I Got Plenty o' Nuttin'

Conor McBride

Mathematically Structured Programming
University of Strathclyde
conor@strictlypositive.org

Abstract. Work to date on combining linear types and dependent types has deliberately and successfully avoided doing so. Entirely fit for their own purposes, such systems wisely insist that types depend only on the replicable sublanguage, thus sidestepping the issue of counting uses of limited-use data either within types or in ways which are only really needed to shut the typechecker up. As a result, the linear implication ('lollipop') stubbornly remains a non-dependent $S \multimap T$. This paper defines and establishes the basic metatheory of a type theory supporting a 'dependent lollipop' $(x : S) \multimap T[x]$, where what the input used to be is in some way commemorated by the type of the output. For example, we might convert list to length-indexed vectors *in place* by a function with type $(l : \mathsf{List}\ X) \multimap \mathsf{Vector}\ X\ (\mathsf{length}\ l)$. Usage is tracked with resource annotations belonging to an arbitrary rig, or 'riNg without Negation'. The key insight is to use the rig's zero to mark information in contexts which is present for purposes of contemplation rather than consumption, like a meal we remember fondly but cannot eat twice. We need no runtime copies of l to form the above vector type. We can have plenty of nothing with no additional runtime resource, and nothing is plenty for the construction of dependent types.

1 Introduction

Girard's linear logic [15] gives a discipline for parsimonious control of resources, and Martin-Löf's type theory [18] gives a discipline for precise specification of programs, but the two have not always seen eye-to-eye.

Recent times have seen attempts to change that. Pfenning blazed the trail in 2002, working with Cervesato on the linear logical framework [8], returning to the theme in 2011 to bring dependency on values to session types in joint work with Toninho and Caires [27]. Shi and Xi have linear typing in ATS [24]. Swamy and colleagues have linear typing in $F^\star$ [25]. Brady's Idris [6] has uniqueness typing, after the fashion of Clean [9]. Vákár has given a categorical semantics [28] where dependency is permitted on the cartesian sublanguage. Gaboardi and colleagues use linear types to study differential privacy [12]. Krishnaswami, Pradic and Benton [16] give a beautiful calculus based on a monoidal adjunction which relates a monoidal closed category of linear types to a cartesian closed category of intuitionistic dependent types.

Linear dependent types are a hot topic, but for all concerned bar me, linearity stops when dependency starts. The application rules (for twain they are, and never shall they meet) from Krishnaswami and coauthors illustrate the puzzle.

$$\frac{\Gamma ; \Delta \vdash e : A \multimap B \quad \Gamma ; \Delta' \vdash e' : A}{\Gamma ; \Delta, \Delta' \vdash e\ e' : B} \qquad \frac{\Gamma ; \Delta \vdash e : \Pi x : X.\ A \quad \Gamma \vdash e' : X}{\Gamma ; \Delta \vdash e\ e' : A[e'/x]}$$

Judgments have an intuitionistic context, Γ, shared in each premise, and a linear context Δ, carved up between the premises. Later types in Γ may depend on earlier variables, so Γ cannot be freely permuted, but in order to distribute resources in the linear application, the linear context *must* admit permutation. Accordingly, types in Δ can depend on variables from Γ, but not on linear variables. How could they? When the linear context is chopped in two, some of the linear variables disappear! Accordingly, the argument in the dependent application is a purely intutionistic term, depending only on shared information, and the result type is thus well formed.

In this paper, I resolve the dilemma, with one idea: *nothing*. Contexts account for *how many* of each variable we have, and when we carve them up, we retain all the variables in each part but we split their quantities, so that we know of which we have none. That is, contexts with typed variables in common are *modules*, in the algebraic sense, with pointwise addition of resources drawn from some rig ('riNg without Negation'). Judgments account for how many of the given term are to be constructed from the resources in the context, and when we are constructing types, that quantity will be *zero*, distinguishing dynamic *consumption* from static *contemplation*. Correspondingly, we retain the ability to contemplate variables which stand for things unavailable for computation. My application rule (for there is but one) illustrates the point.

$$\frac{\Delta_0 \vdash \rho\ f \in (\pi\, x : S) \to T \quad \Delta_1 \vdash \rho\pi\ S \ni s}{\Delta_0 + \Delta_1 \vdash \rho\ f\ s \in T[s : S/x]}$$

The function type is decorated with the 'unit price' π to be paid in copies of the input for each output required, so to make ρ outputs, our resource must be split between ρ functions and $\rho\pi$ arguments. The two contexts Δ_0 and Δ_1 have the same variables, even if some of them have zero resource, so the resulting type makes sense. If the ring has a 1, we may write $(1\, x : S) \to T$ as $(x : S) \multimap T$.

In summary, this paper contributes the definition of a type theory with uniform treatment of **unit-priced dependent function spaces** which is even unto its syntax **bidirectional**, in the sense of Pierce and Turner [22], resulting in a metatheoretic treatment of novel simplicity, including **type preservation** and a proof that **erasure** of all zero-resourced components retains type safety.

2 A Rig of Resources

Let us model resources with a rig, rather as Petricek, Orchard and Mycroft do, in their presentation of *coeffects*. [21].

Definition 1 (rig) *Let $\mathcal{R}$ be a set (whose elements are typically called ρ, π, ϕ), equipped with a value 0, an addition $\rho + \pi$, and a 'multiplication', $\phi\rho$, such that*

$$0 + \rho = \rho \qquad \rho + (\pi + \phi) = (\rho + \pi) + \phi \qquad \rho + \pi = \pi + \rho$$
$$\phi(\rho\pi) = (\phi\rho)\pi \qquad 0\rho = 0 = \rho 0 \qquad \phi(\rho + \pi) = \phi\rho + \phi\pi \qquad (\rho + \pi)\phi = \rho\phi + \pi\phi$$

I was brought up not to presume that a rig has a 1. Indeed, the trivial rig $\{0\}$ will give us the purely intuitionistic type theory we know and love. Moreover, every rig has the trivial sub-rig, a copy of intuitionistic type theory, in which we shall be able to construct objects (types, especially) whose runtime resource footprint is nothing but whose contemplative role allows more precise typechecking. The $\{0, 1\}$ rig gives a purely linear system, but the key example is 'none-one-tons'.

Definition 2 (none-one-tons)

$\rho+\pi$	0	1	ω
0	0	1	ω
1	1	ω	ω
ω	ω	ω	ω

$\rho\pi$	0	1	ω
0	0	0	0
1	0	1	ω
ω	0	ω	ω

The 0 value represents the resource required for usage only in types; 1 resources linear runtime usage; ω indicates relevant usage with no upper limit, or with weakening for ω (as treated in section 12), *arbitrary* usage. With the latter in place, we get three recognizable quantifiers,

informally	$\forall x\!:\!S.\ T$	$(x\!:\!S) \multimap T$	$\Pi x\!:\!S.\ T$
formally	$(0\,x\!:\!S) \to T$	$(1\,x\!:\!S) \to T$	$(\omega\,x\!:\!S) \to T$

where $\forall$ is parametric, effectively an intersection rather than a product, with abstractions and applications erasable at runtime, but Π is *ad hoc* and thus runtime persistent. In making that valuable distinction, I learn from Miquel's Implicit Calculus of Constructions [20]: a dependent intersection collects only the functions which work uniformly for all inputs; a dependent product allows distinct outputs for distinguishable inputs.

Unlike Miquel, I shall remain explicit in the business of typechecking, but once a non-zero term has been checked, we can erase its zero-resourced substructures to obtain an untyped (for types are zero-resourced) λ-term which is the runtime version of it and computes in simulation with the unerased original, as we shall see in section 11. Note that erasure relies on the absence of negation: zero-resourcing should not be an accident of 'cancellation'.

Although I give only the bare functional essentials, one can readily imagine an extension with datatypes and records, where the type of an *in place* sorting algorithm might be $(1\,xs\!:\!\mathsf{List\ Int}) \to \{1\,ys\!:\!\mathsf{OrderedList\ Int};\ 0\,p\!:\!ys \sim xs\}$.

We can, of course, contemplate many other variations, whether for usage analysis in the style of Wadler [30], or for modelling partial knowledge as Gaboardi and coauthors have done [12].

3 Syntax and Computation

The type theory I present is parametrised by its system of sorts (or 'universes'). I keep it predicative: each sort is closed under quantification, but quantifying over

'higher' sorts is forbidden. My aim (for further work) is to support a straightforward normalization proof in the style championed by Abel [2]. Indeed, the resource annotations play no role in normalization, so erasing them and embedding this system into the predicative type theory in Abel's habilitationsschrift [1] may well deliver the result, but a direct proof would be preferable.

Definition 3 (sort ordering) *Let $\mathcal{S}$ be a set of sorts (i,j,k) with a wellfounded ordering $j \succ i$:* $\dfrac{k \succ j \quad j \succ i}{k \succ i}$ $\dfrac{\forall j.\, (\forall i.\, j \succ i \to P[i]) \to P[j]}{\forall k.\, P[k]}$.

Asperti and the Matita team give a bidirectional 'external syntax' for the Calculus of Constructions [4], exploiting the opportunities it offers to omit type annotations. I have been working informally with bidirectional kernel type theories for about ten years, so a precise treatment is overdue. Here, the very syntax of the theory is defined by mutual induction between *terms*, with types specified in advance, and *eliminations* with types synthesized. Types are terms, of course.

Definition 4 (term, elimination)

$R, S, T, s, t ::= *_i$	*sort i*	$e, f ::= x$	*variable*
$\mid (\pi\, x : S) \to T$	*function type*	$\mid f\ s$	*application*
$\mid \lambda x.\, t$	*abstraction*	$\mid s : S$	*annotation*
$\mid \underline{e}$	*elimination*		

Sorts are explicit, and the function type is annotated with a 'price' $\pi \in \mathcal{R}$: the number of copies of the input required to compute each copy of the output. This bidirectional design ensures that λ need not have a domain annotation—that information always comes from the prior type. An elimination becomes a term without annotation: indeed, we shall have *two* candidates for its type.

In the other direction, a term can be used as an elimination only if we give a type at which to check it. Excluding type annotations from the syntax would force terms into β-normal form. Effectively, type annotations mark places where computation is unfinished—the 'cuts', in the language of logical calculi: we see the types of the 'active' terms. I plan neither to write nor to read mid-term type annotations, but rather to work with β-normal forms and typed *definitions*.

Syntactically, terms do not fit where variables go: we must either compute the β-redexes after substitution, as in the hereditary method introduced by Watkins and coauthors [32] and deployed to great effect by Adams [3], or we must find some other way to suspend computation in a valid form. By adding cuts to the syntax of eliminations, I admit a small-step characterization of reduction which allows us to approach the question of type preservation without first establishing β-normalization, which is exactly cut elimination in the sense of Gentzen [14].

The syntax is set up so that a redex pushes a cut towards its elimination. The β-rule replaces a redex *elimination* at a function type by redexes at the domain and range. The υ-rule strips the annotation from a fully computed *term*.

Definition 5 (contraction, reduction, computation) Contraction *schemes are as follows:*

$$(\lambda x.\, t : (\pi\, x{:}S) \to T)\ s \rightsquigarrow_\beta (t{:}T)[s{:}S/x] \qquad \underline{t : T} \rightsquigarrow_\upsilon t$$

Closing $\rightsquigarrow_\beta$ *and* $\rightsquigarrow_\upsilon$ *under all one-hole contexts yields* reduction*,* $s \rightsquigarrow t$*. The reflexive-transitive closure of reduction is* computation*:* $\twoheadrightarrow = \rightsquigarrow^*$*.*

Let us defer the metatheory of computation and build more of the type theory.

4 Typing Rules

What we might be used to calling a *context* is here a *pre*context, Γ.

Definition 6 (precontext, context) $\Gamma ::= \cdot \mid \Gamma, x{:}S$
A Γ*-context is a marking-up of* Γ *with a rig element for each variable.*

$$\frac{}{\cdot \ \textit{is a}\ \ \mathrm{Cx}(\cdot)} \qquad \frac{\Delta \ \textit{is a}\ \ \mathrm{Cx}(\Gamma)}{\Delta, \rho\, x{:}S \ \textit{is a}\ \ \mathrm{Cx}(\Gamma, x{:}S)}$$

If Δ *is a* Γ*-context, we may take* $\lfloor \Delta \rfloor = \Gamma$

It is my tidy habit to talk of "Γ-contexts", rather than slicing "contexts" by $\lfloor - \rfloor$, after the fact: Γ-contexts form an $\mathcal{R}$-module: addition $\Delta + \Delta'$ is pointwise, and multiplication $\phi\Delta$ premultiplies all the rig-annotations on variables in Δ by ϕ, keeping types the same. It is another (similar) habit to suppress the conditions required for wellformedness of expressions like $\Delta + \Delta'$: that is how I am telling you that Δ and Δ' are both Γ-contexts for some Γ.

Definition 7 (prejudgment) *The prejudgment forms,* $\mathcal{P}$*, are as follows.*

type checking	$\Gamma \vdash T \ni t$
type synthesis	$\Gamma \vdash e \in S$

As with contexts, judgments decorate prejudgments with resources from $\mathcal{R}$. We may define a forgetful map $\lfloor - \rfloor$ from judgments to prejudgments.

Definition 8 (judgment) *The judgment forms,* $\mathcal{J}$*, and* $\lfloor - \rfloor : \mathcal{J} \to \mathcal{P}$ *are given as follows.*

type checking	$\lfloor \Delta \vdash \rho\, T \ni t \rfloor = \lfloor \Delta \rfloor \vdash T \ni t$
type synthesis	$\lfloor \Delta \vdash \rho\, e \in S \rfloor = \lfloor \Delta \rfloor \vdash e \in S$

Let us say that t and e are, respectively, the *subjects* of these judgments. Let us further designate Δ and T *inputs* and S an *output*. My plan is to arrange the rules so that, presupposing input to a judgment makes sense, the subjects are validated and sensible output (if any) is synthesized. My policy is "garbage in; garbage out" and its benefit is that information flows as typing unfolds from subjects (once checked) to inputs and outputs, but never from inputs back to

subjects. By restricting the interaction between the parts of the judgments in this way, I arrange a straightforward inductive proof of type preservation.

Readers familiar with traditional presentations may be surprised by the lack of a *context validity* judgment: I was surprised, too. Rather, we must now maintain the invariant that *if* a derivable judgment has valid input, every judgment in its derivation has valid input and output, and its own output is valid.

Any assignment of resources to a valid precontext Γ gives a context Δ. The typing judgments indicate that the resources in Δ are precisely enough to construct the given t or e with multiplicity ρ: it may help to think of ρ as 'how many runtime copies'. The types S and T consume no resources. We may use the derived judgment form $\Gamma \vdash_0 T \ni t$ to abbreviate the all-resources-zero judgment $0\Gamma \vdash 0\ T \ni t$ which is common when checking that types are well formed.

Definition 9 (type checking and synthesis) *Type checking and synthesis are given by the following mutually inductive definition.*

$$\textsc{pre}\quad \frac{\Delta \vdash \rho\ R \ni t}{\Delta \vdash \rho\ T \ni t}\ T \rightsquigarrow R \qquad \textsc{post}\quad \frac{\Delta \vdash \rho\ e \in S}{\Delta \vdash \rho\ e \in R}\ S \rightsquigarrow R$$

$$\textsc{sort}\quad \frac{}{\Gamma \vdash_0 *_j \ni *_i}\ j \succ i \qquad \textsc{var}\quad \frac{}{0\Gamma, \rho\, x{:}S, 0\Gamma' \vdash \rho\ x \in S}$$

$$\textsc{fun}\quad \frac{\Gamma \vdash_0 *_i \ni S \quad \Gamma, x{:}S \vdash_0 *_i \ni T}{\Gamma \vdash_0\ *_i \ni (\pi\, x{:}S) \to T} \qquad \textsc{lam}\quad \frac{\Delta, \rho\pi\, x{:}S \vdash \rho\ T \ni t}{\Delta \vdash \rho\ (\pi\, x{:}S) \to T \ni \lambda x.\, t}$$

$$\textsc{app}\quad \frac{\Delta_0 \vdash \rho\ f \in (\pi\, x{:}S) \to T \quad \Delta_1 \vdash \rho\pi\ S \ni s}{\Delta_0 + \Delta_1 \vdash \rho\ f\ s \in T[s{:}S/x]}$$

$$\textsc{elim}\quad \frac{\Delta \vdash \rho\ e \in S}{\Delta \vdash \rho\ T \ni \underline{e}}\ S \preceq T \qquad \textsc{cut}\quad \frac{\lfloor \Delta \rfloor \vdash_0 *_i \ni S \quad \Delta \vdash \rho\ S \ni s}{\Delta \vdash \rho\ s{:}S \in S}$$

Introduction forms require a type proposed in advance. It may be necessary to PRE-compute that type, e.g., to see it as a function type when checking a lambda. The SORT rule is an axiom, reflecting the presupposition that the context is valid already. Wherever the context is extended, the rules guarantee that the new variable has a valid type: in FUN, it comes from the subject and is checked; in LAM, it is extracted from an input. The FUN rule pushes sort demands inward rather than taking the max of inferred sorts, so it is critical that ELIM allows cumulative inclusion of sorts as we switch from construction to usage.

The VAR or CUT at the heart of an elimination drives the synthesis of its type, which we may need to POST-compute if it is to conform to its use. E.g., to use APP, we must first compute the type of the function to a function type, exposing its domain (to check the argument) and range (whence we get the output type). Presuming context validity, VAR is an axiom enforcing the resource policy. Note how APP splits our resources between function and argument, but with the same *pre*context for both. The usual 'conversion rule' is replaced by ELIM sandwiched between POSTs and PREs computing two types to compatible forms.

Valid typing derivations form $\mathcal{R}$-modules. Whatever you make some of with some things, you can make none of with nothing—plenty for making types.

5 Dependent Iterative Demolition of Lists

Thus far, I have presented a bare theory of functions. Informally, let us imagine some data by working in a context. Underlines $\underline{e}$ help readability only of *meta*theory: I omit them, and sorts to boot. I telescope some function types for brevity. Let us have zero type constructors and plenty of value constructors.

$$0\ \mathsf{List} : (0\ X : *) \to *, \qquad \omega\ \mathsf{nil} : (0\ X : *) \to \mathsf{List}\ X,$$
$$\omega\ \mathsf{cons} : (0\ X : *, 1\ x : X, 1\ xs : \mathsf{List}\ X) \to \mathsf{List}\ X$$

Let us add a traditional *dependent* eliminator, resourced for iterative demolition.

$$\begin{aligned}\omega\ \mathsf{lit} : {} & (0\ X : *, 0\ P : (0\ x : \mathsf{List}\ X) \to *) \to \\ & (1\ n : P\ (\mathsf{nil}\ X)) \to \\ & (\omega\ c : (1\ x : X, 0\ xs : \mathsf{List}\ X, 1\ p : P\ xs) \to P\ (\mathsf{cons}\ X\ x\ xs)) \to \\ & (1\ xs : \mathsf{List}\ X) \to P\ xs\end{aligned}$$

The nil case is used once. In the cons case the tail xs has been reconsituted as p but remains contemplated in types. The intended computation rules conform.

$$\mathsf{lit}\ X\ P\ n\ c\ (\mathsf{nil}\ X) \rightsquigarrow n \qquad \mathsf{lit}\ X\ P\ n\ c\ (\mathsf{cons}\ X\ x\ xs) \rightsquigarrow c\ x\ xs\ (\mathsf{lit}\ X\ P\ n\ c\ xs)$$

Typechecking both sides of each rule, we see that n occurs once on both sides of each, but that c is dropped in one rule and duplicated in the other—some weakening is needed. Meanwhile, for cons, the tail xs has its one use in the recursive call but can still be given as the zero-resourced tail argument of c.

Some familiar list operations can be seen as simple demolitions:

$$\begin{aligned}\omega\ \mathsf{append}\ &: (0\ X : *, 1\ xs : \mathsf{List}\ X, 1\ ys : \mathsf{List}\ X) \to \mathsf{List}\ X \\ \mathsf{append} &= \lambda X.\, \lambda xs.\, \lambda ys.\, \mathsf{lit}\ X\ (\lambda\_.\, \mathsf{List}\ X)\ ys\ (\lambda x.\, \lambda\_.\, \lambda xs'.\, \mathsf{cons}\ X\ x\ xs')\ xs \\ \omega\ \mathsf{reverse}\ &: (0\ X : *, 1\ xs : \mathsf{List}\ X) \to \mathsf{List}\ X \\ \mathsf{reverse} &= \lambda X.\, \lambda xs.\, \mathsf{lit}\ X\ (\lambda\_.\, (1\ ys : \mathsf{List}\ X) \to \mathsf{List}\ X) \\ &\qquad (\lambda ys.\, ys)\ (\lambda x.\, \lambda\_.\, \lambda f.\, \lambda ys.\, f\ (\mathsf{cons}\ X\ x\ ys))\ xs\ (\mathsf{nil}\ X)\end{aligned}$$

Now let us have unary numbers for contemplation only, e.g. of vector length.

$$0\ \mathsf{Nat} : *, \qquad 0\ \mathsf{zero} : \mathsf{Nat}, \qquad 0\ \mathsf{suc} : (0\ n : \mathsf{Nat}) \to \mathsf{Nat}$$
$$0\ \mathsf{Vector} : (0\ X : *, 0\ n : \mathsf{Nat}) \to *, \qquad \omega\ \mathsf{vnil} : (0\ X : *) \to \mathsf{Vector}\ X\ \mathsf{zero},$$
$$\omega\ \mathsf{vcons} : (0\ X : *, 0\ n : \mathsf{Nat}, 1\ x : X, 1\ xs : \mathsf{Vector}\ X\ n) \to \mathsf{Vector}\ X\ (\mathsf{suc}\ n)$$

We can measure length statically, so it does not matter that we drop heads. Then, assuming length computes as above, let us turn lists to $\mathsf{vectors}$.

$$\begin{aligned}0\ \mathsf{length}\ &: (0\ X : *, 0\ xs : \mathsf{List}\ X) \to \mathsf{Nat} \\ \mathsf{length} &= \lambda X.\, \mathsf{lit}\ X\ (\lambda\_.\, \mathsf{Nat})\ \mathsf{zero}\ (\lambda x.\, \lambda\_.\, \mathsf{suc}) \\ \omega\ \mathsf{vector}\ &: (0\ X : *, 1\ xs : \mathsf{List}\ X) \to \mathsf{Vector}\ X\ (\mathsf{length}\ xs) \\ \mathsf{vector} &= \lambda X.\, \mathsf{lit}\ X\ (\lambda xs.\, \mathsf{Vector}\ X\ (\mathsf{length}\ xs)) \\ &\qquad (\mathsf{vnil}\ X)\ (\lambda x.\, \lambda xs.\, \lambda xs'.\, \mathsf{vcons}\ X\ (\mathsf{length}\ xs)\ x\ xs')\end{aligned}$$

In the step case, the zero-resourced list tail, xs, is used only to compute a zero-resourced length: the actual runtime tail is recycled as the vector xs'. The impact is to demolish a list whilst constructing a vector whose length depends on what the list used to be: that is the dependent lollipop in action.

6 Confluence of Computation

We shall need to establish the *diamond property* for computation.

Definition 10 (diamond property) *A binary relation R has the* diamond property *if* $\quad \forall s, p, q.\ sRp \wedge sRq \Rightarrow \exists r.\ pRr \wedge qRr.$

$$\begin{array}{ccc} s & R & p \\ R & \exists & R \\ q & R & r \end{array}$$

Appealing to visual intuition, I will typically depict such propositions, where the existential quantifier governs points and proofs below and right of it.

The Tait–Martin-Löf–Takahashi method [26] is adequate to the task. We introduce a 'parallel reduction', $\triangleright$, which amounts to performing none, any or all of the available contractions in a term, but no further redexes thus arising. Proving the diamond property for $\triangleright$ will establish it for $\twoheadrightarrow$. Here is how.

Lemma 11 (parallelogram) *Let R, P be binary relations with $R \subseteq P \subset R^*$. If P has the diamond property, then so has R^*.*

Proof. If sR^*p then for some m, sR^mp, hence also sP^mp. Similarly, for some n, sP^nq. We may now define a 'parallelogram' t_{ij} for $0 \leq i \leq m, 0 \leq j \leq n$, first taking two sides to be the P-sequences we have, $s = t_{00}Pt_{10}P \ldots Pt_{m0} = p$ and $s = t_{00}Pt_{01}P \ldots Pt_{0n} = q$, then applying the diamond property

$$\text{for } 0 \leq i < m, 0 \leq j < n \qquad \begin{array}{ccc} t_{ij} & P & t_{(i+1)j} \\ P & \exists & P \\ t_{i(j+1)} & P & t_{(i+1)(j+1)} \end{array}$$

Let $r = t_{mn}$. The lower two sides give pP^nr and qP^mr, so pR^*r and qR^*r. □

By design, parallel reduction fits Lemma 11. We have structural rules for each construct, together with υ and β rules.

Definition 12 (parallel reduction) *Let parallel reduction, $\triangleright$, be defined by mutual induction for terms and eliminations, as follows.*

$$\frac{}{*_i \triangleright *_i} \qquad \frac{S \triangleright S' \quad T \triangleright T'}{(\pi\, x{:}S) \to T \triangleright (\pi\, x{:}S') \to T'} \qquad \frac{t \triangleright t'}{\lambda x.\, t \triangleright \lambda x.\, t'} \qquad \frac{e \triangleright e'}{\underline{e} \triangleright \underline{e'}} \qquad \frac{t \triangleright t' \quad T \triangleright T'}{\underline{t{:}T} \triangleright t'}$$

$$\frac{}{x \triangleright x} \qquad \frac{f \triangleright f' \quad s \triangleright s'}{f\ s \triangleright f'\ s'} \qquad \frac{t \triangleright t' \quad T \triangleright T'}{t{:}T \triangleright t'{:}T'} \qquad \frac{t \triangleright t' \quad S \triangleright S' \quad T \triangleright T' \quad s \triangleright s'}{(\lambda x.\, t : (\pi\, x{:}S) \to T)\ s \triangleright (t'{:}T')[s'{:}S'/x]}$$

Lemma 13 (parallel reduction computes) $\quad \rightsquigarrow \subseteq \triangleright \subseteq \twoheadrightarrow.$

Proof. Both inclusions are easy inductions on derivations. □

Crucially, parallel reduction commutes with substitution, because the latter duplicates or drops redexes, but never subdivides them.

Lemma 14 (vectorized substitution) *Admissibly,* $\quad \dfrac{t \triangleright t' \quad \boldsymbol{e} \triangleright \boldsymbol{e'}}{t[\boldsymbol{e}/\boldsymbol{y}] \triangleright t'[\boldsymbol{e'}/\boldsymbol{y}]}.$

Proof. Proceed by structural induction on the derivation of $t \triangleright t'$. Effectively we are lifting a substitution on terms to a substitution on parallel reduction derivations. The only interesting cases are for variables and β-contraction.

For $y_i \triangleright y_i$ where e_i/y_i and e'_i/y_i are in the substitutions 'before' and 'after': here, we substitute the given derivation of $e_i \triangleright e'_i$.

For $(\lambda x.\, t : (\pi\, x{:}S) \to T)\, s \triangleright (t' {:} T')[s' {:} S'/x]$, where our 'argument' inductive hypotheses yield $(s{:}S)[\boldsymbol{e}/\boldsymbol{y}] \triangleright (s' {:} S')[\boldsymbol{e'}/\boldsymbol{y}]$, we may extend the term and derivation substitutions, then use 'body' hypotheses to deduce $t[(s{:}S)[\boldsymbol{e}/\boldsymbol{y}]/x, \boldsymbol{e}/\boldsymbol{y}] \triangleright t'[(s' {:} S')[\boldsymbol{e'}/\boldsymbol{y}]/x, \boldsymbol{e'}/\boldsymbol{y}]$ and similarly for T; the usual composition laws allow us to substitute in phases $-[(s{:}S)[\boldsymbol{e}/\boldsymbol{y}]/x, \boldsymbol{e}/\boldsymbol{y}] = -[s' {:} S'/x][\boldsymbol{e}/\boldsymbol{y}]$ and the β rule then yields $((\lambda x.\, t : (\pi\, x{:}S) \to T)\; s)[\boldsymbol{e}/\boldsymbol{y}] \triangleright (t' : T')[s' {:} S'/x][\boldsymbol{e}/\boldsymbol{y}]$. □

Iterating Lemma 14, we get that if $e \twoheadrightarrow e'$ and $t \twoheadrightarrow t'$, then $t[e/x] \twoheadrightarrow t'[e'/x]$.

Lemma 15 (parallel reduction diamond)
$$\begin{array}{c} s \triangleright p \\ \triangledown\ \exists\ \triangledown \\ q \triangleright r \end{array}$$

Proof. We use induction on the derivation of $s \triangleright p$, then case analysis on the derivation of $s \triangleright q$. Where both use the same rule, the inductive hypotheses yield common parallel reducts. The only interesting cases are where computation is possible but one side declines the opportunity.

For υ, we have $\dfrac{s \triangleright p \quad S \triangleright P}{\underline{s{:}S} \triangleright p} \quad \dfrac{\dfrac{s \triangleright q \quad S \triangleright Q}{s{:}S \triangleright q{:}Q}}{\underline{s{:}S} \triangleright \underline{q{:}Q}}$.

Inductively, we obtain $\begin{array}{c} s \triangleright p \\ \triangledown\ \exists\ \triangledown \\ q \triangleright r \end{array}$ and $\begin{array}{c} S \triangleright P \\ \triangledown\ \exists\ \triangledown \\ Q \triangleright R \end{array}$, then also $\dfrac{q \triangleright r \quad Q \triangleright R}{\underline{q{:}Q} \triangleright r}$.

For β, we have
$$\dfrac{\begin{array}{c} s' \triangleright q' \quad S \triangleright Q \quad S' \triangleright Q' \\ \vdots \qquad\ \vdots \qquad\ \vdots \qquad s \triangleright q \end{array}}{(\lambda x.\, s' : (\pi\, x{:}S) \to S')\; s \triangleright (\lambda x.\, q' : (\pi\, x{:}Q) \to Q')\; q}$$

Induction yields common reducts r', R, R', r, so $(p' {:} P')[p{:}P/x] \triangleright (r' {:} R')[r{:}R/x]$ by Lemma 14, then the β rule gives us the required

$$\dfrac{q' \triangleright r' \quad Q \triangleright R \quad Q' \triangleright R' \quad q \triangleright r}{(\lambda x.\, q' : (\pi\, x{:}Q) \to Q')\; q \triangleright (r' {:} R')[r{:}R/x]}$$

□

Corollary 16 (confluence)
$$\begin{array}{c} s \twoheadrightarrow p \\ \twoheadrightarrow\ \exists\ \twoheadrightarrow \\ q \twoheadrightarrow r \end{array}$$

Proof. Apply Lemma 11 with Lemma 13 and Lemma 15. □

7 Subtyping and its Metatheory

Subtyping is the co- and contravariant lifting of the universe ordering through the function type, where Luo's Extended Calculus of Constructions is covariant in the codomain and *equi*variant in the domain [17]. It is not merely convenient, but crucial in forming polymorphic types like $*_1 \ni (0\,X\!:\!*_0) \to (1\,x\!:\!X) \to X$, as the FUN rule demands $X : *_1$. To establish that cut elimination preserves typing, we must justify the subtyping in the ELIM rule with a proof of subsumption, i.e., that *term* can be lifted from sub- to supertype. While developing its proof, below, I had to adjust the definition to allow a little wiggle room where the application rule performs substitution: type annotations obstruct the proof. Rather than demanding cut elimination to get rid of them, I deploy wilful ignorance.

Definition 17 (subtyping) *Let $\sim$ ('similarity') identify terms and eliminations structurally up to $s\!:\!S \sim s\!:\!S'$. Define subtyping inductively thus.*

$$\frac{S \sim T}{S \preceq T} \qquad \frac{j \succ i}{*_i \preceq *_j} \qquad \frac{S' \preceq S \quad T \preceq T'}{(\pi\, x\!:\!S) \to T \preceq (\pi\, x\!:\!S') \to T'}$$

In particular, subtyping is reflexive, so a term $\underline{e}$ is accepted if its *synthesized* type S and its checked type T have a common reduct.

Note that computation plays no role in subtyping: given that it is deployed at the 'change of direction', we can always use POST and PRE to compute as much as is needed to make this rather rigid syntactic definition apply. The rigidity then makes it easier to establish the crucial metatheoretic properties of subtyping.

Lemma 18 (subtyping transitivity) *Admissibly,* $\dfrac{R \preceq S \quad S \preceq T}{R \preceq T}$.

Proof. Contravariance in the rule for function types obliges us to proceed by induction on the maximum of the heights of the derivations (or, in effect, the 'size change' principle, for which subtyping is a paradigmatic example). If both derivations are by similarity, the result is by similarity. Otherwise, we have either sorts, in which case the transitivity of $\succ$ suffices, or function types, in which case the result follows from the inductive hypotheses. □

We need two results about the interaction between subtyping and computation. If we compute one side, we can compute the other side to match, and if we compute both sides independently, we can compute further to reestablish subtyping. Both will depend on the corresponding fact about similarity.

Lemma 19 (similarity preservation) *Again, with $\exists$ applying below and right,*

$$\begin{array}{c} S \sim T \\ \triangledown \;\exists\; \triangledown \\ S' \sim T' \end{array} \qquad \begin{array}{c} S \sim T \\ \triangledown \quad\; \triangledown \\ S' \quad\; T' \\ \exists\; \triangledown \quad\; \triangledown \\ S'' \sim T'' \end{array} \qquad \begin{array}{c} S \sim T \\ \twoheaddownarrow \;\exists\; \twoheaddownarrow \\ S' \sim T' \end{array} \qquad \begin{array}{c} S \sim T \\ \twoheaddownarrow \quad\; \twoheaddownarrow \\ S' \quad\; T' \\ \exists\; \twoheaddownarrow \quad\; \twoheaddownarrow \\ S'' \sim T'' \end{array}$$

Proof. For $\triangleright$, use copycat induction on derivations. If just one side computes, we need only compute the other. When both compute apart, we need to compute both back together, hence the far left $\exists$. For $\twoheadrightarrow$, we iterate the results for $\triangleright$. $\square$

Lemma 20 (subtyping preservation)

$$\begin{array}{ccc} S \preceq T \\ \downarrow\ \exists\ \downarrow \\ S' \preceq T' \end{array} \qquad \begin{array}{ccc} S \preceq T \twoheadrightarrow T' \\ \exists\ \downarrow \qquad\quad \| \\ S' \quad \preceq \quad T' \end{array} \qquad \begin{array}{ccc} S \preceq T \\ \downarrow \quad \downarrow \\ S' \quad T' \\ \exists\ \downarrow \quad \downarrow \\ S'' \preceq T'' \end{array}$$

Proof. Induction on the derivation of $S \preceq T$. Lemma 19 covers similarity. For sorts, there is no way to compute. For function types, computation occurs only within sources and targets, so the inductive hypotheses deliver the result. $\square$

Lemma 21 (subtyping stability) $S \preceq T \Rightarrow S[r : R/x] \preceq T[r : R'/x]$

Proof. Induction on derivations. Wherever R and R' occur distinctly, $\sim$ ignores. $\square$

The key result about subtyping is that it is justified by the admissibility of subsumption, pushing terms up the ordering. We may extend subtyping pointwise to contexts and establish the following rule, contravariant in contexts.

Theorem 22 (subsumption) *If $\Delta' \preceq \Delta$, then admissibly,*

$$\frac{\Delta \vdash \rho\ S \ni s}{\Delta' \vdash \rho\ T \ni s}\ S \preceq T \qquad \frac{\Delta \vdash \rho\ e \in S}{\exists S'.S' \preceq S \wedge \Delta' \vdash \rho\ e \in S'}$$

Proof. We proceed by induction on typing derivations. For PRE, we make use of Lemma 20. We may clearly allow iterated PRE to advance types by $\twoheadrightarrow$, not just $\rightsquigarrow$. I write $\therefore$ to mark an appeal to the inductive hypothesis.

$$\frac{\Delta \vdash \rho\ R \ni t}{\Delta \vdash \rho\ S \ni t}\ S \rightsquigarrow R \quad \begin{array}{c} S \preceq T \\ \downarrow\ \exists\ \downarrow \\ R \preceq R' \end{array} \quad \frac{\therefore\ \Delta' \vdash \rho\ R' \ni t}{\Delta' \vdash \rho\ T \ni t}$$

For SORT, transitivity of $\succ$ suffices. For FUN, we may pass the inflation of the desired sort through to the premises and appeal to induction.

For LAM, we have $\dfrac{S' \preceq S \quad T \preceq T'}{(\pi\, x : S) \to T \preceq (\pi\, x : S') \to T'} \quad \dfrac{\Delta, \rho\pi\, x : S \vdash \rho\ T \ni t}{\Delta \vdash \rho\ (\pi\, x : S) \to T \ni \lambda x.\, t}$

The contravariance of function subtyping allows us to extend the context with the source subtype and check the target supertype, $\dfrac{\therefore\ \Delta', \rho\pi\, x : S' \vdash \rho\ T' \ni t}{\Delta' \vdash \rho\ (\pi\, x : S') \to T' \ni \lambda x.\, t}$.

For ELIM, we have $\dfrac{\Delta \vdash \rho\ e \in R}{\Delta \vdash \rho\ S \ni \underline{e}}\ R \preceq S \quad S \preceq T$. Inductively, for some $R' \preceq R$ we have $\Delta' \vdash \rho\ e \in R'$ and by Lemma 18, we get $R' \preceq T$ and apply ELIM.

For POST, we may again appeal to Lemma 20. For VAR, we look up the subtype given by the contextual subtyping.

For APP, we have $\dfrac{\Delta_0 \vdash \rho\ f \in (\pi\, x{:}S) \to T \quad \Delta_1 \vdash \rho\pi\ S \ni s}{\Delta_0 + \Delta_1 \vdash \rho\ f\ s \in T[s{:}S/x]}$. Given that Δ_0 and Δ_1 share a precontext, we have $\Delta'_0 \preceq \Delta_0$ and $\Delta'_1 \preceq \Delta_1$. Inductively, we may deduce in succession, $\therefore \exists S', T'.\ S \preceq S' \wedge T' \preceq T \wedge \quad \Delta'_0 \vdash \rho\ f \in (\pi\, x{:}S') \to T'$
$\therefore \Delta'_1 \vdash \rho\pi\ S' \ni s$
from which we obtain $\Delta'_0 + \Delta'_1 \vdash \rho\ f\ s \in T'[s{:}S'/x]$ where Lemma 21 gives, as required, $T'[s{:}S'/x] \preceq T[s{:}S/x]$. □

8 Not *That* Kind of Module System

Above, I claimed that Γ-contexts and typing deriviations yield $\mathcal{R}$-modules. Let us make that formal. Firstly, what is an $\mathcal{R}$-module?

Definition 23 ($\mathcal{R}$-module) *An $\mathcal{R}$-module is a set M with*

$$\begin{array}{rl} \textit{zero} & 0 \quad : M \\ \textit{addition} & - + - : M \times M \to M \\ \textit{scalar multiplication} & - - \ : \mathcal{R} \times M \to M \end{array}$$

which make $(M, 0, +)$ a commutative monoid and are compatible $\mathcal{R}$ in that, for all $m \in M$ $\quad 0m = m \quad (\rho + \pi)m = \rho m + \pi m \quad (\rho\pi)m = \rho(\pi m)$.

The obvious (and, for us, adequate) way to form $\mathcal{R}$-modules is pointwise.

Lemma 24 (pointwise $\mathcal{R}$-modules) *$X \to \mathcal{R}$ is an $\mathcal{R}$-module with*

$$0x = 0 \qquad (f + g)\, x = f\, x + g\, x \qquad (\rho f)\, x = \rho(f\, x)$$

Proof Calculation with rig laws for $\mathcal{R}$. □

By taking $X = 0$ and, we get that $0 \to \mathcal{R} \cong 1$ is an $\mathcal{R}$-module. By taking $X = 1$, we get that $1 \to R \cong \mathcal{R}$ itself is an $\mathcal{R}$-module.

Lemma 25 (contexts $\mathcal{R}$-modules) *Γ-contexts form an $\mathcal{R}$-module.*

Proof The Γ-contexts, Δ are given by functions $\Delta|- : \mathrm{dom}(\Gamma) \to \mathcal{R}$, where $(\Delta, \rho\, x{:}S, \Delta')|x = \rho$. Lemma 24 applies. □

Where the 'coeffects' treatment of resources from Petricek and coauthors [21] retains the non-dependent mode of splitting the context—some variables into one premise, the rest in the other—the potential for type dependency forces more rigidity upon us. The module structure of Γ-contexts lets us send Γ to all premises, splitting up the resources the context gives each of Γ's variables.

What about typing derivations? We can play the same game. Let $\mathcal{T}(X)$ be the set of finitely branching trees whose nodes are labelled with elements of X. The typing rules tell us which elements D, of $\mathcal{T}(\mathcal{J})$ constitute valid deductions of the judgment at the root. We can separate such a tree into a *shape* and a set of *positions*. The elements of $\mathcal{T}(\mathcal{J})$ with a given shape form a module by lifting $\mathcal{R}$ pointwise over positions. That is but a dull fact about syntax, but more interesting is that the module can then be restricted to *valid* derivations.

Definition 26 (shape and positions) *Shapes, d, of derivations inhabit trees $\mathcal{T}(\mathcal{P})$ of prejudgments. The shape of a given derivation is given by taking $\lfloor - \rfloor : \mathcal{T}(\mathcal{J}) \to \mathcal{T}(\mathcal{P})$ to be the functorial action $\mathcal{T}(\lfloor - \rfloor)$ which replaces each judgment with its corresponding prejudgment. Position sets, $\mathrm{Pos} : \mathcal{T}(\mathcal{P}) \to \mathbf{Set}$ and prejudgment positions $\mathrm{Pos}' : \mathcal{P} \to \mathbf{Set}$ are given structurally:*

$$\mathrm{Pos}\left(\frac{d_1 \ldots d_n}{P}\right) = \mathrm{Pos}'(P) + \sum_i \mathrm{Pos}(d_i) \qquad \begin{array}{ll} \mathrm{Pos}'(\Gamma \vdash) & = 0 \\ \mathrm{Pos}'(\Gamma \vdash T \ni t) & = \mathrm{dom}(\Gamma) + 1 \\ \mathrm{Pos}'(\Gamma \vdash e \in S) & = \mathrm{dom}(\Gamma) + 1 \end{array}$$

where 1 *is the unit type with element* $\star$.

That is, each typing prejudgment has a resource position for each quantity in its context and for the number of things to be constructed. A straightforward recursive labelling strategy then yields the following.

Lemma 27 (representing derivations) $\forall d \in \mathcal{T}(\mathcal{P}).$
$$\{D : \mathcal{T}(\mathcal{J}) \mid \lfloor D \rfloor = d\} \cong \mathrm{Pos}(d) \to \mathcal{R}$$

Proof Structural induction. At each node, $\{J : \mathcal{J} \mid \lfloor J \rfloor = P\} \cong \mathrm{Pos}'(P) \to \mathcal{R}$.

P	J	$\mathrm{Pos}'(P) \to \mathcal{R}$
$\Gamma \vdash$	$\Gamma \vdash$	$\_$
$\Gamma \vdash T \ni t$	$\Delta \vdash \rho\ T \ni t$	$x \mapsto \Delta \vert x\ ;\ \star \mapsto \rho$
$\Gamma \vdash e \in S$	$\Delta \vdash \rho\ e \in S$	$x \mapsto \Delta \vert x\ ;\ \star \mapsto \rho$

□

The derivation trees of shape d thus form an unremarkable $\mathcal{R}$-module. Let us establish something a touch more remarkable. In fact it is obvious, because when designing the system, I took care to ensure that any nonzero resource demand in the conclusion of each rule is linearly a factor of the demands in the premises.

Theorem 28 (valid derivation modules) *For any* valid *derivation tree D of shape d, the $\mathcal{R}$-module on $\{D' : \mathcal{T}(\mathcal{J}) \mid \lfloor D' \rfloor = d\}$ refines to an $\mathcal{R}$-module on $\{D' : \mathcal{T}(\mathcal{J}) \mid \lfloor D' \rfloor = d, D'$ valid$\}$.*

Proof It is necessary and sufficient to check closure under addition and scalar multiplication as the latter gives us that $0D$ is a valid zero. The proof is a straightforward induction on d, then inversion of the rules yielding the conclusion. For scalar multiplication, I give the cases for VAR, APP and CUT, as they give the pattern for the rest, showing local module calculuations by writing true equations in places where one side is given and the other is needed.

$$\phi\left(\frac{\Gamma, x : S, \Gamma' \vdash}{0\Gamma, \rho\, x : S, 0\Gamma' \vdash \rho\ x \in S}\right) = \frac{\Gamma, x : S, \Gamma' \vdash}{0\Gamma, \phi\rho\, x : S, 0\Gamma' \vdash \phi\rho\ x \in S}$$

$$\phi\left(\frac{\begin{array}{c}\Delta \vdash \rho\ f \in (\pi\, x : S) \to T \\ \Delta' \vdash \rho\pi\ S \ni s\end{array}}{\Delta + \Delta' \vdash \rho\ f\ s \in T[s : S/x]}\right) = \frac{\begin{array}{c}\phi\Delta \vdash \phi\rho\ f \in (\pi\, x : S) \to T \\ \phi\Delta' \vdash (\phi\rho)\pi\ S \ni s\end{array}}{\phi\Delta + \phi\Delta' \vdash \phi\rho\ f\ s \in T[s : S/x]}$$

$$\phi\left(\frac{\lfloor \Delta \rfloor \vdash_0 *_i \ni S \quad \Delta \vdash \rho\ S \ni s}{\Delta \vdash \rho\ s : S \in S}\right) = \frac{\lfloor \phi\Delta \rfloor \vdash_0 *_i \ni S \quad \phi\Delta \vdash \phi\rho\ S \ni s}{\phi\Delta \vdash \phi\rho\ s : S \in S}$$

For addition, I give just the APP case, which makes essential use of commutativity of the rig's addition and distributivity of multplication over addition.

$$\frac{\begin{array}{c}\Delta_0 \vdash \rho_0\ f \in (\pi\, x{:}S) \to T \\ \Delta_0' \vdash \rho_0 \pi\ S \ni s\end{array}}{\Delta_0 + \Delta_0' \vdash \rho_0\ f\ s \in T[s{:}S/x]} + \frac{\begin{array}{c}\Delta_1 \vdash \rho_1\ f \in (\pi\, x{:}S) \to T \\ \Delta_1' \vdash \rho_1 \pi\ S \ni s\end{array}}{\Delta_1 + \Delta_1' \vdash \rho_1\ f\ s \in T[s{:}S/x]}$$

$$= \frac{\begin{array}{c}\Delta_0 + \Delta_1 \vdash (\rho_0 + \rho_1)\ f \in (\pi\, x{:}S) \to T \\ \Delta_0' + \Delta_1' \vdash (\rho_0 + \rho_1) \pi\ S \ni s\end{array}}{(\Delta_0 + \Delta_1) + (\Delta_0' + \Delta_1') \vdash (\rho_0 + \rho_1)\ f\ s \in T[s{:}S/x]}$$

Hence, valid derivations form an $\mathcal{R}$-module. □

Not only can we multiply by scalars and add. We can also pull out common factors and split up our resources wherever they make multiples and sums.

Lemma 29 (factorization) *If $\Delta \vdash \phi\rho\ T \ni t$ then for some context named Δ/ϕ, $\Delta = \phi(\Delta/\phi)$ and $\Delta/\phi \vdash \rho\ T \ni t$. Similarly, if $\Delta \vdash \phi\rho\ e \in S$ then for some Δ/ϕ, $\Delta = \phi(\Delta/\phi)$ and $\Delta/\phi \vdash \rho\ e \in S$.*

Proof. Induction on derivations. The only interesting case is APP.

$$\frac{\Delta_0 \vdash \phi\rho\ f \in (\pi\, x{:}S) \to T \quad \Delta_1 \vdash (\phi\rho)\pi\ S \ni s}{\Delta_0 + \Delta_1 \vdash \phi\rho\ f\ s \in T[s{:}S/x]}$$

Inductively, $\Delta_0/\phi \vdash \rho\ f \in (\pi\, x{:}S) \to T$, and reassociating, $\Delta_1/\phi \vdash \rho\pi\ S \ni s$, so distribution gives us $\phi(\Delta_0/\phi + \Delta_1/\phi) \vdash \phi\rho\ f\ s \in T[s{:}S/x]$. □

This result does not mean $\mathcal{R}$ has multiplicative inverses, just that to make ϕ things at once, our supplies must come in multiples of ϕ, especially when $\phi = 0$.

Corollary 30 (nothing from nothing) *If $\Delta \vdash 0\ T \ni t$ then $\Delta = 0\lfloor\Delta\rfloor$.*

Proof. Lemma 29 with $\phi = \rho = 0$. □

Lemma 31 (splitting) *If $\Delta \vdash (\phi + \rho)\ T \ni t$ then for some $\Delta = \Delta' + \Delta''$, $\Delta' \vdash \phi\ T \ni t$ and $\Delta' \vdash \rho\ T \ni t$. Similarly, if $\Delta \vdash (\phi + \rho)\ e \in S$ then for some $\Delta = \Delta' + \Delta''$, $\Delta' \vdash \phi\ e \in S$ and $\Delta'' \vdash \rho\ e \in S$.*

Proof. Induction on derivations. The only interesting case is APP.

$$\frac{\Delta_0 \vdash (\phi + \rho)\ f \in (\pi\, x{:}S) \to T \quad \Delta_1 \vdash (\phi + \rho)\pi\ S \ni s}{\Delta_0 + \Delta_1 \vdash (\phi + \rho)\ f\ s \in T[s{:}S/x]}$$

Inductively, $\Delta_0' \vdash \phi\ f \in (\pi\, x : S) \to T$ and $\Delta_0'' \vdash \rho\ f \in (\pi\, x : S) \to T$, and distributing, $\Delta_1' \vdash \phi\pi\ S \ni s$ and $\Delta_1'' \vdash \rho\pi\ S \ni s$, so $\Delta_0' + \Delta_1' \vdash \phi\pi\ f\ s \in T[s{:}S/x]$ and $\Delta_0'' + \Delta_1'' \vdash \rho\pi\ f\ s \in T[s{:}S/x]$. □

9 Resourced Stability Under Substitution

Let us establish that basic thinning and substitution operations lift from syntax (terms and eliminations) to judgments (checking and synthesis). It may seem peculiar to talk of thinning in a precisely resourced setting, but as the *pre*context grows, the context will show that we have *zero* of the extra things.

Notationally, it helps to define *localization* of judgments to contexts, in that it allows us to state properties of derivations more succinctly.

Definition 32 (localization) *Define* $- \vdash - : \textsc{Cx}(\Gamma) \times \mathcal{J} \to \mathcal{J}$

$$\begin{aligned} \Delta \vdash\ \Delta' \vdash \rho\, T \ni t &= \Delta, \Delta' \vdash \rho\, T \ni t \\ \Delta \vdash\ \Delta' \vdash \rho\, e \in S &= \Delta, \Delta' \vdash \rho\, e \in S \end{aligned}$$

Strictly speaking, I should take care when localizing $\Delta \vdash \mathcal{J}$ to freshen the names in $\mathcal{J}$'s local context relative to Δ. For the sake of readability, I shall presume that accidental capture does not happen, rather than freshening explicitly.

Lemma 33 (thinning) *Admissibly,* $\dfrac{\Delta \vdash \mathcal{J}}{\Delta, 0\Gamma \vdash \mathcal{J}}$

Proof. Induction on derivations, with $\mathcal{J}$ absorbing local extensions to the context, so the inductive hypothesis applies under binders. We can thus replay the input derivation with 0Γ inserted. In the APP case, we need that $0\Gamma + 0\Gamma = 0\Gamma$. In the VAR case, inserting 0Γ preserves the applicability of the rule. □

Lemma 34 (substitution stability) *Admissibly,* $\dfrac{\Delta, \phi\, x : S \vdash \mathcal{J} \quad \Delta' \vdash \phi\, e \in S}{\Delta + \Delta' \vdash \mathcal{J}[e/x]}$

Proof. Induction on derivations, effectively substituting a suitable $\Delta', 0\Gamma \vdash \phi\ e \in S$ for every usage of the VAR rule at some $\lfloor \Delta \rfloor, \phi\, x : S, 0\Gamma \vdash \phi\ x \in S$. Most cases are purely structural, but the devil is in the detail of the resourcing, so let us take account. For PRE, LAM, ELIM and POST, the resources in Δ are undisturbed and the induction goes through directly. For SORT and FUN, $\Delta = 0\Gamma$, and Corollary 30 tells us that $\phi = 0$ and hence $\Delta' = 0\Gamma$, pushing the induction through. For VAR with variables other than x, again, $\phi = 0$ and the induction goes through. For VAR at x itself, Lemma 33 delivers the correct resource on the nose. For CUT, we may give all of the es to the term and (multiplying by 0, thanks to Theorem 28) exactly none of them to its type. In the APP case, Lemma 31 allows us to share out our es in exactly the quantities demanded in the premises. □

10 Computation Preserves Typing

The design of the system allows us to prove that computation in all parts of a judgment preserves typing: inputs never become *subjects* at any point in the derivation. While following the broad strategy of 'subject reduction' proofs, exemplified by McKinna and Pollack [19], the result comes out in one delightfully unsubtle dollop exactly because information flows uniformly through the rules.

We can lift $\twoheadrightarrow$ to contexts, simply by permitting computation in types, and we can show that any amount of computation in judgment inputs, and a parallel reduction in the subject, preserves the derivability of judgments upto computation in outputs. We should not expect computation to preserve the types we can synthesize: if you reduce a variable's type in the context, you should not expect to *synthesize* the unreduced type, but you can, of course, still *check* it.

Theorem 35 (parallel preservation)

$$\begin{array}{ccc}\Delta & T & t\\ \downarrow & \downarrow & \triangledown\\ \Delta' & T' & t'\end{array} \Rightarrow \frac{\Delta \vdash \rho\ T \ni t}{\Delta' \vdash \rho\ T' \ni t'} \qquad \begin{array}{ccc}\Delta & e & S\\ \downarrow & \triangledown & \exists\downarrow\\ \Delta' & e' & S'\end{array} \Rightarrow \exists\downarrow \wedge \frac{\Delta \vdash \rho\ e \in S}{\Delta' \vdash \rho\ e' \in S'}$$

Proof We proceed by induction on derivations and inversion of $\triangleright$. Let us work through the rules in turn.

Type Checking For PRE, we have

$$\frac{\Delta \vdash \rho\ R \ni t}{\Delta \vdash \rho\ T \ni t} \qquad \begin{array}{ccccc}\Delta & t & T & \rightsquigarrow & R\\ \downarrow & \triangledown & \downarrow & \exists & \downarrow\\ \Delta' & t' & T' & \twoheadrightarrow & R'\end{array} \qquad \frac{\therefore \Delta' \vdash \rho\ R' \ni t'}{\Delta' \vdash \rho\ T' \ni t'}$$

with the confluence of computation telling me how much computation to do to R if I want T' to check t'. For SORT, subject and checked type do not reduce, but one axiom serves as well as another.

$$\text{given } \Gamma \vdash_0 *_j \ni *_i \quad j \succ i \quad \Gamma \twoheadrightarrow \Gamma' \quad \text{deduce } \Gamma' \vdash_0 *_j \ni *_i$$

For FUN and LAM, respectively, we must have had

$$\frac{\Gamma \vdash_0 *_i \ni S \quad \Gamma, x{:}S \vdash_0 *_i \ni T}{\Gamma \vdash_0 *_i \ni (\pi\, x{:}S) \to T} \qquad \begin{array}{ccc}\Gamma & S & T\\ \downarrow & \triangledown & \triangledown\\ \Gamma' & S' & T'\end{array} \qquad \frac{\therefore \Gamma' \vdash_0 *_i \ni S' \quad \therefore \Gamma', x{:}S' \vdash_0 *_i \ni T'}{\Gamma' \vdash_0 *_i \ni (\pi\, x{:}S') \to T'}$$

$$\frac{\Delta, \rho\pi\, x{:}S \vdash \rho\ T \ni t}{\Delta \vdash \rho\ (\pi\, x{:}S) \to T \ni \lambda x.\, t} \qquad \begin{array}{cccc}\Gamma & S & T & t\\ \downarrow & \downarrow & \downarrow & \triangledown\\ \Gamma' & S' & T' & t'\end{array} \qquad \frac{\therefore \Delta', \rho\pi\, x{:}S' \vdash \rho\ T' \ni t'}{\Delta' \vdash \rho\ (\pi\, x{:}S') \to T' \ni \lambda x.\, t'}$$

For ELIM, we have two cases. For the structural case, we must compute.

$$\frac{\Delta \vdash \rho\ e \in S}{\Delta \vdash \rho\ T \ni \underline{e}}\ S \preceq T \qquad \begin{array}{ccc}\Delta & T & e\\ \downarrow & \downarrow & \triangledown\\ \Delta' & T' & e'\end{array} \qquad \begin{array}{c}S\\ \exists\downarrow\\ S'\end{array} \wedge\ \Delta' \vdash \rho\ e' \in S'$$

$$\begin{array}{ccc}S & \preceq & T\\ \downarrow & & \downarrow\\ S' & & T'\\ \exists\downarrow & & \downarrow\\ S'' & \preceq & T''\end{array} \qquad \begin{array}{l}\dfrac{\dfrac{\Delta' \vdash \rho\ e' \in S'}{\Delta' \vdash \rho\ e' \in S''}\ S' \twoheadrightarrow S''}{\Delta' \vdash \rho\ T'' \ni \underline{e'}}\ S'' \preceq T''\\ \hline \Delta' \vdash \rho\ T' \ni \underline{e'}\end{array} T' \twoheadrightarrow T''$$

Lemma 20 reestablishes subtyping after computation.

For υ-contraction, we have a little more entertainment. We start with

$$\frac{\Delta \vdash \rho\, S \ni s \quad \cdots \quad \Delta \vdash \rho\, s : S \in S'}{\Delta \vdash \rho\, T \ni \underline{s : S}} \qquad \begin{array}{cccc} \Delta & s & S & \preceq & T \\ \downarrow & \triangledown & \downarrow & & \downarrow \\ \Delta' & s' & S' & & T' \\ & & \exists \downarrow & & \downarrow \\ & & S'' & \preceq & T'' \end{array} \quad \therefore\ \Delta' \vdash \rho\, S'' \ni s'$$

Then by Theorem 22 (subsumption), we obtain $\dfrac{\dfrac{\Delta' \vdash \rho\, S'' \ni s'}{\Delta' \vdash \rho\, T'' \ni s'}\ S'' \preceq T''}{\Delta' \vdash \rho\, T' \ni s'}\ T' \twoheadrightarrow T''$

Type Synthesis For POST, we have

$$\frac{\Delta \vdash \rho\, e \in S}{\Delta \vdash \rho\, e \in R} \qquad \begin{array}{ccccc} \Delta & e & S & \rightsquigarrow & R \\ \downarrow & \triangledown & \exists \downarrow & & \exists \downarrow \\ \Delta' & e' & S' & \twoheadrightarrow & R' \end{array} \qquad \frac{\therefore \Delta' \vdash \rho\, e' \in S'}{\Delta' \vdash \rho\, e' \in R'}$$

For VAR, again, one axiom is as good as another

$$\overline{0\Gamma_0, \rho\, x : S, 0\Gamma_1 \vdash \rho\, x \in S} \qquad \begin{array}{ccccc} \Gamma_0 & S & \Gamma_1 \\ \downarrow & \downarrow & \downarrow \\ \Gamma_0' & S' & \Gamma_1' \end{array} \qquad \overline{0\Gamma_0', \rho\, x : S', 0\Gamma_1' \vdash \rho\, x \in S'}$$

The case of CUT is just structural.

$$\frac{\lfloor \Delta \rfloor \vdash_0 *_i \ni S \quad \Delta \vdash \rho\, S \ni s}{\Delta \vdash \rho\, s : S \in S} \qquad \begin{array}{ccc} \Delta & S & s \\ \downarrow & \triangledown & \triangledown \\ \Delta' & S' & s' \end{array} \qquad \frac{\therefore \lfloor \Delta' \rfloor \vdash_0 *_i \ni S' \quad \therefore \Delta' \vdash \rho\, S' \ni s'}{\Delta' \vdash \rho\, s' : S' \in S'}$$

For APP, we have two cases. In the structural case, we begin with

$$\frac{\Delta_0 \vdash \rho\, f \in (\pi\, x : S) \to T \quad \Delta_1 \vdash \rho\pi\ S \ni s}{\Delta_0 + \Delta_1 \vdash \rho\, f\, s \in T[s : S/x]} \qquad \begin{array}{cccc} \Delta_0 + \Delta_1 & f & s \\ \downarrow & \downarrow & \triangledown & \triangledown \\ \Delta_0' + \Delta_1' & f' & s' \end{array}$$

and we should note that the computed context $\Delta_0' + \Delta_1'$ continue to share a common (but more computed) precontext. The inductive hypothesis for the function tells us the type at which to apply the inductive hypothesis for the argument. We obtain $\begin{array}{cc} S & T \\ \exists\downarrow & \exists\downarrow \\ S' & T' \end{array}$ $\dfrac{\therefore \Delta_0' \vdash \rho\, f \in (\pi\, x : S') \to T' \quad \therefore \Delta_1' \vdash \rho\pi\ S' \ni s}{\Delta_0' + \Delta_1' \vdash \rho\, f'\, s' \in T'[s' : S'/x]}$

where Lemma 14 tells us that $T[s : S/x] \twoheadrightarrow T'[s' : S'/x]$. This leaves only the case where application performs β-reduction, the villain of the piece. We have

$$\frac{\begin{array}{l}\Delta_0 \vdash \rho\, (\lambda x.\, t : (\pi\, x : S_0) \to T_0) \in (\pi\, x : S_1) \to T_1 \\ \Delta_1 \vdash \rho\pi\ S_1 \ni s\end{array}}{\Delta_0 + \Delta_1 \vdash \rho\, (\lambda x.\, t : (\pi\, x : S_0) \to T_0)\, s \in T_1[s : S_1/x]} \qquad \begin{array}{ccccccccc} \Delta_0 + \Delta_1 & t & S_0 & T_0 & s & S_0 & T_0 \\ \downarrow & \downarrow & \triangledown & \triangledown & \triangledown & \triangledown & \downarrow & \downarrow \\ \Delta_0' + \Delta_1' & t' & S_0' & T_0' & s' & S_1 & T_1 \end{array}$$

noting that POST might mean we apply at a function type computed from that given. Let us first interrogate the type checking of the function. There will have been some FUN, and after PRE computing $S_0 \twoheadrightarrow S_2$ and $T_0 \twoheadrightarrow T_2$, some LAM:

$$\frac{\lfloor \Delta_0 \rfloor \vdash_0 *_i \ni S_0 \quad \lfloor \Delta_0 \rfloor, x : S_0 \vdash_0 *_i \ni T_0}{\lfloor \Delta_0 \rfloor \vdash_0 *_i \ni (\pi\, x : S_0) \to T_0} \qquad \frac{\Delta_0, \rho\pi\, x : S_2 \vdash \rho\, T_2 \ni t}{\Delta_0 \vdash \rho\, (\pi\, x : S_2) \to T_2 \ni \lambda x.\, t}$$

We compute a common reduct $S_0 \twoheadrightarrow \{S_0', S_1, S_2\} \twoheadrightarrow S_1'$, and deduce inductively

$$\begin{array}{ll} \therefore \lfloor \Delta_0' \rfloor, x : S_1' \vdash_0 *_i \ni T_0' & \therefore \lfloor \Delta_1' \rfloor \vdash_0 *_i \ni S_0' \\ \therefore \Delta_0', \rho\pi\, x : S_1' \vdash \rho\, T_0' \ni t' & \therefore \Delta_1' \vdash \rho\pi\ S_0' \ni s' \end{array}$$

so that CUT and POST give us $\Delta_1' \vdash \rho\pi\ s' : S_0' \in S_1'$. Now, Lemma 34 (stability under substitution) and CUT give us

$$\frac{\lfloor \Delta_0' + \Delta_1' \rfloor \vdash_0 *_i \ni T_0'[s' : S_0'/x] \quad \Delta_0' + \Delta_1' \vdash \rho\, T_0'[s' : S_0'/x] \ni t'[s' : S_0'/x]}{\Delta_0' + \Delta_1' \vdash \rho\, (t' : T_0')[s' : S_0'/x] \in T_0'[s' : S_0'/x]}$$

so POST can compute our target type to a common reduct $T_0 \twoheadrightarrow \{T_0', T_1, T_2\} \twoheadrightarrow T_1'$, and deliver $\Delta_0' + \Delta_1' \vdash \rho\, (t' : T_0')[s' : S_0'/x] \in T_1'[s' : S_1'/x]$. □

Corollary 36 (preservation)

$$\begin{array}{lll} \Delta & T & t \\ \downdownarrows & \downdownarrows & \downdownarrows \\ \Delta' & T' & t' \end{array} \Rightarrow \frac{\Delta \vdash \rho\, T \ni t}{\Delta' \vdash \rho\, T' \ni t'} \qquad \begin{array}{ll} \Delta & e \\ \downdownarrows & \downdownarrows \\ \Delta' & e' \end{array} \Rightarrow \begin{array}{l} S \\ \exists \downdownarrows \wedge \\ S' \end{array} \frac{\Delta \vdash \rho\, e \in S}{\Delta' \vdash \rho\, e' \in S'}$$

Proof. Iteration of Theorem 35. □

11 Erasure to an Implicit Calculus

Runtime programs live in good old lambda calculus and teletype font, to boot.

Definition 37 (programs) $\mathtt{p} ::= x \mid \backslash x \texttt{->} \mathtt{p} \mid \mathtt{p}\ \mathtt{p}$

I introduce two new judgment forms for programs, which arise as the erasure of our existing fully explicit terms.

Definition 38 (erasure judgments) *When $\rho \neq 0$, we may form judgments as follows:* $\Delta \vdash \rho\, T \ni t \blacktriangleright \mathtt{p}$ $\Delta \vdash \rho\, e \in S \blacktriangleright \mathtt{p}$.

That is, programs are nonzero-resourced. Such judgments are derived by an elaborated version of the existing rules which add programs as outputs. For checking, we must omit the type formation rules, but we obtain implicit and explicit forms of abstraction. For synthesis, we obtain implicit and explicit forms of application. In order to ensure that contemplation never involves consumption, we must impose a condition on the rig $\mathcal{R}$ that not only is the presence of negation unnecessary, but also its absence is vital: $\dfrac{\rho + \pi = 0}{\rho = \pi = 0}$.

Definition 39 (checking and synthesis with erasure)

$$\textsc{pre}+ \quad \frac{\Delta \vdash \rho\ R \ni t \ \blacktriangleright \mathtt{p}}{\Delta \vdash \rho\ T \ni t \ \blacktriangleright \mathtt{p}}\ T \rightsquigarrow R$$

$$\textsc{lam}0 \quad \frac{\Delta, 0\,x{:}S \vdash \rho\ T \ni t \ \blacktriangleright \mathtt{p}}{\Delta \vdash \rho\ (\phi\,x{:}S) \to T \ni \lambda x.\,t \ \blacktriangleright \mathtt{p}}\ \rho\phi = 0$$

$$\textsc{lam}+ \quad \frac{\Delta, \rho\pi\,x{:}S \vdash \rho\ T \ni t \ \blacktriangleright \mathtt{p}}{\Delta \vdash \rho\ (\pi\,x{:}S) \to T \ni \lambda x.\,t \ \blacktriangleright \ \backslash x \texttt{->}\ \mathtt{p}}\ \rho\pi \neq 0$$

$$\textsc{elim}+ \quad \frac{\Delta \vdash \rho\ e \in S \ \blacktriangleright \mathtt{p}}{\Delta \vdash \rho\ T \ni \underline{e} \ \blacktriangleright \mathtt{p}}\ S \preceq T$$

$$\textsc{post}+ \quad \frac{\Delta \vdash \rho\ e \in S \ \blacktriangleright \mathtt{p}}{\Delta \vdash \rho\ e \in R \ \blacktriangleright \mathtt{p}}\ S \rightsquigarrow R$$

$$\textsc{var}+ \quad \frac{}{0\Gamma, \rho\,x{:}S, 0\Gamma' \vdash \rho\ x \in S \ \blacktriangleright x}$$

$$\textsc{app}0 \quad \frac{\Delta \vdash \rho\ f \in (\phi\,x{:}S) \to T \ \blacktriangleright \mathtt{p} \quad \lfloor\Delta\rfloor \vdash_0 S \ni s}{\Delta \vdash \rho\ f\ s \in T[s{:}S/x] \ \blacktriangleright \mathtt{p}}\ \rho\phi = 0$$

$$\textsc{app}+ \quad \frac{\Delta \vdash \rho\ f \in (\pi\,x{:}S) \to T \ \blacktriangleright \mathtt{p} \quad \Delta' \vdash \rho\pi\ S \ni s \ \blacktriangleright \mathtt{p}'}{\Delta + \Delta' \vdash \rho\ f\ s \in T[s{:}S/x] \ \blacktriangleright \mathtt{p}\,\mathtt{p}'}\ \rho\pi \neq 0$$

$$\textsc{cut}+ \quad \frac{\lfloor\Delta\rfloor \vdash_0 *_i \ni S \quad \Delta \vdash \rho\ S \ni s \ \blacktriangleright \mathtt{p}}{\Delta \vdash \rho\ s{:}S \in S \ \blacktriangleright \mathtt{p}}$$

We can be sure that in the LAM0 rule, the variable x bound in t occurs nowhere in the corresponding $\mathtt{p}$, because it is bound with resource 0, and it will remain with resource 0 however the context splits, so the rule VAR+ cannot *consume* it, even though VAR can still contemplate it . Accordingly, no variable escapes its scope. We obtain without difficulty that erasure can be performed.

Lemma 40 (erasures exist uniquely and elaborate) *If $\rho \neq 0$, then*

$$\frac{\Delta \vdash \rho\ T \ni t}{\exists!\mathtt{p}.\ \Delta \vdash \rho\ T \ni t \ \blacktriangleright \mathtt{p}} \qquad \frac{\Delta \vdash \rho\ T \ni t \ \blacktriangleright \mathtt{p}}{\Delta \vdash \rho\ T \ni t}$$

$$\frac{\Delta \vdash \rho\ e \in S}{\exists!\mathtt{p}.\ \Delta \vdash \rho\ e \in S \ \blacktriangleright \mathtt{p}} \qquad \frac{\Delta \vdash \rho\ e \in S \ \blacktriangleright \mathtt{p}}{\Delta \vdash \rho\ e \in S}$$

Proof. Induction on derivations. □

The unerased forms may thus be used to form types, as Theorem 28 then gives us that $\dfrac{\Delta \vdash \rho\ T \ni t \ \blacktriangleright \mathtt{p}}{\lfloor\Delta\rfloor \vdash_0 T \ni t} \qquad \dfrac{\Delta \vdash \rho\ e \in S \ \blacktriangleright \mathtt{p}}{\lfloor\Delta\rfloor \vdash_0 e \in S}$.

How do programs behave? They may compute by β reduction, liberally.

Definition 41 (program computation)

$$\frac{}{(\ \backslash x \texttt{->}\ \mathtt{p})\ \mathtt{p}' \ \rightsquigarrow\ \mathtt{p}[\mathtt{p}'/x]} \qquad \frac{\mathtt{p} \ \rightsquigarrow\ \mathtt{p}'}{\backslash x \texttt{->}\ \mathtt{p} \ \rightsquigarrow\ \backslash x \texttt{->}\ \mathtt{p}'} \qquad \frac{\mathtt{p} \ \rightsquigarrow\ \mathtt{p}'}{\mathtt{p}\,\mathtt{p}_a \ \rightsquigarrow\ \mathtt{p}'\,\mathtt{p}_a} \qquad \frac{\mathtt{p} \ \rightsquigarrow\ \mathtt{p}'}{\mathtt{p}_f\ \mathtt{p} \ \rightsquigarrow\ \mathtt{p}_f\ \mathtt{p}'}$$

The key to understanding the good behaviour of computation is to observe that any term which erases to some $\backslash x \texttt{->} \texttt{p}$ must contain a subterm on its left spine typed with the LAM+ rule. On the way to that subterm, appeals to APP0 will be bracketed by appeals to LAM0, ensuring that we can dig out the non-zero λ by computation. Let us now show that we can find the LAM+.

Definition 42 (n-to-ρ-function type) *Inductively, $(\pi x : S) \to T$ is a 0-to-ρ-function type if $\rho\pi \neq 0$; $(\phi x : S) \to T$ is an $n+1$-to-ρ-function type if $\rho\phi = 0$ and T is an n-to-ρ-function type.*

Note that n-to-ρ-function types are stable under substitution and subtyping. Let λ^n denote λ-abstraction iterated n times.

Lemma 43 (applicability) *If $\Delta \vdash \rho\ T \ni t \blacktriangleright \backslash x \texttt{->} \texttt{p}$, then $T \twoheadrightarrow$ some n-to-ρ-function type T' and $t \twoheadrightarrow$ some $\lambda^n \boldsymbol{y}.\, \lambda x.\, t'$ such that*

$$\Delta \vdash \rho\ T' \ni \lambda^n \boldsymbol{y}.\, \lambda x.\, t' \blacktriangleright \backslash x \texttt{->} \texttt{p}$$

If $\Delta \vdash \rho\ e \in S \blacktriangleright \backslash x \texttt{->} \texttt{p}$, then $S \twoheadrightarrow$ some n-to-ρ-function type S' and $e \twoheadrightarrow$ some $\lambda^n \boldsymbol{y}.\, \lambda x.\, t' : S'$ such that

$$\Delta \vdash \rho\ \lambda^n \boldsymbol{y}.\, \lambda x.\, t' : S' \in S' \blacktriangleright \backslash x \texttt{->} \texttt{p}$$

Proof. Proceed by induction on derivations. Rules VAR+ and APP+ are excluded by the requirement to erase to $\backslash x\texttt{->}\texttt{p}$. For PRE+, the inductive hypothesis applies and suffices. For LAM0, the inductive hypothesis tells us how to compute t and T to an abstraction in an n-to-ρ-function type, and we glue on one more $\lambda y.\, -$ and one more $(\phi\, y : S) \to -$, respectively. At LAM+, we can stop. For POST+, the inductive hypothesis gives us a cut at type S', where $S \twoheadrightarrow S'$, so we can take the common reduct $S \twoheadrightarrow \{S', R\} \twoheadrightarrow R'$ and deliver the same cut at type R'. For CUT+, we again proceed structurally. This leaves only the entertainment.

For ELIM+, we had $e \in S$ with $S \preceq T$. Inductively, $e \twoheadrightarrow \lambda \boldsymbol{y}.\, \lambda x.\, t' : S'$ with $S \twoheadrightarrow S'$, some n-to-ρ-function type. Lemma 20 gives us that $T \twoheadrightarrow T'$ with $S' \preceq T'$, also an n-to-ρ-function type. Hence $T' \ni \overline{\lambda \boldsymbol{y}.\, \lambda x.\, t' : S'}$ and then Theorem 35 (preservation) allows the υ-reduction to $T' \ni \overline{\lambda \boldsymbol{y}.\, \lambda x.\, t'}$.

For APP0, the inductive hypothesis gives us $f \twoheadrightarrow \lambda y.\, \lambda^n \boldsymbol{y}.\, \lambda x.\, t' : (\phi\, x : S') \to T'$ with $S \twoheadrightarrow S'$ and $T \twoheadrightarrow T'$, and T' an n-to-ρ-function type. Hence $f\ s \twoheadrightarrow (\lambda^n \boldsymbol{y}.\, \lambda x.\, t' : T')[s : S'/x]$. Preservation tells us the reduct is well typed at some other reduct of $T[s : S/x]$, but the common reduct is the type we need. □

Theorem 44 (step simulation) *The following implications hold.*

$$\frac{\Delta \vdash \rho\ T \ni t \blacktriangleright \texttt{p}}{\exists t'.\, t \twoheadrightarrow t' \wedge\ \Delta \vdash \rho\ T \ni t' \blacktriangleright \texttt{p}'}\ \texttt{p} \rightsquigarrow \texttt{p}'$$

$$\frac{\Delta \vdash \rho\ e \in S \blacktriangleright \texttt{p}}{\exists e', S'.\, e \twoheadrightarrow e' \wedge S \twoheadrightarrow S' \wedge\ \Delta \vdash \rho\ e' \in S' \blacktriangleright \texttt{p}'}\ \texttt{p} \rightsquigarrow \texttt{p}'$$

Proof. Induction on derivations and inversion of program computation. The only interesting case is the APP+ case when the computation takes a β-step.

$$\frac{\Delta \vdash \rho\ f \in (\pi\, x : S) \to T \ \blacktriangleright\ \backslash x \texttt{ -> } \mathtt{p} \quad \Delta' \vdash \rho\pi\ S \ni s \ \blacktriangleright\ \mathtt{p}'}{\Delta + \Delta' \vdash \rho\ f\ s \in T[s:S/x] \ \blacktriangleright\ (\backslash x \texttt{ -> } \mathtt{p})\ \mathtt{p}' \rightsquigarrow \mathtt{p}[\mathtt{p}'/x]}\ \rho\pi \neq 0$$

We have some f whose type is a 0-to-ρ function type and which erases to some $\backslash x$ -> p, so we may invoke Lemma 43 to get that $f \twoheadrightarrow \lambda x.\, t$ at some reduced function type, $(\pi\, x : S') \to T'$, where S' still accepts the argument s, by preservation, erasing to $\mathtt{p}'$. Accordingly, $f\ s \twoheadrightarrow (t : T')[s : S'/x]$, where the latter is still typed at a reduct of $T[s:S/x]$ and erases to $\mathtt{p}[\mathtt{p}'/x]$. □

Accordingly, once we know terms are well typed, contemplation has done its job and we may erase to virtuous and thrifty 0-free programs.

12 Take It Or Leave It

Let us consider liberalising our rig-based resource managemen. At present, the $\{0, 1, \omega\}$ rig imposes a kind of *relevance*, but not traditional relevance, in that it takes at least two uses at multiplicity 1 or one at ω to discharge our spending needs: what if we want traditional relevance, or even the traditional intuitionistic behaviour? Similarly, we might sometimes want to weaken the linear discipline to affine typing, where data can be dropped but not duplicated.

One option is to impose an 'order', $\leq$ on the rig. We can extend it pointwise to Γ-contexts, so $\Delta \leq \Delta'$ if Δ' has at least as many of each variable in Γ as Δ.

The $\leq$ relation should be reflexive and transitive, and at any rate we shall need at least that the order respects the rig operations

$$\frac{}{\rho \leq \rho} \qquad \frac{\rho \leq \pi \quad \pi \leq \phi}{\rho \leq \phi} \qquad \frac{\pi \leq \phi}{\rho + \pi \leq \rho + \phi} \qquad \frac{\pi \leq \phi}{\rho\pi \leq \rho\phi} \qquad \frac{\pi \leq \phi}{\pi\rho \leq \phi\rho}$$

to ensure that valid judgments remain an $\mathcal{R}$-module when we add weakening:

$$\textsc{weak} \quad \frac{\Delta \vdash \rho\ T \ni t}{\Delta' \vdash \rho\ T \ni t}\ \Delta \leq \Delta'$$

To retain Lemmas 29 and 31 (factorization and splitting), we must also be able to factorize and split the ordering, so two more conditions on $\leq$ emerge: factorization and additive splitting.

$$\frac{\rho\pi \leq \rho\phi}{\pi \leq \phi} \qquad \frac{\phi + \rho \leq \pi}{\exists \phi', \rho'.\ \phi \leq \phi' \wedge \rho \leq \rho' \wedge \pi = \phi' + \rho'}$$

Stability under substitution requires no more conditions but a little more work. The following lemma is required to deliver the new case for WEAK.

Lemma 45 (weakening) *If $\rho \leq \rho'$ then*

$$\frac{\Delta' \vdash \rho'\pi\ T \ni t}{\exists \Delta.\ \Delta \leq \Delta' \wedge\ \Delta \vdash \rho\pi\ T \ni t} \qquad \frac{\Delta' \vdash \rho'\pi\ e \in S}{\exists \Delta.\ \Delta \leq \Delta' \wedge\ \Delta \vdash \rho\pi\ e \in S}$$

Proof. Induction on derivations, with the interesting cases being VAR, WEAK and APP: the rest go through directly by inductive hypothesis and replay of the rule. For VAR, form Δ by replacing the $\rho'\pi$ in Δ' by $\rho\pi$, which is smaller.

For WEAK, we must have delivered $\Delta' \vdash \rho'\pi T \ni t$ from some Δ' with $\Delta' \leq \Delta''$ and $\Delta' \vdash \rho'\pi\ T \ni t$. By the inductive hypothesis, there is some $\Delta \leq \Delta'$ with $\Delta \vdash \rho\pi\ T \ni t$ and transitivity gives us $\Delta \leq \Delta''$.

For APP, the fact that $\leq$ respects multiplication allows us to invoke the inductive hypothesis for the argument, and then we combine the smaller contexts delivered by the inductive hypotheses to get a smaller sum. □

As a consequence, we retain stability of typing under substitution and thence type preservation. To retain safe erasure, we must prevent WEAK from bringing a contemplated variable back into a usable position by demanding $\dfrac{\rho \leq 0}{\rho = 0}$.

If we keep $\leq$ discrete, nobody will notice the difference with the rigid system. However, we may now add, for example, $1 \leq \omega$ to make ω capture run-time relevance, or $0 \leq 1 \leq \omega$ for affine typing, or $0, 1 \leq \omega$ (but not $0 \leq 1$) to make $(\omega\, x : S) \to T$ behave like an ordinary intuitionistic function type whilst keeping $(1\, x : S) \to T$ linear: you can throw only plenty away.

13 Contemplation

Our colleagues who have insisted that dependent types should depend only on replicable things have been right all along. The $\mathcal{R}$-module structure of derivations ensures that every construction fit for consumption has an intuitionistic counterpart fit for contemplation, replicable because it is made from nothing.

In a dependent type theory, the variables in types stand for things: which things? It is not always obvious, because types may be used to classify more than one kind of thing. The things that variables in types stand for are special: when substituted for variables, they appear in types and are thus contemplated. A fully dependent type theory demands that everything classified by type has a contemplatable image, not that everything is contemplatable.

Here, we use the same types to classify terms and eliminations in whatever quantity, and also to classify untyped programs after erasure, but it is the *eliminations of quantity zero* which get contemplated when the application rule substitutes them for a variable, and everything classified by a type in one way or another corresponds to just such a thing.

Such considerations should warn us to be wary of jumping to conclusions, however enthusiastic we may feel. We learned from Kohei Honda that session types are linear types, in a way which has been made precise intuitionistically by Caires and Pfenning [7], and classically by Wadler [31], but we should not expect a linear dependent type theory, to be a theory of dependent session types *per se*. The linear dependent type theory in this paper is *not* a theory of session types because contemplated terms give too much information: they represent the *participants* which input and output values according to the linear types.

Dependency in protocol types should concern only the *signals* exchanged by the participants, not the participants' private strategies for generating those signals. In fact, the *signal traffic*, the *participants* and the *channels* are all sorts of things classified by session types, but it is only the signal traffic which must inform dependency. That is another story, and I will tell it another time, but it is based on the same analysis of how dependent type theories work. It seems realistic to pursue programming in the style of Gay and Vasconcelos [13] with dependent session types and linearly managed channels.

What we do have is a basis for a propositions-as-types account of certifying linearly typed programs, where the idealised behaviour of the program can be contemplated in propositions. When proving theorems about functions with unit-priced types, we know to expect uniformity properties when the price is zero: from parametricity, we obtain 'theorems for free' [23, 29]. What might we learn when the price is small but not zero? Can we learn that a function from lists to lists which is parametric in the element type and linear in its input necessarily delivers a permutation? If so, the mission to internalise such results in type theory, championed by Bernardy, Jansson and Paterson [5] is still more crucial.

Looking nearer, I have mentioned the desirability of a normalization proof, and its orthogonality to resourcing. We must also look beyond $\multimap$ and give dependent accounts of other linear connectives: dependent $\otimes$ clearly makes sense with a pattern matching eliminator; dependent $(x : S) \& T[x]$ offers the intriguing choice to select an S or a $T[x]$ whose type mentions what the S used to be, like money or goods to the value of the money. But what are the duals?

It would be good to study datatypes supporting mutation. We have the intriguing prospect of linear induction principles like

$$\begin{array}{l}\forall X : *.\, \forall P : \mathsf{List}\ X \to *. \\ (n : P\ []) \multimap (c : (x : X) \multimap (xsp : (xs : \mathsf{List}\ X) \& P\ xs) \multimap P\ (x :: \mathsf{fst}\ xsp)) \to \\ (xs : \mathsf{List}\ X) \multimap P\ xs\end{array}$$

which allow us at each step in the list either to retain the tail or to construct a P from it, but not both. Many common programs exhibit this behaviour (insertion sort springs to mind) and they seem to fit the heuristic identified by Domínguez and Pardo for when the fusion of paramorphisms an optimisation [11].

What we can now bring to all these possibilities is the separation of contemplation from consumption, ensuring that contemplation requires no resource and can correspondingly be erased. More valuable, perhaps, than this particular technical answer to the challenge of fully integrating linear and dependent types is the learning of the question 'What are the contemplated images?'.

Acknowledgements

Writing was a substitute for sleep in a busy summer of talks in the Netherlands, Germany and England. I thank everyone who interacted with me in the daytimes at least for their forbearance, and some for very useful conversations and feedback, notably James McKinna and Shayan Najd. The ideas were incubated

during my visit to Microsoft Research Cambridge in 2014: I'm grateful to Nick Benton for feedback and encouragement. My talk on this topic at SREPLS in Cambridge provoked a very helpful interaction with Andy Pitts, Dominic Orchard, Tomas Petricek, and Stephen Dolan—his semiring-sensitivity provoked the generality of the resource treatment here [10]. I apologize to and thank the referees of all versions of this paper. Sam Lindley and Craig McLaughlin also deserve credit for helping me get my act together.

The inspiration, however, comes from Phil. The clarity of the connection he draws between classical linear logic and session types [31] is what attracted me to this problem. One day he, Simon Gay and I bashed rules at my whiteboard, trying to figure out a dependent version of that 'Propositions As Sessions' story. The need to distinguish 'contemplation' from 'consumption' emerged that afternoon: it has not yet delivered a full higher-order theory of 'dependent session types', but in my mind, it showed me the penny I had to make drop.

References

1. Andreas Abel. Normalization by Evaluation: Dependent Types and Impredicativity, 2013. Habilitationsschrift.
2. Andreas Abel, Thierry Coquand, and Peter Dybjer. Normalization by Evaluation for Martin-Löf Type Theory with Typed Equality Judgements. In *22nd IEEE Symposium on Logic in Computer Science (LICS 2007), 10-12 July 2007, Wroclaw, Poland, Proceedings*, pages 3–12. IEEE Computer Society, 2007.
3. Robin Adams. Pure type systems with judgemental equality. *J. Funct. Program.*, 16(2):219–246, 2006.
4. Andrea Asperti, Wilmer Ricciotti, Claudio Sacerdoti Coen, and Enrico Tassi. A Bi-Directional Refinement Algorithm for the Calculus of (Co)Inductive Constructions. *Logical Methods in Computer Science*, 8(1), 2012.
5. Jean-Philippe Bernardy, Patrik Jansson, and Ross Paterson. Proofs for free - parametricity for dependent types. *J. Funct. Program.*, 22(2):107–152, 2012.
6. Edwin Brady. Idris, a general-purpose dependently typed programming language: Design and implementation. *J. Funct. Program.*, 23(5):552–593, 2013.
7. Luís Caires and Frank Pfenning. Session types as intuitionistic linear propositions. In Paul Gastin and François Laroussinie, editors, *CONCUR 2010 - Concurrency Theory, 21th International Conference, CONCUR 2010, Paris, France, August 31-September 3, 2010. Proceedings*, volume 6269 of *Lecture Notes in Computer Science*, pages 222–236. Springer, 2010.
8. Iliano Cervesato and Frank Pfenning. A Linear Logical Framework. *Inf. Comput.*, 179(1):19–75, 2002.
9. Edsko de Vries, Rinus Plasmeijer, and David M. Abrahamson. Uniqueness Typing Simplified. In Olaf Chitil, Zoltán Horváth, and Viktória Zsók, editors, *Implementation and Application of Functional Languages, 19th International Workshop, IFL 2007, Freiburg, Germany, September 27-29, 2007. Revised Selected Papers*, volume 5083 of *Lecture Notes in Computer Science*, pages 201–218. Springer, 2007.
10. Stephen Dolan. Fun with semirings: a functional pearl on the abuse of linear algebra. In Greg Morrisett and Tarmo Uustalu, editors, *ACM SIGPLAN International Conference on Functional Programming, ICFP'13, Boston, MA, USA - September 25 - 27, 2013*, pages 101–110. ACM, 2013.

11. Facundo Domínguez and Alberto Pardo. Program fusion with paramorphisms. In *Proceedings of the 2006 International Conference on Mathematically Structured Functional Programming*, MSFP'06, pages 6–6, Swinton, UK, UK, 2006. British Computer Society.
12. Marco Gaboardi, Andreas Haeberlen, Justin Hsu, Arjun Narayan, and Benjamin C. Pierce. Linear dependent types for differential privacy. In Roberto Giacobazzi and Radhia Cousot, editors, *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, Rome, Italy - January 23 - 25, 2013*, pages 357–370. ACM, 2013.
13. Simon J. Gay and Vasco Thudichum Vasconcelos. Linear type theory for asynchronous session types. *J. Funct. Program.*, 20(1):19–50, 2010.
14. Gerhard Gentzen. Untersuchungen über das logische schließen. *Mathematische Zeitschrift*, 29(2-3):176–210, 405–431, 1935.
15. Jean-Yves Girard. Linear logic. *Theor. Comput. Sci.*, 50:1–102, 1987.
16. Neelakantan R. Krishnaswami, Pierre Pradic, and Nick Benton. Integrating Linear and Dependent Types. In Sriram K. Rajamani and David Walker, editors, *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*, pages 17–30. ACM, 2015.
17. Zhaohui Luo. ECC, an Extended Calculus of Constructions. In *Proceedings of the Fourth Annual Symposium on Logic in Computer Science (LICS '89), Pacific Grove, California, USA, June 5-8, 1989*, pages 386–395. IEEE Computer Society, 1989.
18. Per Martin-Löf. An Intuitionistic Theory of Types: Predicative Part. In H. E. Rose and J. C. Shepherdson, editors, *Logic Colloquium 1973*, Amsterdam, 1975. North-Holland Publishing Company.
19. James McKinna and Robert Pollack. Some lambda calculus and type theory formalized. *J. Autom. Reasoning*, 23(3-4):373–409, 1999.
20. Alexandre Miquel. The implicit calculus of constructions. In *TLCA*, pages 344–359, 2001.
21. Tomas Petricek, Dominic A. Orchard, and Alan Mycroft. Coeffects: a calculus of context-dependent computation. In Johan Jeuring and Manuel M. T. Chakravarty, editors, *Proceedings of the 19th ACM SIGPLAN international conference on Functional programming, Gothenburg, Sweden, September 1-3, 2014*, pages 123–135. ACM, 2014.
22. Benjamin C. Pierce and David N. Turner. Local type inference. *ACM Trans. Program. Lang. Syst.*, 22(1):1–44, 2000.
23. John C. Reynolds. Types, abstraction and parametric polymorphism. In *IFIP Congress*, pages 513–523, 1983.
24. Rui Shi and Hongwei Xi. A linear type system for multicore programming in ATS. *Sci. Comput. Program.*, 78(8):1176–1192, 2013.
25. Nikhil Swamy, Juan Chen, Cédric Fournet, Pierre-Yves Strub, Karthikeyan Bhargavan, and Jean Yang. Secure distributed programming with value-dependent types. *J. Funct. Program.*, 23(4):402–451, 2013.
26. Masako Takahashi. Parallel Reductions in λ-Calculus (revised version). *Information and Computation*, 118(1):120–127, 1995.
27. Bernardo Toninho, Luís Caires, and Frank Pfenning. Dependent session types via intuitionistic linear type theory. In Peter Schneider-Kamp and Michael Hanus, editors, *Proceedings of the 13th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, July 20-22, 2011, Odense, Denmark*, pages 161–172. ACM, 2011.

28. Matthijs Vákár. Syntax and Semantics of Linear Dependent Types. *CoRR*, abs/1405.0033, 2014.
29. Philip Wadler. Theorems for free! In *FPCA*, pages 347–359, 1989.
30. Philip Wadler. Is There a Use for Linear Logic? In Charles Consel and Olivier Danvy, editors, *Proceedings of the Symposium on Partial Evaluation and Semantics-Based Program Manipulation, PEPM'91, Yale University, New Haven, Connecticut, USA, June 17-19, 1991*, pages 255–273. ACM, 1991.
31. Philip Wadler. Propositions as sessions. In Peter Thiemann and Robby Bruce Findler, editors, *ACM SIGPLAN International Conference on Functional Programming, ICFP'12, Copenhagen, Denmark, September 9-15, 2012*, pages 273–286. ACM, 2012.
32. Kevin Watkins, Iliano Cervesato, Frank Pfenning, and David Walker. A concurrent logical framework: The propositional fragment. In Stefano Berardi, Mario Coppo, and Ferruccio Damiani, editors, *Types for Proofs and Programs, International Workshop, TYPES 2003, Torino, Italy, April 30 - May 4, 2003, Revised Selected Papers*, volume 3085 of *Lecture Notes in Computer Science*, pages 355–377. Springer, 2003.