Functional Pearl — Programming with Negative, Fractional, Square Root, and Imaginary Types

Roshan P. James
Indiana University
rpjames@indiana.edu

Amr Sabry
Indiana University
sabry@indiana.edu

Abstract

Every functional programmer knows about sum and product types, $a + b$ and $a * b$ respectively. Negative, fractional, square root, and imaginary types, $a - b$, a/b, $\sqrt{a}$, and $a + ib$ respectively, are much less known and their computational interpretation is unfamiliar and often complicated. We show that in a programming model in which information is preserved (such as the model introduced in our recent paper on *Information Effects*), these types have particularly intuitive and natural computational interpretations. Intuitively, values of negative types are values that flow "backwards" to satisfy demands, values of fractional types are values that represent first-class "structural" constraints, and values of square root and imaginary types are values that can be related by arbitrary algebraic constraints. The combination of these negative, fractional, square root, and imaginary types enable greater flexibility in programming by breaking complicated invariants into local ones that can be independently satisfied by a subcomputation. Furthermore, they allow a programmer to express a wide range of programming idioms, including higher-order functions, delimited continuations, speculative computation, logic programming, and more.

Categories and Subject Descriptors D.3.1 [*Formal Definitions and Theory*]; F.3.2 [*Semantics of Programming Languages*]; F.3.3 [*Studies of Program Constructs*]: Type structure

General Terms Languages, Theory

Keywords continuations, information flow, linear logic, logic programming, quantum computing, reversible logic, symmetric monoidal categories, compact closed categories.

1. Introduction

The world of computation we are describing has:

- suppliers,
- consumers, and
- bi-directional transformations

This is the same world described by the papers on the duality of computation but that work only scratched the surface! We have the following features:

- we can start from the supplier and push the values towards the consumer (call-by-value in the duality of computation papers)
- we can start from the consumer and pull the values from the suppliers (call-by-name in the duality of computation papers)
- we can combine the pushing and pulling and values using eta/epsilon for sum types; these allow us to at any point in the middle of the computation create out of nothing a value to send to the consumers and a demand to send to the suppliers.
- we can break a big data structure into fragments described by fractional types; the suppliers and consumers can produce and consume the pieces completely independently of each other. Eventually the pieces will fit together at the consumer to produce the desired output.
- we can break a bi-directional transformation into pieces using square roots
- we can take into account that values have phase (complex numbers), i.e., it is not that they flow towards the consumer or just towards the suppliers; they can be flowing in direction that "30 degrees" towards the consumer for example.

So it is all about breaking dependencies in some sense to allow for maximum autonomy (parallelism) of subcomputations. It is probably the case that to make full use of square root types and imaginary types, we have to move to a vector space. If that's the case, we should probably leave this stuff out and focus on negative and fractional types and only have a short discussion of the polynomials restricted to seven trees in one and similar issues.

The conventional idea is to divide the world into a "real" one and a "virtual" one. In the "real" world, we can define datatypes like `t=t^2+1` but we don't have additive inverses so it makes no sense to talk of negative types and we can't rearrange the terms in the datatype definition. However the observation is that we can map these datatypes to a virtual world that has more structure (a ring that provides additive inverses or a field that also provides multiplicative inverses) and then perform computations in the ring/field. If we perform computations in the ring, then some of these will use additive inverses in ways that cannot be mapped back to the "real" world. Much of current research attempts to characterize which computations done in the ring are valid isomorphisms between datatypes in the "real" world. This is nice but is not what I am after. In fact I am not interested in the ring or the semiring at all. I am interested in the field and I want this field to be **the real world.** This is partly motivated by the fact that Quantum Mechanics seems to demand an underlying field and more generally that the field provides the maximum generality in slicing and dicing computations. So assuming I live in a field and that the negative, fractional, square root, and imaginary types are all "real," how do I compute in this field? Clearly there will be constraints on "measurement" in the

[Copyright notice will appear here once 'preprint' option is removed.]

sense that a full program cannot produce any of the crazy types but that's done outside the formalism in some sense just as in Quantum Computing. The main question I am after is how to compute in this field with first-class negative, fractional, etc. types. As I mentioned in my previous email, we can produce programs that have types `t^3 <-> -1` and they "run" (but only to give infinite loops).

So when a programmer writes the datatype declaration `t = t^2+1`, if we allow negative etc. then this is effectively writing `t = cubicroot{-1}`.The example suggests that the ability to consider values flowing backwards enriches our computational model. This observation goes back to Filinski's Masters thesis where continuations are identified with these negative values. We discuss the connections to Filinski's work and others in more detail in Sec. 8. For now, we note that in a conventional programming language in which values can be copied and deleted at will, extreme care is needed to keep track of negative values. Indeed, in the example above, if it were possible to simply delete the variable corresponding to the debt, we would have produced money out of nothing.

If we are in the field then computations that manipulate these trees can be sliced and diced even at interfaces that expose the cubic root and the imaginary types.

Future work: develop a type system for a "normal language" that has negative, fractional, etc. types as first-class types. More long term, instead of adding one polynomial at a time, we can go to an algebraically closed field. The complex numbers is an obvious choice but I would rather go to something computable like the field of algebraic numbers. Is the adele ring or the p-adics relevant here?

We show a deep symmetry between functions and delimited continuations, values and continuations that arises in Π in a manner that is reminiscent of Filinksi's Symmetric λ-calculus [2]. The symmetry arises by extending Π with a notion of additive duality over the monoid $(+, 0)$ by including *eta* and *eps* operators of Compact Closed Categories. The resulting dual types, which we denote $-b$, have a time traveling "backward information flow" interpretation and allow for the encoding of higher-order function and iteration via the construction of *trace* operators, thereby making the extended language $\Pi^{\eta\epsilon}$ a Turing-complete reversible programming language with higher-order functions and first-class delimited continuations.

We introduced this thesis that computation should be based on isomorphisms that preserve information [3]. Since Filinski, we've had the idea that values and continuations are like mirror images. In a conventional language, the negative (continuation) side is implicit and we introduce information effects on the positive. Trying to recover the duality from this distorted positive side has always been messy. Now it looks clean because we have kept the positive side pure.

The way to think about something of type A is that it is a value we have produced. The way to think about something of type $-A$ is that is a value we have already consumed.

Other interpretations of the types of think about. The first one is arithmetic obviously. Another one is languages consisting of sets of string. The type 0 is the empty set, the type 1 is the set containing the empty word, the + constructor corresponds to union, and the $*$ constructor corresponds to concatenation. The constructor − would not correspond to set difference however. It would correspond to marking the elements in the set as "consumed" so that if we take the union and a "consumed" element appears in the other set, the two cancel. This makes it clear that concatenating a produced a and a consumed b is not the same as concatenating a consumed a and a produced b. They really need to be kept separate. Incidentally, division would be defined as follows:

$$L_1/L_2 = \{x \mid xy \in L_1 \text{ for some } y \in L_2\}$$

Connections to type logical grammars and Lambek calculus

Some background on Π because it is mentioned in the next section.

2. Negative Types: Intuition

Consider the following two ways of purchasing an item that costs $20.00:

1. You give the seller a $20.00 bill.
2. You use a credit card to pay the seller.

In both cases, the seller receives the money immediately but there is a subtle difference. In the first transaction, the money received by the seller corresponds to a value that has been produced earlier in the computation. In the second transaction, the money may or may not exist yet: computationally, a *debt* is generated to compensate for the money received by the seller and this debt travels "backwards" towards the bank where it is (hopefully) reconciled.

For this reason, our treatment of negative values in the context of Π is particularly simple. We will have a type 0 and an isomorphism $0 \leftrightarrow a + (-a)$ that when used in the left-to-right direction allows us to create a value and a corresponding debt out of nothing. Both the value and its negative counterpart can flow anywhere in the system. Because information is preserved, a closed program (which does not have any "dangling" negative values) will eventually have to match the negative value with some corresponding value, effectively using the isomorphism in the right-to-left direction. In more detail, we can model the credit card transaction above using the following program (shown diagrammatically):

FIGURE

CLEAN UP the following based on the figure. In contrast, in our setting, one can start from the empty type 0, introduce a positive value and its negative counterpart, and let each of these flow in arbitrary ways. The entire framework guarantees that neither the value nor its negative counterpart will be deleted or duplicated and hence that, in any closed program, the "debt" corresponding to the negative value is paid off exactly once. Computationally we model the first transaction using essentially the identity function which receives a $20.00 bill as input from the buyer and propagates it on its output to the seller. The second transaction is more complicated. We model it as shown in the circuit below: There are two ways to understand this circuit that are both quite instructive. Let's first examine the type structure of each of the combinators that comprise the circuit. The first combinator on the right outputs $20.00 to the seller. This $20.00 is produced from nothing so to speak by generating an equivalent debt that travels backwards. COMPLETE BASED ON THE FIGURE. The other way is to follow the execution. It goes forward in time so to speak, comes back, and then goes forwards again.

It is critical that the framework in which the negative types are introduced is a framework in which all information is preserved, with no duplication or erasure. This guarantees that the generated debts are paid once and exactly once.

3. Fractional Types: Intuition

The type a/b is prominent in the Lambek-Grishin calculus which is extensively used in computational linguistics. In the common interpretation, a value of type a/b is a value of some type c such that when put in a context of type b, the result is a value of type a. For example, assuming contexts are represented as functions, a value of type `bool`/(`int` → `bool`) is simply a value of type `int`. Indeed in this case, putting a value of type `int` in the context (`int` → `bool`) produces a value of type `bool`.

In our case, the situation is simpler. If the goal is to produce a value of type $a \times b$ and a subcomputation can only produce the a-part, it would have type $(a \times b)/b$. In most settings, this type would

be equivalent to a. However in our setting, the constraint that this value must eventually fit in a larger value that supplies the b is explicitly recorded and must be resolved. In more detail, we have an isomorphism $1 \leftrightarrow a \times (1/a)$ which when used in the left-to-right direction allows the creation of a value and a corresponding constraint on the context. Both the value and the constraint can flow in arbitrary ways during the computation but eventually in a closed program with no "dangling" constraints, the isomorphism should be used in the right-to-left direction to resolve the constraints. To understand the computational interpretation, consider the following example.

The goal is to produce a value of type (`bool` $\times$ `int`). One part of the program, e_1, knows how to produce the value of type `int` (say 3) and another part, e_2, knows how to produce the value of type `bool` (say *true*). Computationally, we model this as follows. The first subcomputation e_1 produces $((), 3)$ and uses the isomorphism to convert () to $(alpha, 1/alpha)$ where *alpha* is a yet-unknown boolean value. Reshuffling the components, e_1 produces $((alpha, 3), 1/alpha)$. Similarly, e_2 can independently produce $((true, beta), 1/beta)$ where *beta* is a yet-unknown integer value. When the two values meet, we can group the components as follows:

$$(alpha, beta)(3, 1/beta)(true, 1/alpha)$$

Using the isomorphism in the right-to-left direction on the last two pairs, forces *alpha* to be resolved to *true* and forces *beta* to be resolved to 3. Both of these pairs then become () and be absorbed. We are left with the pair $(true, 3)$ as desired.

To summarize, one can think of a value of type a/b as a value that imposes a constraint on its context of use: it is a value that requires its context to be of type b and only if that condition is satisfied, can the value be considered as a value of type a. In the simplest case, the type $1/b$ just constraints the context to be of type b. By allowing the fractional types to be first-class, we can separate the generation of constraints from their use. Both the value and the constraint can flow in arbitrary ways during the computation but eventually in a closed program with no "dangling" constraints, the isomorphism should be used in the right-to-left direction to resolve the constraints.

NEED to work out the example in detail (or perhaps another example that's better.

Again it is critical that the framework in which the fractional types are introduced is a framework in which all information is preserved, with no duplication or erasure. This guarantees that the generated constraints are satisfied once and exactly once.

4. Algebraic Types: Intuition

Square root and imaginary types have also appeared in the literature: we relegate the connections to Sec. **??** and proceed with a simple explanation. We have so far extended the set of types to be the rational numbers. Now we will push this and extend the set of types to algebraic numbers. In other words, we will allow datatypes defined by arbitrary polynomials and allow the roots of such polynomials to be types.

Consider first an example in which we want to compute with the sides of a rectangle whose area is 91 and whose length is longer than its width by 6 units. One can solve the quadratic equation to determine that the sides are 7 and 13 and proceed. This however prematuraly forces us to globally solve the constraint. Instead we can let the two sides of the rectangle be x and $x + 6$ and use the following equation to capture the desired constraint:

$$x^2 + 6x - 91 = 0$$

The equation introduces an isomorphism between the type $x^2 + 6x - 91$ and the type 0. We can now proceed to compute with the unknown x, being assured that in a closed program, our computation will eventually be consistent with the solution of the algebraic equation.

Previously the most famous example of a similar nature is the puzzling isomorphism that one can establish between seven binary trees and one. A binary tree is defined by the datatype

$$x = 1 + x * x$$

which can be rearranged to the polynomial equation $x^2 - x + 1 = 0$. By algebraic manipulation we can reason as follows:

$$\begin{array}{lllllllll} x^3 & = & x^2x & = & (x-1)x & = & x^2 - x & = & -1 \\ x^6 & = & 1 & & & & & & \\ x^7 & = & x^6x & = & x & & & & \end{array}$$

Fiore poses the question of why such algebraic manipulation would make sense type theoretically but states that even though some of the intermediate steps make no sense, the final equivalence is valid and can be used to actually construct an isomorphism between x^7 the type of seven binary trees and x the type of binary trees.

Discussion of possible polynomials:

- $b = 1 + 1$. Boring.
- $b = b + 1$. That introduces the natural numbers. No solution for this polynomial over the algebraic numbers. We must extend the numbers with ω to get a solution. We reject this in this paper and prefer to stick to algebraic numbers. The advantage is that all the isomorphisms are valid numerically. With the above type we could subtract b from both sides to show that $0 = 1$ which is nonsense. We can however have infinite types as long as they have algebraic solutions.
- $b^2 = 2$ or $b = \pm\sqrt{2}$. We have introduced the square root of a boolean! If we had superpositions, we could then write a function that performs the square root of negation in the sense that applying it twice would be equivalent to boolean negation.

5. Summary

To summarize negative, fractional, square root, and imaginary types all make sense. What they help you accomplish as a programmer is to disassociate global invariants into local ones that can be satisfied independently by subcomputations with no synchronization or communication. A computation producing something of type a/b does not need to concern itself with who is going to supply the missing b: it just does its part. Conversely faced with a complicated task, a computation might decide to break it into pieces and demand these pieces using negative types.

It is no surprise that these types are closely related to quantum mechanics and that they give us the feel that this is how nature computes. This is speculation however.

In any case, in a framework where information can be copied and deleted, none of this makes much sense. It is critical that these constraints and demands can neither be duplicated nor erased. This gives us the maximum "parallelism" possible.

6. Additive Duality in Π

6.1 Syntax

$$\begin{array}{rcl} \mathit{Values}, v & = & () \mid (v, v) \mid L\ v \mid R\ v \\ \mathit{Combinators}, c & = & \mathit{iso} \mid c \fatsemi c \mid c + c \mid c \times c \end{array}$$

$$\begin{array}{rcl} \mathit{Sequential\ Contexts}, P, F & = & \Box \mid P : c \\ \mathit{Parallel\ Contexts}, D & = & \Box \\ & \mid & D : (P \mid \Box + c \mid F) \\ & \mid & D : (P \mid c + \Box \mid F) \\ & \mid & D : (P \mid \Box \times c\ v \mid F) \\ & \mid & D : (P \mid \Box \times v\ c \mid F) \\ & \mid & D : (P \mid c\ v \times \Box \mid F) \\ & \mid & D : (P \mid v\ c \times \Box \mid F) \end{array}$$

$$\begin{array}{rcl} \mathit{Machine\ States} & = & D[P \mid c\ v \mid F] \mid D[P \mid v\ c \mid F] \\ \mathit{Start\ State} & = & \Box[\Box \mid v\ c \mid \Box] \\ \mathit{Stop\ State} & = & \Box[P \mid c\ v \mid \Box] \end{array}$$

Combinator reconstruction, $P[c]$:

$$\begin{array}{rcl} \Box\,(c) & = & c \\ P : c'(c) & = & P(c' \fatsemi c) \end{array}$$

6.2 Operational Semantics

The small step semantics present for Π below work symmetrically for forward and backward evaluation.

- Basic reduction of an isomorphism. Note that the evaluation leaves the adjoint of the combinator behind. This will become important when we reverse the direction of computation.

$$D[P|v\ \mathit{iso}|F] \quad \mapsto \quad D[P|\mathit{iso}^\dagger v|F]$$

- Sequencing involves pushing and popping from the Future and Past continuations:

$$\begin{array}{rcl} D[P|v\ (c_1 \fatsemi c_2)|F] & \mapsto & D[P|v\ c_1|F : c_2] \\ D[P|c_1\ v|F : c_2] & \mapsto & D[P : c_1|v\ c_2|F] \end{array}$$

- Parallel composition, captures the current Future and Past and extends the parallel context.

$$\begin{array}{rcl} D[P|(L\ v)\ c_1 + c_2|F] & \mapsto & D : (P|\Box + c_2|F)[\Box|v\ c_1|\Box] \\ D[P|(R\ v)\ c_1 + c_2|F] & \mapsto & D : (P|c_1 + \Box|F)[\Box|v\ c_2|\Box] \\ D : (P|\Box + c_2|F)[P'|c_1\ v|\Box] & \mapsto & D[P|(P'(c_1) + {c_2}^\dagger)\ (L\ v)|F] \\ D : (P|c_1 + \Box|F)[P'|c_2\ v|\Box] & \mapsto & D[P|({c_1}^\dagger + P'(c_2))\ (R\ v)|F] \end{array}$$

- Similarly for products:

$$\begin{array}{rcl} D[P|(v_1, v_2)\ c_1 \times c_2|F] & \mapsto & D : (P|\Box \times v_2\ c_2|F)[\Box|v_1\ c_1|\Box] \\ D : (P|\Box \times v_2\ c_2|F)[P'|c_1\ v_1|\Box] & \mapsto & D : (P|P'(c_1)\ v_1 \times \Box|F)[\Box|v_2\ c_2|\Box] \\ D : (P|c_1\ v_1 \times \Box|F)[P'|c_2\ v_2|\Box] & \mapsto & D[P|c_1 \times P'(c_2)\ (v_1, v_2)|F] \end{array}$$

and symmetrical rules for evaluation along the second branch.

$$\begin{array}{rcl} D : (P|v_1\ c_1 \times \Box|F)[P'|c_2\ v_2|\Box] & \mapsto & D : (P, \Box \times P'(c_2)\ v_2, F)[\Box|v_1\ c_1|\Box] \\ D : (P|\Box \times c_2\ v_2|F)[P'|c_1\ v_1|\Box] & \mapsto & D[P|P'(c_1) \times c_2\ (v_1, v_2)|F] \end{array}$$

The later two rules will be relevant only for reverse execution.

6.3 Rules for *eta* and *eps*

The operation *eps* reverses the direction of a particle by reversing the world.

Note that we deviate from the categorical definition of *eta* and *eps* slightly in that they swap the order of $-b$ and b in choice of *eta*. This however does not affect us, because we deal with a symmetric category.

- Grammar

$$\begin{array}{rcl} \mathit{Values}, v & = & () \mid (v, v) \mid L\ v \mid R\ v \mid -v \\ \mathit{Isomorphisms}, \mathit{iso} & = & \ldots \mid \mathit{eta} \mid \mathit{eps} \\ \mathit{Combinators}, c & = & \mathit{iso} \mid c \fatsemi c \mid c + c \mid c \times c \end{array}$$

- Type judgement.

$$\mathit{eta} \quad : 0 \leftrightarrow (-b) + b : \quad \mathit{eps}$$

$$\frac{\vdash v : b}{\vdash -v : -b}$$

- Operational Semantics.

$$\begin{array}{rcl} D[P|(R\ v)\ \mathit{eps}|F] & \mapsto & D^\dagger[F|\mathit{eps}\ (L\ (-v))|P] \\ D[P|(L\ (-v))\ \mathit{eps}|F] & \mapsto & D^\dagger[F|\mathit{eps}\ (R\ v)|P] \end{array}$$

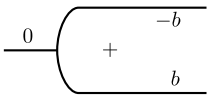
Note: there is NO reduction rule for *eta*.

- The adjoint of a parallel context is defined to be:

$$\begin{array}{rcl} \Box^\dagger & = & \Box \\ (D : (P, \Box + c, F))^\dagger & = & D^\dagger : (F, \Box + c^\dagger, P) \\ (D : (P, c + \Box, F))^\dagger & = & D^\dagger : (F, c^\dagger + \Box, P) \\ (D : (P, \Box \times c\ v, F))^\dagger & = & D^\dagger : (F, \Box \times v\ c, P) \\ (D : (P, \Box \times v\ c, F))^\dagger & = & D^\dagger : (F, \Box \times c\ v, P) \\ (D : (P, c\ v \times \Box, F))^\dagger & = & D^\dagger : (F, v\ c \times \Box, P) \\ (D : (P, v\ c \times \Box, F))^\dagger & = & D^\dagger : (F, c\ v \times \Box, P) \end{array}$$

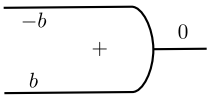
6.4 Diagrams

eta

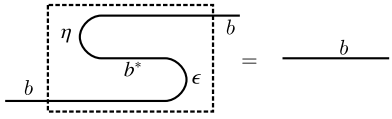


Note that the connective is a +, hence only one of the branches may be inhabited at any time. Thus the action of *eta* is to transfer a backward flowing value on one wire to a forward flowing value on the other wire.

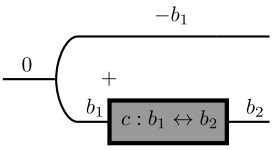
eps



Coherence condition

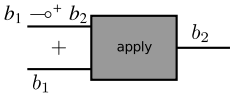


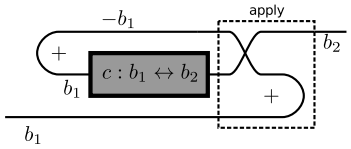
Function



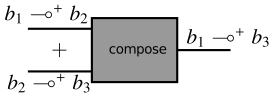
Let us use the shorthand $b_1 \multimap b_2 = -b_1 + b_2$

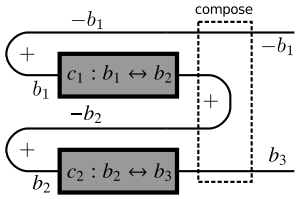
Function application



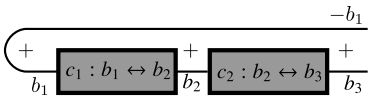


Function composition

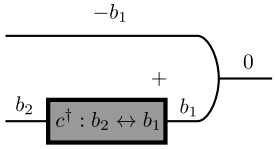




This is also equivalent to sequencing both the computation blocks.

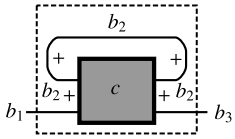


Delimited continuation

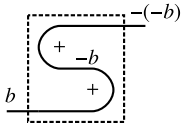


Trace

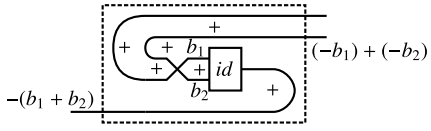
$$\frac{c : b_2 + b_1 \leftrightarrow b_2 + b_3}{trace\ c : b_1 \leftrightarrow b_3}$$



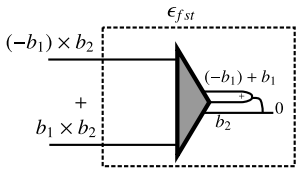
Double Negation
$b \leftrightarrow -(-b)$



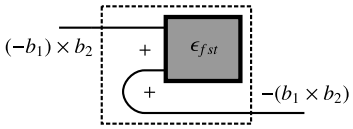
Negation distributes over +.
$-(b_1 + b_2) \leftrightarrow (b_1) + (-b_2)$



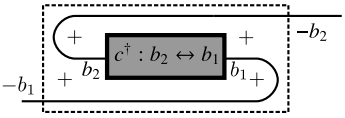
eps_{fst}
$(-b_1) \times b_2 + b_1 \times b_2 \leftrightarrow 0$



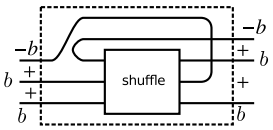
Lifting negation out of $\times$.
$(-b_1) \times b_2 \leftrightarrow -(b_1 \times b_2) \leftrightarrow b_1 \times (-b_2)$



The following isomoprhism can be constructed similarly
$b_1 \times b_2 \leftrightarrow (-b_1) \times (-b_2)$
Lifting a operation of postive types to negated types:
Given $c : b_1 \leftrightarrow b_2$



Observability. Execution of program is defined by $c : b_1 \leftrightarrow b_2$ when evaluated on input $v_1 : b_1$ gives us a value $v_2 : b_2$ on termination. Execution is well defined only if b_1 and b_2 are entirely positive types. Consider the program that

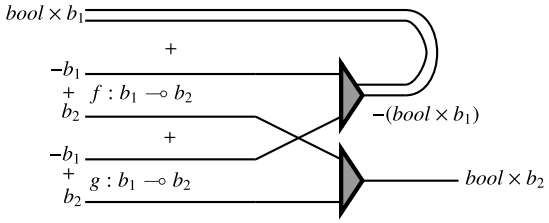


We define observables to be only positive types. The ouputs of programs that ouput mixed positive and negative types are not observable. Also, programs that input mixed positive and negatives types are not executable.

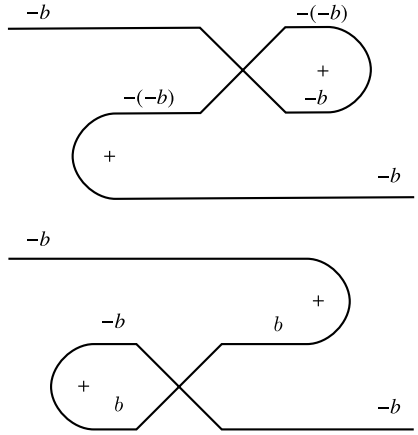
6.5 To think about

- If we built an effect over $\Pi^{\eta\epsilon}$, say *create* and *erase*, are effects structured by an arrow or a monad now?
- Which operations can be lifted to work on negative types?
- Can the operational semantics for $\Pi^{\eta\epsilon}$ be an interpreter that is implemented in Π^o (similar to how the the tree traversal interpreter was implemented). This would imply the existence of a more powerful construction than Int, wherein the products would also be preserved. This is possibly worth a paper in itself.
- These functions aren't really values. There is no value one can produce that denotes a function. These functions are the ability to transform a value - the possibility of transforming a value. In the product encoding of environments, there is no value one can assign to a variable such that it denotes a function.

  Actually this is possible. Consider two functions $f : b_1 \leftrightarrow b_2$ and $g : b_1 \leftrightarrow b_2$. A value of type *bool* is sufficient to discriminate them. Hence the *boo* is the first class representation of the functions and can be thought of as the address of the function.



- It is not fair to say that negative types flow backwards. The following circuits are valid in $\Pi^{\eta\epsilon}$. It is however proper to say that for any type b that flows in a direction, the type $-b$ flows in the reverse direction.



7. Multiplicative Duality in Π

8. Related Work

Filinski proposes that continuations are a *declarative* concept. He, furthermore, introduces a symmetric extension of the λ-calculus in which values and continuations are treated as opposites. This is essentially what we are proposing with one fundamental difference: our underlying language is not the λ-calculus but a language of pure isomorphisms in which information is preserved. This shift of perspective enables us to distill and generalize the duality of values and continuations: in particular, in the conventional λ-calculus setting values and continuations can be erased and duplicated which makes it difficult to maintain the correspondence between a value and its negative counterpart.

Continuations. The idea of using negative types to model information flowing backwards, demand for values, continuations, etc. goes back to at Filinski's thesis. We recall these connections below but we first note that all these systems are complicated because in all these systems information can be ignored, destroyed, or duplicated. Clearly the possibility of erasure of information would mean that our credit card transaction is incorrect. In our work, information is maintained and hence we have a guarantee that, in a closed program, the debt must be accounted and paid for.

Filinski [2]. In his Masters thesis, Filinski proposes that continuations are a *declarative* concept. He, furthermore, introduces a symmetric extension of the λ-calculus in which values and continuations are treated as opposites. This is essentially what we are proposing with one fundamental difference: our underlying language is not the λ-calculus but a language of pure isomorphisms in which information is preserved. This shift of perspective enables us to distill and generalize the duality of values and continuations: in particular, in the conventional λ-calculus setting values and continuations can be erased and duplicated which makes it difficult to maintain the correspondence between a value and its negative counterpart. In contrast, in our setting, one can start from the empty type 0, introduce a positive value and its negative counterpart, and let each of these flow in arbitrary ways. The entire framework guarantees that neither the value nor its negative counterpart will be deleted or duplicated and hence that, in any closed program, the "debt" corresponding to the negative value is paid off exactly once. The forward and backward executions in our framework correspond to call-by-value and call-by-name. This duality was observed by Filinski and others following him but it is particularly clean in our framework.

Subtraction Wadler the reloaded paper, does not consider the subtraction type because its "computational interpretation is not familiar." Curien and Herbelin study duality in classical logic, show that it exchanges call-by-value and call-by-name. They extend classical natural deduction with subtraction but give it no computational meaning. Crolard (in the formulae-as-types interpretation of subtractive logic) address the computational interpretation of substraction. He explains the type $A-B$ as the of *coroutines* with a local environment of type A and a continuation of type B. The description is complicated by what is essentially the desire to enforce linearity constraints so that coroutines cannot access the local environment of other coroutines. Must cite Selinger control categories in this context of duality but I am not sure what to say: it assumes cartesian closed categories for one thing and not symmetric monoidal ones. Ariola, Herbelin, and Sabry also use subtractive types to explain delimited continuations.

Linear Logic In accounts that are linear, the value and continuation that comprise the substractive type need to be constrained to "stay together." This can be achieved by various restrictions. In this work we have no such constraints, the negative value can flow anywhere. The entire system guarantees that any closed program would have to account for it. We don't have to introduce special constraints to achieve that. Zeilberger in the paper on polarity and the logic of delimited continuations uses polarized logic: he shows that if positive and negative values are completely symmetric except that answer types are positive, then the framework accommodates delimited continuations. But he interprets negative values are control operators, or as values defined by the shape of their continuations. We simply interpret values of negative type as values flowing in the "other" direction.

Int Construction. For a traced monoidal category C the Int construction produces a Compact Closed Category called Int C [4]. Further we know that the target of the Int construction is isomorphic to the target of $\mathcal{G}$ construction of Abramsky [1] from Haghverdi. However, note that the $\Pi^{\eta\epsilon}$ is not the same as the image of the Int construction on Π^{o}, since the the later lacks a multiplicative tensor that distrbutes over the additive tensor in Int Π^{o}.

Recursion. In our previous work we introduce recursive types and trace operators. This is dangerous here because infinite loops allow us to prolong paying the debt for as long as we want.

GoI machines We can now encode the GoI machine of Mackie [5, 6].

9. Conclusion

Acknowledgments

This project was partially funded by Indiana University's Office of the Vice President for Research and the Office of the Vice Provost for Research through its Faculty Research Support Program. We also acknowledge support from Indiana University's Institute for Advanced Study.

References

[1] S. Abramsky. Retracing some paths in process algebra. In U. Montanari and V. Sassone, editors, *CONCUR*, volume 1119 of *Lecture Notes in Computer Science*, pages 1–17. Springer, 1996.

[2] A. Filinski. Declarative continuations: an investigation of duality in programming language semantics. In *Category Theory and Computer Science*, pages 224–249, London, UK, 1989. Springer-Verlag.

[3] R. P. James and A. Sabry. Information effects. In *POPL*, 2012.

[4] A. Joyal, R. Street, and D. Verity. Traced monoidal categories. *Math. Proc. Camb. Philos. Soc.*, 119(3):447–468, 1996.

[5] I. Mackie. The geometry of interaction machine. In *POPL*, pages 198–208, 1995.

[6] I. Mackie. Reversible higher-order computations. In *Workshop on Reversible Computation*, 2011.