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

Acknowledgments

Abstract

Contents

1 Introduction

- Datatypes in general
  - Datatypes in Isabelle/HOL are built on Bounded Natural Functors (BNFs) (defined in [TPB12])
  - Structure of the Thesis

2 Background

This Chapter serves to introduce BNFs and their generalization to Map-Restricted Bounded Natural Functors (MRBNFs).

2.1 Bounded Natural Functors (BNFs)

As described in Chapter 1, BNFs are essential for constructing datatypes and co-datatypes in Isabelle/HOL. Especially for defining a datatype with recursion it is required that the type constructor $'a$ list is registered as a BNF, i.e., it fulfills the BNF-axioms. For example the following **datatype** command only succeeds if $'a$ list, is a BNF.

datatype $'a$ ex = A "($'a$ × $'a$ ex) list"

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

The type variables of a BNF are divided into two groups: *live* and *dead* variables or *lives* and *deads*. Live variables can be used for recursive datatype definitions, while dead ones do not allow for this. We take the function type $'a \Rightarrow 'b$ as an example. It's first type argument $'a$ is dead, while the second one $'b$ is live. Thus, of the following the first command succeeds while the second one fails

datatype $'a$ success = S1 | S2 "$'a \Rightarrow 'a$ success"
datatype $'a$ fail = F1 | F2 "$'a$ fail $\Rightarrow 'a$"

2.1.1 BNF constants

A BNF F with l live variables is characterized by one map and l set functions, a bound and a relator.

Map function and functors

The $l+1$-ary 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 recursively applied to the components 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 $'a$ list type as an example, a BNF with one live $'a$, the mapper has the type $\mathsf{map}_{\mathsf{list}} :: ('a \Rightarrow 'a_1) \Rightarrow 'a\ \mathsf{list} \Rightarrow 'a_1\ \mathsf{list}$.

To make F with its mapper a *functor* on the universe of all types, the mapper has to fulfill two axioms [TPB12]. First, mapping the *id* function on all lives over an element should leave it unchanged. This axiom is named MAP_ID. Secondly, mapping two lists of functions over an element, e.g., first $f_1 \ldots f_l$ and then $g_1 \ldots g_l$, should produce the same result as mapping the index-wise composition $(g_1 \circ f_1) \ldots (g_l \circ f_l)$ over it once. We name this axiom MAP_COMP and fully formalize it and all the other axioms in Figure 2.2.

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 components 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.

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. It states that taking the i-th set of an F after mapping $f_1 \ldots f_l$ to it, results in the same set as if i-th set was taken from the original F before the image of f_i was applied to it. Again, the formalization of SET_MAP can be found in Figure 2.2, while Figure 2.1 shows a visualization of it.

$$\begin{tikzcd}
('a_1,\ \ldots\ 'a_l)\ F \arrow[r, "\mathsf{set}_{F,i}"] \arrow[d, "\mathsf{map}_F\ f_1\ \ldots\ f_l"'] & 'a_i\ \mathsf{set} \arrow[d, "image\ f_i"] \\
('b_1,\ \ldots\ 'b_l)\ F \arrow[r, "\mathsf{set}_{F,i}"] & 'b_i\ \mathsf{set}
\end{tikzcd}$$

for all i where $'a_i$ is a live variable of F

Figure 2.1: $\mathsf{set}_{F,i}$ as a natural transformation

Another axiom of the setters and the mapper is the congruency MAP_CONG of the map function. It states that if two (lists of) functions $f_1 \ldots f_l$ and $g_1 \ldots g_l$ are equal when applied to the corresponding sets of all components of an F (obtained through the setters), then mapping these two lists of functions over the F each produces the same result.

Bound and boundedness

Lastly, the BNF needs a bound. This is an infinite cardinal that may depend on the cardinalities of the dead variables, but not on the of the live variables. The bound is used for the SET_BD axiom (formalized in Figure 2.2) to ensure that the component sets obtained by the setters are bounded. This ensures that the branching of a recursively defined datatype is bounded.

Relator and shapes

The relator is used to build a relation on F by relating the components 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 type as follows:

$$\begin{aligned}\mathsf{rel_{prod}} :: ('a_1 \Rightarrow 'a_2 \Rightarrow \mathsf{bool}) \Rightarrow ('b_1 \Rightarrow 'b_2 \Rightarrow \mathsf{bool}) \Rightarrow \\ ('a_1 \times 'b_1) \Rightarrow ('a_2 \times 'b_2) \Rightarrow \mathsf{bool} \\ \mathsf{rel_{prod}}\ R\ S\ p_1\ p_2 := R\ (fst\ p_1)\ (fst\ p_2) \wedge S\ (snd\ p_1)\ (snd\ p_2)\end{aligned}$$

Considering the list type again, we make an interesting observation: There are some $'a$ lists xs and ys that the relator cannot relate, no matter which $'a$ relation is chosen. The relator on lists is index-wise defined, i.e., the $'a$ relation must relate the elements of both lists for each index. Consequently lists of different length cannot be positively related. We think of the length of a list as its *shape*.

2.1.2 BNF-axioms

A BNF is characterized by map and set functions, a relator and a bound. We consider a BNF F with l live variables and use the notation $\overline{f} = f_1 \dots f_l$. Furthermore, when we write i as an index, we assume it to be in the range $1 \leq i \leq l$. We give the BNF-axioms as follows:

While most of these properties are straightforward, we want to explain the preservation of weak pullbacks in more detail.

$$\mathsf{rel}_F\ \overline{R}\ x\ y = \exists z.\ (\forall i.\ \mathsf{set}_{F,i}\ z \subseteq \{(a,b).\ R_i\ a\ b\}) \wedge \mathsf{map}_F\ \overline{fst}\ z = x \wedge \mathsf{map}_F\ \overline{snd}\ z = y \quad (2.9)$$

The idea is that two elements x and y of the type $'a\ F$ are related through a relation R iff there exists a z that acts as a "zipped" version of x and y. The components of this z are R_i-related pairs of the components of x and y, where the first position in the pair corresponds to x and the second one to y.

$$\textsc{map\_id}\text{: } \mathsf{map}_F\ \overline{id}\ x = x \tag{2.1}$$

$$\textsc{map\_comp}\text{: } \mathsf{map}_F\ \overline{g}\ (\mathsf{map}_F\ \overline{f}\ x) = \mathsf{map}_F\ \overline{(g \circ f)}\ x \tag{2.2}$$

$$\textsc{map\_cong}\text{: } (\forall i.\ \forall z \in \mathsf{set}_{F,i}\ x.\ f_i\ z = g_i\ z) \Longrightarrow \mathsf{map}_F\ \overline{f}\ x = \mathsf{map}_F\ \overline{g}\ x \tag{2.3}$$

$$\textsc{set\_map}\text{: } \forall i.\ \mathsf{set}_{F,i}(\mathsf{map}_F\ \overline{f}\ x) = f_i\ `\ \mathsf{set}_{F,i}\ x \tag{2.4}$$

$$\textsc{bd}\text{: } \mathsf{infinite}\ \mathsf{bd}_F \wedge \mathsf{regular}\ \mathsf{bd}_F \wedge \mathsf{cardinal\_order}\ \mathsf{bd}_F \tag{2.5}$$

$$\textsc{set\_bd}\text{: } \forall i.\ |\mathsf{set}_{F,i}\ x| <_o \mathsf{bd}_F \tag{2.6}$$

$$\textsc{rel\_compp\_leq}\text{: } \mathsf{rel}_F\ \overline{R}\ \bullet\ \mathsf{rel}_F\ \overline{Q}\ = \mathsf{rel}_F\ \overline{(R\ \bullet\ Q)} \tag{2.7}$$

$$\textsc{in\_rel}\text{: } \text{Weak Pullback Preservation WP} \tag{2.8}$$

where ` is the image function on sets, • is the composition of relations and $<_o$ is the less than relation on cardinals

Figure 2.2: The BNF axioms

2.1.3 Witnesses

2.1.4 BNF examples

Further examples of BNFs are is the product type $('a,\ 'b)$ prod, a binary type constructor with infix notation $'a\ \times\ 'b$, and the type of finite sets $'a$ fset. The latter 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, $'a$ fset can be shown to be a BNF.

To show, how BNFs can be combined to create new ones, we consider the type constructor $('a,\ 'b)$ plist = $('a\ \times\ 'b)$ 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 $\mathsf{rel}_{\mathsf{plist}}\ R\ S$. The exact definitions are given as such:

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

where we use the standard map, set and relator functions of the list and product type.

To show that $('a, 'b)$ plist is a BNF, we have to prove the BNF-axioms for it. Besides the definitions above, a bound $\mathsf{bd}_{\mathsf{plist}}$ is needed. We chose *natLeq*.

2.2 Subtype

We can carve out a subtype from a type constructor using the **typedef** command.

2.3 Map-Restricted Bounded Natural Functors (MRBNFs)

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. We call type variables that 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.

Consequently, for type constructors with variables that are considered dead in BNF terms, we can declare some of them as free or bound variables, depending on the type. Consider for example the type of distinct lists $'a$ dlist, a subtype of $'a$ list that describes only lists containing pairwise distinct $'a$ elements. The issue with this type is that the standard map function on lists cannot guarantee that the resulting list is still distinct, i.e., that it is still part of the type. Thus in BNF terms the type variable of $'a$ dlist is dead. However, by restricting the mapper to only use bijections, the distinctness of the resulting list can be ensured. Thus, it can be shown that $'a$ dlist is a MRBNF with $'a$ as a bound variable.

MRBNFs can be used in a **binder_datatype** command to produce a datatype with bindings.

- cite [Bla+19]

3 Linearizing MRBNFs

3.1 Linearization of MRBNFs (In theory)

In this section we define the linearization of a MRBNF on a subset of it's *live* variables. The result of the linearization is a new MRBNF with the same variable types (*live*, *dead*, *bound*, *free*), except for the linearized variables that change their type from *live* to *bound*. This means that the map function is now restricted to only allow bijective and small-support functions on these variables. Apart from this change, it is ensured that the MRBNF is *non-repetitive* with respect to the linearized variables. We give a definition non-repetitiveness in the following Subsection 3.1.1. Intuitively it means that the atoms of that type cannot occur multiple times in an element of the type.

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, a 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 have to 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 cannnot exist, since it would have to map two same elements in xs to two differing ones in ys.

Recalling our model of BNFs as containers that have elements of live variables' types in certain positions, we can generalize the notion of lists having the same length to two elements of a type being of equal or equivalent shape. We can express this through the relator by using the top relation that relates everything with each other as the argument. Thus, we give the defninition of equivalent shape and non-repetitiveness for list:

$$\mathsf{eq\_shape}_{\mathsf{list}}\ x\ y = \mathsf{rel}_{\mathsf{list}}\ top\ x\ y \tag{3.1}$$

$$\mathsf{nonrep}_{\mathsf{list}}\ x = \forall y.\ \mathsf{eq\_shape}_{\mathsf{list}}\ x\ y \Longrightarrow (\exists f.\ y = \mathsf{map}_{\mathsf{list}}\ f\ x) \tag{3.2}$$

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.

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, one can easily define a function mapping the distinct elements of x to that list). 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.

For MRBNFs with more than one live variable, we can give a definitions of *non-repetitiveness* and having *equal shape* on a subset of the live variables. In that case, we consider x and y of type $(\alpha, \beta)\ G$ to have equal shape with respect to α, iff they are *equal* in their β atoms and are related with *top* in α as before. Consequently for the map in the nonrep definition, the *id* function is applied to the β atoms, since they are already required to be equal.

$$\mathsf{eq\_shape}^1_G\ x\ y = \mathsf{rel}_G\ top\ (=)\ x\ y \tag{3.3}$$

$$\mathsf{nonrep}^1_G\ x = \forall y.\ \mathsf{eq\_shape}^1_G\ x\ y \Longrightarrow (\exists f.\ y = \mathsf{map}_G\ f\ id\ x) \tag{3.4}$$

3.1.2 Conditions for linearization

A MRBNFs has to fulfill two properties to be linearized. 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): $\exists x.\ \mathsf{nonrep}\ x$

Furthermore, even though MRBNFs are already required to preserve weak pullbacks as defined in Equation 2.9, for the linearization it is required that they preserve *all* pullbacks. Formalized this means that the existance of z in the equation has to be fulfilled uniquely, i.e., for each R-related x and y there existes *exactly one* z fulfilling the property in Equation 2.9. For example the strong pullback preservation is fulfilled by the α list and $\alpha\ \beta$ prod functors but not by α fset, the type constructor for finite sets of αs.

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 *relation 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 some intermediate lemmas which we present in this section.

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

$$[\![\mathsf{rel}_F\ R\ x\ y;\ \mathsf{rel}_F\ Q\ x\ y]\!] \Longrightarrow \mathsf{rel}_F\ (inf\ R\ Q)\ x\ y$$

where the infimum inf of two relations R and Q relates exactly those elements that are related by both R and Q.

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 $\overline{R} = R_1 \ldots R_n$ and $\overline{Q} = Q_1 \ldots Q_n$ of atom-level relations, then x and y are also related with any index-wise combination of $\overline{R}$ or $\overline{Q}$. For each index $1 \leq i \leq n$ 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 $\overline{R}$ is chosen and for all others the Q relation. As an example, this results in the following lemma for $(\alpha, \beta)\ G$ from Subsection 3.1.1:

$$[\![\mathsf{rel}_G\ R_1\ R_2\ x\ y;\ \mathsf{rel}_G\ Q_1\ Q_2\ x\ y]\!] \Longrightarrow \mathsf{rel}_G\ R_1\ Q_2\ x\ y$$

In the specific case, that the MRBNF is linearized on *all* of it's live variables, the goal of the lemma is equal to it's first assumption. Thus, the lemma becomes trivial since exactly the list of relations $\overline{R}$ is chosen.

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.

map peresrving non-repetitiveness

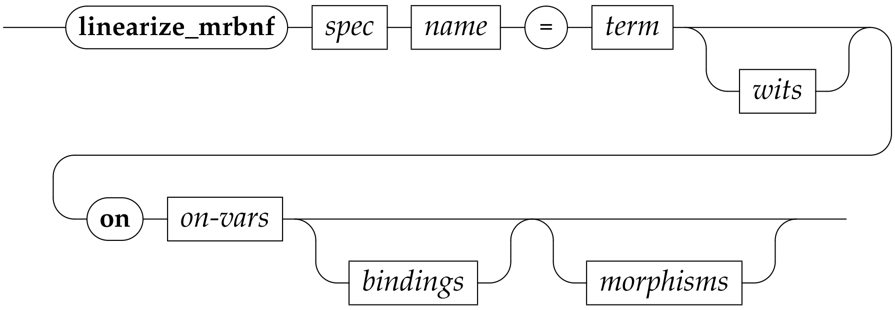
$$[\![\mathsf{nonrep}^1_G\ x;\ \mathsf{bijective} f]\!] \Longrightarrow \mathsf{nonrep}^1_G\ (\mathsf{map}_G\ f\ g\ x)$$

3.1.4 Proving the MRBNF axioms

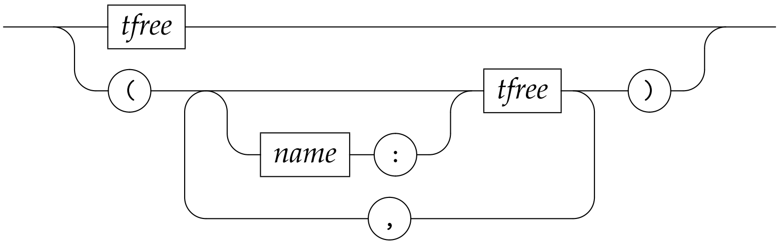
3.1.5 Non-emptines Witnesses

3.2 Linearization of MRBNFs (In Isabelle)

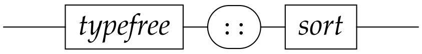
We implement a command that allows the user to linearize an existing MRBNF or BNF on one or multiple of it's live variables. The syntax of the command is given in the following:



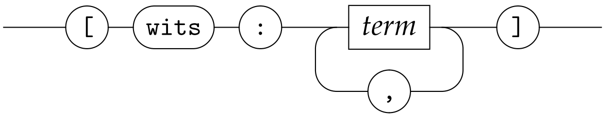
spec



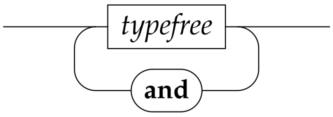
tfree



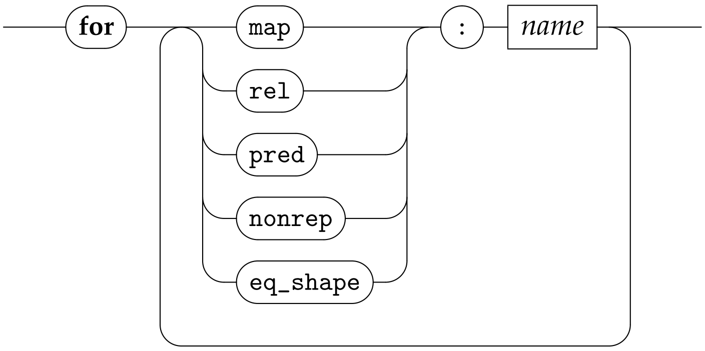
wits



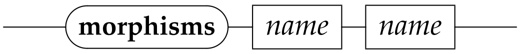
on-vars



bindings



morphisms



With this command, we can linearize our example by writing the following line in Isabelle:

linearize_mrbnf (keys: $'k$ :: var, vals: $'v$) alist = ($'k$ :: var × $'v$) list **on** $'a$

Since for ($'k$ × $'v$) list both type variables are live and we only linearize on $'k$, it is necessary to prove strong pullback preservation for this MRBNF.

After the user has written the command, the conditions for linearization we presented in Subsection 3.1.2 have to be proven, i.e., non-emptines of the linear type and strong pullback preservation.

These conditions are given dynamically to the user. For example, strong pullback preservation only has to be shown, when the resulting MRBNF has live variables remaining. Furthermore, the non-emptines of the non-repetitive type is easily proven when the user specified a non-emptines witness, or a preservable witness of the original type exists. Thus, the user is not asked to show the existance of a non-repetitive element in these cases.

4 Examples

4.1 POPLmark challenge: Pattern

The POPLmark challenge [Ayd+05] 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 proof *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).

We focus on the `record` terms `pattern-let`. A record is a term defined as a set of pairs, where the first element is a label and the second element a term: $\{(\mathtt{l}_j, \mathtt{t}_j)\}$. The labels l within a record must be pairwise distinct. A pattern is defined as either a typed variable or a set of (label, patten) pairs with pairwise distinct labels: $\mathtt{p} ::= \mathtt{x} : \mathtt{T} \mid \{(\mathtt{l}_j, \mathtt{p}_j)\}$

A formalization of part 2B of the POPLmark challenge in Isabelle/HOL is presented by Blanchette et al. [Bla+19]. They use *binder_datatypes* to abstract types, variables and terms. A central notion in this formalization is the *labeled finite set* $('a, 'b)$ lfset that is used in the representation of records and patterns. This type constructor is a subtype of $('a \times 'b)$ fset that only includes elements that are non-repetitive on $'a$. This restriction is necessary, because for both records and patterns the label $'a$ must be mutually distinct, i.e., the set representing them has to be non-repetitive.

While by construction $('a \times 'b)$ fset is a BNF (and an MRBNF since all BNFs are also MRBNFs) with both variables being live, $('a, 'b)$ lfset is a MRBNFs with $'a$ as a bound variable, since it is non-repetitive on $'a$. While this is a linearization, the finite set on 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 on non-repetitiveness specific to this type, it is still possible to manually linearize this MRBNF.

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

```
datatype ('v, 'tv) prepat = PPVar "'v" "'tv typ" | PPRec "(string, ('v, 'tv) prepat) lfset"
```

Abbreviations

BNF Bounded Natural Functor

MRBNF Map-Restricted Bounded Natural Functor

List of Figures

List of Tables

Bibliography

[Ayd+05] 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: *International Conference on Theorem Proving in Higher Order Logics*. Springer. 2005, pp. 50–65.

[Bla+19] J. C. Blanchette, L. Gheri, A. Popescu, and D. Traytel. “Bindings as bounded natural functors.” In: *Proceedings of the ACM on Programming Languages* 3.POPL (2019), pp. 1–34.

[TPB12] 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*. IEEE. 2012, pp. 596–605.