PE_2

Sarah R.F.

April 16, 2020

Get labeling style consistent between typing and reduction rules.

1 Introduction

PE_2 is a system introduced in Abr93 as a term language for proofs in one-sided second-order classical linear logic. This means that its terms and programs correspond to proofs in that logic, and their types correspond to the things being proved. Its reduction rules, in turn, correspond to the process of "cut elimination" in proofs.[1] In this paper, however, I will attempt to give an informal presentation and explanation of it as a programming language in its own right, without touching directly on its relationship to logic.

Cite properly

Talk about how we're gonna be bootstrapping understanding up with fake cases and looping around to the real thing.

2 The grammar

We will begin with the grammars of types, terms, and programs. This is merely to make clear the scope of our concern; I do not recommend trying to absorb all of it at first, nor is it particularly easy to intuit the meaning of many of the constructs from their syntax without prior knowledge of linear logic. Instead, return to this section as a reference when necessary. Having established our domains of discourse, we will work through the typing rules, reduction rules, and intuitive meanings of the various constructs over the course of the remaining sections.

Types are formed from the grammar in Figure 1. 1 and $\bot$ are called the *units*, $\otimes$ and $⅋$ are called the *multiplicatives*, & and $\oplus$ are called the *additives*, ! and ? are called the *exponentials*, and $\forall$ and $\exists$ are called the *quantifiers*. Terms

[1] This correspondence is technically not exact, insofar as the process of cut elimination removes all instances of the cut rule from a proof, while reduction of a PE_2 program will not simplify its construct corresponding to cut if it occurs in certain places.

does this need elaboration?

$A, B ::=$	types
α	type variable (α may be any type variable name)
$\alpha^\perp$	dual type variable (α may be any type variable name)
$\mathbf{1}$	one
$\bot$	bottom
$A \otimes B$	A tensor B (or A times B)
$A \parr B$	A par B (as in "parallel")
$A \mathbin{\&} B$	A with B
$A \oplus B$	A plus B
$!A$	of course A
$?A$	why not A
$\forall \alpha.A$	for all α, A
$\exists \alpha.A$	exists α, A

Figure 1: The grammar of types (with names/pronunciations).

$t, u, v ::=$	terms
x	(x may be any variable name)
$*$	
$\circledast$	
$t \otimes u$	
$t \parr u$	
$x_1 \ldots x_n(P \,[]\, Q)$	($n \geq 0$)
$\mathsf{inl}(t)$	
$\mathsf{inr}(t)$	
$x_1 \ldots x_n(P)$	($n \geq 0$)
$?t$	
$\overline{}$	
$t \mathbin{@} u$	

Figure 2: The grammar of terms.

$P, Q ::=$	programs
$t_1 \perp u_1, \ldots, t_n \perp u_n;\ v_1, \ldots, v_m$	($n \geq 0$, $m \geq 0$)

Figure 3: The grammar of programs.

and programs[2] are formed from the mutually recursive grammars in Figures 2 and 3. Note that the occurrences of the symbols $\otimes$ and $⅋$ in the type and term grammars are not a priori related; terms formed using these symbols will indeed have types formed using these symbols in the type system we establish, but there is no significance to their double usage at the level of the grammar. In what follows, context should make clear whether a given $\otimes$ or $⅋$ refers to the type constructor or term constructor.

3 Duality

A pervasive pair of notions in programming languages are those of *introduction forms* and *elimination forms*. In many languages, each of the basic forms of expression (with a few exceptions) is associated to some type and classified by whether it constructs—"'introduces"—a value of that type, or decomposes/uses—"eliminates"—one. For example, an integer literal is an introduction form for the type int, a field access is an elimination form for a compound type, an anonymous function is an introduction form for a function type, and a function application is an elimination form for one. Then, reduction can be seen as the process of cancelling matching introduction and elimination forms.

explain

These notions correspond to the logical notions of introduction and elimination *rules*, which are a feature of natural deduction systems. But PE_2 is based on sequent calculus, not natural deduction, and its terms have no elimination forms, only introduction forms. That is to say, there is no way to write a term which decomposes/uses one of its subterms in order to compute a final result; the only terms (apart from variables) that may be written are those that are already "literals", in a certain sense, and not subject to reduction.

There is, however, still a way to express computation! Each type A is assigned a *dual type* $A^\perp$, a value of which contains precisely the information that *would* have been present in an elimination form for A. Given a term t of A and u of $A^\perp$, they may be placed into a *coequation*, which looks like $t \perp u$, and it is *coequations* that are subject to reduction; they can simplify, split into multiple coequations, or disappear, making use of the structure of u to eliminate t.

not a term

Because coequations are based on the Cut rule of sequent calculus, we refer to placing t and u in a coequation as "cutting t against u".

The definition of the dualization operation on types is given by recursion in Figure 4. Note that in the first clause, $\alpha^\perp$ on the left-hand side means the application of the dualization operation to the type variable α, but on the right-hand side it means the basic "dual variable" type also written $\alpha^\perp$; the fact that the operation is defined this way for variables resolves the ambiguity that would otherwise exist between the two interpretations of "$\alpha^\perp$". Similarly, on the left-hand side of the second clause, $\alpha^{\perp\perp}$ means the application of the dualization operation to a "dual variable" type.

It is not too hard to see that this operation is in fact an involution—that is, $A^{\perp\perp} = A$ for all A. With a little thought, this shows that it is actually as valid

maybe also say something about type constructors coming in dual pairs?

[2] What I call "programs" here are called "proof expressions" in Abramsky.

$$
\begin{aligned}
\alpha^\perp &= \alpha^\perp \\
\alpha^{\perp\perp} &= \alpha \\
1^\perp &= \bot \\
\bot^\perp &= 1 \\
(A \otimes B)^\perp &= A^\perp \parr B^\perp \\
(A \parr B)^\perp &= A^\perp \otimes B^\perp \\
(A \mathbin{\&} B)^\perp &= A^\perp \oplus B^\perp \\
(A \oplus B)^\perp &= A^\perp \mathbin{\&} B^\perp \\
(!A)^\perp &= ?(A^\perp) \\
(?A)^\perp &= !(A^\perp) \\
(\forall \alpha.A)^\perp &= \exists \alpha.(A^\perp) \\
(\exists \alpha.A)^\perp &= \forall \alpha.(A^\perp)
\end{aligned}
$$

Figure 4: The dual of each type.

to say that $A^\perp$ is the type of things that As can eliminate as it is to say that $A^\perp$ is the type of things that can eliminate As! This means that if $t \perp u$ makes sense, then so does $u \perp t$—if u's type is dual to t's type, then t's type is dual to u's type. Indeed, one of PE_2's basic congruence rules is swapping the sides of a coequation! So in $t \perp u$, we can actually think either of u as the subject acting on the object t to eliminate it, or vice versa—whichever is most convenient or makes the most sense for the case at hand. Alternatively, we can imagine the reduction of a coequation as something akin to the annihilation of a particle and an antiparticle, although this is harder to understand by comparison to familiar languages. Because of this symmetry, we will not generally distinguish between "cutting t against u" and "cutting u against t" as ways of referring to the construction of $t \perp u$ or $u \perp t$.

is this too cute

These notions of "subject" and "object" will be a running theme. To fully understand a value of type A, we should ideally understand both what it means when viewed as the acted-open object of a coequation (in which case something of type $A^\perp$ will act upon it), *and* what it means when viewed as the acting subject of a coequation (in which case it will act upon something of type $A^\perp$). The former is more akin to how values are usually thought of, but the latter is equally (if not more!) critical in this framework.

this is at the intuitive level

We now look at our first concrete example of these phenomena in the form of the additives, & and $\oplus$, beginning from the object perspective for each. We defer the actual typing rules until later, because they involve concepts that we have not yet seen.

$A \& B$ is the type of *lazy pairs* of A and B. A value of this type contains a kind of thunk for producing an A and one for a B. This is what the $x_1 \dots x_n(P \,\square\, Q)$ term form is for. However, we have not yet discussed the language features necessary to understand the significance of the full syntax, so we will for now only consider the simplest case, when the thunks are already values, in which case this construct takes the form $(; t \,\square\, ; u)$, where t has type A and u has type B. Ultimately, the user of such a pair must request one of the two elements (at which point the thunk is forced) and discard the other; they cannot take both.

this feels like a vaguely dishonest description?

$A \oplus B$ is the *disjoint union* of A and B; its values look like $\mathsf{inl}(t)$ for t of type A and $\mathsf{inr}(u)$ for u of type B. The user of such a value must be prepared to eliminate either an A or a B.

We now turn to the subject perspective. Figure 4 states that $(A \,\&\, B)^\perp = A^\perp \oplus B^\perp$, so a value of $A \& B$ viewed as a subject should eliminate objects of type $A^\perp \oplus B^\perp$. This is realized as follows: personifying slightly, the subject examines the object to check whether it is an $\mathsf{inl}(t)$ or an $\mathsf{inr}(u)$. In the $\mathsf{inl}(t)$ case, t is an $A^\perp$, so the subject computes its own first element, an A, and discards its own second thunk. Then the computed A is tasked with continuing to eliminate t. The $\mathsf{inr}(u)$ case is largely similar, except that the second element is computed instead.

This can be summarized by saying that, viewed as a subject, a value of $A \,\&\, B$ is a case or match statement; intuitively speaking, $v \perp (; t \,\square\, ; u)$ means something like

$$\mathsf{case}\ v\ \mathsf{of}\ \{\mathsf{inl}(t') \to t' \perp t \mid \mathsf{inr}(u') \to u' \perp u\}.$$

The two elements of its object role correspond to the two branches of the statement, the laziness corresponds to the fact that branches of a statement are not evaluated until they are encountered, and the fact that only one of the two elements may be retrieved corresponds to the fact that only one of the two branches is taken.

Inverting the subject-object relationship, Figure 4 has $(A \oplus B)^\perp = A^\perp \,\&\, B^\perp$, and a value of $A \oplus B$ as a subject should eliminate objects of type $A^\perp \,\&\, B^\perp$. For $\mathsf{inl}(t)$ with t an A, this is realized by projecting out the first element of the object and then tasking t with eliminating it; for $\mathsf{inl}(u)$, similarly with the second element. Thus, a value of $A \oplus B$ as a subject is a projection: intuitively, $v \perp \mathsf{inl}(t)$ means $v.1 \perp t$ and $v \perp \mathsf{inr}(u)$ means $v.2 \perp u$.

Both of these perspectives are viewpoints on one symmetrical system. We will not yet look at the full reduction rules for these types, since we have not yet discussed the full syntax for & terms, but the special cases of those rules for the special cased syntax we have considered are

$$(; t \,\square\, ; u) \perp \mathsf{inl}(v) \longrightarrow t \perp v \qquad \text{(Case Left)}$$
$$(; t \,\square\, ; u) \perp \mathsf{inr}(v) \longrightarrow u \perp v. \qquad \text{(Case Right)}$$

4 Programs and variables

In PE_2, coequations and terms always appear within a program. As described in Figure 3, a *program* in PE_2 consists of a (possibly empty) sequence of coequations, called its *solution*[3], separated by a semicolon from a (possibly empty)[4] sequence of terms, called its *main body*. Rules like Case Left and Case Right that simplify a coequation or sequence of coequations into a coequation or sequence of coequations are called *reaction rules*, and PE_2 provides a *reaction context rule* that allows them to be applied within the solution of a program:

$$\frac{\Theta \longrightarrow \Xi}{\Theta_1,\, \Theta,\, \Theta_2;\ \overline{t} \ \longrightarrow\ \Theta_1,\, \Xi,\, \Theta_2;\ \overline{t}} \text{ (Reaction Context)}$$

Here Θ_1, Θ, Θ_2, and Ξ may be instantiated with any sequences of coequations (so the comma is actually overloaded to mean concatenation here, as is common), and $\overline{t}$ may be instantiated with any sequence of terms. We will continue to use similar names for such metavariables without comment.

PE_2 also has congruence rules that allow the sides of a coequation to be swapped and the coequations of a solution permuted before and after taking a step. First, there is a relation $\leftrightharpoons$ between sequences of coequations defined by

$$\frac{\begin{array}{c} t \perp u \leftrightharpoons u \perp t \\ t \perp u,\, t' \perp u' \leftrightharpoons t' \perp u',\, t \perp u \\ \Theta \leftrightharpoons \Xi \end{array}}{\Theta_1, \Theta, \Theta_2 \leftrightharpoons \Theta_1, \Xi, \Theta_2}$$

am I underestimating what counts as a context? also, align the rule correctly

Then, using the equivalence relation $\leftrightharpoons^*$ it generates, there is a *magical mixing rule*[5]:

call attn to this

$$\frac{\Theta \leftrightharpoons^* \Theta' \qquad \Theta';\ \overline{t} \longrightarrow \Xi';\ \overline{u} \qquad \Xi' \leftrightharpoons^* \Xi}{\Theta;\ \overline{t} \longrightarrow \Xi;\ \overline{u}} \text{ (Magical Mixing)}$$

In other words: if we can use $\leftrightharpoons$ to convert our program's solution into a form in which a simplification applies, and then convert the new solution into a final form, then we can simplify to that final form.

Is there some way I can say "finish this next bit before thinking about anything in it too carefully"?

Having laid this groundwork, we can now address an elephant in the room. Although I used phrases like "a term t of type A" in the last section, I avoided

[3] As in chemistry, not algebra—the operational semantics of PE_2 draw upon the notion of a "chemical abstract machine", with the idea that coequations are like molecules floating in a solution, and the congruence rules that rearrange coequations are like the Brownian motion that allows molecules in a solution to come into contact and react.

[4] Although (as a nontrivial theorem) no program with an empty main body is well-typed.

[5] This is the technical term.

ever writing $t : A$. This is because terms of PE_2 are not, in fact, assigned types in isolation, and while "a term t of type A" is therefore misleading, the notation $t : A$ is downright wrong! Instead, terms must be considered within a program before they can be given a type. A typing judgment looks like

$$\vdash \Theta;\ t_1 : A_1, \ldots, t_n : A_n.$$

We can see that types are assigned to terms of a program's main body *jointly*—a typing judgment is a simultaneous claim about all of the terms in the main body at once. Furthermore, this does *not* just mean that each individual $v_i : A_i$ is separately true, even on an intuitive level—each $v_i : A_i$ is a statement contingent on the fact that v_i is part of the overall program in the judgment. The simplest explanation for this is that the various terms of a program may be *linked together*, and so they will not function correctly as their claimed type if they are removed from their context. This arises from the behavior of *variables* in PE_2, whose basic typing rule is

$$\frac{}{\vdash ;\ x : A^{\perp}, x : A}\ \text{(Variable)}$$

Since $A^{\perp\perp} = A$, by substituting $A^{\perp}$ for A we also have

$$\frac{}{\vdash ;\ x : A, x : A^{\perp}}\ \text{(Variable)}$$

The rest of the typing rules of PE_2 continue to ensure that any given variable that appears in a program will always have exactly 2 occurrences. The idea is that these occurrences constitute the two endpoints of a kind of "connection". We will turn to the reduction rules next before coming back to the typing rule; do not worry too much about its meaning until then.

$$t \perp x,\ x \perp u \longrightarrow t \perp u \qquad \text{(Communication)}$$

$$x \perp t,\ \Theta;\ \overline{u} \longrightarrow \Theta;\ \overline{u}[t/x] \quad (\text{if } x \text{ occurs substitutable}^6 \text{ in } \overline{u}) \qquad \text{(Cleanup)}$$

Note that although Communication appears to be limited in that it cannot apply to non-adjacent coequations, the existence of the Mixing rule means that this limitation is only apparent, since any two coequations can be moved next to each other prior to simplification. Similarly, one can apply Cleanup to a coequation other than the first by permuting the solution so that the desired one is now at the beginning.

Let us examine the meaning of these reduction rules in the subject/object framework. In the case of Communication, we can view the simplification as either a fusion or as one of the coequations terminating and the other being updated. The former is more amenable to a symmetrical viewpoint; the latter is more amenable to the asymmetrical notion of a variable as a subject or object,

[6] It is possible for a variable to occur in a term without it being possible to substitute for it if it occurs as one of the variables in front of $x_1 \ldots x_n(P \,\square\, Q)$ or $x_1 \ldots x_n(P)$ rather than as a subterm. Such an occurrence is said to be *passive*; others are *active*.

okay maybe this is a little too cute and not quite accurate

clarify phrasing maybe? who's my audience?

and not entirely truthful. does this add much?

sth about being only intuition? value of mentioning symmetry?

and so it is the one we will consider—in particular, we will imagine that the activity of the rule takes place in the first coequation, and that the effects to the second are byproducts of this. Then if x is the subject of the coequation $t \perp x$, Communication expresses that one way it may accomplish its task of eliminating t is by transferring it to the location of the other occurrence of x, thereby delegating the task to u. This completes the purpose of the first coequation, at which point it evaporates, and we are left only with $t \perp u$. If x is instead seen as the *object* of $t \perp x$, then Communication expresses that x can provide a value by finding another coequation $x \perp u$ that the other occurrence of x is in, removing that coequation, and stealing its other side u, once again leaving only $t \perp u$. In any of these cases, we see a behavior where the two occurrences of a variable act together as a kind of proxy to connect t to u.

Cleanup is slightly less symmetric, and it is difficult to interpret the variable in its coequation as anything other than a subject. x's behavior as a subject here is basically similar to the account given for Communication: it moves t to the location of the other occurrence. Note that since there can be only one other occurrence in a well-typed program, and Cleanup is only applicable if that occurrence is in the main body, there is at no point a choice between applying Cleanup and Communication (since Communication requires there to be another occurrence in the solution). The substitution in Cleanup will also only cause an actual change in at most one term of the main body, since exactly one term can include the exactly one other occurrence of x.

Returning to the typing rule, we can now see that $\vdash ; x : A^\perp, x : A$ expresses any of the following equivalent facts:

- the first x can eliminate an A, at the cost (or perhaps benefit) of the second x needing to be treated as a potential A;
- the first x can be treated as a potential $A^\perp$, at the cost (or perhaps benefit) of the second x needing to eliminate an $A^\perp$;
- the second x can eliminate an $A^\perp$, at the cost (or perhaps benefit) of the first x needing to be treated as a potential $A^\perp$;
- the second x can be treated as a potential A, at the cost (or perhaps benefit) of the second x needing to eliminate an A.

However, none of this makes sense anymore if the individual occurrences are considered in isolation, rather than assigning types to both of them simultaneously—hence (one part of) the fundamental need for a simultaneous typing judgment.

Having seen the functionality carried by variables, we can now examine the form of programs more closely. Since terms have no reduction rules, the main body is "inert"; all of the computation takes place in the solution. However, the Cleanup rule allows this action in the solution to cross over and cause a term to be substituted into the main body, if the two occurrences of a variable are on opposite sides of the semicolon. With a few exceptions, the coequations of a well-typed program can generally continue to simplify up until they disappear, so many programs will finish only once the solution is empty. Thus, the main

body is the output of the program, and the coequations produce useful results from a distance by replacing holes in the output (occurrences of variables) with values, via the Cleanup rule.

In general, a judgment like $\vdash \Theta;\ t : A,\ u : B,\ v : C$ indicates that:

1. each of t, u, and v is an "incomplete" value, of type A, B, and C respectively—incomplete in the sense that some of their subterms may be variables instead of standalone values;
2. some of those incompletenesses can be resolved by allowing the coequations in Θ to execute and deposit results into the main body;
3. any remaining incompletenesses "complement each other", such that t, u, and v can all be used in coequations as their claimed types in spite of the missing portions, by "cooperating with each other" to accomplish their tasks.

only in that sense?

We have now seen enough of the language to appreciate the real typing rules for coequations, $\mathsf{inl}()$, $\mathsf{inr}()$, and the "exchange rules" which allow terms and coequations in a program to be permuted in order to assign types, although we will continue to wait on a full examination of & until after some more ground has been covered. In what follows, Γ and Δ are used as metavariables that may be instantiated with any sequences of term-type pairs like $t_1 : A_1, \ldots, t_n : A_n$.

talk @ some point about the purpose of the exchange rules?

$$\frac{\vdash \Theta;\ \Gamma, t : A \qquad \vdash \Xi;\ \Delta, u : A^{\perp}}{\vdash \Theta, \Xi, t \perp u;\ \Gamma, \Delta}\ \text{(Coequation) (†)}$$

$$\frac{\vdash \Theta;\ \Gamma, t : A}{\vdash \Theta;\ \Gamma, \mathsf{inl}(t) : A \oplus B}\ \text{(Plus (i))} \qquad \frac{\vdash \Theta;\ \Gamma, t : B}{\vdash \Theta;\ \Gamma, \mathsf{inr}(t) : A \oplus B}\ \text{(Plus (ii))}$$

$$\frac{\vdash \Theta, t \perp u, t' \perp u', \Xi;\ \Gamma}{\vdash \Theta, t' \perp u', t \perp u, \Xi;\ \Gamma}\ \text{(Xchg (i))} \qquad \frac{\vdash \Theta;\ \Gamma, t : A, u : B, \Delta}{\vdash \Theta;\ \Gamma, u : B, t : A, \Delta}\ \text{(Xchg (ii))}$$

The (†) next to the coequation rule indicates that an important side condition must hold when it is instantiated in order for it to be considered a valid usage: *no one variable may appear in both of the programs in the premises.* Placing this side condition on each rule with multiple premises ensures the aforementioned property that a variable can only appear exactly twice in a well-typed program if it appears at all, because variables are introduced in pairs and rules that combine programs together preserve the property.

One might expect the rule for coequations to instead look something like

$$\frac{\vdash \Theta;\ \Gamma, t : A, u : A^{\perp}}{\vdash \Theta, t \perp u;\ \Gamma}\ \text{(Not a real rule)}$$

However, this would allow undesirable programs to be typed! The simplest example is

$$\dfrac{\dfrac{}{\vdash ;\ x : A^{\perp}, x : A}\ \text{(Variable)}}{\vdash x \perp x;}\ \text{(Fake Coequation)}$$

This program cannot simplify. At a deeper level, the problem is that, as described above, the terms in the main body of a program are incomplete and only able to be used in a coequation as their listed type insofar as they can rely on the other terms of the main body as "collaborators". If one of the terms is cut against a term from another program, this is well and good; but if a coequation is built out of two terms that already rely on each other, this can create a kind of "dependency loop" where each needs to be able to offload the task on the other. Since we regard variables as a means of proxying between two terms in our intuitive framework, we can see this effect fairly clearly in the program above; each x in $x \perp x$ is attempting to call on the other occurrence of x, and in doing so, is attempting to delegate to *itself*. We avoid this kind of trap by insisting that the terms of a coequation be typed independently of each other, with disjoint sets of dependencies.

5 The multiplicatives

This whole section is pretty drafty & needs a lot of feedback & work.

With this understanding of potential "communication" and "dependency" between terms, we can now turn to our next dual pair of type constructors: the multiplicatives $\otimes$ and $\parr$. From the object viewpoint, $A \otimes B$ is the type of *strict* pairs of A and B; a value of this type contains an A value and a B value, and the user of the $A \otimes B$ must accept both, in contrast with &. $A \parr B$, in turn, is the type of *incomplete* pairs of A and B; a value of this type contains an A value and a B value, but which are permitted to together have the kind of "complementary incompleteness" discussed above. Before giving any further explanations, however, I will set down the actual typing and reduction rules so that I can give concrete examples:

I never did talk about constructors coming in pairs...

$$\dfrac{\vdash \Theta;\ \Gamma, t : A \qquad \vdash \Xi;\ \Delta, u : B}{\vdash \Theta, \Xi;\ \Gamma, \Delta, t \otimes u : A \otimes B}\ \text{(Tensor)}\ (\dagger)$$

$$\dfrac{\vdash \Theta;\ \Gamma, t : A, u : B}{\vdash \Theta;\ \Gamma, t \parr u : A \parr B}\ \text{(Par)}$$

$$t \otimes u \perp t' \parr u' \longrightarrow t \perp t',\ u \perp u' \qquad \text{(Pairing)}$$

Note that the same side condition about no shared variables as in the coequation rule applies to the $\otimes$ typing rule.

For both $A \otimes B$ and $A \parr B$, a term contains a subterm of type A and a subterm of type B. However, the typing rule for $\otimes$ requires that the two terms be defined in separate programs before being brought together, whereas the rule for $\parr$ requires that they come from the same program. Thus, the components of a $\otimes$ term are "independent" of one another, can be individually regarded as values in their own right[7], and can, for example, be cut against each other. By contrast, the components of a $\parr$ term cannot in general be separated from each other, and broadly share characteristics with the terms of a program's main body. It is best, therefore, not to think of $\parr$ as a type of pairs, even if its terms have two components. For example, the following typing derivation works for any choice of A:

$$\dfrac{\dfrac{}{\vdash\ ;\ x : A^\perp,\ x : A}\ \text{(Variable)}}{\vdash\ ;\ x \parr x : A^\perp \parr A}\ \text{(Par)}$$

By contrast, inverting the typing rule for $\otimes$ shows that $\vdash\ ;\ t \otimes u : A^\perp \otimes A$ would require *independently* $\vdash\ ;\ t : A^\perp$ and $\vdash\ ;\ u : A$, which is far more in line with how types of pairs are usually understood.[8]

umm, maybe this isn't self-evident?

We have $(A \otimes B)^\perp = A^\perp \parr B^\perp$ and $(A \parr B)^\perp = A^\perp \otimes B^\perp$. The subject viewpoint on $\otimes$ and $\parr$, and the duality between them, is closely related to the earlier discussion about the form of the typing rule for coequations and the hazard of "dependency loops". As a subject, an $A \otimes B$ eliminates an $A^\perp \parr B^\perp$ by tasking each of its components with independently eliminating the corresponding half of the $\parr$. This will work in spite of the fact that the components of the $\parr$ are incomplete, because of the previously-described notion that their incompletenesses are "complementary" in a way that allows cooperation to compensate for them, once the terms are put to actual use. In order for this to succeed, the characteristic independence between the two components of the $\otimes$ is critical: since the halves of the $\parr$ may perform "delegation" through one another, and each is being eliminated by half of the $\otimes$, the two halves of the $\otimes$ are liable to being delegated against each other, and so they must be independent. As an extreme example, consider again the term $x \parr x$ and what happens when something of the form $t \otimes u$ tries to eliminate it:

$$t \otimes u \perp x \parr x \longrightarrow t \perp x,\ u \perp x \leftrightharpoons t \perp x,\ x \perp u \longrightarrow t \perp u.$$

This will end poorly if t and u rely on each other!

An $A \parr B$ as a subject likewise eliminates an $A^\perp \otimes B^\perp$ by matching component to component, but in this viewpoint, it is the two subjects which are entitled

[7] Of course, they may still be interdependent with other terms of the main bodies of their respective originating programs if there are any, but at a minimum they do not share any variables with *each other*.

uhh, that's true, right? also: can I make some kind of stronger statement about "safety to treat something as a value from a given viewpoint"

[8] Actually, it is impossible to write *any* program $\Theta;\ t$ with $\vdash \Theta;\ t : A^\perp \otimes A$, even for particular A—that would correspond to being able to prove a contradiction in linear logic!

to "cooperate". Once again, this is safe because of the independence of the components of the $\otimes$.

Another, arguably more useful viewpoint on the multiplicatives comes from the notion of the *linear function type* $A \multimap B$, which is defined to be shorthand for $A^\perp \parr B$. The key shift in perspective is as follows: when thinking of $t \parr u$ as having type $A \multimap B$ rather than type $A^\perp \parr B$, we generally view it as an object—a *function* (from A to B)—but rather than thinking of its components t and u as both objects, we see t as a subject and u as an object. Conversely, when thinking of $t' \otimes u'$ as having type $(A \multimap B)^\perp$ rather than type $A \otimes B^\perp$, we generally view it as a subject—a *function call* awaiting a function to be attached to—but rather than thinking of its components t' and u' as both subjects, we see t' as an object and u' as a subject. We conceptualize each piece as follows:

- t, of type $A^\perp$, is a subject whose role is to eliminate the argument supplied to the function $t \parr u$.
- u, of type B, is an object whose role is to be the result of the function $t \parr u$. The fact that we have $t \parr u$ rather than $t \otimes u$ means that t and u can be linked, so as t eliminates an argument, it can update u, much like how the main body of a program can be updated by coequations using the Cleanup rule. This allows the result of the function to depend on the argument.
- t', of type A, is an object whose role is to be the argument to the function being called.
- u', of type $B^\perp$, is a subject whose role is to eliminate the result of the function once it returns. It is important that we have $t' \otimes u'$ rather than $t' \parr u'$, because otherwise the argument to the function could depend in some way on the result of the call, which could create dependency cycles.

there are some important differences—how should I talk about them?

"once it returns" really doesn't describe the shape of PE_2's execution...

maybe I should introduce this concept earlier?!!

If we analogize values of type $A^\perp$ to *patterns* that can match and destructure values of type A[9], then in this reading $t \parr u$ becomes analogous to $\lambda t.u$. In turn, the action in $t' \otimes u' \perp v$ is analogous to $u' \perp v(t')$. If we apply both of these analogies, then the reduction Pairing rule looks like

$$u' \perp (\lambda t.u)(t') \longrightarrow t' \perp t,\ u' \perp u,$$

which is quite close in flavor to β-reduction. The resulting $t' \perp t$ is in some sense the processing due to the function consuming its argument, and the $u' \perp u$ is in some sense the processing due to the caller consuming the function result. The shared variables between t and u allow these two pieces of processing to interact.

Under this interpretation, the recurring example of $x \parr x$ in $A^\perp \parr A$ reveals itself as the *identity function* in $A \multimap A$. The previously-mentioned simplification

$$t \otimes u \perp x \parr x \longrightarrow t \perp x,\ u \perp x \leftrightharpoons t \perp x,\ x \perp u \longrightarrow t \perp u$$

[9] Although this analogy breaks down slightly when one tries to interpret, e.g., & as a "pattern" for $\oplus$.

in fact demonstrates why: if t is the argument and u is the consumer of the result, then the call reduces to u acting directly on t—i.e., $x \parr x$ has simply returned its argument!

Some examples to chew on are given in Figure 5. The first four can be loosely analogized to terms in a lambda calculus with pattern matching by the scheme above. In order to interpret $t' \otimes u'$ in its function-call role as a pattern to match arguments of function type against, we will need our lambda calculus to support an unorthodox feature somewhat akin to Haskell's *view patterns*: if e is an expression and p is a pattern, we have a pattern $e \mapsto p$, and a function f can be matched against $e \mapsto p$ by taking $f(e)$ and matching it against p. That is, $e \mapsto p$ is a pattern for functions that map e to something matching p. Then we have the following rough analogies:

Need some simplification examples too.

$$\vdash ;\ x \parr \mathsf{inl}(x) : A \multimap A \oplus B \quad \sim$$
$$\vdash \lambda x.\mathsf{inl}(x) : A \to A + B$$

$$\vdash ;\ \mathsf{inl}(x) \parr x : A \mathbin{\&} B \multimap A \quad \sim$$
$$\vdash \lambda(x, \_).x : A \times B \to A$$

$$\vdash ;\ x \parr y \parr (x \otimes y) : A \multimap B \multimap A \otimes B \quad \sim$$
$$\vdash \lambda x.\lambda y.(x, y) : A \to B \to A \times B$$

$$\vdash ;\ x \parr (\mathsf{inr}(x) \otimes y) \parr y : B \multimap (A \oplus B \multimap C) \multimap C \quad \sim$$
$$\vdash \lambda x.\lambda(\mathsf{inr}(x) \mapsto y).y : A \to (A + B \to C) \to C$$

The fifth example, however, is a trap! It is meant to illustrate a danger in taking the analogies above *too* literally. Although it is only a slight tweak of the fourth example, if we try to apply a similar "translation" to it, we get the following nonsense:

$$\vdash ;\ (\mathsf{inr}(x) \otimes y) \parr x \parr y : (A \oplus B \multimap C) \multimap B \multimap C \quad \sim$$
$$\vdash \lambda(\mathsf{inr}(x) \mapsto y).\lambda x.y : (A + B \to C) \to A \to C$$

x gets used out of scope! This danger arises because most things in PE_2 are fundamentally symmetric, while most of the analogies I have presented rely on selectively choosing an asymmetric perspective, and in particular, the notion of a function has a clear sense of directedness. In a lambda calculus, variables are *bound* in a pattern and *used* in an expression, so it is clear that there is a violation above, since the occurrence as a use appears before the occurrence as a binding. But PE_2 has no such distinction! Neither occurrence of x in $(\mathsf{inr}(x) \otimes y) \parr x \parr y$ is technically "binding", and in fact either can cause the other to be replaced with something else.

Probably make a note of associativity; could be a source of confusion!

$$\dfrac{\dfrac{\dfrac{}{\vdash ;\ x : A^{\perp}, x : A}\ (\text{Variable})}{\vdash x : A^{\perp}, ;\ \mathsf{inl}(x) : A \oplus B}\ (\text{Plus (i)})}{\vdash ;\ x \parr \mathsf{inl}(x) : A \multimap A \oplus B}\ (\text{Par})$$

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\vdash ;\ x : A, x : A^{\perp}}\ (\text{Variable})}{\vdash ;\ x : A, \mathsf{inl}(x) : A^{\perp} \oplus B^{\perp}}\ (\text{Plus (i)})}{\vdash ;\ \mathsf{inl}(x) : A^{\perp} \oplus B^{\perp}, x : A}\ (\text{Xchg (ii)})}{\vdash ;\ \mathsf{inl}(x) \parr x : A \,\&\, B \multimap A}\ (\text{Par})$$

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\vdash ;\ x : A^{\perp}, x : A}\ (\text{Variable}) \quad \dfrac{}{\vdash ;\ y : B^{\perp}, y : B}\ (\text{Variable})}{\vdash ;\ x : A^{\perp}, y : B^{\perp}, x \otimes y : A \otimes B}\ (\text{Tensor})}{\vdash ;\ x : A^{\perp}, y \parr (x \otimes y) : B \multimap A \otimes B}\ (\text{Par})}{\vdash ;\ x \parr y \parr (x \otimes y) : A \multimap B \multimap A \otimes B}\ (\text{Par})$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\vdash ;\ x : B^{\perp}, x : B}\ (\text{Variable})}{\vdash ;\ x : B^{\perp}, \mathsf{inr}(x) : A \oplus B}\ (\text{Plus (ii)}) \quad \dfrac{}{\vdash ;\ y : C, y : C^{\perp}}\ (\text{Variable})}{\vdash ;\ x : B^{\perp}, y : C, \mathsf{inr}(x) \otimes y : (A \oplus B) \otimes C^{\perp}}\ (\text{Tensor})}{\vdash ;\ x : B^{\perp}, \mathsf{inr}(x) \otimes y : (A \oplus B) \otimes C^{\perp}, y : C}\ (\text{Xchg (ii)})}{\vdash ;\ x : B^{\perp}, (\mathsf{inr}(x) \otimes y) \parr y : (A \oplus B \multimap C) \multimap C}\ (\text{Par})}{\vdash ;\ x \parr (\mathsf{inr}(x) \otimes y) \parr y : B \multimap (A \oplus B \multimap C) \multimap C}\ (\text{Par})$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\vdash ;\ x : B^{\perp}, x : B}\ (\text{Variable})}{\vdash ;\ x : B^{\perp}, \mathsf{inr}(x) : A \oplus B}\ (\text{Plus (ii)}) \quad \dfrac{}{\vdash ;\ y : C, y : C^{\perp}}\ (\text{Variable})}{\vdash ;\ x : B^{\perp}, y : C, \mathsf{inr}(x) \otimes y : (A \oplus B) \otimes C^{\perp}}\ (\text{Tensor})}{\vdash ;\ x : B^{\perp}, \mathsf{inr}(x) \otimes y : (A \oplus B) \otimes C^{\perp}, y : C}\ (\text{Xchg (ii)})}{\vdash ;\ \mathsf{inr}(x) \otimes y : (A \oplus B) \otimes C^{\perp}, x : B^{\perp}, y : C}\ (\text{Xchg (ii)})}{\vdash ;\ \mathsf{inr}(x) \otimes y : (A \oplus B) \otimes C^{\perp}, x \parr y : B \multimap C}\ (\text{Par})}{\vdash ;\ (\mathsf{inr}(x) \otimes y) \parr x \parr y : (A \oplus B \multimap C) \multimap B \multimap C}\ (\text{Par})$$

Figure 5: Sample terms with typing derivations that make use of the multiplicative types.

6 Full &

Even draftier.

Should probably talk somewhere in here about how & breaks cut-free-ness of canonical forms, and the *active* occurrence condition on the Cleanup rule.

We are finally in a reasonable position to discuss the full functionality of &.[10] We first lay out the issue that needs to be addressed. In a pair like $\otimes$, the two components must come from separate programs and can share no dependencies, because they must both be usable independently and simultaneously. In a &, there need not be any such restriction; since only one of the two components will ever be used, both can theoretically be linked to the same dependencies without problem. Thus, we might imagine a syntax like $t \,\square\, u$ where t and u are terms, with a typing rule like

$$\frac{\vdash \Theta;\ \Gamma,\ t : A \qquad \vdash \Theta;\ \Gamma,\ u : B}{\vdash \Theta;\ \Gamma,\ t \,\square\, u : A \,\&\, B}\ \text{(Not a real rule) (†)}$$

However, for technical reasons, such as the desirable property that a variable appears in a program exactly twice, it is not as simple as just requiring both halves to be well-typed within the program that the full &-term appears in. Instead, we pass through some indirection. Recalling Figure 2, the full syntax for &-terms is actually $x_1 \ldots x_n(P \,\square\, Q)$, where P and Q are *programs*. Having previously described the components of such a term as thunks, we can see that the solutions of P and Q are the deferred computations in question. However, this leaves the question of why there is a full multi-term main body in each half, and what the variables at the beginning are for! In fact, this is precisely the means by which the indirection is accomplished. The typing constrains the main bodies of P and Q to have $n+1$ terms, where n is the number of variables at the beginning. The *final* term in the main body of each half is the actual *result* of the thunk. The first n terms are a kind of "linking interface"; since they are in the main body of a program with the final term, the final term may be linked to them. Then, *they* can be linked to in the rest of the program outside of the &-term by referring to the variables at the beginning of the term. Intuitively, each x_i binds a means of referring to "the future ith item of the selected component's main body", deferred until the &-term is actually used and a component is selected, and thus indirectly a means of interfacing with the final term of the body.

Here are the full rules. In the typing rule, $\overline{x}$ is to be instantiated with a list of variables distinct from any occurring in either of the premises; $\overline{C}$ is to be instantiated with a list of types; $\overline{t}$, $\overline{u}$, $\overline{x}$, and $\overline{C}$ must be the same length; and $\overline{x} : \overline{C}$ is to be expanded as $x_1 : C_1, \ldots, x_n : C_n$.

$$\frac{\vdash \Theta;\ \overline{t} : \overline{C},\ t : A \qquad \vdash \Xi;\ \overline{u} : \overline{C},\ u : B}{\vdash ;\ \overline{x} : \overline{C}, \overline{x}(\Theta;\ \overline{t},\ t \,\square\, \Xi;\ \overline{u},\ u) : A \,\&\, B}\ \text{(With) (†)}$$

[10] Strictly, this section could go directly after the one on variables and programs, but it is helpful to gain some perspective from the multiplicatives first.

This typing rule is a bit unusual in that it prescribes the form of the entire program in the conclusion, rather than just one or two of its terms or coequations. This means that a typing derivation for a program containing a &-term must "isolate" it before using the (With) rule. We will go over some examples shortly.

In the reduction rules, $\overline{x} \perp \overline{t}$ is to be expanded as $x_1 \perp t_1, \ldots, x_n \perp t_n$.

$$\overline{x}(\Theta;\ \overline{t}, t \parr \Xi;\ \overline{u}, u) \perp \mathsf{inl}(v) \longrightarrow \Theta, \overline{x} \perp \overline{t}, t \perp v \qquad \text{(Case Left)}$$

$$\overline{x}(\Theta;\ \overline{t}, t \parr \Xi;\ \overline{u}, u) \perp \mathsf{inr}(v) \longrightarrow \Xi, \overline{x} \perp \overline{u}, u \perp v \qquad \text{(Case Right)}$$

In one of these reductions, 3 things happen.

1. The coequations from the selected "thunk" are "released into the solution", making them now subject to computation.
2. The final term of the selected "thunk" is cut against the term from the $\oplus$. This is just the content of our previous simplified understanding.
3. Some coequations are generated from the $\overline{x}$ and $\overline{t}$. To understand this, consider the context of the larger program the coequation being simplified exists within. The $\overline{x}$ variables are used elsewhere as "advance references" to whichever of the $\overline{t}$ or $\overline{u}$ is picked. Since this reduction is precisely the point at which one is picked, the "advance references" must now be resolved. This is accomplished by the addition of these coequations, which we can see as binding each x_i to each t_i or u_i.

this is a direct Abramsky quote! (albeit in reference to a different thing)

Let's take a look at a typing derivation.

$$\dfrac{\dfrac{\dfrac{}{\vdash ;\ r : B, r : B^\perp}\ \text{(Variable)}}{\vdash ;\ r : B, \mathsf{inl}(r) : A^\perp \oplus B^\perp}\ \text{(Plus (i))}}{\vdash ;\ \mathsf{inr}(r) : A^\perp \oplus B^\perp, r : B}\ \text{(Xchg (ii))} \qquad \dfrac{\dfrac{\dfrac{}{\vdash ;\ l : A, l : A^\perp}\ \text{(Variable)}}{\vdash ;\ l : A, \mathsf{inl}(l) : A^\perp \oplus B^\perp}\ \text{(Plus (i))}}{\vdash ;\ \mathsf{inl}(l) : A^\perp \oplus B^\perp, l : A}\ \text{(Xchg (ii))}$$

$$\dfrac{\vdash ;\ p : A^\perp \oplus B^\perp, p(;\ \mathsf{inr}(r), r \parr ;\ \mathsf{inl}(l), l) : B \,\&\, A}{\vdash ;\ p \parr p(;\ \mathsf{inr}(r), r \parr ;\ \mathsf{inl}(l), l) : A \,\&\, B \multimap B \,\&\, A}\ \text{(Par)} \qquad \text{(With)}$$

...

7 The units

How do I explain these ;_;

$$\dfrac{}{\vdash ;\ * : \mathbf{1}}\ \text{(One)} \qquad \dfrac{\vdash \Theta;\ \Gamma}{\vdash \Theta;\ \Gamma, \circledast : \perp}\ \text{(Bottom)}$$

$$* \perp \circledast \longrightarrow \qquad \text{(Unit)}$$

8 Interlude: Fixed-width integers

Gotta reorganize so that I've introduced the units by the time we're here.

We now have enough types and operations to carry out a medium-size example: we will define addition on fixed-width binary integers. We begin by defining a type of booleans and the "true" and "false" terms:

$$\mathbf{2} \triangleq \mathbf{1} \oplus \mathbf{1}$$
$$\mathbf{tt} \triangleq \mathsf{inl}(*)$$
$$\mathbf{ff} \triangleq \mathsf{inr}(*)$$

We then have $\vdash ;\ \mathbf{tt} : \mathbf{2}$ and $\vdash ;\ \mathbf{ff} : \mathbf{2}$. The simplest nontrivial function on booleans is negation, and it is defined as follows:

$$\mathrm{not} \triangleq o(;\ \mathbf{ff}, \circledast \,[]\ ;\ \mathbf{tt}, \circledast) \parr o$$

$$\dfrac{\dfrac{\dfrac{}{\vdash ; \mathbf{ff} : \mathbf{2}}\ (\text{known})}{\vdash ;\ \mathbf{ff} : \mathbf{2}, \circledast : \bot}\ (\text{Bottom}) \qquad \dfrac{\dfrac{}{\vdash ; \mathbf{tt} : \mathbf{2}}\ (\text{known})}{\vdash ;\ \mathbf{tt} : \mathbf{2}, \circledast : \bot}\ (\text{Bottom})}{\dfrac{\dfrac{\vdash ;\ o : \mathbf{2}, o(;\ \mathbf{ff}, \circledast \,[]\ ;\ \mathbf{tt}, \circledast) : \bot \,\&\, \bot}{\vdash ;\ o(;\ \mathbf{ff}, \circledast \,[]\ ;\ \mathbf{tt}, \circledast) : \bot \,\&\, \bot, o : \mathbf{2}}\ (\text{Xchg (ii)})}{\vdash ;\ o(;\ \mathbf{ff}, \circledast \,[]\ ;\ \mathbf{tt}, \circledast) \parr o : \mathbf{2} \multimap \mathbf{2}}\ (\text{Par})}\ (\text{With})$$

Unpacking this:

$$\overbrace{o(;\ \underbrace{\mathbf{ff}}_{\text{becomes } o}\ ,\ \underbrace{\circledast}_{\text{eliminates the } \mathbf{1} \text{ in arg}}\ \underbrace{[]}_{\text{"arg is } \mathbf{tt}\text{" branch}}\ ;\ \underbrace{\mathbf{tt}, \circledast}_{\text{"arg is } \mathbf{ff}\text{" branch}})}^{\text{switch on the argument}} \parr \overbrace{o}^{\text{return forward reference}}$$

Rigorously speaking, we now cannot use not more than once in a program, or in a program that uses the variable name o already for something else; but in practice, we shall use a convention that whenever we define a name for a term or program, all future uses of this name in building larger terms or programs will carry an implicit operation of variable freshening to ensure no conflicts and to allow multiple uses.

motivate better

Here is what happens if "not" is applied to, say, **tt**, using the variable r to capture the output:

I should probably explain that concept somewhere, huh

$$\dfrac{\dfrac{}{\vdash ;\ \mathbf{tt} : \mathbf{2}}\ \text{(known)} \qquad \dfrac{}{\vdash ;\ r : \mathbf{2}, r : \mathbf{2}^\perp}\ \text{(Variable)}}{\vdash ;\ r : \mathbf{2}, \mathbf{tt} \otimes r : \mathbf{2} \otimes \mathbf{2}^\perp}\ \text{(Tensor)} \qquad \dfrac{}{\vdash ;\ \text{not} : \mathbf{2} \multimap \mathbf{2}}\ \text{(known)}$$
$$\vdash \mathbf{tt} \otimes r \perp \text{not};\ r : \mathbf{2}\ \text{(Coequation)}$$

$$\mathbf{tt} \otimes r \perp \text{not};\ r \xrightarrow{\text{(Pairing)}} \mathbf{tt} \perp o(;\ \mathbf{ff}, \circledast \square ;\ \mathbf{tt}, \circledast), r \perp o;\ r \leftrightharpoons^*$$
$$o(;\ \mathbf{ff}, \circledast \square ;\ \mathbf{tt}, \circledast) \perp \mathbf{tt}, r \perp o;\ r \xrightarrow{\text{(Case Left)}} o \perp \mathbf{ff}, \circledast \perp *, r \perp o;\ r \leftrightharpoons^*$$
$$* \perp \circledast, r \perp o, o \perp \mathbf{ff};\ r \xrightarrow{\text{(Unit)}} r \perp o, o \perp \mathbf{ff};\ r \xrightarrow{\text{(Communication)}} r \perp \mathbf{ff};\ r \xrightarrow{\text{(Cleanup)}} ;\ \mathbf{ff}$$

Next, we define the type $\mathbb{FW}_k$ for $k \geq 0$ as the k-fold $\otimes$-power of $\mathbf{2}$:

this is too wide. also: note on importance of working thru this

$$\mathbb{FW}_0 \triangleq \mathbf{1}$$
$$\mathbb{FW}_{k+1} \triangleq \mathbb{FW}_k \otimes \mathbf{2}$$

Thus, $\mathbb{FW}_k$ is the type of k-tuples of booleans. If we further identify $\mathbf{tt}$ with 1 and $\mathbf{ff}$ with 0, we can see these as bitstrings of length k, and hence use them to represent elements of $\mathbb{Z}/2^k\mathbb{Z}$. For example, we can represent 10 as a term in $\mathbb{FW}_6 = (((((\mathbf{1} \otimes \mathbf{2}) \otimes \mathbf{2}) \otimes \mathbf{2}) \otimes \mathbf{2}) \otimes \mathbf{2}) \otimes \mathbf{2}$ by:

$$\vdash ;\ (((((* \otimes \mathbf{ff}) \otimes \mathbf{ff}) \otimes \mathbf{tt}) \otimes \mathbf{ff}) \otimes \mathbf{tt}) \otimes \mathbf{ff} : \mathbb{FW}_6$$

We will abbreviate this to just $\vdash ;\ 001010 : \mathbb{FW}_6$ for convenience.

wait, we don't really use this

By recursion on k, we define:

$$\text{zero}_0 \triangleq *$$
$$\text{zero}_{k+1} \triangleq \text{zero}_k \otimes \mathbf{ff}$$

So for example, zero_6 is 000000, in our shorthand. For each k, we have:

$$\vdash ;\ \text{zero}_k : \mathbb{FW}_k$$

We then define the successor operation, suc_k, with the intent that $\vdash ;\ \text{suc}_k : \mathbb{FW}_k \multimap \mathbb{FW}_k$. In the case of $\mathbb{FW}_0$, this is just the identity, since there is only one zero-width integer.

$$\text{suc}_0 \triangleq x \parr x$$

$$\text{suc}_{k+1} \triangleq (\overbrace{h}^{\text{high bits}} \parr h o (\overbrace{\underbrace{\text{suc}_k \perp h_1 \otimes h_1'}_{\text{“}h_1' = \text{suc}_k(h_1)\text{”}};\ h_1}^{\text{“}h \text{ in branch 1”}}, h_1' \otimes \mathbf{ff}, \circledast \square ;\ \overbrace{h_2}^{\text{“}h \text{ in branch 2”}}, h_2 \otimes \mathbf{tt}, \circledast)) \parr o$$

Finally, we can define full addition. Writing out a direct term for the addition function add_{k+1} would involve two occurrences of add_k as subterms, one on each

side of a &-term, which would cause a combinatorial explosion of the size of the term in practice. To avoid this, we will place *one* occurrence of it outside of the &-term and then make reference to it from *inside* by means of the "forward reference" variables. However, in order to accomplish this, we will need to define some *programs* rather than terms. For each $k \geq 0$, we define a solution add_k^Θ and a term add_k^t, such that $\vdash \mathrm{add}_k^\Theta;\ \mathrm{add}_k^t : \mathbb{FW}_k \multimap \mathbb{FW}_k \multimap \mathbb{FW}_k$.

$$\mathrm{add}_0^\Theta;\ \mathrm{add}_0^t \triangleq ;\ \circledast \parr x \parr x$$

$$\mathrm{add}_{k+1}^\Theta;\ \mathrm{add}_{k+1}^t \triangleq \mathrm{add}_k^\Theta,\ \overbrace{\mathrm{add}_k^t \perp a_h \otimes r}^{\text{"}r(x) = \mathrm{add}_k^t(a_h, x)\text{"}};\ (\underbrace{a_h}_{\text{high bits of arg } a} \parr \overbrace{bro(P \,\square\, Q)}^{\text{switch on low bit of arg } a}) \parr b \parr o$$

$$\text{where } P \triangleq \mathrm{suc}_k \perp b_1 \otimes (\overbrace{b'_{h1} \parr b'_{l1}}^{\text{high and low bit[s] of } b+1});\ b_1,\ \underbrace{b'_{h1} \otimes o_{h1}}_{\text{becomes "call to" } r},\ o_{h1} \otimes b'_{l1},\ \circledast$$

$$\text{where } Q \triangleq ;\ b_{h2} \parr b_{l2},\ \underbrace{b_{h2} \otimes o_{h2}}_{\text{becomes "call to" } r},\ o_{h2} \otimes b_{l2},\ \circledast$$

am I sure that this is well-typed?

9 Type variables and the quantifiers

Hmm, doing an awful lot of casual assumption about familiarity on the part of the reader here...

PE_2 has support for *parametric polymorphism*; its level of power is roughly on par with System F. This is accomplished using the type variables, dual type variables, and quantifiers. Type variables play a familiar role; the purpose of dual type variables is to allow reference to the dual of a quantified-over type. Their existence introduces one minor subtlety worth noting: capture-avoiding substitution of a type for a type variable is defined in the case of dual variables by

$$\alpha^\perp[A/\alpha] = A^\perp.$$

The remainder of the definition is utterly conventional.

A term of type $\forall\alpha.A$ is a term which is usable as type $A[B/\alpha]$ for all choices of B. A term of type $\exists\alpha.A$ is a term which is usable as type $A[B/\alpha]$ for *some* choice of B. So a term with a $\forall$ type is one which is parametrically polymorphic, while a term with an $\exists$ type has no special properties, but could be any of a wider range of things than something with a more concrete type. There are no special term forms for these types; it is simply possible to generalize or abstract the type of any term.[11] The rules for doing so are:

reference some kind of tutorial on System F for readers who aren't familiar, maybe? am I past the point of trying to target people who might not know what System F is?

$$\frac{\vdash \Theta;\ \Gamma,\ t : A}{\vdash \Theta;\ \Gamma,\ t : \forall\alpha.A}\ (\mathrm{All})\ (*) \qquad \frac{\vdash \Theta;\ \Gamma,\ t : A[B/\alpha]}{\vdash \Theta;\ \Gamma,\ t : \exists\alpha.A}\ (\mathrm{Exists})$$

[11] That is, PE_2 uses Curry-style rather than Church-style type quantifiers.

The "$(*)$" next to the (All) rule indicates the standard restriction that α must not occur free in any other type from Γ. This prevents us from generalizing types that must match other types; for example, we would like to allow the first of these derivations but not the second:

$$\dfrac{\dfrac{\dfrac{}{\vdash\ ;\ x : \alpha^\perp,\ x : \alpha}\ \text{(Variable)}}{\vdash\ ;\ x \parr x : \alpha \multimap \alpha}\ \text{(Par)}}{\vdash\ ;\ x \parr x : \forall\alpha.\alpha \multimap \alpha}\ \text{(All)} \qquad \dfrac{\dfrac{\dfrac{}{\vdash\ ;\ x : \alpha^\perp,\ x : \alpha}\ \text{(Variable)}}{\vdash\ ;\ x : \alpha^\perp,\ x : \forall\alpha.\alpha}\ \text{(All)}}{\vdash\ ;\ x \parr x : \alpha \multimap \forall\alpha.\alpha}\ \text{(Par)}$$

Because there are no special term forms, there are no associated reduction rules either. We do have $(\forall\alpha.A)^\perp = \exists\alpha.(A^\perp)$ and $(\exists\alpha.A)^\perp = \forall\alpha.(A^\perp)$, but the existing reduction rules can already handle any coequation between terms with these types! If we have $t \perp u$ with t from $\exists\alpha.A$ and u from $\forall\alpha.(A^\perp)$, then there is some B such that t can instead be regarded as having type $A[B/\alpha]$. We can further safely use u as type $(A^\perp)[B^\perp/\alpha]$, since it is a $\forall$, and this is fairly easily equal to $A[B/\alpha]^\perp$. Then whatever rules exist for coequations between $A[B/\alpha]$ and $A[B/\alpha]^\perp$ apply here too!

wait, is that rigorously true?

double check this!

10 The exponentials

D r a f t y

Talk about the failure of $A \otimes B \multimap A \parr B$ somewhere? Also, would it be better to bring up values being used exactly once earlier?

PE_2 has no fundamental mechanism for duplicating arbitrary values. Variables let values be *moved* from one place to another, and values $A^\perp$ let values of A be *used up*, but that is all. Indeed, it turns out that we cannot write any term t with $\vdash\ ;\ t : \forall\alpha.\alpha \multimap \alpha \otimes \alpha$.[12] Ultimately, this should be expected: since terms can have dependencies on other terms, duplicating them may well be unsafe: if a term is duplicated and then used, its dependencies may be used up or modified as well, meaning that the duplicate will no longer work. That said, there *are* types whose values can be duplicated, as an operation specific to that type. For example:

wait, am I obfuscating between "terms" and "values"?

$$\vdash\ ;\ \circledast \parr (* \otimes *) : \mathbf{1} \multimap \mathbf{1} \otimes \mathbf{1}$$
$$\vdash\ ;\ o(;\mathbf{tt} \otimes \mathbf{tt},\ \circledast\ ;\ \square\ ;\mathbf{ff} \otimes \mathbf{ff},\ \circledast) \parr o : \mathbf{2} \multimap \mathbf{2} \otimes \mathbf{2}$$

...but what about *deleting* them?

In order to support more general and more polymorphic patterns of duplication and deletion, PE_2 has the final dual pair of types we will examine: the *exponentials*, ! and ?. It is easiest to understand values of $!A$ as objects and those of $?A$ as subjects, so that is the approach we will primarily take. A value of $!A$ is, as an object, a *safely copyable and deletable factory* for

say something about what that implies about the type?

[12] We *do* have $\vdash\ ; x \parr x(;\ x_1, x_1\ \square\ ;\ x_2, x_2) : \forall\alpha.\alpha \multimap \alpha \,\&\, \alpha$, though! But this doesn't count as "duplication", since we can only get one component back out.

As. Dually, a value of $?A = (!(A^\perp))^\perp$ is, as a subject, something that consumes *several* (or possibly *zero*) As. Roughly speaking, $!A$ means something like $1 \mathbin{\&} A \mathbin{\&} (A \otimes A) \mathbin{\&} (A \otimes A \otimes A) \mathbin{\&} \ldots$, while $?A$ means something like $\bot \oplus A \oplus (A \parr A) \oplus (A \parr A \parr A) \oplus \ldots$—except that such a hypothetical "infinitary &" would presumably allow different implementations for each choice of how many As, while duplicates of an $!A$ value are guaranteed to produce the same A.

We will start with the term constructors and typing rules for ?, because they are more straightforward. We can build a $?A$ in one of three ways:

1. We can take an existing t from A and "put it into ?", which is written $?t$ and called "dereliction". This is a subject which eliminates an $!(A^\perp)$ by requesting only one $A^\perp$, and feeding it to t.

2. We can write $\_$, which is called "weakening". This is a subject which discards its object.

3. If we have two interdependent $?A$s, t and u, we can combine them into one, which is written $t \mathbin{@} u$ and called "contraction". This is a subject which eliminates an $!(A^\perp)$ by duplicating it and feeding the copies to t and u.

Here are the actual typing rules:

$$\frac{\vdash \Theta;\ \Gamma,\ t : A}{\vdash \Theta;\ \Gamma,\ ?t : ?A}\ \text{(Dereliction)}$$

$$\frac{\vdash \Theta;\ \Gamma}{\vdash \Theta;\ \Gamma,\ \_ : ?A}\ \text{(Weakening)}$$

$$\frac{\vdash \Theta;\ \Gamma,\ t : ?A,\ u : ?A}{\vdash \Theta;\ \Gamma,\ t \mathbin{@} u : ?A}\ \text{(Contraction)}$$

! has only one kind of term and only one typing rule, but it's more complicated, so we'll start with an approximation. The underlying reason why duplication of arbitrary values is unsafe is that they may have dependencies; so we could, say, define a kind of term $!t$ and require that for $!t$ to be well-typed, it must not have any dependencies:

$$\frac{\vdash ;\ t : A}{\vdash ;\ !t : !A}\ \text{(Fake Of Course)}$$

This works, but it is more limiting than it needs to be. The critical observation is that *duplication of something with dependencies is still safe if the dependencies are duplicated too*; so rather than restricting to *no* dependencies, we restrict to only *copyable* dependencies. This is achieved through the same kind of indirection as in & (which also introduces some laziness): an !-term is of the form

$x_1 \ldots x_n(P)$, where the program P has $n+1$ terms in its main body in any well-typed case. As with &, the final term is the "result": in this case, it will be the A the !-term is obligated to be able to manufacture. Then, by requiring P to be self-contained and having all dependencies in the surrounding program proxied through the first n terms via the "forward reference" variables (again as in &), we can regulate what kind of dependencies are allowed. In particular: anything x_i eventually gets cut against will have to have a type dual to the ith term of P's main body. So if we restrict the non-result terms of P's main body to have ? types, the external dependencies of the !-term will have to have ! types, and hence be themselves copyable! The fact that the non-result terms need to have ? types rather than ! types may make more sense if they are thought of as subjects that eliminate the dependencies of the result term.

The typing rule for ! is (where $?\overline{A}$ means $?A_1, \ldots, ?A_n$):

$$\frac{\vdash \Theta;\ \bar{t} : ?\overline{A},\ t : A}{\vdash ;\ \overline{x} : ?\overline{A},\ \overline{x}(\Theta;\ \bar{t},\ t) : !A}\ \text{(Of Course)}$$

This is subject to most of the same notes as (With): $\bar{t}$, $\overline{A}$, and $\overline{x}$ should be lists of terms, types, and variables, respectively, all of the same length; none of $\overline{x}$ should already appear in the program; and $\overline{x} : ?\overline{A}$ means $x_1 : ?A_1, \ldots, x_n : ?A_n$.

The first reduction rule is fairly straightforward and analogous to (Case Left/Right):

$$\overline{x}(\Theta;\ \bar{t},\ t) \perp ?u \longrightarrow \Theta,\ \overline{x} \perp \bar{t},\ t \perp u \qquad \text{(Read)}$$

The next one, for _, is a bit subtler. One might expect $t \perp \_ \longrightarrow$, but this is incorrect! It is not safe to simply discard t; its dependencies must be discarded as well, and transitively. So we have:

$$\overline{x}(P) \perp \_ \longrightarrow x_1 \perp \_, \ldots, x_n \perp \_ \qquad \text{(Discard)}$$

The new coequations replace the "forward reference to P's dependency eliminators" meaning of the xs with a "immediate reference to a deletion eliminator" meaning.

Finally, we must give a rule for contraction. This is trickier still, especially since it must involve variable freshening, because it involves duplicating a term, and that term may involve variables—we do not have the luxury of simply declaring that freshening is a convention when we are giving actual reduction rules!

In order to accommodate the need for freshening, one approach we can take—and (nearly) the one Abramsky takes—is to first say that each variable name can be split into a *base name* and a *suffix*, where the suffix consists of any trailing αs and βs; for example, $x_3\alpha\alpha\beta\alpha$ has base name x_3 and suffix $\alpha\alpha\beta\alpha$.[13]

[13] Abramsky uses ls and rs rather than αs and βs; but he also fixes some bijection between variable names and pairs rather than talking about trailing symbols, so his scheme does not have an issue in the event that he uses variable names l and r, while I do—and I *have* used variable names l and r above!

Then, if we have a variable name, term, or program z, we write z^α to mean the result of appending a α to the suffix of every variable name in it, and z^β to mean the result of appending a β to the suffix of every variable name in it. Finally, we must redefine our recurring (†) condition to the following strengthened version: no variable in either premise of a rule with the condition can occur in the other premise, *or become* a variable in the other premise by means of some applications of these freshening operations. If we don't do this strengthening, then putting freshening in the reduction rule for contraction could cause the creation of variable names that already existed. One easy way of satisfying this condition is to simply ensure that no one base name appears in both premises, although this is only sufficient, not necessary. In any case, every example shown thus far that has satisfied the old version of (†) has also satisfied the new version, since we have not used α or β in our variable names, and hence having distinct names has implied having distinct base names.

With this obtuse diversion out of the way, we can finally give a reduction rule for contraction.

$$\overline{x}(P) \perp u \mathbin{@} v \longrightarrow \overline{x} \perp (\overline{x}^\alpha \mathbin{@} \overline{x}^\beta),\ \overline{x}(P)^\alpha \perp u,\ \overline{x}(P)^\beta \perp v \qquad \text{(Copy)}$$

Here $\overline{x} \perp (\overline{x}^\alpha \mathbin{@} \overline{x}^\beta)$ means $x_1 \perp (x_1^\alpha \mathbin{@} x_1^\beta), \ldots, x_n \perp (x_n^\alpha \mathbin{@} x_n^\beta)$.

There are two main parts to the right-hand side of this rule. The easier part is $\overline{x}(P)^\alpha \perp u, \overline{x}(P)^\beta \perp v$. We simply make two copies of the !-term, freshen the variables in both of them to avoid conflicts, and then cut each of them against their corresponding eliminator. The other part, $\overline{x} \perp (\overline{x}^\alpha \mathbin{@} \overline{x}^\beta)$, is trickier. This is the part that ensures duplication of $\overline{x}(P)$'s dependencies. It functions somewhat similarly to the rule for weakening: each coequation changes the meaning of x_i from "forward reference to a dependency eliminator of P" to "immediate reference to a 'duplicate and then cut against forward references to dependency eliminators of P^α and P^β' eliminator". Of course, *that* duplication may introduce coequations like this too, so the duplication process will continue to propagate upward through the dependency tree as necessary until it reaches !-terms that are self-contained.

Let's look at some examples. Just as some types A may support $A \multimap A \otimes A$, some types may support $A \multimap !A$. This means, intuitively, that it is possible to consume all of the information from an A in one go and then repackage it into a copyable format. Types for which this may be impossible include things like most &s, because it is only possible to get one of the components out.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\vdash ;\ * : \mathbf{1}}\ \text{(One)}}{\vdash ;\ (;\ *) : !\mathbf{1}}\ \text{(Of Course)}}{\vdash ;\ (;\ *) : !\mathbf{1},\ \circledast : \bot}\ \text{(Bottom)}}{\vdash ;\ \circledast : \bot,\ (;\ *) : !\mathbf{1}}\ \text{(Xchg (ii))}}{\vdash ;\ \circledast \parr (;\ *) : \mathbf{1} \multimap !\mathbf{1}}\ \text{(Par)}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\vdash ;\ \mathbf{tt} : \mathbf{2}}\ \text{(known)}}{\vdash ;\ (;\ \mathbf{tt}) : !\mathbf{2}}\ \text{(Of Course)}}{\vdash ;\ (;\ \mathbf{tt}) : !\mathbf{2}, \circledast : \bot}\ \text{(Bottom)} \qquad \dfrac{\dfrac{\dfrac{}{\vdash ;\ \mathbf{ff} : \mathbf{2}}\ \text{(known)}}{\vdash ;\ (;\ \mathbf{ff}) : !\mathbf{2}}\ \text{(Of Course)}}{\vdash ;\ (;\ \mathbf{ff}) : !\mathbf{2}, \circledast : \bot}\ \text{(Bottom)}}{\dfrac{\dfrac{\vdash ;\ o : !\mathbf{2}, o(;\ (;\ \mathbf{tt}), \circledast \,\square\, ;\ (;\ \mathbf{ff}), \circledast) : \bot \,\&\, \bot}{\vdash ;\ o(;\ (;\ \mathbf{tt}), \circledast \,\square\, ;\ (;\ \mathbf{ff}), \circledast) : \bot \,\&\, \bot, o : !\mathbf{2}}\ \text{(Xchg (ii))}}{\vdash ;\ o(;\ (;\ \mathbf{tt}), \circledast \,\square\, ;\ (;\ \mathbf{ff}), \circledast) \parr o : \mathbf{2} \multimap !\mathbf{2}}\ \text{(Par)}}\ \text{(With)}$$

The exponentials are *called* "exponentials" because they "turn additives into multiplicatives": we can convert between $!(A \,\&\, B)$ and $!A \otimes !B$, and between $?(A \oplus B)$ and $?A \parr ?B$.

$$\dfrac{\dfrac{\dfrac{\vdots}{\vdash ;\ ?\mathsf{inl}(a) : ?(\alpha^\perp \oplus \beta^\perp), a : \alpha}}{\vdash ;\ w_1 : ?(\alpha^\perp \oplus \beta^\perp), w_1(;\ ?\mathsf{inl}(a), a) : !\alpha}\ \text{(Of Course)} \qquad \begin{array}{c}\vdots\\ \vdash ;\ w_2 : ?(\alpha^\perp \oplus \beta^\perp), w_2(;\ ?\mathsf{inr}(b), b) : !\beta\end{array}}{\begin{array}{c}\vdash ;\ w_1 : ?(\alpha^\perp \oplus \beta^\perp), w_2 : ?(\alpha^\perp \oplus \beta^\perp), w_1(;\ ?\mathsf{inl}(a), a) \otimes w_2(;\ ?\mathsf{inr}(b), b) : !\alpha \otimes !\beta\\ \vdots\ \text{(Xchg (ii), Xchg (ii), Contraction)}\\ \vdash ;\ w_1(;\ ?\mathsf{inl}(a), a) \otimes w_2(;\ ?\mathsf{inr}(b), b) : !\alpha \otimes !\beta, w_1 \mathbin{@} w_2 : ?(\alpha^\perp \oplus \beta^\perp)\\ \vdots\ \text{(Xchg (ii), Par, All, All)}\\ \vdash ;\ (w_1 \mathbin{@} w_2) \parr (w_1(;\ ?\mathsf{inl}(a), a) \otimes w_2(;\ ?\mathsf{inr}(b), b)) : \forall\alpha.\forall\beta.!(\alpha \,\&\, \beta) \multimap !\alpha \otimes !\beta\end{array}}\ \text{(Tensor)}$$

And:

$$\vdash ;\ (a \parr b) \parr ab(;\ a',\ b',\ a'b'(;\ ?a'',\ \_,\ a'' \,\square\, ;\ \_,\ ?b'',\ b'')) : \forall\alpha.\forall\beta.!\alpha \otimes !\beta \multimap !(\alpha \& \beta)$$

The cases for ? and $\parr$ are essentially the same, but with the sides of the $\parr$ swapped.