

# 1 Programming with Dependent Additive Pairs<sup>\*</sup>

2 Vít Šefl

3 Charles University, Czech Republic  
4 sefl@ksvi.mff.cuni.cz

5 **Abstract.** Linear logic gives us additive pairs in the form of the additive  
6 conjunction. Intuitionistic type theory gives us dependent pairs in the  
7 form of the dependent sum type. What happens when we combine these  
8 two kinds of pairs together? And is this new pair type useful in practice?  
9 To answer these questions, we employ quantitative type theory, which  
10 can describe both substructural and dependent types simultaneously.  
11 We introduce dependent additive pairs and show how these pairs can  
12 be used in three completely different scenarios: folding data structures  
13 using linear recursion schemes, computing resource-aware proofs, and  
14 defining additive versions of inductive and coinductive types. Each of  
15 these scenarios is then illustrated by an implementation in the Janus  
16 language.

## 17 1 Introduction

18 Many programming languages use type systems. Their main purpose is to detect  
19 a wide variety of bugs before the program is even run. Throughout the years, type  
20 systems have become quite sophisticated, supporting features such as subtyping,  
21 parametric polymorphism, or metatypes. Today, some of the most expressive  
22 type systems are based on *dependent type theories*, such as the Martin-Löf type  
23 theory [11]. As the name suggests, a dependent type is a type which can depend  
24 on a value. A standard example is a vector: a list that contains its length in its  
25 type. Dependent types can be exploited to express very detailed properties of  
26 programs, which can then be automatically checked by the computer.

27 However, dependent types are poorly suited to describe how a program uses  
28 its values. In particular, any value can be freely duplicated or discarded. A pro-  
29 grammer might want to ensure that a value representing an important resource,  
30 such as an open file, is not simply discarded. *Substructural type systems* seek to  
31 address this issue by restricting certain operations on variables. The most well  
32 known subclass of substructural type systems are linear type systems. In these  
33 systems, variables are typically split into two subsets: unrestricted and linear.  
34 The key restriction is that linear variables must be used exactly once. Other  
35 restrictions give rise to different systems, such as affine or relevant type systems.

36 *Quantitative type theory* (QTT) [3] is a recent attempt at combining depen-  
37 dent and substructural types into a single theory. In QTT, variables are not split

---

<sup>\*</sup> This work was supported by the Charles University grant SVV-260699

38 into subsets. Instead, each variable is associated with an element of a semiring,  
 39 so called *multiplicity*, which keeps track of how that particular variable is used.  
 40 If the same variable occurs in different contexts, semiring operations are used to  
 41 combine its constituent multiplicities into a single value.

42 Different semirings give rise to various kinds of substructural systems. A  
 43 single-element semiring results in a system in which all variables are unrestricted.  
 44 The usual semiring on natural numbers results in a system where multiplicity  
 45 tracks exact usage of a given variable, with 1 corresponding to linear use. The  
 46 zero-one-many semiring avoids unnecessarily precise multiplicities while still sup-  
 47 porting irrelevant, linear, and unrestricted variables.

48 In QTT, multiplicities can also appear in some types. These indexed types  
 49 replace the need for multiple versions of each connective, as is the case in linear  
 50 logic. For example, instead of having linear  $(A \multimap B)$  and unrestricted  $(A \rightarrow B)$   
 51 functions, QTT has a single (dependent) function type  $(x \text{ ? } A) \rightarrow B$ . Linear  
 52 and unrestricted functions can be obtained by choosing a suitable value for  $\sigma$ .  
 53 However, since the value of  $\sigma$  can be arbitrary, each element of the semiring  
 54 thus gives rise to a different variant of the function type. For example, in the  $\mathbb{N}$   
 55 semiring, each number  $n$  gives a function that must use its parameter exactly  $n$   
 56 times.

57 Substructural types can be further classified as either *multiplicative* or *ad-*  
 58 *ditive*. Multiplicative types split resources. When introducing a new value, the  
 59 resources are divided into groups and each group is used to construct a part  
 60 of the value. When eliminating a value, each part must be used to ensure no  
 61 resources are discarded. Additive types preserve resources. When introducing a  
 62 new value, each part of the value has access to all resources. When eliminating  
 63 a value, only one part must be used to ensure no resources are duplicated.

64 Since QTT supports both dependent and substructural types, it gives us a  
 65 unique opportunity to explore dependent versions of multiplicative and additive  
 66 types. One example is the previously mentioned function type  $(x \text{ ? } A) \rightarrow B$ .  
 67 Another example is the dependent additive pair  $(x : A) \& B$ , which is a gener-  
 68 alization of the additive conjunction found in linear logic and is the main focus  
 69 of this work.

## 70 1.1 Goals

71 The primary objective of this work is to identify and describe practical uses  
 72 of dependent additive pairs. In particular, we seek problems in resource-aware  
 73 programming that are best solved by these pairs. The solutions should make use  
 74 of both the type dependency and the additive nature of this type. More generally,  
 75 we also seek novel applications of these pairs where the type dependency can be  
 76 useful but is not strictly necessary.

77 A secondary objective is to provide an implementation of each solution in a  
 78 language capable of expressing dependent additive pairs, which will demonstrate  
 79 the correctness of our solutions and allow the reader to easily verify.

## 80 1.2 Contributions

81 To achieve the stated goals, we identify three distinct and novel scenarios that are  
82 best solved by dependent additive pairs. Firstly, we implement a linear *paramor-*  
83 *phism*, which is a recursion scheme that allows access to both the recursive result  
84 and the remainder of the structure. Secondly, we show how to compute resource-  
85 aware proofs, which demonstrate that a witness satisfies a given property while  
86 allowing their resources to be shared. Thirdly, we define additive versions of in-  
87 ductive lists and coinductive streams and implement linear versions of commonly  
88 used operations. Finally, we provide implementation in the Janus language [14].

## 89 2 Related Work

90 As mentioned in the introduction, some type systems split variables into multi-  
91 ple subsets that restrict how these variables can be used. Type dependency can  
92 also be added to these systems. An early example of such a dependent theory  
93 is Cervesato and Pfenning’s Linear Logical Framework [7]. LLF splits the typ-  
94 ing context into two parts: linear and intuitionistic. One major downside of this  
95 approach is that a dependent type may only refer to variables from the intu-  
96 itionistic context. A more recent example of this split-context system is given by  
97 Krishnaswami et al. [10].

98 Semirings have also been used before to keep account of variable usage. One  
99 such example is given by Brunel et al. [6]. Semiring annotations are used with  
100 the exponential modality, which is then generalized into a full *coeffect* system.  
101 The typing context is not split. Unlike in QTT, it contains linear and *discharged*  
102 variables, which carry the semiring annotations.

103 The key insight behind QTT is provided by McBride [12]. In his type system,  
104 semiring zero represents computational irrelevance. Types are then treated as  
105 computationally irrelevant and variable occurrences there do not count towards  
106 the total multiplicity, which allows types to depend on any kind of variable.  
107 This idea is further reinforced by providing a type-erasing translation which also  
108 removes the computationally irrelevant parts.

109 QTT itself is described by Atkey [3]. This work addresses a problem with  
110 inadmissible substitution as well as extending the theory with dependent mul-  
111 tiplicative pairs and booleans. A categorical model is also provided. Note that  
112 other additive types are only available via encoding involving booleans and func-  
113 tion types.

114 A graded dependent type system, an approach similar to QTT, is given by  
115 Choudhury et al. [8]. The key difference is that types are not forced to use the  
116 semiring zero for all variables. This change gives a finer control over resources  
117 at the type level and also places fewer restrictions on the semiring.

118 QTT forms the basis of some programming languages. Idris 2 is a purely  
119 functional, general purpose programming language developed by Brady et al. [5]  
120 and based on the zero-one-many flavor of QTT. Brady [4] provides compelling  
121 examples of combining dependent and substructural types to increase the type

122 safety of programs, such as dependent session types: a two party communication  
 123 channels that enforce a protocol at the type level.

124 QTT has been extended with dependent additive pairs and annotated elim-  
 125 inators in our previous work [14]. Multiplicity annotations help resolve a few  
 126 undesirable interactions between weakening and eliminators of the sum and pair  
 127 types. This extended theory serves as the basis of the Janus language.

## 128 3 Quantitative Type Theory

### 129 3.1 Semirings

130 QTT uses positive semirings to keep track of variable usage. A semiring is a tuple  
 131  $(S, +, \cdot, 0, 1)$  where  $S$  is a set,  $+$  and  $\cdot$  are binary operations on  $S$ ,  $0$  and  $1$  are  
 132 elements of  $S$  such that  $(S, +, 0)$  is a commutative monoid,  $(S, \cdot, 1)$  is a monoid,  
 133  $\cdot$  distributes over  $+$ , and  $0a = 0 = a0$ . A positive semiring further satisfies the  
 134 following two properties: if  $a + b = 0$  then  $a = 0$  and  $b = 0$ , if  $ab = 0$  then  $a = 0$   
 135 or  $b = 0$ .

136 Without semiring positivity, we could have variables that are used nonzero  
 137 times in two different contexts but their overall usage is still zero. Similar issue  
 138 can occur with nested contexts, which are handled by semiring multiplication.

139 In this work, we use the zero-one-many semiring. Its elements are  $0$ ,  $1$ , and  
 140  $\omega$ , which represents more than one use. The definitions of addition and multi-  
 141 plication follow from these two equations:  $\omega \cdot \omega = \omega$  and  $\forall \rho. \rho + \omega = \omega$ .

### 142 3.2 Syntax

143 An overview of the syntax of the particular QTT flavor used in this work is given  
 144 below.

$$\begin{aligned}
 \pi, \sigma &::= 0 \mid 1 \mid \omega \\
 \Gamma &::= \cdot \mid \Gamma, x \text{ ? } M \\
 M, N, O &::= x \mid \mathcal{U} \\
 &\mid \lambda x. M \mid MN \mid (x \text{ ? } M) \rightarrow N \\
 &\mid (M, N) \mid \mathbf{let}_\sigma (x, y) = M \mathbf{in} N \mid (x \text{ ? } M) \otimes N \\
 &\mid () \mid \mathbf{let} () = M \mathbf{in} N \mid \mathbf{1} \\
 &\mid \mathbf{inl} M \mid \mathbf{inr} M \mid \mathbf{case}_\sigma M \mathbf{of} \{ \mathbf{inl} x \rightarrow N; \mathbf{inr} y \rightarrow O \} \mid M \oplus N \\
 &\mid \mathbf{case} M \mathbf{of} \{ \} \mid \perp \quad (\text{no introduction}) \\
 &\mid \langle M, N \rangle \mid \mathbf{fst} M \mid \mathbf{snd} M \mid (x : M) \& N \\
 &\mid \langle \rangle \mid \top \quad (\text{no elimination})
 \end{aligned}$$

145 Going from top to bottom, we have multiplicities  $\pi, \sigma$ ; contexts  $\Gamma$ ; and terms  
 146  $M, N, O$ . When unambiguous, the empty context  $\cdot$  is omitted. Contexts can be

147 scaled and added together. Context addition is not used in this work and we  
 148 thus only define scaling by  $\pi$ :

$$\pi(\cdot) = \cdot \quad \pi(\Gamma, x \overset{\sigma}{:} M) = \pi\Gamma, x \overset{\pi\sigma}{:} M$$

149 A term can be a variable or the universe constant  $\mathcal{U}$ , representing the type of  
 150 types. Each of the remaining lines then describes introduction, elimination, and  
 151 formation (in this order) of all the types present in the system: dependent func-  
 152 tion, dependent multiplicative pair, multiplicative unit, additive sum, additive  
 153 zero, dependent additive pair, and additive unit.

154 We also need a modified typing judgment that takes into account the multi-  
 155 plicities:

$$\Gamma \vdash M \overset{\sigma}{:} N$$

156 That is, given the context  $\Gamma$ , we can show that the term  $M$  is usable  $\sigma$   
 157 times and has the type  $N$ . A key restriction is that  $\sigma$  must be either 0 or 1.  
 158 The judgment thus only communicates whether the term  $M$  can be used in a  
 159 computationally relevant context.

160 For further details, we encourage the reader to check the original presen-  
 161 tation given by Atkey [3]. However, there are some notable differences worth  
 162 emphasizing. Firstly, types and terms are not separated and instead of the **El**  
 163 decoder, we have the universe constant  $\mathcal{U}$ . The boolean type has been replaced  
 164 with three additive types: sum, unit, and pair. And finally, the multiplicative  
 165 pair and additive sum eliminators contain multiplicity annotations.

### 166 3.3 Weakening

167 As currently presented, the  $\omega$  multiplicity is only applicable when a variable is  
 168 used at least twice in a relevant context. However, we would like to associate  $\omega$   
 169 with unrestricted use. The system must be able to treat other multiplicities as  
 170  $\omega$ , which can be accomplished by adding an ordering to the semiring and using  
 171 a weakening rule, as described by McBride [12]. In particular,  $\sigma \leq \pi$  means that  
 172 a variable with  $\sigma$  uses may be treated as a variable with  $\pi$  uses.

173 Unsurprisingly, such ordering needs to be reflexive, transitive and must re-  
 174 spect the semiring operations. A suitable ordering is  $0 \leq \omega$  and  $1 \leq \omega$ . Since we  
 175 want 1 to represent linear use,  $0 \leq 1$  must not hold.

176 Contexts can also be ordered. If the contexts  $\Gamma_1$  and  $\Gamma_2$  only differ in multi-  
 177 plicities ( $0\Gamma_1 = 0\Gamma_2$ ), we define  $\Gamma_1 \leq \Gamma_2$  as a pointwise extension of the semiring  
 178 ordering  $\leq$ . In other words,  $\Gamma_1 \leq \Gamma_2$  iff the multiplicity of each variable in  $\Gamma_1$  is  
 179 less than or equal to the multiplicity of the same variable in  $\Gamma_2$ . The weakening  
 180 rule then states that if the typing judgment holds in a typing context  $\Gamma_1$ , it also  
 181 holds in any greater typing context  $\Gamma_2$ :

$$\frac{\Gamma_1 \vdash M \overset{\sigma}{:} T \quad \Gamma_1 \leq \Gamma_2}{\Gamma_2 \vdash M \overset{\sigma}{:} T} \text{WEAK}$$

### 182 3.4 Annotated Eliminators

183 While weakening works well in most cases, there is an issue with its interaction  
184 with eliminators of some types. Consider the following judgment:

$$p \multimap (- : \mathbb{N}) \otimes \mathbb{N} \vdash \mathbf{let} (x, y) = p \mathbf{in} x + x : \mathbb{N}$$

185 If we remove the multiplicities, we obtain a judgment that is valid in an  
186 intuitionistic setting. One of the goals of weakening in the system is the ability  
187 to treat  $\omega$  as unrestricted use, turning that fragment of the system into a regular  
188 intuitionistic type theory.

189 However, adding weakening does not make this judgment valid. Atkey's elim-  
190 ination rule for multiplicative pairs states that while checking the subterm  $x + x$ ,  
191 the variables  $x$  and  $y$  must be added to the typing context with multiplicity 1,  
192 which prevents  $y$  from being discarded and  $x$  from being duplicated. Notice that  
193 the weakening rule *can* be used to treat the single use of  $p$  in the eliminator as  
194  $\omega$  to match the multiplicity of  $p$  in the typing context. The problem is thus the  
195 inability of the eliminator to use  $p$  multiple times.

196 In theory, we could have only a single copy of  $p$  in the typing context and  
197 use the exponential modality for the elements of the pair, but such solution is  
198 not very flexible since the programmer must know ahead of the time where all  
199 such values will be required.

200 Instead, the eliminator is extended with a multiplicity annotation [14] which  
201 lets it consume the eliminated pair  $\omega$  times. To ensure resource correctness, this  
202 multiplicity is propagated to the freshly bound variables. At that point, the  
203 weakening rule may be applied, allowing us to treat the zero uses of  $y$  as  $\omega$  uses.  
204 The following judgment is now valid:

$$p \multimap (- : \mathbb{N}) \otimes \mathbb{N} \vdash \mathbf{let}_\omega (x, y) = p \mathbf{in} x + x : \mathbb{N}$$

205 Apart from the dependent multiplicative pair  $(x : S) \otimes T$ , this annotation  
206 may also be added to the eliminator of the additive sum  $S \oplus T$ . However, in  
207 the case of additive sums, the annotation cannot be 0 as the eliminator provides  
208 computationally relevant information.

## 209 4 Dependent Additive Pairs

210 An additive pair consists of two elements that have access to the same resources.  
211 This property can lead to seemingly unsound resource use. Consider the following  
212 judgment:

$$x : T \vdash \langle x, x \rangle : T \& T$$

213 Each element is forced to use the variable  $x$  and yet  $x$  is still considered  
214 linear. The trick lies in the elimination: only one element of the additive pair  
215 can be extracted and the other must necessarily be discarded. The end result is  
216 that the variable  $x$  is indeed used once.

217 In the original presentation of QTT, an additive pair  $S \& T$  is represented  
 218 by a function from booleans to either  $S$  or  $T$ . Similar encoding can be used in  
 219 our presentation, since booleans are a special case of the more general additive  
 220 sum type. Indeed, we can define the boolean type  $\mathbf{2}$  as  $\mathbf{1} \oplus \mathbf{1}$ . An additive pair  
 221  $S \& T$  is then defined by the following term:

$$S : \mathcal{U}, T : \mathcal{U} \vdash (b : \mathbf{2}) \rightarrow \mathbf{case}_1 b \text{ of } \{\mathbf{inl} \ t \rightarrow S; \mathbf{inr} \ f \rightarrow T\} : \mathcal{U}$$

222 Introduction and elimination follow immediately from this definition. Note  
 223 that in a computationally relevant context, the units contained in the additive  
 224 sum need to be eliminated as well. For brevity, this step is not explicitly written  
 225 out and is only implied by using  $()$  in place of the bound variable.

226 Notice that the resource use matches our expectations. The eliminator of  
 227 additive sums has a property similar to the additive pair introduction: each  
 228 branch of the case analysis has access to the same resources. Resource soundness  
 229 of the pair elimination comes from the fact that function application counts as a  
 230 use of that function. In particular, if we wish to extract both elements of such a  
 231 pair, we need to apply the function twice and thus use it twice. As an example,  
 232 we can implement an operation to swap the elements of the pair.

$$p : \mathbf{1} \ S \& T \vdash \lambda b. \mathbf{case}_1 b \text{ of } \{\mathbf{inl} \ () \rightarrow p \ (\mathbf{inr} \ ()); \mathbf{inr} \ () \rightarrow p \ (\mathbf{inl} \ ())\} : \mathbf{1} \ T \& S$$

233 However, this approach has two major downsides. Firstly, dependent type  
 234 theories often lack function extensionality principle, which complicates reasoning  
 235 about functions. Proving that two additive pairs are the same thus becomes quite  
 236 difficult. Adding function extensionality as an axiom presents other problems.  
 237 Using a different notion of equality, such as in observational type theory [2] or  
 238 homotopy type theory [16], might be possible but is out of the scope of this  
 239 work.

240 The second, bigger problem is that this encoding is incapable of expressing  
 241 type dependency between  $S$  and  $T$ . Since the two alternatives of the additive  
 242 sum type elimination are independent, neither type has access to a value of the  
 243 other type.

244 Instead of using an encoding, dependent additive pairs are built into the  
 245 system as one of its base types. Formation, introduction and elimination are  
 246 defined by the following rules:

$$\begin{array}{c} \frac{0\Gamma \vdash S : \mathcal{U} \quad 0\Gamma, x : S \vdash T : \mathcal{U}}{0\Gamma \vdash (x : S) \& T : \mathcal{U}} \&-F \qquad \frac{\Gamma \vdash M : S \quad \Gamma \vdash N : T[M/x]}{\Gamma \vdash \langle M, N \rangle : (x : S) \& T} \&-I \\[10pt] \frac{\Gamma \vdash M : (x : S) \& T}{\Gamma \vdash \mathbf{fst} \ M : S} \&-E_1 \qquad \frac{\Gamma \vdash M : (x : S) \& T}{\Gamma \vdash \mathbf{snd} \ M : T[\mathbf{fst} \ M/x]} \&-E_2 \end{array}$$

247 In the dependent additive pair  $(x : S) \& T$ , the type  $T$  can refer to the value  
 248 of the first element via the variable  $x$ . As expected, the introduction rule gives  
 249 both elements of the pair access to the entire context  $\Gamma$ .

250 Notice that the second elimination rule uses both **fst**  $M$  and **snd**  $M$ , seem-  
 251 ingly resulting in unsound resource use. However, because the first element is  
 252 accessed in a computationally irrelevant context, resource soundness is not af-  
 253 fected. The behavior of these eliminators is as follows:

$$\begin{aligned} \mathbf{fst} \langle M, N \rangle &\rightsquigarrow M \\ \mathbf{snd} \langle M, N \rangle &\rightsquigarrow N \end{aligned}$$

## 254 5 Programming with Dependent Additive Pairs

255 In this section, we discuss three distinct scenarios that benefit from the use  
 256 of dependent additive pairs. We restrict ourselves to linear uses of these pairs.  
 257 Indeed, in the presence of weakening, additive and multiplicative pairs are mostly  
 258 interchangeable.

259 Each definition found in this section is also implemented in Janus. Janus is  
 260 a language based on the extended QTT mentioned in Section 3. It comes with  
 261 a type checker and an interactive evaluator, which are implemented in Haskell  
 262 and their source code is available online [15]. It is provided as a Cabal package  
 263 and can be built and run with any recent version of **ghc** and **cabal**.

264 The examples are provided as **.jns** files and are available online.<sup>1</sup> These  
 265 files can be loaded into Janus with the **:load** command, which performs type  
 266 checking and adds the new definitions to the context. The user may then evaluate  
 267 them or query their type using the **:type** command. If querying a type is not  
 268 sufficient, the **.jns** files contain very detailed type information.

### 269 5.1 Linear Folds

270 In functional programming, operations that eliminate values of recursively de-  
 271 fined data types are typically called *folds*. Consider the case of a singly-linked  
 272 list type: **List**. The constant **Nil** represents an empty list, **Cons** introduces a  
 273 non-empty list. We can define a simple fold operation with these two equations:

$$\begin{aligned} \mathbf{fold} \ f \ z \ \mathbf{Nil} &= z \\ \mathbf{fold} \ f \ z \ (\mathbf{Cons} \ x \ xs) &= f \ x \ (\mathbf{fold} \ f \ z \ xs) \end{aligned}$$

274 To borrow a term from the category theory, these simple folds are called  
 275 *catamorphisms*. In more technical terms, a catamorphism is a unique homomor-  
 276 phism out of an initial algebra. Since we are in a dependent setting, we would  
 277 like to give this operation a fully dependent type. If we replace the return type  
 278 with a dependent *motive*  $P : \mathbf{List} \ A \rightarrow \mathcal{U}$ , we obtain the following:

$$\begin{aligned} A : \mathcal{U}, P : \mathbf{List} \ A \rightarrow \mathcal{U} \vdash \mathbf{fold} : (f : (x : A) \rightarrow (r : P \ ?) \rightarrow P \ (\mathbf{Cons} \ x \ ?)) \rightarrow \\ (z : P \ \mathbf{Nil}) \rightarrow (l : \mathbf{List} \ A) \rightarrow P \ l \end{aligned}$$

---

<sup>1</sup> <https://github.com/vituscze/dependent-additive-pairs>



279 However, we are unable to specify the type of  $f$ . The type of  $r$  as well as  
 280 the type of the result need to mention the list  $xs$ , which is not available at this  
 281 point. The only way to solve this problem is to add an additional parameter to  
 282  $f$ . We get different versions of **fold** depending on how this extra parameter is  
 283 used. Since we are also in a substructural setting, we can specify and enforce this  
 284 usage. Let us analyze the previous equations to see how the other parameters  
 285 are used.

286 We can see that if the function  $f$  consumes the elements and the recursive  
 287 results linearly, the whole list is also consumed linearly. The value  $z$  is also used  
 288 exactly once. The only input that is not used linearly is the function  $f$  itself,  
 289 which can be used any number of times, including zero. However, thanks to  
 290 weakening, the  $\omega$  multiplicity can be used to describe this usage. In this case,  
 291 the additional parameter is not used in a relevant context. We obtain a resource-  
 292 aware version of the previous type.

$$\begin{aligned} A \multimap \mathcal{U}, P \multimap (l \multimap \mathbf{List} A) \rightarrow \mathcal{U} \vdash \mathbf{fold} \multimap \\ (f \multimap (x \multimap A) \rightarrow (xs \multimap \mathbf{List} A) \rightarrow (r \multimap P xs) \rightarrow P (\mathbf{Cons} x xs)) \rightarrow \\ (z \multimap P \mathbf{Nil}) \rightarrow (l \multimap \mathbf{List} A) \rightarrow P l \end{aligned}$$

293 If we allow the function  $f$  to use the additional parameter, we obtain a  
 294 *paramorphism*. More generally, a paramorphism is a generalized catamorphism  
 295 which allows the combining function access to both the recursive result and the  
 296 remaining structure. We can again express this fact using two equations:

$$\begin{aligned} \mathbf{para} f z \mathbf{Nil} &= z \\ \mathbf{para} f z (\mathbf{Cons} x xs) &= f x (xs, \mathbf{para} f z xs) \end{aligned}$$

297 In a substructural setting, we have an additional decision to make regarding  
 298 the function  $f$ . If  $f$  uses both elements of the pair, we need to use a multiplicative  
 299 pair. In this case, the variable  $xs$  is used twice and thus the list cannot be  
 300 consumed linearly. If  $f$  uses exactly one of the elements of the pair, we need to  
 301 use an additive pair. The variable  $xs$  is now used exactly once and the list can  
 302 be consumed linearly. However, we cannot guarantee that the value  $z$  is used  
 303 once. We obtain the following type:

$$\begin{aligned} A \multimap \mathcal{U}, P \multimap (l \multimap \mathbf{List} A) \rightarrow \mathcal{U} \vdash \mathbf{para} \multimap \\ (f \multimap (x \multimap A) \rightarrow (p \multimap (xs : \mathbf{List} A) \& P xs) \rightarrow P (\mathbf{Cons} x (\mathbf{fst} p))) \rightarrow \\ (z \multimap P \mathbf{Nil}) \rightarrow (l \multimap \mathbf{List} A) \rightarrow P l \end{aligned}$$

304 These linear paramorphisms are useful whenever only a part of the structure  
 305 needs to be traversed. Examples include various insertion and deletion oper-  
 306 ations. A deletion operation can be linear as long as the deleted element is  
 307 returned alongside the rest of the structure.

308 The original **fold** can be implemented using **para** quite easily. However, since  
 309 catamorphisms are also the eliminators of inductive types, it should be possible  
 310 to implement **para** in terms of **fold**. Since the combining function does not

311 have access to the rest of the list, it has to reconstruct it. The reconstructed  
 312 list and recursive result are stored in an additive pair. The following definition  
 313 demonstrates the desired semantics:

$$\mathbf{para} = \lambda f \ z \ l. \mathbf{snd} \ (\mathbf{fold} \ (\lambda x \ xs \ p. \langle \mathbf{Cons} \ x \ (\mathbf{fst} \ p), f \ x \ p \rangle) \ \langle \mathbf{Nil}, z \rangle \ l)$$

314 However, this definition requires some changes to satisfy the type checker. We  
 315 have two choices for the motive  $P$ :  $\lambda \_ . (l' : \mathbf{List} \ A) \& P \ l'$ , or  $\lambda l. (\_ : \mathbf{List} \ A) \& P \ l$ .  
 316 The first one fails when applying the final **snd** because the first element is *not*  
 317 the list  $l$ . The second one fails when applying  $f$  because  $xs$  and **fst**  $p$  are different  
 318 lists. In both cases, the type checker is not convinced that the original and the  
 319 reconstructed list are the same.

320 The motive offers us a hint. Notice we are either ignoring the lambda param-  
 321 eter or the first element of the pair. However, to use both of these, we will need  
 322 an identity type. In particular, we need the following constants for the formation  
 323 and introduction:

$$\begin{aligned} A : \mathcal{U}, x : A, y : A \vdash x \equiv y : \mathcal{U} \\ A : \mathcal{U} \vdash \mathbf{refl} : (x : A) \rightarrow x \equiv x \end{aligned}$$

324 We use *based path induction* [13] as the eliminator.

$$\begin{aligned} A : \mathcal{U}, x : A, y : A \vdash \mathbf{J} : (P : (y' : A) \rightarrow (\_ : x \equiv y') \rightarrow \mathcal{U}) \rightarrow \\ (f : P \ x \ (\mathbf{refl} \ x)) \rightarrow (p : x \equiv y) \rightarrow P \ y \ p \end{aligned}$$

325 The new motive contains the additive pair and the proof that the original  
 326 and reconstructed list are the same:

$$\mathbf{Triple} = \lambda l. (p : (l' : \mathbf{List} \ A) \& P \ l') \otimes (\mathbf{fst} \ p \equiv l)$$

327 Of course, the fold now needs to construct this proof as it goes and then use it  
 328 at the very end. The latter can be accomplished by using the identity  $l' \equiv l$  to  
 329 rewrite the type of the result from  $P \ l'$  to  $P \ l$ . The following function uses the  
 330 proof contained in **Triple**  $l$  to produce  $P \ l$ .

$$\mathbf{extract} = \lambda t. \mathbf{let}_1 \ (p, q) = t \ \mathbf{in} \ \mathbf{J} \ (\lambda l' \_ . P \ l') \ (\mathbf{snd} \ p) \ q$$

331 For the former, we start with the proof  $\mathbf{Nil} \equiv \mathbf{Nil}$ . The inductive step asks  
 332 us to prove  $\mathbf{Cons} \ x \ (\mathbf{fst} \ p) \equiv \mathbf{Cons} \ x \ l$  given  $\mathbf{fst} \ p \equiv l$ , which is accomplished by  
 333 using the congruence property. That is, if  $p : x \equiv y$  then  $\mathbf{cong} \ f \ p : f \ x \equiv f \ y$ .  
 334 Putting it all together, we obtain the following:

$$\begin{aligned} \mathbf{para} = \lambda f \ z \ l. \mathbf{extract} \ (\mathbf{fold} \ (\lambda x \ xs \ t. \mathbf{let}_1 \ (p, q) = t \ \mathbf{in} \\ (\langle \mathbf{Cons} \ x \ (\mathbf{fst} \ p), f \ x \ p \rangle, \mathbf{cong} \ (\lambda l. \mathbf{Cons} \ x \ l) \ q)) \ (\langle \mathbf{Nil}, z \rangle, \mathbf{refl} \ \mathbf{Nil}) \ l) \end{aligned}$$

335 This definition is now correct and matches the type given earlier. The type of  
 336 **para** is thus justified. Note that in practice, paramorphisms would not be defined

337 in terms of catamorphisms, as the need to reconstruct the structure removes one  
 338 of their main benefits.

339 While we focused on list paramorphisms here, this definition can be general-  
 340 ized to any tree-like structure. For example, the combining function for a binary  
 341 tree paramorphism would have the following type:

$$(x : A) \rightarrow (l : (t : \mathbf{Tree} A) \& P t) \rightarrow (r : (t : \mathbf{Tree} A) \& P t) \rightarrow \\ P (\mathbf{Node} x (\mathbf{fst} l) (\mathbf{fst} r))$$

## 342 5.2 Resource-Aware Proofs

343 Dependent type theories use dependent pairs to express existential quantifica-  
 344 tion. The first element of such a pair is typically called a *witness*, as it is a value  
 345 that witnesses the inhabitation of the type of the second element.

346 In some cases, the witness can be computationally relevant. This kind of  
 347 witness is typically found in operations that prove some correctness properties  
 348 as their output. There are also cases where the witness is mainly used to specify  
 349 the type of the second element, even though it might carry computationally  
 350 relevant information itself. This use case can be found whenever the indices of  
 351 dependent types cannot (or should not) be specified.

352 In the intersection of these two cases are operations that compute a relevant  
 353 witness *and* a relevant dependent value that hides one or more of its indices.  
 354 Consider a **filter** operation on vectors. Instead of using a natural number as the  
 355 witness, we could use an entire **List**.

$$A : \mathcal{U}, n : \mathbb{N} \vdash \mathbf{filter} : (p : A \rightarrow \mathbf{2}) \rightarrow (xs : \mathbf{Vec} A n) \rightarrow \\ (l : \mathbf{List} A) \times \mathbf{Vec} A (\mathbf{length} l)$$

356 The list and the vector have the same length. But since the length of the  
 357 list is not a part of its type, the length of the vector is still effectively hidden.  
 358 However, the user has a much more interesting choice: if the hidden index is no  
 359 longer necessary, the witness still carries all the useful information.

360 **Partition** The **filter** example does not quite fit into our substructural setting.  
 361 The input list cannot be used linearly since its elements might be discarded.  
 362 Instead of using a different example, we will adjust the **filter** operation as it  
 363 allows us to demonstrate a couple of useful techniques.

364 First of all, if we want the operation to be linear, it also needs to return the  
 365 elements that have been filtered out, ideally in a separate list. Such operation is  
 366 sometimes called **partition**. Since the lists contain different elements, they need  
 367 to be in a multiplicative pair.

$$A : \mathcal{U} \vdash \mathbf{partition} : (p : (a : A) \rightarrow \mathbf{2}) \rightarrow (l : \mathbf{List} A) \rightarrow \\ (\_ : \mathbf{List} A) \otimes \mathbf{List} A$$

368 The first problem we encounter is that applying the predicate  $p$  consumes the  
 369 element of the list, leaving us with nothing to put into the result. Changing the  
 370 multiplicity of the first parameter to zero would solve this issue, but predicates  
 371 that are not allowed to inspect their input are generally not useful.

372 Instead, we require the predicate to also return a new version of the input,  
 373 such as  $p \dot{\vdash} (a \dot{\vdash} A) \rightarrow (- \dot{\vdash} \mathbf{2}) \otimes A$ . With this change, implementing **partition** is  
 374 easy. Now, suppose we want to also return a description of the resulting partition.  
 375 We can use the following type:

$$\begin{aligned}
 & A \dot{\vdash} \mathcal{U}, xs \ ys \ zs \dot{\vdash} \mathbf{List} \ A \vdash \mathbf{Union} \ xs \ ys \ zs \dot{\vdash} \mathcal{U} \\
 & A \dot{\vdash} \mathcal{U}, xs \ ys \ zs \dot{\vdash} \mathbf{List} \ A \vdash \mathbf{Left} \dot{\vdash} (x \dot{\vdash} A) \rightarrow (u \dot{\vdash} \mathbf{Union} \ xs \ ys \ zs) \rightarrow \\
 & \quad \mathbf{Union} \ (\mathbf{Cons} \ x \ xs) \ ys \ (\mathbf{Cons} \ x \ zs) \\
 & A \dot{\vdash} \mathcal{U}, xs \ ys \ zs \dot{\vdash} \mathbf{List} \ A \vdash \mathbf{Right} \dot{\vdash} (x \dot{\vdash} A) \rightarrow (u \dot{\vdash} \mathbf{Union} \ xs \ ys \ zs) \rightarrow \\
 & \quad \mathbf{Union} \ xs \ (\mathbf{Cons} \ x \ ys) \ (\mathbf{Cons} \ x \ zs) \\
 & A \dot{\vdash} \mathcal{U} \vdash \mathbf{Stop} \dot{\vdash} \mathbf{Union} \ \mathbf{Nil} \ \mathbf{Nil} \ \mathbf{Nil}
 \end{aligned}$$

376 **Union**  $xs \ ys \ zs$  is a proof that the lists  $xs$  and  $ys$  can be interleaved to obtain  
 377 the list  $zs$ . The introductions **Left** and **Right** are used to express whether the  
 378 first element of the result came from the first or the second list. The elimination  
 379 is left out as it will not be needed in this case. Since the type of the result is  
 380 quite large, it might be useful to split it into a couple of auxiliary definitions.

$$\begin{aligned}
 \mathbf{Result}_1 &= \lambda A. (- \dot{\vdash} \mathbf{List} \ A) \otimes \mathbf{List} \ A \\
 \mathbf{Result}_2 &= \lambda A \ zs \ r_1. \mathbf{let}_0 \ (xs, ys) = r_1 \ \mathbf{in} \ \mathbf{Union} \ xs \ ys \ zs \\
 \mathbf{Result} &= \lambda A \ zs. (r_1 : \mathbf{Result}_1 \ A) \ \& \ \mathbf{Result}_2 \ A \ zs \ r_1 \\
 \mathbf{Pred} &= \lambda A. (x \dot{\vdash} A) \rightarrow (- \dot{\vdash} \mathbf{2}) \otimes A
 \end{aligned}$$

381 With that, we can state the full type of the **partition** operation as follows:

$$A \dot{\vdash} \mathcal{U} \vdash \mathbf{partition} \dot{\vdash} (p \dot{\vdash} \mathbf{Pred} \ A) \rightarrow (l \dot{\vdash} \mathbf{List} \ A) \rightarrow \mathbf{Result} \ A \ l$$

382 Since the type of the result depends on the input list, we need to use the  
 383 dependent **fold** defined earlier. The base case has the type **Result**  $A \ \mathbf{Nil}$  and  
 384 only has a single valid value:  $\langle (\mathbf{Nil}, \mathbf{Nil}), \mathbf{Stop} \rangle$ . The inductive case can be broken  
 385 down into three steps. Firstly, we define auxiliary functions that add the new  
 386 element to one of the **Result**<sub>1</sub> lists.

$$\begin{aligned}
 \mathbf{add}'_l &= \lambda x \ r_1. \mathbf{let}_1 \ (l, r) = r_1 \ \mathbf{in} \ (\mathbf{Cons} \ x \ l, r) \\
 \mathbf{add}'_r &= \lambda x \ r_1. \mathbf{let}_1 \ (l, r) = r_1 \ \mathbf{in} \ (l, \mathbf{Cons} \ x \ r)
 \end{aligned}$$

387 Secondly, we use these definitions to add the new elements to the whole  
 388 **Result**. However, since the first element does not reduce to a pair, the type  
 389 of the second element remains some form of **Result**<sub>2</sub>, rather than reducing to  
 390 **Union**. We can fix this by inspecting the first element to force reduction.

$$\mathbf{add}_1 = \lambda x \ r. \langle \mathbf{add}'_l \ x \ (\mathbf{fst} \ r), \mathbf{let}_0 \ (-, -) = \mathbf{fst} \ r \ \mathbf{in} \ \mathbf{Left} \ x \ (\mathbf{snd} \ r) \rangle$$

391 However, this definition has a problem similar to the one before. This time  
 392 it is the term **snd**  $r$  whose type does not reduce. Without access to dependent  
 393 pattern matching [9], we need to eliminate into a function type. That way, the  
 394 type of the argument reduces and can be given to **Left**. The resulting function  
 395 is then applied to **snd**  $r$ .

$$\begin{aligned}\mathbf{add}_l &= \lambda x r. \langle \mathbf{add}'_l x (\mathbf{fst} r), (\mathbf{let}_0 (-, -) = \mathbf{fst} r \text{ in } \lambda r_2. \mathbf{Left} x r_2) (\mathbf{snd} r) \rangle \\ \mathbf{add}_r &= \lambda x r. \langle \mathbf{add}'_r x (\mathbf{fst} r), (\mathbf{let}_0 (-, -) = \mathbf{fst} r \text{ in } \lambda r_2. \mathbf{Right} x r_2) (\mathbf{snd} r) \rangle\end{aligned}$$

396 We can easily check that all auxiliary definitions have the expected types:

$$\begin{aligned}\dots \vdash \mathbf{add}'_l \mathbf{add}'_r : (x : A) \rightarrow (r_1 : \mathbf{Result}_1 A) \rightarrow \mathbf{Result}_1 A \\ \dots \vdash \mathbf{add}_l \mathbf{add}_r : (x : A) \rightarrow (r : \mathbf{Result} A \text{ } zs) \rightarrow \mathbf{Result} A (\mathbf{Cons} x zs)\end{aligned}$$

397 Finally, we can apply the predicate and then use **add<sub>l</sub>** or **add<sub>r</sub>** depending  
 398 on the result.

$$\begin{aligned}\mathbf{step} &= \lambda p x r. \mathbf{let}_1 (b, x') = p x \text{ in} \\ &\quad \mathbf{case}_1 b \text{ of } \{\mathbf{inl} () \rightarrow \mathbf{add}_l x' r; \mathbf{inr} () \rightarrow \mathbf{add}_r x' r\}\end{aligned}$$

399 However, both **add<sub>l</sub>**  $x' r$  and **add<sub>r</sub>**  $x' r$  produce **Result**  $A (\mathbf{Cons} x' xs)$  as  
 400 we were forced to use  $x'$ , which does not match **Result**  $A (\mathbf{Cons} x xs)$  required  
 401 by **fold**. The problem is that the predicate is allowed to return any value and  
 402 thus we cannot assume that  $x$  and  $x'$  are the same. We can force it to return the  
 403 same value by adding the identity type to the definition of **Pred**.

$$\mathbf{Pred} = \lambda A. (x : A) \rightarrow (- : \mathbf{2}) \otimes (x' : A) \otimes (x' \equiv x)$$

404 The proof can be extracted with a second **let** term. It can then be used to  
 405 rewrite the type of the result, which is done by using the substitutivity of the  
 406 identity type. In particular, if  $v : P x$  and  $p : x \equiv y$  then **subst**  $P p v : P y$ . We  
 407 can now fix the **step** function.

$$\begin{aligned}\mathbf{step} &= \lambda p x r. \mathbf{let}_1 (b, s) = p x \text{ in } \mathbf{let}_1 (x', q) = s \text{ in} \\ &\quad \mathbf{subst} (\lambda x. \mathbf{Result} A (\mathbf{Cons} x xs)) q \\ &\quad (\mathbf{case}_1 b \text{ of } \{\mathbf{inl} () \rightarrow \mathbf{add}_l x' r; \mathbf{inr} () \rightarrow \mathbf{add}_r x' r\})\end{aligned}$$

408 And finally, we can put it all together to implement the **partition** operation  
 409 itself.

$$\mathbf{partition} = \lambda p l. \mathbf{fold} (\mathbf{step} p) \langle (\mathbf{Nil}, \mathbf{Nil}), \mathbf{Stop} \rangle l$$

410 It should be noted that the reduction behavior of **Result<sub>2</sub>** might not be  
 411 desirable in some situations. Even though it will eventually reduce to a **Union**,  
 412 the type checker cannot see that without eliminating the pair first. In cases like  
 413 this, it is generally recommended to move the computation to the indices of the

414 type. We can define projections **Fst** and **Snd** for the multiplicative pair that  
 415 may be used in types. We can then define another version of **Result<sub>2</sub>**.

$$\mathbf{Result}_2 = \lambda A \ zs \ r_1. \mathbf{Union} \ (\mathbf{Fst} \ r_1) \ (\mathbf{Snd} \ r_1) \ zs$$

416 We have implemented **partition** for both versions, but here we only present  
 417 the one that does not require additional auxiliary definitions to function.

418 **Insertion** As mentioned previously, many insertion operations may be imple-  
 419 mented by using a paramorphism. We can reuse the **Union** type to implement  
 420 a sorted list insertion operation that also produces a resource-aware proof. In  
 421 particular, if the list  $l$  is a result of inserting a new element  $x$  into the list  $xs$ ,  
 422 we expect **Union**  $xs$  (**Cons**  $x$  **Nil**)  $l$  to hold. We shall abbreviate **Cons**  $x$  **Nil**  
 423 as  $[x]$ . We begin with a couple of auxiliary definitions.

$$\begin{aligned} \mathbf{Result} &= \lambda A \ x \ xs. (l : \mathbf{List} \ A) \ \& \ \mathbf{Union} \ xs \ [x] \ l \\ \mathbf{Cmp} &= \lambda A. (x : A) \rightarrow (y : A) \rightarrow \\ &\quad (b : \mathbf{2}) \otimes (x' : A) \otimes (y' : A) \otimes (x' \equiv x) \otimes (y' \equiv y) \end{aligned}$$

424 As before, we use the identity type to make sure the comparison function re-  
 425 turns the same values it was given. We can now state the full type of a dependent  
 426 **insert** operation.

$$A : \mathcal{U} \vdash \mathbf{insert} : (c : \mathbf{Cmp} \ A) \rightarrow (x : A) \rightarrow (xs : \mathbf{List} \ A) \rightarrow \mathbf{Result} \ A \ x \ xs$$

427 The base case is seemingly trivial: we only need to insert  $x$  into the empty list.  
 428 However, because **para** requires the base case to have multiplicity  $\omega$ , simply using  
 429  $[x]$  would require  $x : A$ . We instead eliminate into a linear function. The function  
 430 can be discarded and thus the  $\omega$  multiplicity is not a problem. In particular, we  
 431 use the following motive:

$$A : \mathcal{U} \vdash \lambda xs. (x : A) \rightarrow \mathbf{Result} \ A \ x \ xs : (xs : \mathbf{List} \ A) \rightarrow \mathcal{U}$$

432 The base case is then trivial.

$$\mathbf{base} = \lambda x. \langle [x], \mathbf{Right} \ x \ \mathbf{Stop} \rangle$$

433 The inductive case is more interesting. If the inserted element  $x$  is smaller  
 434 than or equal to  $y$  (according to the comparison function), we have found the in-  
 435 sersion point and no further recursion is necessary. Recall that a paramorphism  
 436 gives us access to an additive pair  $r$  containing the rest of the list and the recur-  
 437 sive result. We ignore the recursive result and return **Cons**  $x$  (**Cons**  $y$  (**fst**  $r$ )).  
 438 We also need to construct a proof of the following type:

$$\mathbf{Union} \ (\mathbf{Cons} \ y \ (\mathbf{fst} \ r)) \ [x] \ (\mathbf{Cons} \ x \ (\mathbf{Cons} \ y \ (\mathbf{fst} \ r)))$$

439 Clearly,  $x$  must have come from the **Right** list. We then need a simple lemma  
 440 to show that **Union**  $l$  **Nil**  $l$  holds for any  $l$ . The proof consists of a **Left** for each  
 441 element of  $l$  and a **Stop** at the end.

$$\text{lem} = \lambda l. \text{fold } (\lambda x \text{ xs } r. \text{Left } x \text{ } r) \text{ Stop } l$$

442 Putting it all together, we obtain the **done** function that handles the non-  
 443 recursive case.

$$\text{done} = \lambda x \text{ y } r. \langle \text{Cons } x (\text{Cons } y (\text{fst } r)), \text{Right } x (\text{lem } (\text{Cons } y (\text{fst } r))) \rangle$$

444 If  $x$  is greater than  $y$ , we must recursively insert  $x$  into the sublist by using  
 445 the second element of the additive pair, giving us a new list and also a proof.

$$\dots \vdash \text{snd } r \text{ } x \dot{=} (l : \text{List } A) \ \& \ \text{Union } (\text{fst } r) \ [x] \ l$$

446 Of course, we need to add the element  $y$  back to the list and return the  
 447 list **Cons**  $y$  (**fst** (**snd**  $r \text{ } x$ )). Additionally, we need to construct a proof of the  
 448 following type:

$$\text{Union } (\text{Cons } y (\text{fst } r)) \ [x] \ (\text{Cons } y (\text{fst } (\text{snd } r \text{ } x)))$$

449 The element  $y$  must have come from the **Left** list this time. The remain-  
 450 ing proof obligation is satisfied by using the second element of (**snd**  $r$ )  $x$ , the  
 451 induction hypothesis. The following **go** function handles the recursive case:

$$\text{go} = \lambda x \text{ y } r. \langle \text{Cons } y (\text{fst } (\text{snd } r \text{ } x)), \text{Left } y (\text{snd } (\text{snd } r \text{ } x)) \rangle$$

452 Combining the functions **done** and **go** gives us a single step of the insertion.  
 453 Note that we use a shortcut to represent the use of four **let**<sub>1</sub> eliminations required  
 454 to unpack the result of the comparison  $c \text{ } x \text{ } y$ .

$$\begin{aligned} \text{step} &= \lambda c \text{ y } r \text{ } x. \text{let}_1 (b, (x', (y', (p_x, p_y)))) = c \text{ } x \text{ } y \text{ in} \\ &\quad \text{case}_1 b \text{ of } \{\text{inl } () \rightarrow \text{done } x' \text{ } y' \text{ } r; \text{inr } () \rightarrow \text{go } x' \text{ } y' \text{ } r\} \end{aligned}$$

455 The result of this function has the type **Result**  $A \text{ } x' (\text{Cons } y' (\text{fst } r))$ , which  
 456 does not match the type required by **para**. As before, we use the proofs  $p_x$  and  
 457  $p_y$  to rewrite this type. However, we now need to use the **subst** operation twice.

$$\begin{aligned} \text{step} &= \lambda c \text{ y } r \text{ } x. \text{let}_1 (b, (x', (y', (p_x, p_y)))) = c \text{ } x \text{ } y \text{ in} \\ &\quad \text{subst } (\lambda x. \text{Result } A \text{ } x (\text{Cons } y (\text{fst } r))) \text{ } p_x \\ &\quad (\text{subst } (\lambda y. \text{Result } A \text{ } x' (\text{Cons } y (\text{fst } r))) \text{ } p_y \\ &\quad (\text{case}_1 b \text{ of } \{\text{inl } () \rightarrow \text{done } x' \text{ } y' \text{ } r; \text{inr } () \rightarrow \text{go } x' \text{ } y' \text{ } r\})) \end{aligned}$$

458 And finally, we have everything needed to define the dependent **insert** oper-  
 459 ation itself.

$$\text{insert} = \lambda c \text{ } x \text{ } xs. \text{para } (\text{step } c) \text{ base } xs \text{ } x$$

460 Notice that the linearity of **insert** provides some guarantees for free. In  
 461 particular, we know that the value  $x$  must be present in the list. Similarly, none of  
 462 the elements of the original list could be discarded or duplicated. The computed  
 463 proof makes these guarantees explicit, allowing their further use in other proofs.  
 464 Additionally, it shows that the insertion does not change the relative positions  
 465 of the original elements.

### 466 5.3 Inductive and Coinductive Types

467 So far we have seen additive pairs used with other data types. Let us consider  
 468 what happens when these pairs are used to define a data type. Going back to  
 469 the **List** type, we can see that its definition does not explicitly mention pairs.  
 470 However, inductive types can be represented as least fixed points. **List**  $A$  is the  
 471 least fixed point of the following type function:

$$\mathbf{ListF} = \lambda X. \mathbf{1} \oplus (- \multimap A) \otimes X$$

472 This representation reveals the implicit use of a pair type. If we replace the  
 473 multiplicative pair with an additive pair and compute the new fixed point we  
 474 obtain the following type:

$$\begin{aligned} A \multimap \mathcal{U} \vdash \mathbf{List}^+ A \multimap \mathcal{U} \\ A \multimap \mathcal{U} \vdash \mathbf{Nil}^+ \multimap \mathbf{List}^+ A \\ A \multimap \mathcal{U}, p \multimap (- : A) \& \mathbf{List}^+ A \vdash \mathbf{Cons}^+ p \multimap \mathbf{List}^+ A \end{aligned}$$

475 If additive pairs represent a choice between two resources, an additive list  
 476 represents a choice between  $n$  resources, where  $n$  is not known ahead of time.  
 477 However, before we even attempt to define the eliminator, we quickly run into  
 478 an issue. In a linear context, we cannot create a list with a single element.

$$A \multimap \mathcal{U}, x \multimap A \not\vdash \mathbf{Cons}^+ \langle x, \mathbf{Nil}^+ \rangle \multimap \mathbf{List}^+ A$$

479 The second element of the pair discards  $x$ . We could fix it by also changing  
 480 the multiplicative unit  $\mathbf{1}$  to the additive unit  $\top$ . However, we would be treating  
 481 symptoms rather than the cause, which is that a choice between zero resources  
 482 does not make sense. We therefore want nonempty lists, which can be accom-  
 483 plished by replacing  $\mathbf{Nil}^+$  with  $\mathbf{Last}^+$ .  $\mathbf{Last}^+ x$  represents a list with a single  
 484 element  $x$ . Let us analyze the behavior of the eliminator so that we can assign  
 485 the correct multiplicities.

$$\begin{aligned} \mathbf{fold}^+ f z (\mathbf{Last}^+ x) &= z x \\ \mathbf{fold}^+ f z (\mathbf{Cons}^+ p) &= f \langle \mathbf{fst} p, \mathbf{fold}^+ f z (\mathbf{snd} p) \rangle \end{aligned}$$

486 The function  $f$  is discarded in one case and duplicated in the other and thus  
 487 needs the  $\omega$  multiplicity. However, the function  $z$  also needs  $\omega$ , as it is discarded  
 488 in the second case.

$$\begin{aligned} A \multimap \mathcal{U}, P \multimap \mathcal{U} \vdash \mathbf{fold}^+ \multimap (f \multimap (- \multimap (- : A) \& P) \rightarrow P) \rightarrow \\ (z \multimap (- \multimap A) \rightarrow P) \rightarrow (l \multimap \mathbf{List}^+ A) \rightarrow P \end{aligned}$$



489 Additive lists admit some common list operations, such as **map**<sup>+</sup>. However,  
 490 unlike the normal **map**, we can guarantee that the mapped function is used  
 491 linearly. Notice that since the combining function given to **fold**<sup>+</sup> is used  $\omega$  times,  
 492 referencing the mapped function inside it would not count as linear use. Instead,  
 493 we eliminate into a function type and thread the mapped function through the  
 494 entire fold. That is, instead of the usual type **List**  $B$  we eliminate into the type  
 495  $(f : (- : A) \rightarrow B) \rightarrow \mathbf{List} B$ . The result is then applied to the mapped function.

$$\begin{aligned} \mathbf{map}^+ &= \lambda f l. \mathbf{fold}^+ (\lambda p f. \mathbf{Cons}^+ \langle f (\mathbf{fst} p), \mathbf{snd} p f \rangle) \\ &\quad (\lambda x f. \mathbf{Last}^+ (f x)) l f \end{aligned}$$

496 A linear operation that can be implemented on additive lists but not normal  
 497 lists is **replicate**. Since we are using nonempty lists, we need to ensure that  
 498 **replicate** is not used with zero, or generate a list one element longer. We also  
 499 need to define natural numbers, though for this example only the non-dependent  
 500 eliminator **rec** with the usual semantics will suffice.

$$P : \mathcal{U} \vdash \mathbf{rec} : (f : (- : P) \rightarrow P) \rightarrow (z : P) \rightarrow (n : \mathbb{N}) \rightarrow P$$

501 Like before, the replicated value needs to be threaded through, this time in  
 502 an additive pair.

$$\begin{aligned} \mathbf{replicate} &= \lambda n x. \mathbf{fst} (\mathbf{rec} (\lambda p. \langle \mathbf{Cons}^+ \langle \mathbf{snd} p, \mathbf{fst} p \rangle, \mathbf{snd} p \rangle) \\ &\quad \langle \mathbf{Last}^+ x, x \rangle n) \end{aligned}$$

503 Many linear operations that can be implemented only on additive lists gener-  
 504 ate the list from a single seed value. Types that are defined using these unfolding  
 505 operations are called coinductive types. As the name suggests, they are dual to  
 506 inductive types. If an inductive type corresponds to a least fixed point, a coin-  
 507 ductive type corresponds to a greatest fixed point. Values of such types are  
 508 potentially infinite.

509 While inductive definitions need to exhibit termination, the same cannot be  
 510 required of coinductive definitions, which may produce infinite values and thus  
 511 do not always terminate. Instead, coinductive definitions need to exhibit *pro-*  
 512 *ductivity*. A productive definition always produces a new piece of the final value  
 513 after a finite amount of time. Coinductive types thus naturally lend themselves  
 514 to describing processes that always progress but might not terminate, such as a  
 515 Turing machine simulation.

516 In a linear setting, an infinite value makes sense only if each of its elements  
 517 uses the same finite resources. Coinductive types thus naturally lend themselves  
 518 to definitions using additive pairs. Let us consider infinite **Streams** as an exam-  
 519 ple. When defining a coinductive type, the eliminators are regarded as primary.

$$\begin{aligned} A : \mathcal{U} \vdash \mathbf{Stream} A : \mathcal{U} \\ A : \mathcal{U}, s : \mathbf{Stream} A \vdash \mathbf{head} s : A \\ A : \mathcal{U}, s : \mathbf{Stream} A \vdash \mathbf{tail} s : \mathbf{Stream} A \end{aligned}$$

Introduction is defined by specifying what happens when an eliminator is applied to it. Just like the behavior of eliminators of inductive types can be expressed using pattern matching, we can express the behavior of this introduction using copattern matching [1].

$$\begin{aligned}\mathbf{head} (\mathbf{unfold} \ f \ s) &= \mathbf{fst} \ (f \ s) \\ \mathbf{tail} (\mathbf{unfold} \ f \ s) &= \mathbf{unfold} \ f \ (\mathbf{snd} \ (f \ s))\end{aligned}$$

No matter which eliminator is used, the seed value  $s$  is used exactly once. The generating function  $f$  is duplicated in the second case and thus needs to have the  $\omega$  multiplicity. We obtain the following type for the introduction:

$$S : \mathcal{U}, A : \mathcal{U} \vdash \mathbf{unfold} : (f : (- : A) \& S) \rightarrow (s : S) \rightarrow \mathbf{Stream} \ A$$

Some of the operations defined on additive lists earlier can be expressed much more naturally using streams. For example, we can define **repeat**, an infinite version of **replicate**. If we want to create streams that consist of more than one distinct value, we can use its generalization, the **iterate** operation.

$$\begin{aligned}\mathbf{repeat} &= \lambda a. \mathbf{unfold} \ (\lambda a. \langle a, a \rangle) \ a \\ \mathbf{iterate} &= \lambda f \ a. \mathbf{unfold} \ (\lambda a. \langle a, f \ a \rangle) \ a\end{aligned}$$

We also expect streams to support a mapping operation. Just like before, the mapped function cannot be directly referenced in the generating function. However, unlike before, we do not have full control over the output of **unfold** and thus cannot use a function type. Since we need both the stream and the function to be accessible, we must use a multiplicative pair. That is, instead of the usual seed type  $\mathbf{Stream} \ A$ , we use  $(f : (- : A) \rightarrow B) \otimes \mathbf{Stream} \ A$ .

$$\mathbf{map}_s = \lambda f \ s. \mathbf{unfold} \ (\lambda p. \mathbf{let}_1 \ (f, s) = p \mathbf{in} \langle f \ (\mathbf{head} \ s), (f, \mathbf{tail} \ s) \rangle) \ (f, s)$$

While we focused on the standard infinite streams here, coinductive types can also contain type dependencies. In that case, the type of at least one eliminator mentions the other eliminators and a dependent additive pair is required to specify the type of the generating function in the introduction.

## 6 Conclusion

The most important aspect of additive types is that they extend resource handling with the notion of choice. This choice comes in two different flavors: *external* choice, which happens during introduction; and *internal* choice, which happens during elimination.

External choice is typically provided by an additive sum. This type is commonly found in substructural systems and has many well-documented use cases. Internal choice is less common, typically provided by an additive pair. Many substructural systems either do not support this type at all or only support it via an inconvenient encoding, which cannot express any form of type dependency.

In this work, we showed that additive pairs in general and dependent additive pairs in particular are not only an interesting theoretical construct but also a practical tool for solving problems in resource-aware programming. Specifically, we identified three distinct kinds of problems that are best solved by these pairs, and successfully implemented solutions in the Janus language. We hope this work inspires further adoption of both dependent and ordinary additive pairs in QTT and other substructural systems, as well as a wider use of these pairs in this style of programming.

## References

1. Abel, A., Pientka, B., Thibodeau, D., Setzer, A.: Copatterns: Programming Infinite Structures by Observations. In: Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. p. 27–38. POPL '13, Association for Computing Machinery, New York, NY, USA (2013). <https://doi.org/10.1145/2429069.2429075>
2. Altenkirch, T., McBride, C., Swierstra, W.: Observational Equality, Now! In: Proceedings of the 2007 Workshop on Programming Languages Meets Program Verification. p. 57–68. PLPV '07, Association for Computing Machinery, New York, NY, USA (2007). <https://doi.org/10.1145/1292597.1292608>
3. Atkey, R.: Syntax and Semantics of Quantitative Type Theory. In: Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science. p. 56–65. LICS '18, Association for Computing Machinery, New York, NY, USA (2018). <https://doi.org/10.1145/3209108.3209189>
4. Brady, E.: Idris 2: Quantitative Type Theory in Practice. In: Möller, A., Sridharan, M. (eds.) 35th European Conference on Object-Oriented Programming (ECOOP 2021). Leibniz International Proceedings in Informatics (LIPIcs), vol. 194, pp. 9:1–9:26. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany (2021). <https://doi.org/10.4230/LIPIcs.ECOOP.2021.9>
5. Brady, E., et al.: Idris 2. <https://github.com/idris-lang/Idris2> (2023)
6. Brunel, A., Gaboardi, M., Mazza, D., Zdanczewic, S.: A Core Quantitative Coeffect Calculus. In: Proceedings of the 23rd European Symposium on Programming Languages and Systems - Volume 8410. p. 351–370. Springer-Verlag, Berlin, Heidelberg (2014). [https://doi.org/10.1007/978-3-642-54833-8\\_19](https://doi.org/10.1007/978-3-642-54833-8_19)
7. Cervesato, I., Pfenning, F.: A Linear Logical Framework. Information and Computation **179**(1), 19–75 (2002). <https://doi.org/10.1006/inco.2001.2951>
8. Choudhury, P., Eades, H., Eisenberg, R., Weirich, S.: A Graded Dependent Type System with a Usage-Aware Semantics. Proc. ACM Program. Lang. **5**(POPL) (Jan 2021). <https://doi.org/10.1145/3434331>
9. Coquand, T.: Pattern Matching with Dependent Types. In: Nordström, B., Petersson, K., Plotkin, G. (eds.) Proceedings of the 1992 Workshop on Types for Proofs and Programs. pp. 71–83. Båstad, Sweden (1992)
10. Krishnaswami, N., Pradic, P., Benton, N.: Integrating Linear and Dependent Types. SIGPLAN Not. **50**(1), 17–30 (Jan 2015). <https://doi.org/10.1145/2775051.2676969>
11. Martin-Löf, P.: An intuitionistic theory of types: Predicative part. In: Rose, H., Shepherdson, J. (eds.) Logic Colloquium '73, Studies in Logic and the Foundations of Mathematics, vol. 80, pp. 73–118. Elsevier (1975). [https://doi.org/10.1016/S0049-237X\(08\)71945-1](https://doi.org/10.1016/S0049-237X(08)71945-1)

- 598 12. McBride, C.: I Got Plenty o' Nuttin'. In: A List of Successes That Can Change  
599 the World. pp. 207–233. Springer International Publishing, Cham (2016). [https://doi.org/10.1007/978-3-319-30936-1\\_12](https://doi.org/10.1007/978-3-319-30936-1_12)  
600
- 601 13. Paulin-Mohring, C.: Inductive definitions in the system Coq rules and properties.  
602 In: Bezem, M., Groote, J.F. (eds.) Typed Lambda Calculi and Applications. pp.  
603 328–345. Springer Berlin Heidelberg, Berlin, Heidelberg (1993). <https://doi.org/10.1007/BFb0037116>  
604
- 605 14. Šefl, V., Svoboda, T.: Additive Types in Quantitative Type Theory. In: Ciabat-  
606 toni, A., Pimentel, E., de Queiroz, R.J.G.B. (eds.) Logic, Language, Information,  
607 and Computation. pp. 250–262. Springer International Publishing, Cham (2022).  
608 [https://doi.org/10.1007/978-3-031-15298-6\\_16](https://doi.org/10.1007/978-3-031-15298-6_16)
- 609 15. Svoboda, T., Šefl, V.: Janus. <https://github.com/svobot/janus> (2023)
- 610 16. The Univalent Foundations Program: Homotopy Type Theory: Univalent Founda-  
611 tions of Mathematics. <https://homotopytypetheory.org/book>, Institute for Ad-  
612 vanced Study (2013)