A Tale of Two Multiset-like Types in Isabelle/HOL

Mathias Schack Rabing
University of Copenhagen, Department of Computer Science

Dmitriy Traytel
University of Copenhagen, Department of Computer Science

Abstract

Finite multisets (also known as bags) are a fundamental data structure that generalizes finite sets to record the elements' multiplicities. In the Isabelle proof assistant, finite multisets are defined as the subtype of functions from elements to natural numbers consisting of functions that return non-zero multiplicities for finitely many elements. This representation is negative: the elements occur to the left of the function arrow. To allow (co)datatypes to (co)recurse through multisets, an alternative positive representation as quotients of finite lists of elements modulo permutations is used. Using Isabelle, we define two orthogonal generalizations of multisets and the respective positive and negative representations. First, we view *countable multisets* either as functions from elements to extended natural numbers that return non-zero multiplicities for countably many elements or, alternatively, as quotients of lazy lists modulo infinitary permutations. Second, we view *weighted sets* either as functions from elements to optional weights from a well-behaved algebraic structure that return None for all but finitely many elements or, alternatively, as quotients of element-weight lists modulo permutation and regrouping by element. For both types, we establish the necessary functorial structure to support (co)recursion and exemplify this support in two case studies.

2012 ACM Subject Classification Security and privacy → Logic and verification

Keywords and phrases multiset, bounded natural functor, (co)induction, (co)recursion, Isabelle/HOL

Digital Object Identifier 10.4230/LIPIcs.Isabelle.2026.1

1 Introduction

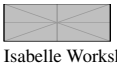
Multisets (also known as bags), which generalize sets to objects recording every element's multiplicity, occur pervasively in mathematics and computer science. Their applications range from counting polynomial roots and prime factors [7] to termination proving [11], automated reasoning [2], database query semantics [13], and text and image processing [24, 26]. All these applications focus on *finite* multisets, in which finitely many elements are assigned a finite (natural-numbered) multiplicity. For example, the well-foundedness of Dershowitz and Manna's ordering [11] relies on these two degrees of finiteness.

Many proof assistants come with a library of finite multisets. Isabelle/HOL defines multisets as the subtype of functions from elements to natural numbers that return 0 for all but finitely many elements:

typedef $'a\ multiset = \{f :: 'a \Rightarrow nat.\ \mathsf{finite}\ \{x.\ f\ x > 0\}\}$

This representation is *negative* because the element type occurs to the left of the function arrow. Applications such as nested and hereditary multisets [4] require datatypes to nest recursion through multisets. To this end, the type of multisets must be registered as a bounded natural functor (BNF) [5, 27] (Section 2), which among other things involves defining the relational lifting on multisets $\mathsf{rel\_mset} :: ('a \Rightarrow 'b \Rightarrow bool) \Rightarrow 'a\ multiset \Rightarrow 'b\ multiset \Rightarrow bool$ and proving that it subdistributes over relation composition $\circ$ as follows: $\mathsf{rel\_mset}\ R \circ \mathsf{rel\_mset}\ S \leq \mathsf{rel\_mset}\ (R \circ S)$, where $\leq$ is the pointwise order on binary predicates. This definition and proof proceed in the library by establishing an alternative *positive* view on finite multisets as quotients of lists modulo permutations and inheriting the subdistributivity property from the relational lifting on lists.

Occasionally, the aforementioned two degrees of finiteness are too limiting. For example, analytic functions such as $\sin(x)$ may have infinitely many roots and flat functions such as $f(x) = e^{-1/x^2}$ for $x \neq 0$ and $f(0) = 0$ may have roots of infinite multiplicity: f has one at 0. In other contexts, such as weighted graphs and automata, one prefers flexible notions of multiplicity over going infinite [8, 14, 18].



We define and study two orthogonal generalizations of multisets in Isabelle. In both cases, we register the type as a BNF and establish the equivalence of a positive and a negative representation. We also exemplify both new types in two case studies involving nested (co)recursion through the type.

First, we consider countable multisets (Section 3), which are equivalently represented as (1) a subtype of functions from elements to extended natural numbers ($\mathbb{N} \cup \{\infty\}$) that return non-zero multiplicities for countably many elements, and (2) a quotient of lazy lists, the list's coinductive siblings, modulo infinitary permutations. We lift the BNF structure to the new type via the quotient representation [12]. We further define the subtype of countably infinite multisets and, inheriting the BNF structure from countable multisets, use it for an alternative modeling of uniform terms in Mazza's infinitary lambda calculus [21], recently formalized by Van Brügge et al. [28] (Section 4).

Second, we consider weighted sets (Section 5), which are equivalently represented as (1) a subtype of functions from elements to optional weights from a suitable type class that return a weight different from None for finitely many elements, and (2) a quotient of element-weight lists modulo permutation and regrouping by element. As with countable multisets, we lift the BNF structure to the new type via the quotient representation. This lifting requires us to formalize a result by Gumm and Schröder [14] while mildly generalizing it. As a case study, we define Kidney and Wu's coinductive graphs [18] that nest corecursion through weighted sets and define depth-first-search corecursively (Section 6).

We see our contribution as both library development and methodology showcase. On the library side, we provide two new types ready to be used in (co)datatype applications. On the methodology side, we use state-of-the-art BNF transfer techniques via quotients and subtypes [12] optimizing proof reuse. While the proof obligations stemming from this transfer are structurally similar, the involved proofs differ significantly for our two types. We also highlight the benefits of being able to flexibly switch between different type representations. Beyond witnessing their usefulness in our case studies, we justify our focus on the two selected multiset-like types by discussing the challenges of further generalizations (Section 7). Our formalization is available as supplementary material [23].

Related Work Both finite multiset representations, the negative one as functions and the positive one as quotients of lists, have their merits. Isabelle allows one to seamlessly switch between representations, when both defining functions and reasoning about them [16]. Proof assistants based on dependent type theory may be more sensitive as to how the type is defined and tend to favor positive quotient representations [1, 10, 17]. Joram and Veltri [17] even discuss different quotient and higher inductive types representations for finite multisets and their suitability for defining the final coalgebra of the multiset functor. In Isabelle, one obtains the final coalgebra of the multiset functor using the **codatatype** command [5], and the only requirement here is that the type is a BNF—it does not matter which representation was chosen to define it, but some representations are more suitable than others to prove the BNF properties.

When it comes to our generalizations, countable multisets and weighted sets, we are not aware of any formalizations of these types, let alone (co)recursion support through them in other proof assistants. Mildly related are Isabelle's types of bounded sets and probability mass functions (PMFs) [15], which are both registered as BNFs. Bounded sets generalize the BNFs of finite and countable sets to sets whose cardinality is bounded by the cardinality of a given type argument κ. Inhabitants of this type are sets—no multiplicites are involved. PMFs can be seen as a subtype of functions from elements to real numbers between 0 and 1 that sum to 1 for a countable subset of elements. Thus it is a sibling of both our types, although significantly different from both. In particular, it is unclear if PMFs have a positive quotient representation. PMFs' BNF properties are proved by reasoning about conditional probabilities.

Closest to our definition of weighted sets, Gumm and Schröder [14] consider the finite multiset functor over an abelian (i.e., commutative) monoid and prove, under an additional assumption called refinability, weak pullback preservation, which is equivalent in the presence of other BNF properties to relator subdistributivity. We generalize from abelian monoids to abelian semigroups, i.e., rather than requiring weights to come with a neutral element, we inject one ourselves by using the option type.

2 Subtypes, Quotients, (Co)datatypes, and Bounded Natural Functors

We recall the three main ways of introducing new types in Isabelle/HOL: subtypes, quotients, and (co)datatypes. (In fact, subtypes are the only primitive way built into Isabelle's higher-order logic, while the other two are reduced to subtypes in an internal construction.)

We already have seen an example subtype defining multisets using Isabelle's **typedef** command. The command requires the user to prove non-emptiness of the supplied set. We have omitted this easy proof for multisets: pick $f = \lambda x.\ 0$. After the proof is completed, the **typedef** command introduces the new type and provides the user with the isomorphism witnesses Rep_multiset $::$ *'a multiset* $\Rightarrow$ (*'a* $\Rightarrow$ *nat*) and Abs_multiset $::$ (*'a* $\Rightarrow$ *nat*) $\Rightarrow$ *'a multiset*. The isomorphism is formally captured by the subtype theorem type_definition Rep_multiset Abs_multiset {*f*. finite {*x*. *f x* > 0}} where the predicate type_definition *Rep Abs A* combines three properties (i) $\forall x.\ Rep\ x \in A$, (ii) $\forall x.\ Abs\ (Rep\ x) = x$, and (iii) $\forall y \in A.\ Rep\ (Abs\ y) = y$. The subtype theorem is used by the tools Lifting and Transfer [16] for defining constants on subtypes from their counterpart on the raw type and prove their properties. For example, we can define the multiset addition and prove that it is commutative:

```
lift_definition madd :: 'a multiset ⇒ 'a multiset ⇒ 'a multiset is λf g x. f x + g x by simp
lemma madd_commute : madd M N = madd N M by transfer auto
```

The **lift_definition** command requires a proof that the resulting function satisfies the multiset property, i.e., returns 0 for all but finitely many elements, assuming the arguments f and g do. Dually, the *transfer* proof method reduces the goal about multisets to one about functions: $\lambda x.\ f\ x + g\ x = \lambda x.\ g\ x + f\ x$. We may assume that f and g satisfy the multiset property; yet, this is not needed here.

Alternatively, we could have defined multisets as a quotient of lists modulo permutations. There are several ways to formulate the "modulo permutations" bit. One for which it is easy to prove that it yields an equivalence relation is to use the library function count_list that counts the number of occurrences of an element in a list and to demand that equivalent lists yield the same count for all elements:

```
abbreviation eq_mset xs ys ≡ (∀z. count_list xs z = count_list ys z)
quotient_type 'a multiset = 'a list/eq_mset by (auto intro!: equivpI reflpI sympI transpI)
```

The **quotient_type** command introduces the new *multiset* type as a quotient of the existing *list* type. The relationship between the old and the new type is expressed formally using newly introduced functions that convert between multisets and Hilbert-chosen list representatives via the quotient theorem Quotient eq_mset abs_multiset rep_multiset cr_multiset. Here, the predicate Quotient *R abs rep T* combines four properties characterizing a (possibly partial) quotient: (i) $\forall x.\ abs\ (rep\ x) = x$, (ii) $\forall x.\ R\ (rep\ x)\ (rep\ x)$, (iii) $\forall y\ z.\ R\ y\ z \leftrightarrow (R\ y\ y \wedge R\ z\ z \wedge abs\ y = abs\ z)$, and (iv) $T = \lambda y\ x.\ R\ y\ y \wedge abs\ y = x$. The last property can be seen as the definition of the correspondence cr_multiset relating a list to a multiset. As with subtypes, Lifting and Transfer can use the quotient theorem to transfer definitions and proofs accross type boundaries. A familiar example follows:

```
lift_definition madd :: 'a multiset ⇒ 'a multiset ⇒ 'a multiset is λxs ys. xs@ys by simp
lemma madd_commute : madd M N = madd N M by transfer simp
```

Here, multiset addition is defined by appending list representatives. The command requires a proof that append (@) is well-behaved on equivalence classes: equivalent pairs of lists eq_mset *xs xs'* and eq_mset *ys ys'* yield equivalent results eq_mset (*xs*@*ys*) (*xs'*@*ys'*). Similarly, the *transfer* proof method reduces the commutativity of multisets to an equivalence of lists: eq_mset (*xs*@*ys*) (*ys*@*xs*).

Lifting and Transfer operate on the level of abstraction of the subtype and quotient theorems. This means that it does not matter which way the type has been originally defined: e.g., when working with a subtype, we may switch to the quotient view by establishing a corresponding quotient theorem (after defining a suitable equivalence relation and abstraction and representation functions), and vice versa.

datatype *enat* = enat *nat* | ∞
codatatype *en* = eZ | eS *en*
datatype *'a option* = None | Some *'a*
datatype *'a list* = Nil | Cons *'a* (*'a list*)
codatatype *'a llist* = LNil | LCons *'a* (*'a llist*)
datatype *'a mtree* = Node *'a* ((*'a mtree*) *multiset*)

Figure 1 A few example datatypes and codatatypes.

$\mathsf{map}_F :: ('a \Rightarrow 'b) \Rightarrow 'a\ F \Rightarrow 'a\ F$ $\quad$ $\mathsf{set}_F :: 'a\ F \Rightarrow 'a\ set$
$\mathsf{rel}_F :: ('a \Rightarrow 'b \Rightarrow bool) \Rightarrow 'a\ F \Rightarrow 'a\ F \Rightarrow bool$ $\quad$ $\mathsf{bd}_F :: (bd\_type_F \times bd\_type_F)\ set$

(1) $\mathsf{map}_F\ \mathsf{id} = \mathsf{id}$ $\quad$ (2) $\mathsf{map}_F\ (f \circ g) = \mathsf{map}_F\ f \circ \mathsf{map}_F\ g$ $\quad$ (3) $\mathsf{set}_F \circ \mathsf{map}_F\ f = \mathsf{image}\ f \circ \mathsf{set}_F$
(4) $(\forall a \in \mathsf{set}_F\ x.\ f\ a = g\ a) \longrightarrow \mathsf{map}_F\ f\ x = \mathsf{map}_F\ g\ x$ $\quad$ (5) $\mathsf{infinite\_regular\_card\_order}\ \mathsf{bd}_F$
(6) $|\mathsf{set}_F\ x| <_o \mathsf{bd}_F$ $\quad$ (7) $\mathsf{rel}_F\ R \circ \mathsf{rel}_F\ S \leq \mathsf{rel}_F\ (R \circ S)$
(8) $\mathsf{rel}_F\ R\ x\ y \longleftrightarrow (\exists z.\ \mathsf{set}_F\ z \subseteq \{(a, b).\ R\ a\ b\} \wedge \mathsf{map}_F\ \mathsf{fst}\ z = x \wedge \mathsf{map}_F\ \mathsf{snd}\ z = y)$

Figure 2 BNF operators and their properties. (Free variables are implicitly universally quantified.)

A rather different way (and maybe a more familiar one for most users) of introducing new types is given by the **datatype** and **codatatype** commands [5]. Figure 1 gives some examples. We have already mentioned the standard *list* and *option* types. The coinductive sibling of lists, the codatatype of lazy lists *llist*, models potentially infinite sequences, which may be constructed from an infinite number of LCons applications (forever neglecting LNil). We will use lazy lists to define countable multisets.

Sometimes, a datatype and a codatatype may end up defining isomorphic types. This is the case for extended natural numbers: *enat* is their datatype representation in Isabelle's standard library, whereas *en* is our equivalent coinductive representation. Both representations are useful: datatypes allow us to define functions that proceed by recursion on one of the function's arguments; codatatypes support functions that corecursively construct an inhabitant of the codatatype in the codomain. Sometimes, the latter is the only option: there may be no datatype argument present to recurse on. Unfortunately, unlike with subtype/quotients, there is no direct facility to present an existing datatype as a codatatype or vice versa. We need to define both isomorphic types to be able to use recursion and corecursion on the respective variant. However, we can use Lifting and Transfer to transfer definitions and proofs once we have established the isomorphism between the datatype and the codatatype using the following subtype theorem.

```
fun nat_to_en :: nat ⇒ en where nat_to_en 0 = eZ | nat_to_en (Suc n) = eS (nat_to_en n)
corec einf :: en where einf = eS einf
definition enat_to_en n = (case n of enat n ⇒ nat_to_en n | _ ⇒ einf)
lemma type_definition enat_to_en (inv enat_to_en) UNIV <proof>
```

Here, **fun** recurses on natural numbers and **corec** corecurses on *en*. In our formalization, we occasionally use the more primitive commands **primrec** and **primcorec** for primitive (co)recusion, but in the paper we abstract over such details. In the subtype theorem, inv denotes the inverse of a function, which only makes sense for injective functions, and UNIV denotes the set of all elements of a type, here en.

The final example, *mtree*, demonstrates nested recursion. It defines a type of trees in which every node contains a label and a multiset of children. Isabelle's (co)datatypes support nested recursion through datatypes such as *list*, codatatypes such as *llist*, and non-(co)datatypes such as *multiset* alike. The key to this flexibility is an abstract interface, called bounded natural functor (BNF) [27], which characterizes types that may nest corecursion semantically. A type can be registered as a BNF if it provides four operators satisfying eight properties shown in Figure 2 for an abstract type *'a F*. (BNFs may have arbitrary arity; we restrict our attention to unary BNFs, which may have additional passive parameters.)

It is instructive to consider the BNF operator instances for *list*, *llist*, and *multiset*. The map functions are standard: for lists and lazy lists the function argument is applied to every component in the linear structure; for multisets applying a non-injective function may trigger a multiplicity summation for previously different elements—this happens naturally when the map function is lifted via the quotient

of lists representation. The set function should return all finitely many elements contained in a list or multiset and all countably many elements contained in a lazy list. The cardinality bound is connected to these sets' sizes: natLeq :: (*nat*×*nat*) *set* for lists and multisets and card_suc natLeq :: (*nat suc*× *nat suc*) *set* for lazy lists; these are $\aleph_0$ and $\aleph_1$ represented by well-founded relations in Isabelle's cardinals library [6]. The relational lifting (short: relator) for (lazy) lists relates two (lazy) lists of equal length by pairing their elements position-wise and checking that the relation is satisfied for all pairs. The multiset relator is hardest to grasp: it is again best to think of it in terms of the quotient of lists representation: two multisets are related if there exist two lists representing them that are related by the list relator.

The BNF operators are crucial in the definitional and reasoning principles for (co)datatypes with nested (co)recursion. For example, the multiset set function is used to access the children in *mtree*'s induction principle. If *mtree* was a codatatype, the multiset relator would be used to define a bisimulation on such trees, which prominently shows up in the coinduction principle. (Co)recursive functions use the map function to apply the (co)recursive calls through the nesting type.

The BNF properties establish the functoriality of map (1,2), that set is a natural transformation to the powerset functor (3), a congruence property relating set and map (4), infiniteness and regularity of the cardinal bound (5), boundedness of the returned set (6), relator subdistributivity (7) and an alternative characterization of the relator in terms of set and map (8). They are chosen such that BNFs are closed under composition and the (co)datatype constructions, which means that (co)datatypes like (lazy) lists are automatically registered as BNFs by the respective command introducing them.

BNFs are, however, not closed in general under subtypes and quotients, which means that *multiset* must be registered manually. In some cases, the BNF structure can be lifted across subtypes and quotients. Fürer et al. [12] identify sufficient conditions for when this is the case and automate this lifting in form of the **lift_bnf** command. For a subtype defined by a subset A on a raw type $'a\ F$, F's BNF structure can be lifted to the new type if A is closed under the map function in the following sense (S1) $\forall f.\ \forall x \in A.\ \mathsf{map}_F\ f\ x \in A$ and (S2) $\forall z.\ \mathsf{map}_F\ \mathsf{fst}\ z \in A \longrightarrow \mathsf{map}_F\ \mathsf{snd}\ z \in A \longrightarrow z \in A$. A stronger condition implying both, (S) $\forall f.\ \forall x.\ \mathsf{map}_F\ f\ x \in A \longleftrightarrow x \in A$, is often sufficient in practice. For a quotient defined by an equivalence relation $\equiv$ on a raw type $'a\ F$, F's BNF structure can be lifted to the new type if (Q1) $\forall R\ S.\ R \circ S \neq \bot \implies \mathsf{rel}_F\ R \circ (\equiv) \circ \mathsf{rel}_F\ S \leq (\equiv) \circ \mathsf{rel}_F\ (R \circ S) \circ (\equiv)$ and (Q2) $\forall \mathcal{A}.\ \mathcal{A} \neq \varnothing \longrightarrow (\bigcap \mathcal{A}) \neq \varnothing \longrightarrow (\bigcap_{A \in \mathcal{A}} (\equiv) \backsim \{x.\ \mathsf{set}_F \subseteq A\}) \subseteq (\equiv) \backsim \{x.\ \mathsf{set}_F \subseteq (\bigcap \mathcal{A})\}$, where $R \backsim A$ denotes the relational image $\{y.\ \exists x \in A.\ R\ x\ y\}$. Condition (Q2) is unproblematic in cases where $\forall x\ y.\ x \equiv y \longrightarrow \mathsf{set}_F\ x = \mathsf{set}_F\ y$, which will be the case for all quotients we consider.

Multisets inherit the BNF structure of lists via the quotient view. Because equivalent lists have the same sets of elements, we are only concerned with proving condition (Q1). To this end we assume $\mathsf{rel}_{list}\ R\ a\ b$, eq_mset $b\ c$, and $\mathsf{rel}_{list}\ S\ c\ d$ for arbitrary but fixed lists a, b, c, and d, and show (eq_mset $\circ$ rel_{list} $(R \circ S)$ $\circ$ eq_mset) $a\ d$. The key *left-invariance* property for doing so observes that one can reorder the equivalence and the list relator: $\forall R\ xs\ ys\ xs'.\ \mathsf{rel}_{list}\ R\ xs\ ys \longrightarrow$ eq_mset $xs'\ xs \longrightarrow \exists ys'.\ \mathsf{rel}_{list}\ R\ xs'\ ys' \wedge$ eq_mset $ys'\ ys$. Left-invariance follows by induction on the list relator (which corresponds to simultaneous induction over the two lists xs and ys of equal length). Continuing with our proof of condition (Q1), left-invariance yields a list c' such that $\mathsf{rel}_{list}\ S\ b\ c'$ and eq_mset $c'\ d$. Together with eq_mset's reflexivity and rel_{list}'s subdistributivity (7), we obtain the desired relation chain: eq_mset $a\ a$, $\mathsf{rel}_{list}\ (R \circ S)\ a\ c'$, and eq_mset $c'\ d$.

3 Countable Multisets

In this section, we define our first new type of countable multisets, register them as a BNF, and prove the equivalence between a negative and a positive representation. There are a few design questions of what constitutes a countable multiset. For example, a first naive attempt would be to only replace finite by countable in the definition of multiset, while keeping the multiplicities' type as nat. This

would not work so well in terms of the BNF structure as any meaningful map function would need to compute a countably infinite sum of non-zero multiplicities when applied with a constant function argument to a countably infinite multiset. Thus, using nat for multiplicities is not enough; enat is.

We now know how a negative representation for countable multisets would look like. But our experience showed that for multisets the quotient representation was more fruitful for transferring the BNF structure (in fact functions from the multisets' negative representation are not a BNF in $'a$). Thus, we define countable multisets as a quotient, mimicking the multisets' positive representation.

abbreviation eq_cmset *lxs lys* ≡ (∀*z*. count_llist *lxs z* = count_llist *lys z*)
quotient_type $'a$ *cmset* = $'a$ *llist*/eq_cmset **by** (*auto intro!: equivpI reflpI sympI transpI*)

Lists have been replaced by lazy lists. We use count_llist :: $'a$ *llist* ⇒ $'a$ ⇒ *enat* that counts how often an element occurs in a lazy list. Unlike its list counterpart count_list, this function was not already available in Isabelle's library of lazy lists [20]. While count_list is easily defined by recursion on the list, this is one example where we have no argument to recurse on. Fortunately, the codomain *enat* is isomorphic to our *en* codatatype, so that we can corecurse on *en* and transfer the resulting function to *enat*.

corec count_llist_en :: $'a$ *llist* ⇒ $'a$ ⇒ *en* **where**
  count_llist_en *lxs x* = if *x* ∈ set_{llist} *lxs* then eS (ltl (ldropWhile ((≠) *x*) *lxs*)) else eZ
lift_definition count_llist :: $'a$ *llist* ⇒ $'a$ ⇒ *enat* **is** count_llist_en .

Here, ltl returns the tail of a lazy lists and ldropWhile *P lxs* applies ltl iteratively starting from *lxs* as long as the lazy list's head satisfies *P*. (If all elements satisfy *P*, then LNil is returned.)

3.1 BNF Structure

We proceed by lifting the BNF structure from lazy lists to the quotient type: **lift_bnf** $'a$ *cmset*. This command yields proof obligations corresponding to criteria (Q1) and (Q2) from Section 2. As indicated, (Q2) follows easily because equivalent lazy lists have the same sets of elements: ∀*lxs lys*. eq_cmset *lxs lys* ⟶ set_{llist} *lxs* = set_{llist} *lys*, which itself follows from the fact that *x* ∈ set_{llist} *lxs* holds if and only if count_llist *lxs x* ≠ enat 0 does.

For (Q1), we observe that left-invariance also holds for lazy lists:

lemma rel_{llist} *R lxs lys* ⟶ eq_cmset *lxs*′ *lxs* ⟶ ∃*lys*′. rel_{llist} *R lxs*′ *lys*′ ∧ eq_cmset *lys*′ *lys*

Equipped with this fact we can proceed proving (Q1) precisely as for finite multisets. However, proving that left-invariance holds is significantly more complicated for lazy lists than for lists, because we cannot use induction: relators of codatatypes are coinductive. Instead, we opt for an alternative characterization of eq_cmset, which lets us more directly construct the witness needed for left invariance.

lemma eq_cmset *lxs*′ *lxs* ⟷ (∃π. lbij_on π *lxs* ∧ lpermute π *lxs* = *lxs*′)

This characterization states that two lazy lists are equivalent if an only if there exists a bijection π on the indices of the first lazy list *lxs* such that rearranging the elements of *lxs* according to π we obtain the second lazy list. The statement involves the following definitions that we introduced:

abbreviation lbij_on π *lxs* ≡ bij_betw π {*k*. enat *k* < llength *lxs*} {*k*. enat *k* < llength *lxs*}
corec lupt :: *nat* ⇒ *enat* ⇒ *nat* llist **where**
  lupt *i j* = if enat *i* ≥ *j* then LNil else LCons *i* (lupt (Suc *i*) *j*)
definition lpermute π *lxs* = map_{llist} (λ*i*. lnth *lxs* (π *i*)) (lupt 0 (llength *lxs*))

Here, bij_betw *f A B* states that *f* is a bijection between sets *A* and *B*, llength *lxs* returns the length of the lazy list *lxs* as an extended natural number, and lnth *lxs i* returns the element at index *i* in *lxs*.

The alternative characterization's left-to-right direction is among the technically most intricate proofs in our formalization. It involves explicitly constructing the bijection by means of a recursive function that finds the index of the i-th occurrence of an element in a lazy list (omitted). Equipped with this characterization, however, proving left-invariance is easy, which concludes the proofs required by **lift_bnf**. We have thereby registered countable multisets as a BNF, inheriting lazy list's BNF structure. In particular, the command defines all the lifted BNF operators and proves their properties.

3.2 Equivalence Between the Positive and the Negative View

We want to view our countable multisets as a subtype of functions from elements to extended natural numbers that return 0 all but for countably many elements. A good first step is to define the representation function, which given a countable multiset will return the corresponding function. Since we already have a corresponding function on lazy lists, we just lift it to countable multisets:

lift_definition cmcount :: *'a cmset* ⇒ *'a* ⇒ *enat* **is** count_llist **by** *auto*

We can now state the desired subtype theorem:

lemma type_definition cmcount (inv cmcount) {f. countable {x. f x ≠ enat 0}}

This lemma reduces to proving that cmcount is injective (a prerequisite for using inv anyway) and surjective on {f. countable {x. f x ≠ enat 0}}. The former is easy; the latter is not. Surjectivity means that given such a function f, we must exhibit a countable multiset M, so that cmcount $M = f$. Equivalently, we must exhibit a lazy list *lxs*, so that count_llist *lxs* $= f$. Because {x. f x ≠ enat 0} is countable, we obtain a distinct lazy list *lxs* of precisely these elements for which f returns a non-zero value. We replicate elements according to f's multiplicity, obtaining a lazy list of lazy lists: *lxss* = map_{llist} (λx. map_{llist} (λ_. x) (lupt 0 (f x))) *lxs*. Flattening *lxss* gives us the desired lazy list; however flattening is a complex operation that must preserve all elements' (possibly infinite) multiplicities. To this end, we define the function lmerge :: *'a llist llist* ⇒ *'a llist* (omitted), which satisfies the counting property count_llist (lmerge *lxss*) z = lsum (map_{llist} (λ*lxs*. count_llist *lxs* z) *lxss*). Here, lsum sums a lazy list of extended natural numbers. It has a similar issue as count_llist: no argument to recurse on. Instead, we define it corecursively on *en* and transfer it to *enat*, just like we did for count_llist.

The counting property is another "hard nut" in our formalization. In fact, we started with the definition of lmerge used by Castro Gonçalves Silva et al. [25], but ended up changing to a different traversal strategy that avoids interleaving infinite lazy lists, because of difficulties in getting the coinductive proofs through. (Our new traversal strategy is inspired by the one used for a similar operator smerge on streams in Isabelle's standard library. Note that no counting property has been proved for the stream operator.) Ultimately, the precise definition of lmerge does not matter: the counting property is all we need to know about it to conclude the surjectivity proof with the witness lmerge *lxss*.

As a final missing ingredient to conveniently work with the subtype view on countable multisets, we establish correspondences between BNF operators and other functions we defined via the quotient link and operators on the subtype's raw type. As a simple example, the empty multiset cmempty has been defined by lifting LNil via the quotient. The corresponding function is λ_. enat 0. All correspondences are expressed as transfer rules [16] and selected by the *transfer* proof method when working with one or the other view. The map function has the most intricate correspondance: map_*cmset* had been defined via map_*llist*. In the raw type world, it correponds to λf M b. isum M {a. f $a = b$}, where isum :: (*'a* ⇒ *enat*) ⇒ *'a set* ⇒ *enat* is an infinite sum operator for *enat* we introduced. It is inspired by Buday and Popescu's [9] infinite sum operator on non-negative *real* numbers, which serves as a lightweight alternative to the generic but heavy, integral-based operators from Isabelle's library. Our isum is even more lightweight, i.e., uses fewer assumptions in theorems about it, because *enat* has the top element ∞ which serves as an upper bound in all sums, i.e., unlike with *real* divergence is impossible.

3.3 Countably Infinite Multisets

We introduce the subtype of countably infinite multisets, i.e., these multisets that are represented by infinite lazy lists. Lifting the BNF structure is easy via this subtype because the map function on countable multisets preserves the infiniteness (recall the strengthened criterion (S) from Section 2).

lift_definition cminfinite :: *'a cmset* ⇒ *bool* **is** linfinite <proof>
lemma *cminfinite_map* : cminfinite (map$_{cmset}$ *f M*) ⟷ cminfinite *M* <proof>
typedef *'a cinfmset* = {*M* :: *'a cmset*. cminfinite *M*} <proof>
lift_bnf *'a cinfmset* **by** (*auto simp: cminfinite_map*)

Here, linfinite is the coinductive predicate characterizing infinite lazy lists: those that neglect LNil forever. The last command defines the BNF operators, proves their properties, and registers countably infinite multisets as a BNF. We also introduce several helpful functions that will be used in the next section:

lift_definition cinfmcount :: *'a cinfmset* ⇒ *'a* ⇒ *enat* **is** cmcount .
lift_definition cmreplace :: *'a cmset* ⇒ *'a* ⇒ *'a* ⇒ *'a cmset* **is**
  λf a b. f(*a* := *f a* − 1, *b* := *f b* + 1) <proof>
lift_definition cinfmreplace :: *'a cinfmset* ⇒ *'a* ⇒ *'a* ⇒ *'a cinfmset* **is** *cmreplace* <proof>

Note that lifting to *cinfmset* always proceeds via *cmset*. Thereby, it does not matter via which representation the respective function on *cmset* has been defined. For example, cmcount has been defined via the quotient of lazy lists, whereas cmreplace used the subtype of functions route.

4 Case Study I: Uniform Infinitary Lambda Calculus

It is time to reap the fruits of hard BNF registration labor. We consider a result due to Mazza [21], which establishes an isomorphism between the standard lambda calculus and an infinitary affine lambda calculus, in which variables may be used at most once. Infinite replication of variables allows Mazza to bridge the gap between the at most once restrictions and unrestricted use. Van Brügge et al. [28] have formalized Mazza's result as a case study for their new tool for reasoning about syntax with complex bindings. Their tool is based on map-restricted bounded natural functions (MRBNFs), a generalization of BNFs, while at the same time being well-integrated with Isabelle's database of registered BNFs. Their formalization uses the following type for the terms of the infinitary calculus:

binder_datatype *'a iterm* = iVar *'a* | iApp (*'a iterm*) ((*'a iterm*) *stream*)
  | iLam ((*x*:*'a*) *dstream*) (*t*:*'a iterm*) **binds** *x* **in** *t*

The **binder_datatype** command is the core of their syntax with bindings infrastructure. For our purposes, we can mostly ignore the additional annotations describing the binding structure and treat the declaration as a normal **datatype**. (In fact, this is how the internal **binder_datatype** construction starts out anyway.) Note that, iApp applies a head term to an infinite sequence (*stream*) of terms, while iLam abstracts over an infinite sequence of *distinct* variables (*dstream*).

Mazza focuses on (self-)renaming-equivalent infinitary terms (also called uniform), in which the elements' order in all streams becomes irrelevant. Quoting Mazza: "Intuitively, renaming equivalence relates terms that 'look alike' under any permutation of their application sequences." Van Brügge et al. formalize renaming equivalence using an inductive predicate. Countable multisets allow us to express parts of this predicate's invariants directly in the type. We therefore change *iterm* to be unordered:

binder_datatype *'a iterm* = iVar *'a* | iApp (*'a iterm*) ((*'a iterm*) *cinfmset*)
  | iLam ((*x*:*'a*) *cinfset*) (*t*:*'a iterm*) **binds** *x* **in** *t*

The new definition nests recursion through countably infinite multisets, which is possible because we have registered this type as a BNF. Also distinct streams have been replaced by countably infinite sets

(*cinfset*), another type we introduced. This type is not a BNF because it is not closed under mapping with non-injective functions, but it also does not have to be as we do not nest recursion through this type. (It is an MRBNF though, which is just what **binder_datatype** needs.)

Van Brügge et al. fix an uncountably infinite type of variables *ivar* and from that point on work with monomorphic terms: **type_synonym** *itrm* = *ivar iterm*. We redefine their β-reduction and reprove a strong rule induction principle for it using their infrastructure. The earlier stream-based β-reduction mapped the i-th variable in the distinct stream to the i-th term in the argument stream. With our unordered countably infinite sets and multisets, we non-deterministically consider all valid substitutions from variables to terms at once. Formally, β-reduction is defined as follows:

definition imkSubst $X\ A\ M = \{f :: ivar \Rightarrow itrm.\ |\{x.\ f\ x \neq \mathsf{iVar}\ x\}| <_o |\mathsf{UNIV} :: ivar\ set| \wedge$
$(\forall x \in A.\ f\ x = \mathsf{iVar}\ x) \wedge (\forall e.\ \mathsf{isum}\ (\lambda x.\ \underline{\mathsf{if}}\ f\ x = e\ \underline{\mathsf{then}}\ 1\ \underline{\mathsf{else}}\ 0)\ (\mathsf{set}_{cinfset}\ X) = \mathsf{cinfmcount}\ M\ e)\}$
inductive istep **where**
$f \in \mathsf{imkSubst}\ X\ (\mathsf{set}_{iterm}\ e - \mathsf{set}_{cinfset}\ X)\ M \longrightarrow \mathsf{istep}\ (\mathsf{iApp}\ (\mathsf{iLam}\ X\ e)\ M)\ (\mathsf{isubst}\ f\ e)$
$|\ \mathsf{istep}\ e\ e' \longrightarrow \mathsf{istep}\ (\mathsf{iLam}\ X\ e)\ (\mathsf{iLam}\ X\ e')\ |\ \mathsf{istep}\ e\ e' \longrightarrow \mathsf{istep}\ (\mathsf{iApp}\ e\ M)\ (\mathsf{iApp}\ e'\ M)$
$|\ e' \in \mathsf{set}_{cinfmset} \longrightarrow \mathsf{istep}\ e'\ e'' \longrightarrow \mathsf{istep}\ (\mathsf{iApp}\ e\ M)\ (\mathsf{iApp}\ e\ (\mathsf{cinfmreplace}\ M\ e'\ e''))$

Here, imkSubst describes the set of valid substitutions, i.e., functions from variables to terms of small support (i.e., replacing only countably many variables with some other term) that additionally do not replace variables from a given set A and are consistent with respect to the multiplicities: if term e has count n in M, then there must be n different variables in X that f maps to e. (Note that n may be ∞.)

We have not finished upgrading the full isomorphism proof to the new type. The key point we want to highlight is that once appropriate abstractions are established—such as BNF registration and a hierarchy of subtype and quotient theorems—we can work with more unusual types smoothly.

5 Weighted Sets

We define our second new type, weighted sets, generalizing the natural number multiplicities from finite multisets to generic weights from a well-behaved algebraic structure. More precisely, our weights form a *refinable abelian semigroup* represented by the typeclass *ref_ab_semigroup_add* that we introduce. Abelian semigroups already exist in Isabelle as a type class (*ab_semigroup_add*, where *_add* means that we write + for the semigroup operator)—we merely add the refinability aspect, which we will define in Subsection 5.1. We will argue that refinable abelian semigroups are in fact a necessary condition on weights to register weighted sets as a BNF. As usual, we will develop a positive and a negative representation for weighted sets and prove them equivalent.

Unlike *nat* and *enat* used in finite and countable multisets, (refinable) abelian semigroups do not necessarily have a neutral element (e.g., $\{n :: nat.\ n > 42\}$ is a valid instance). To express non-membership in the weighted set, we use optional weights: None implies non-membership. Semigroup addition is lifted to *option*: $\mathsf{None} + w = w + \mathsf{None} = w$ and $\mathsf{Some}\ v + \mathsf{Some}\ w = \mathsf{Some}\ (v + w)$.

Learning from our experience with (countable) multisets, we anticipate that the positive representation is more fruitful for obtaining the BNF structure. Compared to the positive representation of finite multisets, lists are replaced by key–value lists where the value is our weight. Weighted sets are (positively) represented as a quotient of such key–weight lists in the following sense: two key–weight lists are considered equivalent if for every key the combined weight (using +) is the same in both lists. Formally:

abbreviation sum_key *kws a* ≡ fold $(\lambda(\_, w)\ y.\ \mathsf{Some}\ w + y)$ (filter $(\lambda(a', \_).\ a = a')$ *kws*) None
definition eq_wset *kvs kws* = $(\forall a.$ sum_key *kvs a* = sum_key *kws a*$)$
quotient_type $('a, 'w)$ *wset* = $('a \times 'w)$ *list*/eq_wset
by (*auto intro!: equivpI reflpI sympI transpI simp add: eq_wset_def*)

For brevity, we write $=_w$ for eq_wset and $=_m$ for the multiset equivalence on lists eq_mset.

5.1 Refinable Abelian Semigroups

We restrict our attention to weights from a well-behaved algebraic structure. Gumm and Schröder [14] introduced the concept of refinable monoids, which is at the core of what we call well-behaved.

▶ **Definition 1** ((m,n)-refinable). Given $m, n \in \mathbb{N}$, a monoid $\mathcal{M}$ is called (m,n)-refinable, if for any $r_1, ..., r_m, c_1, ..., c_n \in \mathcal{M}$ with $r_1 + ... + r_m = c_1 + ... + c_n$ one can find an $m \times n$-matrix $(a_{i,j})$ of elements of $\mathcal{M}$, whose row sums are $r_1, ..., r_m$ and whose column sums are $c_1, ..., c_n$. Visually:

$$\begin{array}{ccc|c} a_{1,1} & \dots & a_{1,n} & r_1 \\ \vdots & \ddots & \vdots & \vdots \\ a_{m,1} & \dots & a_{m,n} & r_m \\ \hline c_1 & \dots & c_n & \end{array}$$

They also prove a key lemma reducing the dimensionality of the matrix for abelian monoids.

▶ **Lemma 2** ((2, 2)-refinability [14]). For any $m, n > 1$ we have: An abelian monoid is (m, n)-refinable, if and only if it is $(2, 2)$-refinable.

We generalize mildly to work with abelian semigroups instead of abelian monoids: an abelian semigroup $'w$ is refinable if the option type $'w$ *option* is $(2, 2)$-refinable. Note that for every abelian semigroup $'w$ the option type $'w$ *option* forms an abelian monoid with the addition lifted as showed earlier and None being the neutral element. We introduce a new type class *ref_ab_semigroup_add* capturing the notion of refinable abelian semigroups. From now on, whenever we write $'w$ for the type of weights we will implicitly assume that the type belongs to the *ref_ab_semigroup_add* type class.

5.2 BNF Structure

The BNF structure on weighted sets is obtained by lifting the BNF structure on key–weight lists via the quotient representation in a similar fashion as for finite and countable multisets. As before, (Q2) follows easily from the observation that equivalent key–weight lists must have the same keys: $\forall kvs\ kws.\ kvs =_w kws \longrightarrow$ set (map fst kvs) = set (map fst kws). We are left with proving (Q1). Unfortunately, unlike for finite and countable multisets left-invariance does not hold for key–weight lists, which we demonstrate next. First, the relator structure on key–weight lists we are about to lift is oblivious to weights or how to add them. Rather it expects related keys to align and to have the same weight. Formally, the key–weight list relator is defined as: $kvs \simeq_R kws = \mathsf{rel}_{list}\ (\lambda(a, v)\ (b, w).\ R\ a\ b \wedge v = w)\ kvs\ kws$. Now consider $R = \lambda\_\ \_.$ True, $kvs = [(a, 1 :: nat), (a, 1)]$, $kws = [(a, 1), (b, 1)]$, and $kvs' = [(a, 2)]$ such that we have $kvs \simeq_R kws$ and $kvs' =_w kvs$. Left-invariance demands the existence of kws' such that (i) $kvs' \simeq_R kws'$ and (ii) $kws' =_w kws$. However, no such kws' exists, because it would need to have length one to satisfy (i) and contain both values a and b to satisfy (ii).

The problem is that if the weights in kvs and kvs' are not the same, then the transformations the weight-oblivious relator allows will also yield different weights. Our solution is to further refine the weights, which we call splitting, in a way that we can match the weights from kvs and kvs' one-to-one. Refinability is key for this being possible, which we will prove later. The splitting is defined as follows:

▶ **Definition 3** ($\ll$). For $kvs, kvs' :: ('a \times 'w)$ list, let $kvs' \ll kvs$ hold if and only if kvs can be obtained from kvs' by adding the values of neighboring elements with the same key.

For example, $[(a, 1), (a, 3), (a, 2), (b, 4), (b, 5)] \ll [(a, 6), (b, 9)]$. We can now obtain the left- and right-invariance theorems for our splitting relation, instead of our equivalence relation.

▶ **Lemma 4** (split_left). $kvs \simeq_R kws \longrightarrow kvs' \ll kvs \longrightarrow \exists kws'.\ kvs' \simeq_R kws' \wedge kws' \ll kws$

▶ **Lemma 5** (split_right). $kvs \simeq_R kws \longrightarrow kws' \ll kws \longrightarrow \exists kvs'.\ kvs' \simeq_R kws' \wedge kvs' \ll kvs$

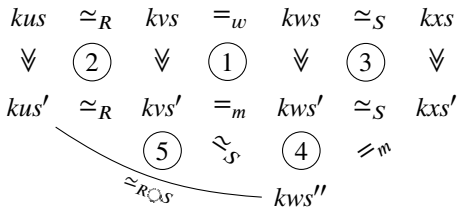


Figure 3 Proof of subdistributivity (Q1) for weighted sets.

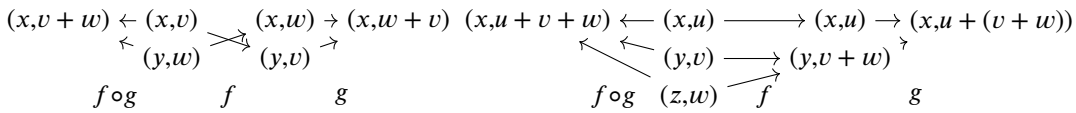


Figure 4 Illustration that map composition implies commutativity and associativity

Both lemmas are proved similarly, by observing that the weights in *kvs* and *kws* must be element-wise equal and thus any split performed in either list can be replicated in the other.

Furthermore, we need the following lemma that lets us split two equivalent key–weight lists into two key–weight lists that are equivalent as lists, i.e., using the equivalence defining finite multisets, which is a much stronger requirement then the key–weight list equivalence.

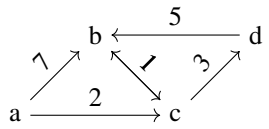
▶ **Lemma 6** (split_exists). $kvs =_w kws \longrightarrow \exists kvs'\ kws'.\ kvs' \ll kvs \wedge kws' \ll kws \wedge kvs' =_m kws'$

To prove this lemma, define $kvs_a = \mathsf{filter}\ (\lambda(a', \_).\ a = a')\ kvs$ for every key a and kws_a analogously. From $kvs =_w kws$ it follows that the sums of the weights in kvs_a and kws_a are equal. We can thus use refinability of our weights to obtain two lists kvs'_a and kws'_a such that $kvs'_a \ll kvs_a$, $kws'_a \ll kws_a$, and $kvs'_a =_m kws'_a$. By concatenating kvs'_a for all keys a in *kvs* and rearranging them (and similarly for *kws*, which has the same set of keys), we can obtain the desired lists kvs' and kws'.

Now, we prove (Q1) following the diagram in Figure 3. The top row corresponds to the relation compostition from (Q1)'s assumption. The conclusion follows the lower path from *kus* to *kxs* while noting that $kus' \ll kus$ implies $kus' =_w kus$ and similarly $kxs' \ll kxs$ and $kxs'' =_m kxs'$ imply $kxs'' =_w kxs$. From the top row the rest of the diagram can be constructed following the numbered subdiagrams, each justified as follows: (1) Lemma 6; (2) Lemma 5; (3) Lemma 4; (4) left-invariance for the finite multiset equivalence; (5) relator subdistributivity (7).

We now argue that the refinable abelian semigroup structure on weights is necessary for lifting the BNF structure via the quotient of key–weight lists. Both commutativity and associativity of addition are necessary because of map's functoriality. Figure 4 illustrates this visually on two examples: the arrows explain what the respective mapped function does, e.g., on the left f swaps keys x and y, whereas g maps both keys to x. If commutativity would not hold, the order in which we sum the elements in the underlying key–weight list would become relevant. The diagram on the right explains the need for associativity through a similar example. To justify refinability, let $R = S = \lambda\_\ \_.\ \mathsf{True}$. Moreover, let W be the weighted set generated by $[(x, a), (y, b)]$ and W' be the weighted set generated by $[(x, c), (y, d)]$. Finally, assume $a+b = c+d$. Then we have $(\mathsf{rel}_{wset}\ R \circ \mathsf{rel}_{wset}\ S)\ W\ W'$. By the BNF relator's subdistributivity (7), we also obtain $(\mathsf{rel}_{wset}\ (R \circ S))\ W\ W'$. Then using the BNF relator's characterization (8) we would obtain a weighted set Z of tuples whose projections are W and W'. We define $a_{(n,m)}$ as the weight of the key (n, m) in Z. Then $a_{(x,x)}$, $a_{(x,y)}$, $a_{(y,x)}$ and $a_{(y,y)}$ are the weights that ensure refinability. Note that some of $a_{(n,m)}$ may be None in case this key does not occur in Z.

datatype *vert* = a | b | c | d



fun graph :: *vert* ⇒ (*vert*, *nat*) *wset* **where**
graph a = ⦇ 7 ▷ b, 2 ▷ c ⦈
graph b = ⦇ 1 ▷ c ⦈
graph c = ⦇ 3 ▷ d, 1 ▷ b ⦈
graph d = ⦇ 5 ▷ b ⦈

Figure 5 An example of a weighted graph with four vertices

5.3 Equivalence Between the Positive and the Negative View

Weighted sets can also be thought of as functions from elements to optional weights that return None for all but finitely many elements. We define the corresponding representation function and state the expected subtype theorem for this negative representation.

lift_definition weight :: ('*a*, '*w*) *wset* ⇒ '*a* ⇒ '*w* option **is** sum_key **by** (*simp add: eq_wset_def*)
lemma type_definition weight (inv weight) {*f*. finite {*x*. *f x* ≠ None}}

The subtype theorem follows easily by showing that weight is surjective on {*f*. finite {*x*. *f x* ≠ None}} and injective. To facilitate working with the new type, we define standard functions on weighted sets and establish correspondences between their positive and negative representations. For example, the empty weighted set wempty is defined as [] positively and as *λ*_. None negatively. Similarly, addition of weighted sets, wadd is defined as *λxs ys*. *xs*@*ys* positively and as *λf g x*. *f x* + *g x* negatively.

6 Case Study II: Coinductive Graphs

Our second case study follows Kidney and Wu's recent coinductive modeling of graphs in Agda [18]. We specifically focus on developing the examples presented in their Section 4.3 [18]. We adopt notation similar to that of the original authors to facilitate comparison. A weighted set with *n* elements e_k having weights w_k is written as ⦇ w_1 ▷ e_1, w_2 ▷ e_2, ..., w_n ▷ e_n ⦈.

A first representation of weighted graphs uses functions from vertices to weighted sets of neighboring vertices. Figure 5 shows an example graph in this representation. The weight type used there is *nat*. Our formalization follows Kidney and Wu and requires that the weight type comes with even more structure than a refinable abelian semigroup, namely that it is a semi-semiring (type class *ssr*, which from now on becomes our default for '*w*), which satisfies the following:

1. *ssr* is a subclass of *ref_ab_semigroup_add*. Here, + should be thought of as the minimum, as it is used to choose between two paths (and we prefer minimal paths).
2. *ssr* has an additional operator ⊗ with a neutral element one. ⊗ is used to add weights together when constructing longer paths from shorter ones.
3. The operators + and ⊗ are distributive.

The proofs are conducted on weights that are instances of *ssr*, but our examples use natural numbers as weights. Since we want to modify the meaning of addition from + to min, we actually work with a type copy *nat*′, on which we lift define + on *nat*′ as min on *nat*, ⊗ on *nat*′ as + on *nat*, and one on *nat*′ as 0 on *nat*. To instantiate *nat*′ as a *ssr*, the following properties must be true:

1. + on *nat*′ is commutative and associative. *Proof:* min is commutative and associative on *nat*. Moreover, + on *nat*′ is refinable. *Proof:* Assume *a* + *b* = *c* + *d*, i.e., min *a b* = min *c d* on *nat*. Without loss of generality, assume *a* > *b*, *c* > *d* and *b* = *d*. The following diagram shows the desired refinement:

None	Some *c*	Some *c*
Some *a*	Some *b*	Some *d*
Some *a*	Some *b*	

$W = \wr\ 1 \triangleright b,\ 2 \triangleright c\ \wr$
$f\ b = \wr\ 1 \triangleright a,\ 3 \triangleright d\ \wr$
$f\ c = \wr\ 4 \triangleright d\ \wr$
$f\ \_ = \wr\ \wr$

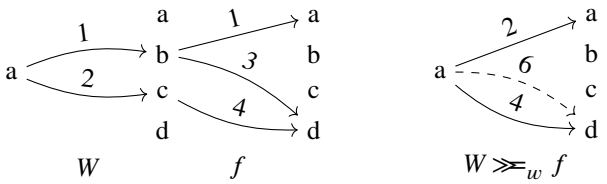


Figure 6 An example of wbind ($\gg\!=_w$).

$F = \wr\ 7 \triangleright \mathsf{Inl}\ \wr\ 1 \triangleright \mathsf{Inr}\ a$
$,\ 2 \triangleright \mathsf{Inr}\ b\ \wr$
$,\ 3 \triangleright \mathsf{Inr}\ c\ \wr$

$k\ b = \wr\ 3 \triangleright \mathsf{Inr}\ 0\ \wr$
$k\ c = \wr\ 5 \triangleright \mathsf{Inr}\ 1$
$,\ 6 \triangleright \mathsf{Inr}\ 2\ \wr$
$k\ \_ = \wr\ \wr$

$F \gg\!= k \equiv$
$\wr\ 7 \triangleright \mathsf{Inl}\ \wr\ 5 \triangleright \mathsf{Inr}\ 0\ \wr$
$,\ 8 \triangleright \mathsf{Inr}\ 1,\ 9 \triangleright \mathsf{Inr}\ 2\ \wr$

Figure 7 An example of bind_forest ($\gg\!=$) omitting the Forest constructors for brevity.

2. one is the neutral element of $\otimes$. *Proof:* 0 is the neutral element of + on *nat*.
3. $\otimes$ and + on *nat*′ are distributive. *Proof:* min and + are distributive on *nat*.

We next define a monadic structure on this graph representation: wreturn $a = \wr$ one $\triangleright\ a\ \wr$ and wbind :: $('v \times 'w)\ wset \Rightarrow ('v \Rightarrow ('u \times 'w)\ wset) \Rightarrow ('u \times 'w)\ wset$, which we denote as the infix $\gg\!=_w$. The definition of wbind combines the weights of the composed parts as illustrated in Figure 6. The dashed line is a path that is superseded by a shorter path. The formal definition of wbind is obtained by lifting from the negative representation of weighted sets:

```
lift_definition wbind :: ('v, 'w) wset ⇒ ('v ⇒ ('u, 'w) wset) ⇒ ('u, 'w) wset is
  λ(f :: 'v ⇒ 'w option) (k :: 'v ⇒ 'u ⇒ 'w option) (el :: 'u).
  Finite_Set.fold (λ(_, w) w'. Some w + w') None
    {(x, w). ∃w1 w2. f x = Some w1 ∧ k x el = Some w2 ∧ w = w1 ⊗ w2})
```

Kidney and Wu's main innovation is to switch from the presented graph representation as a one-step neighboring function to a coinductive representation that supports multiple neighboring steps:

```
codatatype ('v, 'w) forest = Forest ((('v, 'w) forest + 'v, 'w) wset)
```

An inhabitant of this type has a weighted set of either immediate neighbors $'v$ or corecursive subgraphs $('v, 'w)$ *forest*. This nested representation is useful because it enables coinductive algorithms on infinite structures with guardedness that is not available in a single flat one-step function to *wset*.

We define the monadic bind for *forest*, called bind_forest and written infix as $\gg\!=$. The intuition for $F \gg\!= k$ is that it will apply k to the leaves and combine weights using $\otimes$ via wbind. Figure 7 shows an example of this operation. Kidney and Wu give the following characteristic equation for $\gg\!=$, rewritten in our notation:

$$F \gg\!= k \equiv \mathsf{Forest}\ (\mathsf{un\_Forest}\ F \gg\!=_w \mathsf{case\_sum}\ (\mathsf{wreturn} \circ \mathsf{Inl} \circ (\lambda G.\ G \gg\!= k))\ (\mathsf{un\_Forest} \circ k))$$

Here, un_Forest is the selector of *forest*, inverse to the constructor Forest and case_sum performs a case distinction on the sum type, which has constructors Inl and Inr. Our definition is slightly more complex: it ensures that the corecursive call is applied immediately below the guarding constructor and through the map function of *wset*, which guarantees productivity by primitive corecursion [5].

```
corec bind_forest :: ('v, 'w) forest ⇒ ('v ⇒ ('u, 'w) forest) ⇒ ('u, 'w) Forest where
  bind_forest F k = Forest (map_wset (case_sum (λG. Inl (bind_forest G k)) id)
    (un_Forest F ≫=w case_sum (wreturn ∘ Inl) (map_wset Inr ∘ un_Forest ∘ k))))
```

In this definition, $\gg\!=_w$'s intermediate output has type $(('v, 'w)\ forest + ('u, 'w)\ forest + 'u,\ 'w)\ wset$. For each key of F if it is a forest then wreturn has been applied such that only a trivial merge of weights

occur, moreover Inl is used to indicate that a corecursive call must happen here. Otherwise the key has type $'v$ and continuation k is composed along with Inr, which stops the corecursion for this key.

We then prove that our definition is equivalent to what Kidney and Wu have intended by showing that the above characteristic equation holds for our constant. This follows easily from the fact that the map function of *wset* commutes with $\gg\!=_w$: $\mathsf{map}_{wset}\ f\ (W \gg\!=_w g) = W \gg\!=_w (\mathsf{map}_{wset}\ f \circ g)$.

By repeatedly applying $\gg\!=$ to a *forest* Kidney and Wu propose a depth-first-search operator $\mathsf{dfs} :: ('v \Rightarrow ('v, 'w)\ \mathit{forest}) \Rightarrow 'v \Rightarrow ('v, 'w)\ \mathit{forest}$ with the following characteristic equation:

$$\mathsf{dfs}\ k\ x \equiv \mathsf{Forest}\ (\mathsf{wadd}\ (\mathsf{wreturn}\ (\mathsf{Inr}\ x))\ (\mathsf{un\_Forest}\ (\mathsf{dfs\_forest}\ (k\ x)\ k)))$$

Unfortunately, this definition is not productive in general. Take $k = \lambda\_.\ \wr\ n \triangleright \mathsf{Inr}\ a\ \wr$. Applying k repeatedly would never finish constructing a weight; it would just keep adding n to the weight. To address this issue, we follow Kidney and Wu and introduce the type *iforest* which forces the outermost *wset* layer to contain a proper *forest* rather than a single vertex. This additional layer serves as a guard and is sufficient for the productivity of dfs with the adjusted type $('v \Rightarrow ('v, 'w)\ \mathit{iforest}) \Rightarrow 'v \Rightarrow ('v, 'w)\ \mathit{forest}$:

$$\textbf{datatype}\ ('v,\ 'w)\ \mathit{iforest} = \mathsf{iForest}\ (((('v,\ 'w)\ \mathit{forest}, 'w)\ \mathit{wset})$$

The codatatype introduces a selector $\mathsf{un\_iForest} :: ('v,\ 'w)\ \mathit{iforest} \Rightarrow (('v,\ 'w)\ \mathit{forest},\ 'w)\ \mathit{wset}$, which is inverse to iForest. We define an embedding of *iforest* to *forest*:

$$\textbf{definition}\ \sigma\_\mathsf{forest} :: ('v,\ 'w)\ \mathit{iforest} \Rightarrow ('v,\ 'w)\ \mathit{forest}\ \textbf{where}$$
$$\sigma\_\mathsf{forest}\ F = \mathsf{Forest}\ (\mathsf{map}_{wset}\ \mathsf{Inl}\ (\mathsf{un\_iForest}\ F))$$

Using *iforest*, we can now define depth-first-search productively:

$$\textbf{corec}\ \mathsf{dfs\_forest} :: ('v,\ 'w)\ \mathit{forest} \Rightarrow ('v \Rightarrow ('v,\ 'w)\ \mathit{iforest}) \Rightarrow ('v,\ 'w)\ \mathit{forest}\ \textbf{where}$$
$$\mathsf{dfs\_forest}\ F\ k = \mathsf{Forest}\ (\mathsf{map}_{wset}\ (\mathsf{case\_sum}\ (\lambda G.\ \mathsf{Inl}\ (\mathsf{dfs\_forest}\ G\ k))\ \mathsf{Inr})$$
$$(\mathsf{un\_Forest}\ F \gg\!=_w \mathsf{case\_sum}\ (\mathsf{wreturn} \circ \mathsf{Inl})$$
$$(\lambda x.\ \mathsf{wadd}\ (\mathsf{map}_{wset}\ \mathsf{Inr}\ (\mathsf{wreturn}\ x))\ (\mathsf{map}_{wset}\ \mathsf{Inl}\ (\mathsf{un\_iForest}\ (k\ x))))))$$
$$\textbf{definition}\ \mathsf{dfs}\ k\ x = \mathsf{Forest}\ (\mathsf{wadd}\ (\mathsf{wreturn}\ (\mathsf{Inr}\ x))\ (\mathsf{un\_Forest}\ (\mathsf{dfs\_forest}\ (\sigma\_\mathsf{forest}\ (k\ x))\ k)))$$

The function dfs_forest uses a similar trick to become primitively corecursive as bind_forest. The definitions are in fact quite similar but there are two key differences: First, dfs_forest records the intermediate steps of the corecursive exploration of the graph by injecting $\mathsf{map}_{wset}\ \mathsf{Inr}\ (\mathsf{wreturn}\ x)$ the currently explored vertex x with weight one. Second, the corecursion continues even after applying k, by using Inl instead of Inr.

We then prove that our definition is equivalent to the characteristic equation, when limiting the scope to productive graphs exhibited via the type *iforest*:

$$\textbf{lemma}\ \mathsf{dfs\_forest}\ F\ k = F \gg\!=$$
$$(\lambda x.\ \mathsf{Forest}\ (\mathsf{wadd}\ (\mathsf{wreturn}\ (\mathsf{Inr}\ x))\ (\mathsf{un\_Forest}\ (\mathsf{dfs\_forest}\ (\sigma\_\mathsf{forest}\ (k\ x))\ k))))$$
$$\textbf{lemma}\ \mathsf{dfs}\ F\ x = \mathsf{Forest}\ (\mathsf{wadd}\ (\mathsf{wreturn}\ (\mathsf{Inr}\ x))\ (\mathsf{un\_Forest}\ (\sigma\_\mathsf{forest}\ (F\ x) \gg\!= \mathsf{dfs}\ F)))$$

The hard core of these proofs proceed by coinduction on the *forest* codatatype. We lack the space to explain the proof by coinduction in detail, but the **codatatype** command provides convenient theorems and proof method infrastructure for making it work reasonably well [5].

We remark that Kidney and Wu had approached these corecursive equations very differently by building on the theory of completely iterative monads (CIMs) [22] and developing infrastructure for solving iterative equations for this theory in Cubical Agda [29]. We were able to define the guarded dfs, by using the most basic coinduction support Isabelle provides. What made this work is the Isabelle infrastructure building on the flexible BNF abstraction together with our registration of *wset* as a BNF.

The restriction to primitive corecursion was mildly annoying to work around: one would argue that the $\gg\!=$ and dfs_forest equations are as guarded as their defining equations we use with **corec**. In principle, **corec** should be able to see this as it supports powerful up-to corecursion principles based on the notion of *friends* [3]. Intuitively, a friend of corecursion is a function returning a codatatype that consumes at most one constructor before producing at least one. Our equations would need $\gg\!=_w$ and wadd to be recognized as friends. However, these functions do not return codatatypes and are therefore out of scope. A more flexible framework for heterogeneous corecursion-friendly operators has been proposed recently in Rocq [19]—its integration in Isabelle would likely help with the described issue and thus further simplify our case study.

7 Discussion

Our formalization spans around 8 000 lines of Isabelle definitions and proofs. The bulk of these stems from the two libraries developing the types of countable (and countably infinite) multisets and weighted sets in 2 200 and 2 500 lines, respectively. Roughly these lines split evenly between the BNF registration and the equivalence to the negative representation, with a smaller chunk belonging to the actual definition of functions on the new types including transfer rule setup. The case studies contribute around 1 600 (lambda calculus) and 1 000 (graphs) lines. The former is only mildly adapted from Van Brügge et al. [28]'s formalization. The rest splits among the less prominently explained but nontheless useful and reusable libraries of countably infinite sets and infinite sums returning extended natural numbers.

Our work greatly benefits from Isabelle's advanced infrastructure for subtypes and quotients and for BNF-based (co)datatypes. While switching between subtype and quotient views is easy, (co)datatypes do not have such a feature, which leads to some redundancy in our work (e.g., *enat* and *en*). Generally, a tool that allows any type, no matter how it has been defined, to be presented as a (co)datatype, subject to some proof obligations, would allow Isabelle users to construct an even finer web of interconnections between types and to transport results along these interconnections.

We have presented two multiset-like types. A natural question to ask is why these two and whether this is the end of the tale of multiset-like types. Stepping back to the negative representation of weighted sets it seems natural to consider the following generalization of both our types:

typedef $('a, 'w, 'k)\ kwset = \{f :: 'a \Rightarrow 'w\ option.\ |\{x.\ f\ x \neq \mathsf{None}\}| <_o |\mathsf{UNIV} :: 'k\ set|\}$

Note that by choosing $'k = nat$ and restricting $'w$ to be of class *ref_ab_semigroup_add* we obtain weighted sets. Similarly, by choosing $'k = nat\ suc$ (so that $|\mathsf{UNIV} :: 'k\ set| = \mathsf{card\_suc}\ (\mathsf{natLeq})$, i.e., $\aleph_1$) and $'w$ to be the subtype $\{n.\ n \neq \mathsf{enat}\ 0\}$, where we remove enat 0 to avoid duplicated ways to indicate non-membership, we obtain countable multisets. A natural general question to ask is: Under which constraints on the involved type parameters is the resulting type a BNF? For example, we have already discussed some neccessary constraints on the weights $'w$. While we will not give a definitive answer to the stated question, we will discuss why it seems difficult to go beyond the two types we consider, e.g., infinitary extensions of weighted sets or uncountable *enat*-based multisets.

Countably Infinite Commutativity

A structure satisfying countably infinite commutativity ensures that infinite sums are invariant under permutation of summands. We show that this property is necessary for an extension of weighted sets to a countable domain. Suppose we have a BNF for *wset* extended in this way and consider s a weighted set with infinitely many keys. BNFs must to satisfy functoriality: $\mathsf{map}_{wset}\ (f \circ g)\ s = \mathsf{map}_{wset}\ f\ (\mathsf{map}_{wset}\ g\ s)$. By using this property when f is a constant function $\lambda\_.\ k$ and g is a permutation σ, we obtain: $\mathsf{map}_{wset}\ (\lambda\_.\ k)\ s = \mathsf{map}_{wset}\ ((\lambda\_.\ k) \circ \sigma)\ s = \mathsf{map}_{wset}\ (\lambda\_.\ k)\ (\mathsf{map}_{wset}\ \sigma\ s)$,

which implies countably infinite commutativity for the involved weights:

$$w_1 + w_2 + w_3 + ... = w_{\sigma^{-1}(1)} + w_{\sigma^{-1}(2)} + w_{\sigma^{-1}(3)} + ...$$

The countable infinite commutativity holds for natural numbers, but becomes false when negative numbers are added, even though the binary sum operation is commutative.

Limits of $(2, 2)$-Refinability for Infinite Sets

In Section 5 we established that refinability is a necessary condition on weights in our approach of lifting the BNF structure via the quotient representation, and we proved this condition equivalent to (2,2)-refinability for finite weighted set. However, an extension beyond finiteness requires full (m, n)-refinability, where m and n may be countable infinities. This property is not implied from $(2, 2)$-refinability, which we will demonstrate by the following counter example:

Consider $(\mathbb{N} \times \mathbb{N}) \cup \{\infty\}$ with the following addition structure: $\infty + x = x + \infty = \infty$ for all x. When ∞ is not occurring then addition for pairs is computed element-wise. Moreover, infinite sums of non-zero weights result in ∞. Then, $(\mathbb{N} \times \mathbb{N}) \cup \{\infty\}$ is (2,2)-refinable: Consider $x + y = a + b$. If all elements are not ∞, then a refinement exists since $\mathbb{N} \times \mathbb{N}$ is $(2, 2)$-refinable. Otherwise, let $x = \infty$. Then, either a or b must be ∞ too. Without loss of generality consider $a = \infty$. Then the following refinement can be created:

∞	y	∞
b	$(0, 0)$	b
∞	y	

However $(\mathbb{N} \times \mathbb{N}) \cup \{\infty\}$ is not (∞, ∞)-refinable. A counterexample is that $(1, 0) + (1, 0) + \cdots = (0, 1) + (0, 1) + \cdots = \infty$, but no refinement exists for the two infinite sequences.

Beyond Countability

Once we consider even larger cardinalities for multisets or weighted sets, apart from (somewhat valid) motivation questions, it becomes unclear which types can serve as the raw type for the positive quotient construction. Linear codatatypes like lazy lists are no longer large enough. Unfortunately, reasoning about infinite tree data structures modulo permutation seems a daunting prospect.

In this case, it seems that the quotient route may need to be abandoned in favor of a direct relator definition and subdistributivity proof. This approach is, however, also far from obvious—even for finite multiset proving subdistributivity required phrasing the relator in terms of lists.

Conclusion

We have now sketched the need for two additional conditions on the weight type when extending to countable weighted sets: Countable infinite commutativity of sums and (∞,x)-refinability. These conditions are not necessarily straightforward to instantiate, making it more labour intensive to achieve new types. Moreover, the conditions exclude some interesting structures such as integers and real numbers, from serving as valid weight types. And beyond countable sets, a path to obtaining a BNF structure is unclear at best. These observations serve as evidence that our two generalizations might lie along some Pareto-optimal curve for multiset-like structures through which one may (co)recurse.

Acknowledgements and AI declaration: We thank Andrei Popescu for discussing aspects of our formaliztion and for pointing us to Buday's and his infinite sum operator on non-negative reals [9], which we adapt to extended natural numbers. No part of our formalization [23] was produced by LLMs or other generative AI tools. This work is supported by the Independent Research Fund Denmark (DFF Sapere Aude grant 3120-00057B, titled *DISCOVER*: *Distributed Streaming Computations Verified*).

References

1 Jeremy Avigad, Leonardo de Moura, and Jared Roesch. Programming in Lean, 2016. accessed 15.02.2026. URL: https://avigad.github.io/programming_in_lean/programming_in_lean.pdf.

2 Leo Bachmair and Harald Ganzinger. Resolution theorem proving. In John Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning (in 2 volumes)*, pages 19–99. Elsevier and MIT Press, 2001. doi:10.1016/B978-044450813-3/50004-7.

3 Jasmin Christian Blanchette, Aymeric Bouzy, Andreas Lochbihler, Andrei Popescu, and Dmitriy Traytel. Friends with benefits - implementing corecursion in foundational proof assistants. In Hongseok Yang, editor, *ESOP 2017*, volume 10201 of *LNCS*, pages 111–140. Springer, 2017. doi:10.1007/978-3-662-54434-1_5.

4 Jasmin Christian Blanchette, Mathias Fleury, and Dmitriy Traytel. Nested multisets, hereditary multisets, and syntactic ordinals in Isabelle/HOL. In Dale Miller, editor, *FSCD 2017*, volume 84 of *LIPIcs*, pages 11:1–11:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017. doi:10.4230/LIPICS.FSCD.2017.11.

5 Jasmin Christian Blanchette, Johannes Hölzl, Andreas Lochbihler, Lorenz Panny, Andrei Popescu, and Dmitriy Traytel. Truly modular (co)datatypes for Isabelle/HOL. In Gerwin Klein and Ruben Gamboa, editors, *ITP 2014*, volume 8558 of *LNCS*, pages 93–110. Springer, 2014. doi:10.1007/978-3-319-08970-6_7.

6 Jasmin Christian Blanchette, Andrei Popescu, and Dmitriy Traytel. Cardinals in Isabelle/HOL. In Gerwin Klein and Ruben Gamboa, editors, *ITP 2014*, volume 8558 of *LNCS*, pages 111–127. Springer, 2014. doi:10.1007/978-3-319-08970-6_8.

7 Wayne D. Blizard. Multiset theory. *Notre Dame J. Formal Log.*, 30(1):36–66, 1989. doi:10.1305/NDJFL/1093634995.

8 Filippo Bonchi, Marcello M. Bonsangue, Michele Boreale, Jan J. M. M. Rutten, and Alexandra Silva. A coalgebraic perspective on linear weighted automata. *Inf. Comput.*, 211:77–105, 2012. doi:10.1016/J.IC.2011.12.002.

9 Gergely Buday and Andrei Popescu. Countable sums and discrete (sub)distributions. *Arch. Formal Proofs*, 2024. URL: https://www.isa-afp.org/entries/Countable_Sums_and_Discrete_Distributions.html.

10 Solange Coupet-Grimal and William Delobel. An effective proof of the well-foundedness of the multiset path ordering. *Appl. Algebra Eng. Commun. Comput.*, 17(6):453–469, 2006. doi:10.1007/S00200-006-0020-Y.

11 Nachum Dershowitz and Zohar Manna. Proving termination with multiset orderings. *Commun. ACM*, 22(8):465–476, 1979. doi:10.1145/359138.359142.

12 Basil Fürer, Andreas Lochbihler, Joshua Schneider, and Dmitriy Traytel. Quotients of bounded natural functors. *Log. Methods Comput. Sci.*, 18(1), 2022. doi:10.46298/LMCS-18(1:23)2022.

13 Paolo Guagliardo and Leonid Libkin. A formal semantics of SQL queries, its validation, and applications. *Proc. VLDB Endow.*, 11(1):27–39, 2017. doi:10.14778/3151113.3151116.

14 H. Peter Gumm and Tobias Schröder. Monoid-labeled transition systems. In Andrea Corradini, Marina Lenisa, and Ugo Montanari, editors, *CMCS 2001*, volume 44 of *Electronic Notes in Theoretical Computer Science*, pages 185–204. Elsevier, 2001. doi:10.1016/S1571-0661(04)80908-3.

15 Johannes Hölzl, Andreas Lochbihler, and Dmitriy Traytel. A formalized hierarchy of probabilistic system types - proof pearl. In Christian Urban and Xingyuan Zhang, editors, *ITP 2015*, volume 9236 of *LNCS*, pages 203–220. Springer, 2015. doi:10.1007/978-3-319-22102-1_13.

16 Brian Huffman and Ondrej Kuncar. Lifting and transfer: A modular design for quotients in Isabelle/HOL. In Georges Gonthier and Michael Norrish, editors, *CPP 2013*, volume 8307 of *LNCS*, pages 131–146. Springer, 2013. doi:10.1007/978-3-319-03545-1_9.

17 Philipp Joram and Niccolò Veltri. Constructive final semantics of finite bags. In Adam Naumowicz and René Thiemann, editors, *ITP 2023*, volume 268 of *LIPIcs*, pages 20:1–20:19. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2023. doi:10.4230/LIPICS.ITP.2023.20.

18 Donnacha Oisín Kidney and Nicolas Wu. Formalising graph algorithms with coinduction. *Proc. ACM Program. Lang.*, 9(POPL):1657–1686, 2025. doi:10.1145/3704892.

19 Jaewoo Kim, Yeonwoo Nam, and Chung-Kil Hur. Coco: Corecursion with compositional heterogeneous productivity. *Proc. ACM Program. Lang.*, 10(POPL):2643–2671, 2026. doi:10.1145/3776733.

20 Andreas Lochbihler. Coinductive. *Arch. Formal Proofs*, 2010. URL: https://www.isa-afp.org/entries/Coinductive.shtml.

21 Damiano Mazza. An infinitary affine lambda-calculus isomorphic to the full lambda-calculus. In *LICS 2012*, pages 471–480. IEEE Computer Society, 2012. doi:10.1109/LICS.2012.57.

22 Stefan Milius. Completely iterative algebras and completely iterative monads. *Inf. Comput.*, 196(1):1–41, 2005. doi:10.1016/J.IC.2004.05.003.

23 Mathias Schack Rabing and Dmitriy Traytel. Formalization of countable multisets and weighted sets, 2026. accessed 15.05.2026. URL: https://traytel.bitbucket.io/multiset-artifact.zip.

24 Gerard Salton and Chris Buckley. Term-weighting approaches in automatic text retrieval. *Inf. Process. Manag.*, 24(5):513–523, 1988. doi:10.1016/0306-4573(88)90021-0.

25 Rafael Castro Gonçalves Silva, Laouen Fernet, and Dmitriy Traytel. Nondeterministic asynchronous dataflow in Isabelle/HOL. In Yannick Forster and Chantal Keller, editors, *ITP 2025*, volume 352 of *LIPIcs*, pages 30:1–30:20. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2025. doi:10.4230/LIPICS.ITP.2025.30.

26 Josef Sivic and Andrew Zisserman. Video google: A text retrieval approach to object matching in videos. In *ICCV 2003*, pages 1470–1477. IEEE Computer Society, 2003. doi:10.1109/ICCV.2003.1238663.

27 Dmitriy Traytel, Andrei Popescu, and Jasmin Christian Blanchette. Foundational, compositional (co)datatypes for higher-order logic: Category theory applied to theorem proving. In *LICS 2012*, pages 596–605. IEEE Computer Society, 2012. doi:10.1109/LICS.2012.75.

28 Jan van Brügge, Andrei Popescu, and Dmitriy Traytel. Animating MRBNFs: Truly modular binding-aware datatypes in Isabelle/HOL. In Yannick Forster and Chantal Keller, editors, *ITP 2025*, volume 352 of *LIPIcs*, pages 11:1–11:20. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2025. doi:10.4230/LIPICS.ITP.2025.11.

29 Andrea Vezzosi, Anders Mörtberg, and Andreas Abel. Cubical Agda: A dependently typed programming language with univalence and higher inductive types. *J. Funct. Program.*, 31:e8, 2021. doi:10.1017/S0956796821000034.