Symposium on Logic in Computer Science*, 2012, pp. 596–605. DOI: `10.1109/LICS.2012.75`.

[2] J. C. Blanchette, J. Hölzl, A. Lochbihler, L. Panny, A. Popescu, and D. Traytel, "Truly modular (co)datatypes for isabelle/hol," in *Interactive Theorem Proving*, G. Klein and R. Gamboa, Eds., Cham: Springer International Publishing, 2014, pp. 93–110, ISBN: 978-3-319-08970-6.

[3] J. van Brügge, A. Popescu, and D. Traytel, "Animating MRBNFs: Truly Modular Binding-Aware Datatypes in Isabelle/HOL," in *16th International Conference on Interactive Theorem Proving (ITP 2025)*, Y. Forster and C. Keller, Eds., ser. Leibniz International Proceedings in Informatics (LIPIcs), vol. 352, Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2025, 11:1–11:20, ISBN: 978-3-95977-396-6. DOI: `10.4230/LIPIcs.ITP.2025.11`.

[4] J. C. Blanchette, L. Gheri, A. Popescu, and D. Traytel, "Bindings as bounded natural functors," *Proc. ACM Program. Lang.*, vol. 3, no. POPL, Jan. 2019. DOI: `10.1145/3290335`.

[5] J. C. Blanchette, A. Popescu, and D. Traytel, "Cardinals in isabelle/hol," in *Interactive Theorem Proving*, G. Klein and R. Gamboa, Eds., Cham: Springer International Publishing, 2014, pp. 111–127, ISBN: 978-3-319-08970-6.

[6] J. C. Blanchette, A. Popescu, and D. Traytel, "Witnessing (co)datatypes," in *Programming Languages and Systems*, J. Vitek, Ed., Berlin, Heidelberg: Springer Berlin Heidelberg, 2015, pp. 359–382, ISBN: 978-3-662-46669-8.

[7] B. E. Aydemir, A. Bohannon, M. Fairbairn, J. N. Foster, B. C. Pierce, P. Sewell, D. Vytiniotis, G. Washburn, S. Weirich, and S. Zdancewic, "Mechanized metatheory for the masses: The poplmark challenge," in *Theorem Proving in Higher Order Logics*, J. Hurd and T. Melham, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 50–65, ISBN: 978-3-540-31820-0.