Additive Types in Quantitative Type Theory*

Vít Šefl and Tomáš Svoboda

Faculty of Mathematics and Physics, Charles University, Prague, Czech Republic
sefl@ksvi.mff.cuni.cz, svoboda@posteo.net

Abstract. Dependent type theories enable us to reason about properties of our programs, while substructural type theories enable us to reason about their resource use. Quantitative type theory seamlessly combines dependent and substructural types by employing positive semirings to keep track of variable usage and computational contexts. Existing treatments of this theory typically focus on multiplicative connectives such as functions or multiplicative pairs. In this work, we extend quantitative type theory with additive zero, unit, unions and dependent pairs, as well as annotated eliminators for various types. We then explain how these additive types can be used to better describe resource use and how they integrate with existing programming techniques. Finally, we implement an interpreter for a language based on these extensions.

1 Introduction

Type systems allow us to describe and exhibit certain program properties in a way that can be checked by a computer. Today, the most expressive type systems are typically based on various *dependent type theories*, such as calculus of constructions or Martin-Löf type theory. A typical example of a dependent type is the vector: a list of elements whose length is part of the type. Functions that operate on these vectors need to respect not only the type of the elements, but also the lengths of the vectors. This restriction further constrains possible implementations, reducing the chance of errors. Thanks to their expressiveness, dependent type theories are found in proof assistants or programming languages where verification is paramount.

However, whereas a proof assistant does not mind that an assumption is used twice, a programmer might want to avoid the duplication of an extensive data structure. *Substructural type systems* can be used to describe how a program manipulates its resources. These type systems restrict when variables can be duplicated or discarded. Combinations of these restrictions give rise to different substructural type systems. A well-known example is the linear type system. In this system, variables come in two flavors: unrestricted and linear. Linear variables cannot be discarded or duplicated and must thus be used exactly once.

Since dependent types and substructural types are not mutually exclusive, there have been attempts at combining them into a single type system. Historically, these attempts often restricted how dependent and substructural types are able to interact, such as only allowing dependency on unrestricted variables.

* This work was supported by the Charles University grant SVV-260588

Quantitative type theory [1] (abbreviated QTT) is a recent type theory which seamlessly combines dependent and substructural types. QTT drops the distinction between linear and unrestricted variables. Instead, each variable is annotated by *multiplicity*, an element of some semiring, which describes how the variable is used. Semiring operations describe how the multiplicity changes when the variable is used in different contexts. As an example, multiplication by zero ensures that a variable occurrence in a type does not count towards its use.

Depending on the semiring, QTT can describe a wider variety of substructural types. Apart from the usual linear and intuitionistic function types, we can have functions that cannot use its parameter in a relevant context, functions that must use its parameter exactly n times, at most n times, and so on. Similar variations also exist for other types, such as pairs or modalities. Moreover, QTT can also describe dependent versions of these types.

Substructural type systems distinguish between two kinds of types: *multiplicative* and *additive*. When constructing a value of a multiplicative type, the resources are split into multiple groups, which are then used to create a part of the value. Additive types do not split the resources. When constructing a value, each part can access all resources. To ensure that these resources are not used more than once, only one part of the resulting value can be accessed.

1.1 Contributions

We introduce the concept of an annotated eliminator, which enables the use of the weakening rule on the bound variables rather than the eliminated expression. We extend the theory with multiple additive types: zero, unit, unions and pairs. In particular, we define dependent additive pairs, which do not exist in the normal substructural type theories. We show how these types can be used in practice and how they interact with existing programming techniques. Finally, we build a bidirectional type system and implement it in an interpreter.[1]

1.2 Structure

The work is structured as follows. In the second section, we discuss other approaches to additive types. The third section provides an overview of key concepts: semirings, multiplicities, and weakening. Multiplicative types, together with the annotated eliminator, are introduced in the fourth section. Additive types are introduced in the fifth section. This section also discusses possible uses for these types, focusing primarily on dependent additive pairs. The sixth section details the process of building bidirectional type checking for the modified theory. The seventh section describes the interpreter and its implementation.

2 Related Work

The idea of combining linear and dependent types is not new. One of the earliest such systems is Cervesato and Pfenning's Linear Logical Framework [5]. In this

[1] `https://github.com/svobot/janus`

system, the typing context is split into a *linear* part and an *intuitionistic* part. However, this split also creates some restrictions. As an example, a dependent type cannot mention linear variables. A more recent example of a system that distinguishes between linear and intuitionistic parts is Krishnaswami et al. [8].

Semirings have also been used to keep track of usage. Brunel et al. [4] use semiring annotations with exponential modality. This idea is then generalized to encompass *coeffects*. The typing context is not split and contains both linear and *discharged* variables, the latter of which contains a semiring annotation.

McBride [10] describes a dependent type theory in which semiring zero is used to represent computational irrelevance. By treating types as a computationally irrelevant context, this theory allows types to depend on both linear and intuitionistic variables. The work also describes weakening as well as a translation into an untyped lambda calculus which erases irrelevant portions of the program.

However, the system described by McBride does not admit substitution. This problem is fixed by Atkey [1], who also extends the theory with dependent multiplicative pairs and booleans, and provides a categorical model.

Note that QTT essentially disables resource checking for the irrelevant fragment, which might be undesirable in some cases. Choudhury et al. [6] describe a graded dependent type system which does not treat types in a special way and can thus be applied to a wider range of semirings. Interestingly, this system contains additive union eliminator which can consume the given value more than once. However, this feature is not applied to any other eliminator in the system.

QTT has also been successfully implemented into programming languages. A prime example is the Idris2 language, developed by Brady et al. [2,3]. Another example is the language Juvix [11], which features dependent additive pairs.

3 Quantitative Type Theory

3.1 Semirings

QTT uses semirings to keep track of variable usage. A semiring is a tuple $(S, +, \cdot, 0, 1)$ where S is a set, $+$ and $\cdot$ are functions of type $S \times S \to S$, 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 $0a = 0 = a0$. Since variable usage cannot be negative, the semiring also needs to be *positive*. A positive semiring additionally requires: $a + b = 0$ implies $a = 0$ and $b = 0$; $ab = 0$ implies $a = 0$ or $b = 0$.

Different semirings give rise to different variations of QTT. As an example, the $(\mathbb{N}, +, \cdot, 0, 1)$ semiring tracks exact usage. In this work, we use the zero-one-many semiring $(\{0, 1, \omega\}, +, \cdot, 0, 1)$ where ω represents more than one use. In this semiring, $\omega \cdot \omega = \omega$ and $\rho + \omega = \omega$, the rest follows from the semiring properties. After we introduce weakening, we will be able to identify ω with any use, much like unrestricted variables in linear type systems.

3.2 Typing Rules

The standard typing judgment $\Gamma \vdash M : T$ does not provide enough information to correctly account for variable usage. Firstly, the typing context Γ needs to be

extended. Instead of containing pairs (x, T) (written as $x : T$) where x is a name of a variable and T is its type, the context will now contain triples (x, T, π) (written as $x \overset{\pi}{:} T$) where π is an element of the semiring.

Secondly, the judgment itself needs to track the usage context. There are circumstances where variable occurrences do not count toward that variable's usage. In the extended judgment $\Gamma \vdash M \overset{\sigma}{:} T$, the usage context is given by σ, an element of the semiring. In theory, any element can be used. However, this work follows the description given by Atkey [1] where σ can only be semiring's zero (irrelevant context) or one (relevant context).

Finally, the typing rules need to correctly handle extended typing contexts. The variable rule associates the multiplicity of the variable in the typing context with the multiplicity in the judgment itself. Other multiplicities must be zero.

$$\frac{}{0\Gamma_1, x \overset{\sigma}{:} S, 0\Gamma_2 \vdash x \overset{\sigma}{:} S} \textsc{Var}$$

In a linear type system, we need to distinguish between linear functions (written as $A \multimap B$) that must use its parameter exactly once, and unrestricted functions (written as $A \to B$). In QTT, these two types are a special case of a more general function type, $(x \overset{\pi}{:} S) \to T$. Functions of this type can use its parameter according to the multiplicity π. In our system, linear functions correspond to $(x \overset{1}{:} S) \to T$ and unrestricted functions to $(x \overset{\omega}{:} S) \to T$. This restriction is reflected in one of the typing rules for function application.

$$\frac{\Gamma_1 \vdash M \overset{\sigma}{:} (x \overset{\pi}{:} S) \to T \qquad \Gamma_2 \vdash N \overset{1}{:} S}{\Gamma_1 + \sigma\pi\Gamma_2 \vdash M\ N \overset{\sigma}{:} T[N/x]} \to\text{-E}_1$$

The rule follows a naming scheme used throughout this work: a type, a class (formation, introduction, or elimination), and, if necessary, an index. The variable usages from the function and the argument are added together. Additionally, since the function can use its parameter π-times, the usage from the argument needs to be scaled by π. Scaling of the context by π, $\pi\Gamma$, is done by multiplying the multiplicity of each variable by π. Adding two contexts, $\Gamma_1 + \Gamma_2$, is done by adding the multiplicities of matching variables.

3.3 Weakening

In this system, each expression has a unique variable usage. In some situations, such rigidity is undesirable. In linear type theories, unrestricted variables can be used any amount of times, including once. As mentioned previously, we would like to use ω from the $\{0, 1, \omega\}$ semiring to represent unrestricted use. The system thus needs to be able to treat the multiplicities 0 and 1 as ω, which can be accomplished by adding an ordering to the semiring.

The ordering, $\leq$, needs to be reflexive, transitive, and needs to respect the semiring operations. A suitable ordering for our semiring is defined by $0 \leq \omega$ and $1 \leq \omega$. $0 \leq 1$ is not present since it would allow us to treat unused variables

as linear. We can now extend this ordering to typing contexts: $\Gamma_1 \leq \Gamma_2$ iff the multiplicity of each variable in Γ_1 is less than or equal to the multiplicity of the matching variable in Γ_2. The weakening rule states that relaxing a typing context does not change the validity of the judgment.

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

4 Multiplicative Types

4.1 Multiplicative Pairs

When introducing a multiplicative pair, the typing context is split into two parts and each part is then used to construct one element of the pair. When eliminating a multiplicative pair, both elements of the pair must be used to ensure no resources are discarded.

In QTT, the multiplicative pair type $(x \mathrel{:^{\pi}} S) \otimes T$ additionally contains multiplicity annotation π which specifies how many times can the first element be used. The type of the second element can also depend on the value of the first element. The following rules summarize formation, introduction, and elimination. The constant $\mathcal{U}$ is the type of types.

$$\frac{0\Gamma \vdash S \mathrel{:^{0}} \mathcal{U} \qquad 0\Gamma, x \mathrel{:^{0}} S \vdash T \mathrel{:^{0}} \mathcal{U}}{0\Gamma \vdash (x \mathrel{:^{\pi}} S) \otimes T \mathrel{:^{0}} \mathcal{U}}\ \otimes\text{-F}$$

$$\frac{\sigma\pi = 0 \qquad 0\Gamma \vdash M \mathrel{:^{0}} S \qquad \Gamma \vdash N \mathrel{:^{\sigma}} T[M/x]}{\Gamma \vdash (M,\, N) \mathrel{:^{\sigma}} (x \mathrel{:^{\pi}} S) \otimes T}\ \otimes\text{-I}_0 \qquad \frac{\Gamma_1 \vdash M \mathrel{:^{1}} S \qquad \Gamma_2 \vdash N \mathrel{:^{\sigma}} T[M/x]}{\sigma\pi\Gamma_1 + \Gamma_2 \vdash (M,\, N) \mathrel{:^{\sigma}} (x \mathrel{:^{\pi}} S) \otimes T}\ \otimes\text{-I}_1$$

$$\frac{\Gamma_1 \vdash M \mathrel{:^{\sigma}} (x \mathrel{:^{\pi}} S) \otimes T \qquad 0\Gamma_1, z \mathrel{:^{0}} (x \mathrel{:^{\pi}} S) \otimes T \vdash U \mathrel{:^{0}} \mathcal{U} \qquad \Gamma_2, x \mathrel{:^{\sigma\pi}} S, y \mathrel{:^{\sigma}} T \vdash N \mathrel{:^{\sigma}} U[(x,\, y)/z]}{\Gamma_1 + \Gamma_2 \vdash \mathbf{let}\ (x,\, y) = M\ \mathbf{in}\ N \mathrel{:^{\sigma}} U[M/z]}\ \otimes\text{-E}$$

4.2 Multiplicative Unit

Unit types also come in two variants. The multiplicative unit uses no resources and although it cannot be discarded, it can always be eliminated.

$$\frac{0\Gamma \vdash}{0\Gamma \vdash \mathbf{1} \mathrel{:^{0}} \mathcal{U}}\ \mathbf{1}\text{-F} \qquad \frac{0\Gamma \vdash}{0\Gamma \vdash (\,) \mathrel{:^{\sigma}} \mathbf{1}}\ \mathbf{1}\text{-I}$$

$$\frac{\Gamma_1 \vdash M \mathrel{:^{\sigma}} \mathbf{1} \qquad 0\Gamma_1, x \mathrel{:^{0}} \mathbf{1} \vdash S \mathrel{:^{0}} \mathcal{U} \qquad \Gamma_2 \vdash N \mathrel{:^{\sigma}} S[(\,)/x]}{\Gamma_1 + \Gamma_2 \vdash \mathbf{let}\ (\,) = M\ \mathbf{in}\ N \mathrel{:^{\sigma}} S[M/x]}\ \mathbf{1}\text{-E}$$

4.3 Exponential Modality

Multiplicative units and pairs can be combined to form the exponential type $!_\pi S$, which is defined as $(\_ \overset{\pi}{:} S) \otimes \mathbf{1}$. Values of this type are introduced and eliminated by combining the introductions and eliminations of the constituent parts. Introduction, $!x$, is defined as $(x, ())$. Elimination, $\mathbf{let}\ !x = M\ \mathbf{in}\ N$, is defined as $\mathbf{let}\ (x, i) = M\ \mathbf{in}\ \mathbf{let}\ () = i\ \mathbf{in}\ N$.

4.4 Extended Eliminator

However, the multiplicative pair eliminator does not interact well with weakening. Suppose the typing context contains $p \overset{\omega}{:} (\_ \overset{1}{:} S) \otimes T$ (abbreviated $S \otimes T$). Unless we want to eliminate p multiple times, we first need to apply the weakening rule to obtain $p \overset{1}{:} S \otimes T$. Now the elimination $\mathbf{let}\ (x, y) = p\ \mathbf{in}\ N$ can be used, which adds the variables x and y to the context. Notice that their multiplicities are 1 and thus the variables cannot be duplicated or discarded.

The expression $\mathbf{let}\ (x, y) = p\ \mathbf{in}\ (y, y)$ does not seem to violate any resource constraints. Indeed, if p is unrestricted, so should be its elements. The only way around this problem is to know beforehand that the elements of p will be used in this way and change the type to $!_\omega S \otimes !_\omega T$.

We solve this problem by adding a *multiplicity annotation* to the eliminator, allowing it to consume the pair more than once. To ensure no resources are discarded, the multiplicity is propagated to the variables representing the elements of the pair. In the example, the weakening rule can now be used on x instead of p, allowing us to discard it in the expression (y, y).

$$\frac{\begin{array}{c}\Gamma_1 \vdash M \overset{\sigma}{:} (x \overset{\pi}{:} S) \otimes T \\ 0\Gamma_1, z \overset{0}{:} (x \overset{\pi}{:} S) \otimes T \vdash U \overset{0}{:} \mathcal{U} \qquad \Gamma_2, x \overset{\rho\sigma\pi}{:} S, y \overset{\rho\sigma}{:} T \vdash N \overset{\sigma}{:} U[(x, y)/z]\end{array}}{\rho\Gamma_1 + \Gamma_2 \vdash \mathbf{let}_\rho\ (x, y) = M\ \mathbf{in}\ N \overset{\sigma}{:} U[M/z]}\ \otimes\text{-E}$$

Since multiplicative pairs have a single introduction, the eliminator does not provide any computationally relevant information apart from the two elements. In particular, the eliminator functions properly even when the annotation is zero.

With this modification, the exponential type can be introduced as needed and does not have to be added beforehand. Indeed, we can easily transform ω copies of $S \otimes T$ into a single copy of $!_\omega S \otimes !_\omega T$ with the following function.

$$\vdash \lambda p.\ \mathbf{let}_\omega\ (x, y) = p\ \mathbf{in}\ (!x, !y) \overset{1}{:} (\_ \overset{\omega}{:} S \otimes T) \to !_\omega S \otimes !_\omega T$$

In fact, using this extended eliminator, we can show that $!_\omega(S \otimes T)$ and $!_\omega S \otimes !_\omega T$ are isomorphic.

$$\begin{array}{l}\vdash \lambda p'.\ \mathbf{let}\ !p = p'\ \mathbf{in}\ \mathbf{let}_\omega\ (x, y) = p\ \mathbf{in}\ (!x, !y) \\ \quad \overset{1}{:} (\_ \overset{1}{:} !_\omega(S \otimes T)) \to !_\omega S \otimes !_\omega T \\ \vdash \lambda p.\ \mathbf{let}_1\ (x', y') = p\ \mathbf{in}\ \mathbf{let}\ !x = x'\ \mathbf{in}\ \mathbf{let}\ !y = y'\ \mathbf{in}\ !(x, y) \\ \quad \overset{1}{:} (\_ \overset{1}{:} !_\omega S \otimes !_\omega T) \to !_\omega(S \otimes T)\end{array}$$

These definitions can also be used with dependent pairs. However, we have to account for the change of the type of the first element. We can construct the type $!_\omega((x \overset{1}{:} S) \otimes T)$ under the assumption $x \overset{0}{:} S \vdash T \overset{0}{:} \mathcal{U}$. However, if the type of the first element is $!_\omega S$, it needs to be unwrapped before it can be used in T. The second type is then $(x \overset{1}{:} !_\omega S) \otimes !_\omega T[(\mathbf{let}\ !y = x\ \mathbf{in}\ y)/x]$.

We can also add the multiplicity annotation to the unit eliminator. In the $\{0, 1, \omega\}$ semiring, the only useful annotation is 1, which corresponds to the regular eliminator. If the context contains ω copies of a unit, no elimination is necessary due to weakening. This annotation can be useful with other semirings.

5 Additive Types

5.1 Additive Unions

Disjoint unions are an ideal example of an additive type. When constructing a value of this type, the typing context is passed without any changes. Disjoint unions have two introductions and one elimination, which performs case analysis.

$$\frac{0\Gamma \vdash S \overset{0}{:} \mathcal{U} \qquad 0\Gamma \vdash T \overset{0}{:} \mathcal{U}}{0\Gamma \vdash S \oplus T \overset{0}{:} \mathcal{U}}\ \oplus\text{-F}$$

$$\frac{\Gamma \vdash M \overset{\sigma}{:} S \qquad 0\Gamma \vdash T \overset{0}{:} \mathcal{U}}{\Gamma \vdash \mathbf{inl}\ M \overset{\sigma}{:} S \oplus T}\ \oplus\text{-I}_1 \qquad \frac{\Gamma \vdash M \overset{\sigma}{:} T \qquad 0\Gamma \vdash S \overset{0}{:} \mathcal{U}}{\Gamma \vdash \mathbf{inr}\ M \overset{\sigma}{:} S \oplus T}\ \oplus\text{-I}_2$$

$$\frac{\begin{array}{c}\rho \neq 0 \qquad \Gamma_1 \vdash M \overset{\sigma}{:} S \oplus T \qquad 0\Gamma_1, z \overset{0}{:} S \oplus T \vdash U \overset{0}{:} \mathcal{U} \\ \Gamma_2, x \overset{\rho\sigma}{:} S \vdash N \overset{\sigma}{:} U[\mathbf{inl}\ x/z] \qquad \Gamma_2, y \overset{\rho\sigma}{:} T \vdash O \overset{\sigma}{:} U[\mathbf{inr}\ y/z]\end{array}}{\rho\Gamma_1 + \Gamma_2 \vdash \mathbf{case}_\rho\ M\ \mathbf{of}\ \{\mathbf{inl}\ x \to N,\ \mathbf{inr}\ y \to O\} \overset{\sigma}{:} U[M/z]}\ \oplus\text{-E}$$

The eliminator is annotated with multiplicity, just like in the case of multiplicative pairs. The requirement that $\rho \neq 0$ is crucial. In that case, although the elements of the disjoint union cannot be used in a relevant context, the union as a whole carries computationally relevant information: the choice of introduction.

We can use the extended eliminator to show that $!_\omega(S \oplus T)$ and $!_\omega S \oplus !_\omega T$ are isomorphic. Like before, only one direction requires this extension.

$$\vdash \lambda s'.\ \mathbf{let}\ !s = s'\ \mathbf{in}\ \mathbf{case}_\omega\ s\ \mathbf{of}\ \{\mathbf{inl}\ x \to \mathbf{inl}\ !x,\ \mathbf{inr}\ y \to \mathbf{inr}\ !y\}$$
$$\overset{1}{:}\ (\_ \overset{1}{:} !_\omega(S \oplus T)) \to !_\omega S \oplus !_\omega T$$
$$\vdash \lambda s.\ \mathbf{case}_1\ s\ \mathbf{of}\ \{\mathbf{inl}\ x' \to \mathbf{let}\ !x = x'\ \mathbf{in}\ !\mathbf{inl}\ x,\ \mathbf{inr}\ y' \to \mathbf{let}\ !y = y'\ \mathbf{in}\ !\mathbf{inr}\ y\}$$
$$\overset{1}{:}\ (\_ \overset{1}{:} !_\omega S \oplus !_\omega T) \to !_\omega(S \oplus T)$$

5.2 Additive Zero

The empty type has no introductions and a single elimination.

$$\frac{0\Gamma \vdash}{0\Gamma \vdash \bot \overset{0}{:} \mathcal{U}}\ \bot\text{-F} \qquad \frac{\Gamma \vdash M \overset{\sigma}{:} \bot \qquad 0\Gamma, x \overset{0}{:} \bot \vdash S \overset{0}{:} \mathcal{U}}{\Gamma \vdash \mathbf{case}\ M\ \mathbf{of}\ \{\} \overset{\sigma}{:} S[M/x]}\ \bot\text{-E}$$

5.3 Additive Pairs

Defining the additive version of a pair type presents a problem. We cannot split the resources but we still need to construct both elements of the pair. The only option is to give the same resources to both elements, which seemingly results in resource duplication. We can avoid such duplication by restricting the eliminator to only ever access one of the elements.

Such a type can be constructed with the tools we already have. Since the eliminator of unions shares the typing context between both cases, we can use it to introduce an additive pair. The content of the union is not important, only the introduction is. We thus use the simplest union type: $\mathbf{1} \oplus \mathbf{1}$ (written as $\mathbf{2}$).

To allow the elements of the pair to have different types, we need to use a dependent type for the result of the elimination. We thus define the type of additive pairs $S \,\&\, T$ as $(b \mathbin{:^1} \mathbf{2}) \to \mathbf{case}_1\ b\ \mathbf{of}\ \{\mathbf{inl}\, t \to S,\ \mathbf{inr}\, f \to T\}$.

Introduction, $\langle x, y\rangle$, is defined as $\lambda b.\ \mathbf{case}_1\ b\ \mathbf{of}\ \{\mathbf{inl}\, t \to x,\ \mathbf{inr}\, f \to y\}$. We also need to consume the units t and f, but the eliminators are left out for brevity. Eliminations, $\mathbf{fst}\, M$ and $\mathbf{snd}\, M$, are defined as $M\ (\mathbf{inl}(\,))$ and $M\ (\mathbf{inr}(\,))$. Importantly, we cannot extract both elements of an additive pair without applying the function multiple times. This restriction guarantees resource soundness.

However, while this encoding has the necessary properties to qualify as an additive pair, it is not without issues. In dependent type theories, we often use the identity type to ask whether two values are equal. Functions are notoriously hard to compare, since these theories often lack the function extensionality principle.

Moreover, the definition of $S \,\&\, T$ does not allow any dependency between S and T. Both types are on separate branches of the additive union eliminator and thus must be independent. Notice that dependency between S and T is, in principle, possible. Even though we are only allowed to access one element of the pair in a relevant context, this restriction does not extend to irrelevant contexts.

Instead of using a combination of unions and functions, additive pairs, $(x : S) \,\&\, T$, are defined explicitly. The variable x can occur in the type T. We use the same syntax for introduction, $\langle x, y\rangle$, and elimination, $\mathbf{fst}\ M$ and $\mathbf{snd}\ M$.

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

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

5.4 Additive Unit

The multiplicative unit can be viewed as a neutral element to multiplicative pairs. Indeed, the types $(x \mathbin{:^1} S) \otimes \mathbf{1}$ and S are isomorphic. Similarly, an additive unit, $\top$, should be a neutral element to additive pairs.

One side of the isomorphism is trivial. When given $p \mathbin{:^1} S \,\&\, \top$, we can extract a value of type S by using $\mathbf{fst}\ p$. The other direction presents a problem. When

given $s \overset{1}{:} S$, we need to construct a pair $\langle s, \langle\rangle\rangle$ where $\langle\rangle$ is the additive unit introduction. However, the use of the variable s is inconsistent between the two elements. The introduction $\langle\rangle$ must thus count as a use of the variable s. The additive unit represents some unknown, but unused resources. And since there is no general way to consume unknown resources, the unit must not be eliminated.

We can define $\top$ as a function $(e \overset{1}{:} \bot) \to \mathbf{case}\ e\ \mathbf{of}\ \{\}$. A value of the empty type can be used to consume any variable, for example by eliminating it into a function and then applying that function to the desired variables. This function cannot be eliminated as it would require us to provide a value of the empty type.

We can also define the type $\top$ and its introduction $\langle\rangle$ explicitly. Notice the introduction works in any context Γ, not just 0Γ. No elimination is present.

$$\frac{0\Gamma \vdash}{0\Gamma \vdash \top \overset{0}{:} \mathcal{U}}\ \top\text{-F} \qquad\qquad \frac{0\Gamma \vdash}{\Gamma \vdash \langle\rangle \overset{?}{:} \top}\ \top\text{-I}$$

5.5 Usage

Since an additive zero, multiplicative pairs, and additive unions are commonly used types in functional programming, we focus on possible uses of an additive unit and additive pairs. Additionally, we restrict ourselves to linear uses of these types. If values of these types can be discarded or duplicated, they are functionally equivalent to the normal, intuitionistic pairs and units.

The uses of the additive unit are rather limited. Once a value is introduced, it can never be removed. Using it to signal some kind of resource error is also not ideal, since such errors cannot be handled. However, because the additive unit is a neutral element to additive pairs, it can be used as the base case for type operations. Suppose we have a type of lists, **List**, together with two introductions, **Cons** and **Nil**, and an eliminator, **foldr**. When given a list of types, we can construct an *additive tuple* of these types as follows.

$$\vdash \mathbf{foldr}\ \mathcal{U}\ \mathcal{U}\ (\lambda T\ R.\ (\_ : T)\ \&\ R)\ \top \overset{0}{:} (\_ \overset{0}{:} \mathbf{List}\ \mathcal{U}) \to \mathcal{U}$$

In functional programming, higher-order functions are frequently used in place of control flow statements. Take for example the **if** function. It takes a value of type $\mathbf{2}$, two values of some type T and returns a value of type T. We make the function polymorphic by quantifying over the type T, which gives us the type $(T \overset{0}{:} \mathcal{U}) \to \mathbf{2} \to T \to T \to T$.

If we attempt to mark the other parameters with multiplicities, we soon run into a problem. The parameter of type $\mathbf{2}$ is used exactly once, but of the two following parameters, only one is used and the other is discarded. One possible solution is to use the multiplicity ω and rely on weakening to discard one of the values. This change severely limits this function's usefulness in linear contexts.

The requirement that exactly one value must be used can be expressed using an additive pair, which allows us to give the **if** function a fully linear type: $(T \overset{0}{:} \mathcal{U}) \to (b \overset{1}{:} \mathbf{2}) \to (c \overset{1}{:} (\_ : T)\ \&\ T) \to T$. Indeed, whenever we need to express this kind of 1-of-n requirement, we can use the aforementioned additive tuple.

A *paramorphism* is a recursion scheme where the function handling the recursive step has access to the result of the recursion and the remainder of the original structure. To ensure the elements of a given structure are consumed once, the combining function must either use the recursive result or the remainder, but not both. Since the base case of the recursion might not be reached, we must be allowed to discard it. We define the list paramorphism, **para**, via **foldr**.

$$\begin{aligned}
&\vdash \lambda S\ T\ c\ n\ l.\ \mathbf{snd}\ (\mathbf{foldr}\ S\ ((_ : \mathbf{List}\ S)\ \&\ T)\\
&\qquad (\lambda a\ b.\ \langle \mathbf{Cons}\ a\ (\mathbf{fst}\ b),\ c\ a\ b\rangle)\ \langle \mathbf{Nil},\ n\rangle\ l)\\
&\quad \overset{\omega}{\vdots}\ (S \overset{0}{\vdots} \mathcal{U}) \to (T \overset{0}{\vdots} \mathcal{U}) \to (c \overset{\omega}{\vdots} (a \overset{1}{\vdots} S) \to (b \overset{1}{\vdots} (_ : \mathbf{List}\ S)\ \&\ T) \to T) \to\\
&\qquad (n \overset{\omega}{\vdots} T) \to (l \overset{1}{\vdots} \mathbf{List}\ S) \to T
\end{aligned}$$

Note that in practice, **para** would either be given directly or implemented recursively. The **foldr** implementation needlessly reconstructs the original list.

We can use this recursion scheme to implement an insertion operation for sorted lists. Once we find the point of insertion, we do not need to traverse through the rest of the list and we can use the remainder. At first glance, this operation cannot be linear: the elements need to be compared and then used in the resulting list. To solve this problem, we can use a comparison operation that also returns the compared values, such as $\mathbf{leq} \overset{\omega}{\vdots} (p \overset{1}{\vdots} A \otimes A) \to \mathbf{2} \otimes A \otimes A$.

$$\begin{aligned}
&\vdash \lambda v\ l.\ \mathbf{para}\ A\ ((_ \overset{1}{\vdots} A) \to \mathbf{List}A)\\
&\qquad (\lambda a\ w\ i.\ \mathbf{let}_1\ (b,\ p) = \mathbf{leq}\ (a,\ i)\ \mathbf{in}\ \mathbf{let}_1\ (a',\ i') = p\ \mathbf{in}\\
&\qquad\quad \mathbf{if}\ (\mathbf{List}\ A)\ b\ \langle \mathbf{Cons}\ a'\ (\mathbf{snd}\ w\ i'),\ \mathbf{Cons}\ i'\ (\mathbf{Cons}\ a'\ (\mathbf{fst}\ w))\rangle)\\
&\qquad (\lambda i.\ \mathbf{Cons}\ i\ \mathbf{Nil})\ l\ v \overset{\omega}{\vdots} (v \overset{1}{\vdots} A) \to (l \overset{1}{\vdots} \mathbf{List}\ A) \to \mathbf{List}\ A
\end{aligned}$$

A dependent additive pair type can be used to generalize the type of **para** and allow the type of the result to depend on the input list. This function can be defined via a dependent **foldr**, but the implementation has to carry a proof that the original and reconstructed lists are the same, and is skipped for brevity.

$$\begin{aligned}
&\vdash \mathbf{para}' \overset{\omega}{\vdots} (S \overset{0}{\vdots} \mathcal{U}) \to (T \overset{0}{\vdots} (l \overset{0}{\vdots} \mathbf{List}\ S) \to \mathcal{U}) \to\\
&\qquad (c \overset{\omega}{\vdots} (a \overset{1}{\vdots} S) \to (b \overset{1}{\vdots} (r : \mathbf{List}\ S)\ \&\ T\ r) \to T\ (\mathbf{Cons}\ a\ (\mathbf{fst}\ b))) \to\\
&\qquad (n \overset{\omega}{\vdots} T\ \mathbf{Nil}) \to (l \overset{1}{\vdots} \mathbf{List}\ S) \to T\ l
\end{aligned}$$

6 Bidirectional Type System

In contrast with the declarative presentation of the typing rules, there also exists the *bidirectional* form, which reduces the number of explicit type annotations the user of a language with these typing rules is expected to provide, thus the implementation of the typing algorithm is based on a bidirectional form of the rules.

In bidirectional presentation of the typing rules, judgments come in two types: the *type checking* judgment, $\Gamma \vdash M \overset{\sigma}{\Leftarrow} S$, and the *type synthesising* judgment,

$\Gamma \vdash M \overset{\sigma}{\Rightarrow} S$. The difference is that in type checking the type S is an input of the judgment and the derivation proves that the term M can indeed be of type S; whereas, in type synthesis the type S is an output derived from the term M.

Every term M that synthesises type S can also have its type checked using S, thus, in bidirectional formulation of typing rules, we need to decide which terms can have their type synthesised and which terms can only have its type checked. In our work we follow the Pfenning recipe [7]:

> If the rule is an *introduction* rule, make the principal judgment *checking* and if the rule is an *elimination* rule, make the principal judgment *synthesising*.

A *principal judgment* is the judgment containing the logical connective that is being introduced or eliminated.

In the bidirectional form, the order in which the premises are derived is important, because one judgment may synthesise a type that is later used in a different judgment. We derive them from left to right. For example, in the following rule we first synthesise the type $(x \overset{\pi}{:} S) \to T$ from M and only then can we use the S to check N with.

$$\frac{\Gamma_1 \vdash M \overset{\sigma}{\Rightarrow} (x \overset{\pi}{:} S) \to T \qquad \Gamma_2 \vdash N \overset{1}{\Leftarrow} S}{\Gamma_1 + \sigma\pi\Gamma_2 \vdash M\ N \overset{\sigma}{\Rightarrow} T[N/x]} \to\text{-E}_1$$

The rules above are translated into their bidirectional form using the recipe. We introduce an additional type-synthesising term, $M : S$, and an associated rule that forms it from two type-checking terms, M and S.

$$\frac{0\Gamma \vdash S \overset{0}{\Leftarrow} \mathcal{U} \qquad \Gamma \vdash M \overset{\sigma}{\Leftarrow} S}{\Gamma \vdash M : S \overset{\sigma}{\Rightarrow} S}\ \textsc{Ann}$$

7 Implementation

This work is accompanied by an interpreter of a language based on hitherto described extensions to QTT. To implement an efficient typing algorithm, the typing rules had to be modified somewhat, as we describe below.

7.1 Language Interpreter

The accompanying implementation is written in Haskell and, while originally based on the dependently typed λ-calculus interpreter *LambdaPi* [9], has had all its parts rewritten from scratch, making use of modern Haskell packages: parser is written using the parser combinator library *Megaparsec*; text-based user interface is based on the *Haskeline* library, wrapped in *Repline*; and the pretty-printer uses the *Prettyprinter* package.

The Haskell modules that constitute our implementation are documented throughout, using Haddock-style comments.

7.2 Typing Algorithm

To translate the typing rules into an efficient algorithm, we would need to have a way of a priori deciding how to split the resources in a given context when deriving multiple premises; for example, in the rule $\to$-E$_1$, we need to split the context $\Gamma_1 + \sigma\pi\Gamma_2$ into Γ_1 and Γ_2. Instead, we consider an additional output for our judgments: recorded resource *usage* of the judged term. Usage accumulates the resource consumption from throughout the term and the typing algorithm uses it to check that an appropriate amount of every resource is available. Thus, in the example of rule $\to$-E$_1$, there is no need to know how to split $\Gamma_1 + \sigma\pi\Gamma_2$, instead the premises produce usages Ψ from M and Ω from N, which get combined into $\Psi + \Omega$ and returned as the usage of $M\ N$. The addition on usages works analogously to contexts.

To judge that the resource usage of a term is appropriate for the given context, we check that the collected usage of a variable is less than or equal to, with regards to the given semiring ordering $\leq$, the multiplicity with which it occurs in the context of a judgment. For *global variables*, i.e. variables that are free in the whole term, this check is done at the end of the typing algorithm, but for the *local variables*, i.e. variables that have a binding occurrence somewhere in the term, the check is done after accumulating its usage from its scope.

Since the usage check takes the weakening into consideration, we no longer need the explicit Weak rule. Consequently, our typing rules almost completely *syntax-directed*; that is, the derivation rule which can be used is determined by the syntax of the term. There are two exceptions to this: first, when constructing a multiplicative pair, we first check if $\sigma\pi = 0$ and use the rule $\otimes$-I$_0$, otherwise we use $\otimes$-I$_1$; and second, when applying a function, we synthesise the type of the function first and then check if $\sigma\pi = 0$, using the rule $\to$-E$_0$ if true and $\to$-E$_1$ otherwise.

8 Conclusion

Additive types are an important part of substructural type systems. Indeed, one of the most commonly used types in functional programming is the disjoint union, which is typically only available in the additive version. However, other additive types are used rather sparingly. QTT allows us to explore these types in a fully dependent setting.

We present an extended theory containing an additive unit and dependent additive pairs. Thanks to the annotated eliminators, the theory is also more expressive in the presence of the weakening rule. A variety of examples show how these extensions may be used. We demonstrate the viability of the theory by providing a bidirectional type system and its implementation in an interpreter.

We hope that this work encourages both existing and future treatments of QTT to consider these additive types as first class citizens, rather than relying on an inconvenient and limited encoding via functions.

References

1. 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
2. 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
3. Brady, E., et al.: Idris2. `https://github.com/idris-lang/Idris2` (2022)
4. 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
5. Cervesato, I., Pfenning, F.: A Linear Logical Framework. Information and Computation **179**(1), 19–75 (2002). https://doi.org/10.1006/inco.2001.2951
6. 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
7. Dunfield, J., Krishnaswami, N.: Bidirectional Typing. ACM Comput. Surv. **54**(5) (May 2021). https://doi.org/10.1145/3450952
8. 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
9. Löh, A., McBride, C., Swierstra, W.: A Tutorial Implementation of a Dependently Typed Lambda Calculus. Fundamenta Informaticae **102**(2), 177–207 (2010). https://doi.org/10.3233/fi-2010-304
10. 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
11. The Anoma Team: Juvix. `https://github.com/anoma/juvix` (2022)