Programming with Dependent Additive Pairs*

Vít Šefl

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

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

1 Introduction

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

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

Quantitative type theory (QTT) [3] is a recent attempt at combining dependent and substructural types into a single theory. In QTT, variables are not split

* This work was supported by the Charles University grant SVV-260699

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

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

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

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

Since QTT supports both dependent and substructural types, it gives us a unique opportunity to explore dependent versions of multiplicative and additive types. One example is the previously mentioned function type $(x \overset{\sigma}{:} A) \to B$. Another example is the dependent additive pair $(x : A) \mathbin{\&} B$, which is a generalization of the additive conjunction found in linear logic and is the main focus of this work.

1.1 Goals

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

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

1.2 Contributions

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

2 Related Work

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

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

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

QTT itself is described by Atkey [3]. This work addresses a problem with inadmissible substitution as well as extending the theory with dependent multiplicative pairs and booleans. A categorical model is also provided. Note that other additive types are only available via encoding involving booleans and function types.

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

QTT forms the basis of some programming languages. Idris 2 is a purely functional, general purpose programming language developed by Brady et al. [5] and based on the zero-one-many flavor of QTT. Brady [4] provides compelling examples of combining dependent and substructural types to increase the type safety of programs, such as dependent session types: a two party communication channels that enforce a protocol at the type level.

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

3 Quantitative Type Theory

3.1 Semirings

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

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

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

3.2 Syntax

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

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

Going from top to bottom, we have multiplicities π, σ; contexts Γ; and terms M, N, O. When unambiguous, the empty context $\cdot$ is omitted. Contexts can be scaled and added together. Context addition is not used in this work and we thus only define scaling by π:

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

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

We also need a modified typing judgment that takes into account the multiplicities:

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

That is, given the context Γ, we can show that the term M is usable σ times and has the type N. A key restriction is that σ must be either 0 or 1. The judgment thus only communicates whether the term M can be used in a computationally relevant context.

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

3.3 Weakening

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

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

Contexts can also be ordered. If the contexts Γ_1 and Γ_2 only differ in multiplicities ($0\Gamma_1 = 0\Gamma_2$), we define $\Gamma_1 \leq \Gamma_2$ as a pointwise extension of the semiring ordering $\leq$. In other words, $\Gamma_1 \leq \Gamma_2$ iff the multiplicity of each variable in Γ_1 is less than or equal to the multiplicity of the same variable in Γ_2. The weakening rule then states that if the typing judgment holds in a typing context Γ_1, it also holds in any greater typing context Γ_2:

$$\frac{\Gamma_1 \vdash M \overset{\sigma}{:} T \qquad \Gamma_1 \leq \Gamma_2}{\Gamma_2 \vdash M \overset{\sigma}{:} T}\ \textsc{Weak}$$

3.4 Annotated Eliminators

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

$$p \overset{\omega}{:} (\_ \overset{1}{:} \mathbb{N}) \otimes \mathbb{N} \vdash \textbf{let } (x, y) = p \textbf{ in } x + x \overset{1}{:} \mathbb{N}$$

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

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

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

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

$$p \overset{\omega}{:} (\_ \overset{1}{:} \mathbb{N}) \otimes \mathbb{N} \vdash \textbf{let}_\omega\ (x, y) = p \textbf{ in } x + x \overset{1}{:} \mathbb{N}$$

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

4 Dependent Additive Pairs

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

$$x \overset{1}{:} T \vdash \langle x, x \rangle \overset{1}{:} T \,\&\, T$$

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

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

$$S \overset{0}{:} \mathcal{U}, T \overset{0}{:} \mathcal{U} \vdash (b \overset{1}{:} \mathbf{2}) \to \mathbf{case}_1\ b\ \mathbf{of}\ \{\mathbf{inl}\ t \to S; \mathbf{inr}\ f \to T\} \overset{0}{:} \mathcal{U}$$

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

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

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

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

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

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

$$\frac{0\Gamma \vdash S \overset{0}{:} \mathcal{U} \qquad 0\Gamma, x \overset{0}{:} S \vdash T \overset{0}{:} \mathcal{U}}{0\Gamma \vdash (x : S) \mathbin{\&} T \overset{0}{:} \mathcal{U}}\ \&\text{-F} \qquad \frac{\Gamma \vdash M \overset{\sigma}{:} S \qquad \Gamma \vdash N \overset{\sigma}{:} T[M/x]}{\Gamma \vdash \langle M, N \rangle \overset{\sigma}{:} (x : S) \mathbin{\&} T}\ \&\text{-I}$$

$$\frac{\Gamma \vdash M \overset{\sigma}{:} (x : S) \mathbin{\&} T}{\Gamma \vdash \mathbf{fst}\ M \overset{\sigma}{:} S}\ \&\text{-E}_1 \qquad \frac{\Gamma \vdash M \overset{\sigma}{:} (x : S) \mathbin{\&} T}{\Gamma \vdash \mathbf{snd}\ M \overset{\sigma}{:} T[\mathbf{fst}\ M/x]}\ \&\text{-E}_2$$

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

Notice that the second elimination rule uses both **fst** M and **snd** M, seemingly resulting in unsound resource use. However, because the first element is accessed in a computationally irrelevant context, resource soundness is not affected. The behavior of these eliminators is as follows:

$$\textbf{fst}\ \langle M, N\rangle \rightsquigarrow M$$
$$\textbf{snd}\ \langle M, N\rangle \rightsquigarrow N$$

5 Programming with Dependent Additive Pairs

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

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

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

5.1 Linear Folds

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

$$\textbf{fold}\ f\ z\ \textbf{Nil} = z$$
$$\textbf{fold}\ f\ z\ (\textbf{Cons}\ x\ xs) = f\ x\ (\textbf{fold}\ f\ z\ xs)$$

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

$$A : \mathcal{U}, P : \textbf{List}\ A \to \mathcal{U} \vdash \textbf{fold} : (f : (x : A) \to (r : P\ ?) \to P\ (\textbf{Cons}\ x\ ?)) \to (z : P\ \textbf{Nil}) \to (l : \textbf{List}\ A) \to P\ l$$

[1] `https://github.com/vituscze/dependent-additive-pairs`

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

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

$$
\begin{aligned}
&A \overset{0}{:} \mathcal{U}, P \overset{0}{:} (l \overset{0}{:} \textbf{List } A) \to \mathcal{U} \vdash \textbf{fold} \overset{1}{:} \\
&\quad (f \overset{\omega}{:} (x \overset{1}{:} A) \to (xs \overset{0}{:} \textbf{List } A) \to (r \overset{1}{:} P\ xs) \to P\ (\textbf{Cons}\ x\ xs)) \to \\
&\quad (z \overset{1}{:} P\ \textbf{Nil}) \to (l \overset{1}{:} \textbf{List } A) \to P\ l
\end{aligned}
$$

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

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

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

$$
\begin{aligned}
&A \overset{0}{:} \mathcal{U}, P \overset{0}{:} (l \overset{0}{:} \textbf{List } A) \to \mathcal{U} \vdash \textbf{para} \overset{1}{:} \\
&\quad (f \overset{\omega}{:} (x \overset{1}{:} A) \to (p \overset{1}{:} (xs : \textbf{List } A)\ \&\ P\ xs) \to P\ (\textbf{Cons}\ x\ (\textbf{fst}\ p))) \to \\
&\quad (z \overset{\omega}{:} P\ \textbf{Nil}) \to (l \overset{1}{:} \textbf{List } A) \to P\ l
\end{aligned}
$$

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

The original **fold** can be implemented using **para** quite easily. However, since catamorphisms are also the eliminators of inductive types, it should be possible to implement **para** in terms of **fold**. Since the combining function does not have access to the rest of the list, it has to reconstruct it. The reconstructed list and recursive result are stored in an additive pair. The following definition demonstrates the desired semantics:

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

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

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

$$\begin{aligned} &A \overset{0}{:} \mathcal{U}, x \overset{0}{:} A, y \overset{0}{:} A \vdash x \equiv y \overset{0}{:} \mathcal{U} \\ &\qquad\qquad A \overset{0}{:} \mathcal{U} \vdash \mathbf{refl} \overset{1}{:} (x \overset{0}{:} A) \to x \equiv x \end{aligned}$$

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

$$\begin{aligned} A \overset{0}{:} \mathcal{U}, x \overset{0}{:} A, y \overset{0}{:} A \vdash \mathbf{J} \overset{1}{:} &(P \overset{0}{:} (y' \overset{0}{:} A) \to (\_ \overset{0}{:} x \equiv y') \to \mathcal{U}) \to \\ &(f \overset{1}{:} P\ x\ (\mathbf{refl}\ x)) \to (p \overset{1}{:} x \equiv y) \to P\ y\ p \end{aligned}$$

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

$$\mathbf{Triple} = \lambda l.\ (p \overset{1}{:} (l' : \mathbf{List}\ A)\ \&\ P\ l') \otimes (\mathbf{fst}\ p \equiv l)$$

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

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

For the former, we start with the proof $\mathbf{Nil} \equiv \mathbf{Nil}$. The inductive step asks us to prove $\mathbf{Cons}\ x\ (\mathbf{fst}\ p) \equiv \mathbf{Cons}\ x\ l$ given $\mathbf{fst}\ p \equiv l$, which is accomplished by using the congruence property. That is, if $p \overset{1}{:} x \equiv y$ then $\mathbf{cong}\ f\ p \overset{1}{:} f\ x \equiv f\ y$. Putting it all together, we obtain the following:

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

This definition is now correct and matches the type given earlier. The type of **para** is thus justified. Note that in practice, paramorphisms would not be defined in terms of catamorphisms, as the need to reconstruct the structure removes one of their main benefits.

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

$$\begin{aligned}&(x \overset{1}{:} A) \to (l \overset{1}{:} (t : \mathbf{Tree}\ A)\ \&\ P\ t) \to (r \overset{1}{:} (t : \mathbf{Tree}\ A)\ \&\ P\ t) \to \\ &\quad P\ (\mathbf{Node}\ x\ (\mathbf{fst}\ l)\ (\mathbf{fst}\ r))\end{aligned}$$

5.2 Resource-Aware Proofs

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

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

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

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

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

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

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

$$\begin{aligned}A \overset{0}{:} \mathcal{U} \vdash \mathbf{partition} \overset{1}{:} (p \overset{\omega}{:} (a \overset{1}{:} A) \to \mathbf{2}) \to (l \overset{1}{:} \mathbf{List}\ A) \to \\ (\_ \overset{1}{:} \mathbf{List}\ A) \otimes \mathbf{List}\ A\end{aligned}$$

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

Instead, we require the predicate to also return a new version of the input, such as $p \overset{\omega}{:} (a \overset{1}{:} A) \to (\_ \overset{1}{:} \mathbf{2}) \otimes A$. With this change, implementing **partition** is easy. Now, suppose we want to also return a description of the resulting partition. We can use the following type:

$$\begin{aligned}
A \overset{0}{:} \mathcal{U}, xs\ ys\ zs \overset{0}{:} \mathbf{List}\ A &\vdash \mathbf{Union}\ xs\ ys\ zs \overset{0}{:} \mathcal{U} \\
A \overset{0}{:} \mathcal{U}, xs\ ys\ zs \overset{0}{:} \mathbf{List}\ A &\vdash \mathbf{Left} \overset{1}{:} (x \overset{1}{:} A) \to (u \overset{1}{:} \mathbf{Union}\ xs\ ys\ zs) \to \\
&\qquad \mathbf{Union}\ (\mathbf{Cons}\ x\ xs)\ ys\ (\mathbf{Cons}\ x\ zs) \\
A \overset{0}{:} \mathcal{U}, xs\ ys\ zs \overset{0}{:} \mathbf{List}\ A &\vdash \mathbf{Right} \overset{1}{:} (x \overset{1}{:} A) \to (u \overset{1}{:} \mathbf{Union}\ xs\ ys\ zs) \to \\
&\qquad \mathbf{Union}\ xs\ (\mathbf{Cons}\ x\ ys)\ (\mathbf{Cons}\ x\ zs) \\
A \overset{0}{:} \mathcal{U} &\vdash \mathbf{Stop} \overset{1}{:} \mathbf{Union}\ \mathbf{Nil}\ \mathbf{Nil}\ \mathbf{Nil}
\end{aligned}$$

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

$$\begin{aligned}
\mathbf{Result_1} &= \lambda A.\ (_ \overset{1}{:} \mathbf{List}\ A) \otimes \mathbf{List}\ A \\
\mathbf{Result_2} &= \lambda A\ zs\ r_1.\ \mathbf{let}_0\ (xs, ys) = r_1\ \mathbf{in}\ \mathbf{Union}\ xs\ ys\ zs \\
\mathbf{Result} &= \lambda A\ zs.\ (r_1 : \mathbf{Result_1}\ A)\ \&\ \mathbf{Result_2}\ A\ zs\ r_1 \\
\mathbf{Pred} &= \lambda A.\ (x \overset{1}{:} A) \to (_ \overset{1}{:} \mathbf{2}) \otimes A
\end{aligned}$$

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

$$A \overset{0}{:} \mathcal{U} \vdash \mathbf{partition} \overset{1}{:} (p \overset{\omega}{:} \mathbf{Pred}\ A) \to (l \overset{1}{:} \mathbf{List}\ A) \to \mathbf{Result}\ A\ l$$

Since the type of the result depends on the input list, we need to use the dependent **fold** defined earlier. The base case has the type **Result** A **Nil** and only has a single valid value: $\langle(\mathbf{Nil}, \mathbf{Nil}), \mathbf{Stop}\rangle$. The inductive case can be broken down into three steps. Firstly, we define auxiliary functions that add the new element to one of the $\mathbf{Result_1}$ lists.

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

Secondly, we use these definitions to add the new elements to the whole **Result**. However, since the first element does not reduce to a pair, the type of the second element remains some form of $\mathbf{Result_2}$, rather than reducing to **Union**. We can fix this by inspecting the first element to force reduction.

$$\mathbf{add_l} = \lambda x\ r.\ \langle \mathbf{add}'_\mathbf{l}\ x\ (\mathbf{fst}\ r), \mathbf{let}_0\ (\_, \_) = \mathbf{fst}\ r\ \mathbf{in}\ \mathbf{Left}\ x\ (\mathbf{snd}\ r)\rangle$$

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

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

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

$$\begin{aligned}&\ldots \vdash \mathbf{add'_l}\ \mathbf{add'_r} \overset{1}{:} (x \overset{1}{:} A) \to (r_1 \overset{1}{:} \mathbf{Result_1}\ A) \to \mathbf{Result_1}\ A \\ &\ldots \vdash \mathbf{add_l}\ \mathbf{add_r} \overset{1}{:} (x \overset{1}{:} A) \to (r \overset{1}{:} \mathbf{Result}\ A\ zs) \to \mathbf{Result}\ A\ (\mathbf{Cons}\ x\ zs)\end{aligned}$$

Finally, we can apply the predicate and then use $\mathbf{add_l}$ or $\mathbf{add_r}$ depending on the result.

$$\begin{aligned}\mathbf{step} = \lambda p\ x\ r.\ &\mathbf{let}_1\ (b, x') = p\ x\ \mathbf{in} \\ &\quad \mathbf{case}_1\ b\ \mathbf{of}\ \{\mathbf{inl}\ () \to \mathbf{add_l}\ x'\ r; \mathbf{inr}\ () \to \mathbf{add_r}\ x'\ r\}\end{aligned}$$

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

$$\mathbf{Pred} = \lambda A.\ (x \overset{1}{:} A) \to (\_ \overset{1}{:} \mathbf{2}) \otimes (x' \overset{1}{:} A) \otimes (x' \equiv x)$$

The proof can be extracted with a second **let** term. It can then be used to rewrite the type of the result, which is done by using the substitutivity of the identity type. In particular, if $v \overset{1}{:} P\ x$ and $p \overset{1}{:} x \equiv y$ then $\mathbf{subst}\ P\ p\ v \overset{1}{:} P\ y$. We can now fix the **step** function.

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

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

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

It should be noted that the reduction behavior of $\mathbf{Result_2}$ might not be desirable in some situations. Even though it will eventually reduce to a **Union**, the type checker cannot see that without eliminating the pair first. In cases like this, it is generally recommended to move the computation to the indices of the type. We can define projections **Fst** and **Snd** for the multiplicative pair that may be used in types. We can then define another version of $\mathbf{Result_2}$.

$$\mathbf{Result_2} = \lambda A\ zs\ r_1.\ \mathbf{Union}\ (\mathbf{Fst}\ r_1)\ (\mathbf{Snd}\ r_1)\ zs$$

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

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

$$\begin{aligned}\mathbf{Result} &= \lambda A\ x\ xs.\ (l : \mathbf{List}\ A)\ \&\ \mathbf{Union}\ xs\ [x]\ l \\ \mathbf{Cmp} &= \lambda A.\ (x \overset{1}{:} A) \to (y \overset{1}{:} A) \to \\ &\qquad\qquad (b \overset{1}{:} \mathbf{2}) \otimes (x' \overset{1}{:} A) \otimes (y' \overset{1}{:} A) \otimes (\_ \overset{1}{:} x' \equiv x) \otimes (y' \equiv y)\end{aligned}$$

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

$$A \overset{0}{:} \mathcal{U} \vdash \mathbf{insert} \overset{1}{:} (c \overset{\omega}{:} \mathbf{Cmp}\ A) \to (x \overset{1}{:} A) \to (xs \overset{1}{:} \mathbf{List}\ A) \to \mathbf{Result}\ A\ x\ xs$$

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

$$A \overset{0}{:} \mathcal{U} \vdash \lambda xs.\ (x \overset{1}{:} A) \to \mathbf{Result}\ A\ x\ xs \overset{0}{:} (xs \overset{0}{:} \mathbf{List}\ A) \to \mathcal{U}$$

The base case is then trivial.

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

The inductive case is more interesting. If the inserted element x is smaller than or equal to y (according to the comparison function), we have found the insertion point and no further recursion is necessary. Recall that a paramorphism gives us access to an additive pair r containing the rest of the list and the recursive result. We ignore the recursive result and return $\mathbf{Cons}\ x\ (\mathbf{Cons}\ y\ (\mathbf{fst}\ r))$. We also need to construct a proof of the following type:

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

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

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

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

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

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

$$\cdots \vdash \mathbf{snd}\ r\ x \stackrel{1}{:} (l : \mathbf{List}\ A)\ \&\ \mathbf{Union}\ (\mathbf{fst}\ r)\ [x]\ l$$

Of course, we need to add the element y back to the list and return the list **Cons** y (**fst** (**snd** r x)). Additionally, we need to construct a proof of the following type:

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

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

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

Combining the functions **done** and **go** gives us a single step of the insertion. Note that we use a shortcut to represent the use of four $\mathbf{let}_1$ eliminations required to unpack the result of the comparison $c\ x\ y$.

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

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

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

And finally, we have everything needed to define the dependent **insert** operation itself.

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

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

5.3 Inductive and Coinductive Types

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

$$\mathbf{ListF} = \lambda X.\ \mathbf{1} \oplus (\_ \overset{1}{:} A) \otimes X$$

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

$$\begin{aligned} A \overset{0}{:} \mathcal{U} &\vdash \mathbf{List}^{+}\ A \overset{0}{:} \mathcal{U} \\ A \overset{0}{:} \mathcal{U} &\vdash \mathbf{Nil}^{+} \overset{1}{:} \mathbf{List}^{+}\ A \\ A \overset{0}{:} \mathcal{U}, p \overset{1}{:} (\_ : A)\ \&\ \mathbf{List}^{+}\ A &\vdash \mathbf{Cons}^{+}\ p \overset{1}{:} \mathbf{List}^{+}\ A \end{aligned}$$

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

$$A \overset{0}{:} \mathcal{U}, x \overset{1}{:} A \nvdash \mathbf{Cons}^{+}\ \langle x, \mathbf{Nil}^{+} \rangle \overset{1}{:} \mathbf{List}^{+}\ A$$

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

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

The function f is discarded in one case and duplicated in the other and thus needs the ω multiplicity. However, the function z also needs ω, as it is discarded in the second case.

$$\begin{aligned} A \overset{0}{:} \mathcal{U}, P \overset{0}{:} \mathcal{U} \vdash \mathbf{fold}^{+} \overset{1}{:} (f \overset{\omega}{:} (\_ \overset{1}{:} (\_ : A)\ \&\ P) \to P) \to \\ (z \overset{\omega}{:} (\_ \overset{1}{:} A) \to P) \to (l \overset{1}{:} \mathbf{List}^{+}\ A) \to P \end{aligned}$$

Additive lists admit some common list operations, such as $\mathbf{map}^{+}$. However, unlike the normal **map**, we can guarantee that the mapped function is used linearly. Notice that since the combining function given to $\mathbf{fold}^{+}$ is used ω times, referencing the mapped function inside it would not count as linear use. Instead, we eliminate into a function type and thread the mapped function through the entire fold. That is, instead of the usual type $\mathbf{List}\ B$ we eliminate into the type $(f \overset{1}{:} (\_ \overset{1}{:} A) \to B) \to \mathbf{List}\ B$. The result is then applied to the mapped function.

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

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

$$P \overset{0}{:} \mathcal{U} \vdash \mathbf{rec} \overset{1}{:} (f \overset{\omega}{:} (\_ \overset{1}{:} P) \to P) \to (z \overset{1}{:} P) \to (n \overset{1}{:} \mathbb{N}) \to P$$

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

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

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

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

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

$$\begin{aligned} A \overset{0}{:} \mathcal{U} &\vdash \mathbf{Stream}\ A \overset{0}{:} \mathcal{U} \\ A \overset{0}{:} \mathcal{U}, s \overset{1}{:} \mathbf{Stream}\ A &\vdash \mathbf{head}\ s \overset{1}{:} A \\ A \overset{0}{:} \mathcal{U}, s \overset{1}{:} \mathbf{Stream}\ A &\vdash \mathbf{tail}\ s \overset{1}{:} \mathbf{Stream}\ A \end{aligned}$$

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].

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

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 ω multiplicity. We obtain the following type for the introduction:

$$S \overset{0}{:} \mathcal{U}, A \overset{0}{:} \mathcal{U} \vdash \textbf{unfold} \overset{1}{:} (f \overset{\omega}{:} (\_ \overset{1}{:} S) \to (\_ : A) \mathbin{\&} S) \to (s \overset{1}{:} S) \to \textbf{Stream}\ A$$

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.

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

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 **Stream** A, we use $(f \overset{1}{:} (\_ \overset{1}{:} A) \to B) \otimes \textbf{Stream}\ A$.

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

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: Møller, 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., Zdancewic, 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`
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., Pettersson, 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`

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