I Got Plenty o' Nuttin'

Witheld

Witheld

witheld

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. 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 can have plenty of nothing with no additional runtime resource, and nothing is plenty for the construction of dependent types.

Categories and Subject Descriptors D.3.2 [*Programming Languages*]: Formal Definitions and Theory

Keywords linear types, dependent types

1. Introduction

Girard's linear logic [14] gives a discipline for parsimonious control of resources, and Martin-Löf's type theory [17] 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 [25]. Shi and Xi have linear typing in ATS [22]. Swamy and colleagues have linear typing in $F^\star$ [23]. Brady's Idris [6] has uniqueness typing, after the fashion of Clean [9]. Vákár has given a categorical semantics [26] where dependency is permitted on the cartesian sublanguage. Gaboardi and colleagues use linear types to study differential privacy [11]. Krishnaswami, Pradic and Benton [15] 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.

CONF 'yy, Month d–d, 20yy, City, ST, Country.
Copyright © 20yy ACM 978-1-nnnn-nnnn-n/yy/mm. . . $15.00.
http://dx.doi.org/10.1145/nnnnnnn.nnnnnnn

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 friends illustrate the puzzle.

$$\frac{\Gamma; \Delta \vdash e : A \multimap B \quad \Gamma; \Delta' \vdash e' \multimap A}{\Gamma; \Delta, \Delta' \vdash e\, e' : B}$$

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

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 [20], with. . .
- . . . basic metatheory, 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.

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*

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

I was brought up not to presume that a rig has a 1. Indeed, the trivial rig $\{0\}$ will give us the purely contemplative type theory we know and love. However, the key interesting example rig to bear in mind is the 'none-one-tons' rig.

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 weakening for ω, arbitrary usage. With the latter in place, we get three recognizable quantifiers,

$$\begin{array}{lcl} (0\,x\!:\!S)\to T & \text{is} & \forall x\!:\!S.\,T \\ (1\,x\!:\!S)\to T & \text{is} & (x\!:\!S)\multimap T \\ (\omega\,x\!:\!S)\to T & \text{is} & (x\!:\!S)\to T \end{array}$$

where $\forall$ is parametric, effectively an intersection rather than a product, with abstractions and applications erasable at run time, but Π is *ad hoc* and thus run time persistent. In making that valuable distinction, I learn from Miquel's Implicit Calculus Constructions [19]. I shall be quite 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 run time incarnation of its zero-resourced soul and computes in simulation with it, as we shall see in section 10.

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 is something like

$$(1\,xs\!:\!\mathsf{List\ Int})\to\{1\,ys\!:\!\mathsf{OrderedList\ Int};\ 0\,p\!:\!ys\sim xs\}$$

We can, of course, contemplate many other variations, whether for usage analysis in the style of Wadler [28], or for modelling partial knowledge as Gaboardi and friends have done [11].

3. Syntax

The type theory I present is parametrised by its system of sorts (or 'universes', if you prefer). I will keep it predicative: each sort is closed under quantification, but quantifying over 'higher' sorts is forbidden. My intention, ultimately, 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. I confess, that remains further work.

DEFINITION 3 (sort ordering). *Let $\mathcal{S}$ be a set of sorts (i,j,k) equipped with a wellfounded ordering $j \succ i$. That is,*

$$\frac{k\succ j \quad j\succ i}{k\succ i} \qquad \frac{\forall j.\,(\forall i.\,j\succ i\to P[i])\to P[j]}{\forall k.\,P[k]}$$

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

DEFINITION 4 (term, elimination).

$$\begin{array}{rcll} R,S,T,s,t & ::= & *_i & \textit{sort } i \\ & | & (\pi\,x\!:\!S)\to T & \textit{function type, price } \pi \\ & | & \lambda x.\,t & \textit{abstraction} \\ & | & \underline{e} & \textit{elimination} \\ e,f & ::= & x & \textit{variable} \\ & | & f\ s & \textit{application} \\ & | & s\!:\!S & \textit{type annotation/cut} \end{array}$$

Sorts are embedded explicitly, 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. The bidirectional design of the system ensures that abstractions need not have a domain annotation—that information always comes from the prior type. We can turn an elimination into a term without annotation: indeed, we will have *two* candidates for the type of such a thing.

In the other direction, a term can be used as an elimination only if we provide enough type information to check it. If we exclude type annotations from the syntax, we force terms into β-normal form. Effectively, the type annotations mark the places where computation is still to be done, or the 'cuts', in the language of logical calculi: we directly see the types of the 'active' terms. Let me reassure you that I ultimately intend human beings neither to write nor to read these mid-term type annotations, but rather to work with β-normal forms and typed *definitions*.

Syntactically, we cannot substitute terms for variables and leave them at that: we must either compute the β-redexes which arise, as in the hereditary substitution method introduced by Watkins and friends [30] and deployed to great effect by Robin Adams [3], or we must find some other way to suspend computation in a valid form. By the device of adding cuts to the syntax of eliminations, I facilitate 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 [13].

4. Computation

The syntax is arranged so that each redex pushes a cut towards its elimination.

The β-reduction rule removes a redex *elimination* at a function type but creates redexes at the domain and range. The υ-reduction rule removes the type annotation for an 'uncut' *term* which can compute no further.

DEFINITION 5 (contraction, reduction, computation). *The* contraction *schemes are given as follows.*

$$\begin{array}{rcl} (\lambda x.\,t : (\pi\,x\!:\!S)\to T)\ s & \leadsto_\beta & (t\!:\!T)[s\!:\!S/x] \\ \underline{t : T} & \leadsto_\upsilon & t \end{array}$$

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

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

DEFINITION 6 (diamond property). *A binary relation R has the* diamond property *if*

$$\forall s, p, q.\ sRp \wedge sRq \Rightarrow \exists r.\ pRr \wedge qRr$$

Appealing to a more direct visual intuition, I will typically state such propositions diagramatically, $\begin{array}{ccc} s & R & p \\ R & \exists & R \\ q & R & r \end{array}$ where the existential quantifier governs the points and proofs below and right of it.

The Tait–Martin-Löf–Takahashi method [24] is entirely adequate to the task. We introduce the notion of 'parallel reduction', $\triangleright$, which amounts to performing none, any or all of the available contractions in a term, but no further redexes arising as a result. Proving the diamond property for $\triangleright$ will establish the diamond property for $\twoheadrightarrow$. Here is how.

LEMMA 7 (parallelogram). *Let R, P be binary relations such that $R \subseteq P \subset R^*$. 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

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

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

Parallel reduction is constructed to fit Lemma 7: anything you contract but I do not, I can contract to match you, and vice versa.

DEFINITION 8 (parallel reduction). *Let parallel reduction, $\triangleright$, be defined by mutual induction for terms and eliminations, as follows.*

$$\frac{}{*_i \triangleright *_i} \qquad \frac{S \triangleright S' \quad T \triangleright T'}{(\pi\, x\!:\!S) \to T \triangleright (\pi\, x\!:\!S') \to T'}$$

$$\frac{t \triangleright t'}{\lambda x.\, t \triangleright \lambda x.\, t'} \qquad \frac{e \triangleright e'}{\underline{e} \triangleright \underline{e'}} \qquad \frac{t \triangleright t' \quad T \triangleright T'}{\underline{t\!:\!T} \triangleright t'}$$

$$\frac{}{x \triangleright x} \qquad \frac{f \triangleright f' \quad s \triangleright s'}{f\ s \triangleright f'\ s'} \qquad \frac{t \triangleright t' \quad T \triangleright T'}{t\!:\!T \triangleright t'\!:\!T}$$

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

That is, we have structural rules for each syntactic construct, together with υ and β rules.

LEMMA 9 (parallel reduction computes).

$$\leadsto\ \subseteq\ \triangleright\ \subseteq\ \twoheadrightarrow$$

PROOF Both inclusions are easy inductions on derivations. □

Crucially, parallel reduction commutes with substitution, because substitution can duplicate redexes or drop them entirely, but not break them into awkward pieces.

LEMMA 10 (parallel substitutions). *It is admissible that*

$$\frac{t \triangleright t' \quad \vec{e} \triangleright \vec{e'}}{t[\vec{e}/\vec{y}] \triangleright t'[\vec{e'}/\vec{y}]}$$

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

- $y_i \triangleright y_i$ where e_i/y_i and e'_i/y_i are in the substitutions 'before' and 'after': here, we substitute the given derivation of $e_i \triangleright e'_i$
- $(\lambda x.\, t : (\pi\, x\!:\!S) \to T)\ s \triangleright (t'\!:\!T')[s'\!:\!S'/x]$
  where our 'argument' inductive hypotheses allow us to deduce $(s : S)[\vec{e}/\vec{y}] \triangleright (s' : S')[\vec{e'}/\vec{y}]$, so we may extend the term and derivation substitutions, then use our 'body' hypotheses to deduce

  $$t[(s\!:\!S)[\vec{e}/\vec{y}]/x, \vec{e}/\vec{y}] \triangleright t'[(s'\!:\!S')[\vec{e'}/\vec{y}]/x, \vec{e'}/\vec{y}]$$

  and similarly for T; the usual composition laws for substitution allow us to phase the substitution

  $$-[(s\!:\!S)[\vec{e}/\vec{y}]/x, \vec{e}/\vec{y}] = -[s'\!:\!S'/x][\vec{e}/\vec{y}]$$

  and the β rule then yields

  $$((\lambda x.\, t : (\pi\, x\!:\!S) \to T)\ s)[\vec{e}/\vec{y}] \triangleright (t' : T')[s'\!:\!S'/x][\vec{e}/\vec{y}]$$

  as required. □

COROLLARY 11 (computation substitution). *If $e \twoheadrightarrow e'$ and $t \twoheadrightarrow t'$ then $t[e/x] \twoheadrightarrow t'[e'/x]$.*

PROOF Iterate Lemma 10. □

LEMMA 12 (parallel reduction diamond). $\begin{array}{ccc} s & \triangleright & p \\ \triangledown & \exists & \triangledown \\ q & \triangleright & r \end{array}$

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

  $$\frac{s \triangleright p \quad S \triangleright P}{\underline{s\!:\!S} \triangleright p} \qquad \frac{\dfrac{s \triangleright q \quad S \triangleright Q}{s\!:\!S \triangleright q\!:\!Q}}{\underline{s\!:\!S} \triangleright \underline{q\!:\!Q}}$$

  Inductively, we obtain r and R such that $p \triangleright r$, $q \triangleright r$, $P \triangleright R$ and $Q \triangleright R$, then deduce

  $$p \triangleright r \qquad \frac{q \triangleright r \quad Q \triangleright R}{\underline{q\!:\!Q} \triangleright r}$$

- For β, we have

  $$\frac{s' \triangleright p' \quad S \triangleright P \quad S' \triangleright P' \quad s \triangleright p}{(\lambda x.\, s' : (\pi\, x\!:\!S) \to S')\ s \triangleright (p'\!:\!P')[p\!:\!P/x]}$$

  $$\frac{\begin{array}{cccc} s' \triangleright q' & S \triangleright Q & S' \triangleright Q' & \\ \vdots & \vdots & \vdots & s \triangleright q \end{array}}{(\lambda x.\, s' : (\pi\, x\!:\!S) \to S')\ s \triangleright (\lambda x.\, q' : (\pi\, x\!:\!Q) \to Q')\ q}$$

  Inductively, we obtain common reducts r', R, R', r. Lemma 10 gives us that

  $$(p'\!:\!P')[p\!:\!P/x] \triangleright (r'\!:\!R')[r\!:\!R/x]$$

  and the β rule gives us

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

  as required. □

COROLLARY 13 (confluence). $\begin{array}{ccc} s & \twoheadrightarrow & p \\ \downarrow\!\!\!\downarrow & \exists & \downarrow\!\!\!\downarrow \\ q & \twoheadrightarrow & r \end{array}$

PROOF Apply Lemma 7 with Lemma 9 and Lemma 12. $\square$

My rearrangement of the calculus into explicitly bidirectional form has in no way disturbed the confluence of computation or the usual tools which establish that fact.

5. Typing Rules

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

DEFINITION 14 (precontext, context).

$$\Gamma \ ::= \ \cdot \mid \Gamma, x\!:\!S$$

A Γ-context is a marking-up of Γ with a rig element for each variable.

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

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

It is habitual for dependently typed programmers to talk of "Γ-contexts", rather than slicing "contexts" by $\lfloor - \rfloor$, after the fact. And it is a tidy habit: the Γ-contexts turn out to form an $\mathcal{R}$-module: addition $\Delta + \Delta'$ is pointwise, and multiplication $\phi\Delta$ just premultiplies all the rig-annotations on variables in Δ by ϕ, keeping types the same. It is another (in fact, the same) dependently typed programmer's 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. The proof of the lemma will be along shortly.

DEFINITION 15 (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 amount to the decoration of prejudgments with resource information in $\mathcal{R}$. We may define a forgetful map $\lfloor - \rfloor$ from judgments to prejudgments.

DEFINITION 16 (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". The benefit of this policy 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 for a straightforward inductive proof of type preservation.

If you are familiar with more traditional presentations of type theories, you may be as surprised as I am by the absence of a *context validity* judgment: I am learning, too. We shall recover context validity later, in order to check that the obverse of my policy also holds: 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 run time copies'. The types S and T do not consume any 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 17 (type checking and synthesis). *Type checking and synthesis are given by the following mutually inductive definition.*

PRE $$\frac{\Delta \vdash \rho\, R \ni t}{\Delta \vdash \rho\, T \ni t}\ T \rightsquigarrow R$$

SORT $$\frac{}{\Gamma \vdash_0 *_j \ni *_i}\ j \succ i$$

FUN $$\frac{\Gamma \vdash_0 *_i \ni S \quad \Gamma, x\!:\!S \vdash_0 *_i \ni T}{\Gamma \vdash_0 *_i \ni (\pi\, x\!:\!S) \to T}$$

LAM $$\frac{\Delta, \rho\pi\, x\!:\!S \vdash \rho\, T \ni t}{\Delta \vdash \rho\ (\pi\, x\!:\!S) \to T \ni \lambda x.\, t}$$

ELIM $$\frac{\Delta \vdash \rho\, e \in S}{\Delta \vdash \rho\, T \ni \underline{e}}\ S \preceq T$$

POST $$\frac{\Delta \vdash \rho\, e \in S}{\Delta \vdash \rho\, e \in R}\ S \rightsquigarrow R$$

VAR $$\frac{}{0\Gamma, \rho\, x\!:\!S, 0\Gamma' \vdash \rho\, x \in S}$$

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

CUT $$\frac{\lfloor\Delta\rfloor \vdash_0 *_i \ni S \quad \Delta \vdash \rho\, S \ni s}{\Delta \vdash \rho\, s\!:\!S \in S}$$

Introduction forms require a type proposed in advance. It may be necessary to precompute that type, for example, to put it into the form of a function type when checking a lambda. The SORT rule is an axiom, rather than requiring a valid context, reflecting the presupposition that the context is valid already. Note that the ELIM rule checks that we have synthesized a *subtype* S of the demanded type T. Subtyping facilitates upward travel through the predicative hierarchy of types, and is the stuff of the next section.

Elimination forms have their types synthesized. Again, we presuppose that the context is valid, hence the variable rule is an axiom, enforcing, moreover, the resource policy. It may be necessary to compute the type after synthesis, in order to see that it may be further eliminated. For example, when we synthesize the type for the function in an application, we should computed until that type reveals the domain and range. We may add contexts which are resourcings of the same precontext, allowing us to divide our riches in the application rule.

The possible resourcings of a given typing derivation will also turn out to constitute an $\mathcal{R}$-module. In particular, the *zero* of the module will amount to the observation that if you can make some of a thing with some stuff, then you can make none of it with no stuff. The typechecker leads a contemplative existence in which nothing is plenty.

6. Subtyping and its Metatheory

Subtyping is the co- and contravariant lifting of the universe ordering through the function type, where Luo's Extended Calculus od

Construction is covariant in the codomain and *equi*variant in the domain [16]. In order to establish that cut elimination preserves typing, we will have to justify the subtyping in the ELIM rule with a proof that subsumption holds, allowing a *term* to be lifted from sub- to supertype. In the course of the subsumption proof, given below, I found that I needed to adjust the definition to allow a little wiggle room where the application rule performs substitution: type annotations in cuts obstruct the proof. Rather than demanding cut elimination to get rid of the annotations, I propose to ignore them by brute force.

DEFINITION 18 (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 19 (subtyping transitivity). *Admissibly,*

$$\frac{R \preceq S \quad S \preceq T}{R \preceq T}$$

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 20 (similarity preservation).

$$\begin{array}{ccc} S & \sim & T \\ \triangledown & \exists & \triangledown \\ S' & \sim & T' \end{array} \qquad \begin{array}{cccc} & S & \sim & T \\ & \triangledown & & \triangledown \\ & S' & & T' \\ \exists & \triangledown & & \triangledown \\ & S'' & \sim & T'' \end{array} \qquad \begin{array}{ccc} S & \sim & T \\ \twoheaddownarrow & \exists & \twoheaddownarrow \\ S' & \sim & T' \end{array} \qquad \begin{array}{cccc} & S & \sim & T \\ & \twoheaddownarrow & & \twoheaddownarrow \\ & S' & & T' \\ \exists & \twoheaddownarrow & & \twoheaddownarrow \\ & S'' & \sim & T'' \end{array}$$

(Again, $\exists$ applies below and right.)
PROOF For the $\rhd$ pair, we just need a copycat induction on derivations of $\rhd$. For $\twoheadrightarrow$, we iterate the results for $\rhd$. □

LEMMA 21 (subtyping preservation).

$$\begin{array}{ccc} S & \preceq & T \\ \twoheaddownarrow & \exists & \twoheaddownarrow \\ S' & \preceq & T' \end{array} \qquad \begin{array}{ccccc} & S & \preceq T & \twoheadrightarrow & T' \\ \exists & \twoheaddownarrow & & & \| \\ & S' & \preceq & & T' \end{array} \qquad \begin{array}{cccc} & S & \preceq & T \\ & \twoheaddownarrow & & \twoheaddownarrow \\ & S' & & T' \\ \exists & \twoheaddownarrow & & \twoheaddownarrow \\ & S'' & \preceq & T'' \end{array}$$

PROOF Proceed by induction on the derivation of $S \preceq T$. Lemma 20 covers the similarity case. In the case of sorts, there is no way to compute. In the case of function types, computation must occur only within sources and targets, so the inductive hypotheses deliver the result. □

LEMMA 22 (subtyping stability).

$$S \preceq T \Rightarrow S[r\!:\!R/x] \preceq T[r\!:\!R'/x]$$

PROOF Induction on derivations. Similarity ignores the distinction between R and R' wherever the substitution occurs. □

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

THEOREM 23 (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 21. We may clearly allow iterated PRE to advance types by $\twoheadrightarrow$, not just $\leadsto$. I write $\therefore$ to mark an appeal to the inductive hypothesis.

$$\frac{\Delta \vdash \rho\, R \ni t}{\Delta \vdash \rho\, S \ni t}\, S \leadsto R \quad \begin{array}{ccc} S & \preceq & T \\ \twoheaddownarrow & \exists & \twoheaddownarrow \\ R & \preceq & R' \end{array} \quad \frac{\therefore \Delta \vdash \rho\, R' \ni t}{\Delta \vdash \rho\, T \ni t}$$

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

$$\frac{S' \preceq S \quad T \preceq T'}{(\pi\, x\!:\!S) \to T \preceq (\pi\, x\!:\!S') \to T'} \qquad \frac{\Delta, \rho\pi\, x\!:\!S \vdash \rho\, T \ni t}{\Delta \vdash \rho\, (\pi\, x\!:\!S) \to T \ni \lambda x.\, t}$$

The contravariance of function subtyping allows us to extend the context with the source subtype and check the target supertype.

$$\frac{\therefore \Delta', \rho\pi\, x\!:\!S' \vdash \rho\, T' \ni t}{\Delta' \vdash \rho\, (\pi\, x\!:\!S') \to T' \ni \lambda x.\, t}$$

For ELIM, we have

$$\frac{\Delta \vdash \rho\, e \in R}{\Delta \vdash \rho\, S \ni \underline{e}}\, R \preceq S \quad S \preceq T$$

Inductively, for some $R' \preceq R$ we have $\Delta' \vdash \rho\ e \in R'$ and by Lemma 19, we get $R' \preceq T$ and apply ELIM. For POST, we may again appeal to Lemma 21. For VAR, we look up the subtype given by the contextual subtyping. For APP, we have

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

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,

$$\begin{array}{ll} \therefore \exists S', T'.\ S \preceq S' \wedge T' \preceq T \wedge & \Delta'_0 \vdash \rho\ f \in (\pi\, x\!:\!S') \to T' \\ & \therefore \Delta'_1 \vdash \rho\pi\ S' \ni s \end{array}$$

from which we obtain

$$\Delta'_0 + \Delta'_1 \vdash \rho\ f\ s \in T'[s\!:\!S'/x]$$

where Lemma 22 gives, as required, $T'[s\!:\!S'/x] \preceq T[s\!:\!S/x]$. □

7. 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 24 ($\mathcal{R}$-module). *An $\mathcal{R}$-module is a set M with*

zero	0	:	M
addition	$- + -$	:	$M \times M \to M$
scalar multiplication	$- \, -$	:	$\mathcal{R} \times M \to M$

which make $(M, 0, +)$ *a commutative monoid and are, moreover, compatible with* $\mathcal{R}$*'s operations, in that, for all* $m \in M$

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

The obvious way to form $\mathcal{R}$-modules is pointwise. We shall not require anything more serpentine.

LEMMA 25 (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 26 (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 25 applies. □

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. In some sense, though, that is but a boring fact about syntax: the point is that the module can then be restricted to *valid* derivations.

DEFINITION 27 (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,* $\mathrm{Pos} : \mathcal{T}(\mathcal{P}) \to \mathbf{Set}$ *are given recursively:*

$$\mathrm{Pos}\left(\frac{P_1 ... P_n}{P}\right) = \mathrm{Pos}'(P_1) + \sum_i \mathrm{Pos}(P_i)$$

where prejudgment positions $\mathrm{Pos}' : \mathcal{P} \to \mathbf{Set}$ *are given by*

$$\begin{array}{lcl} \mathrm{Pos}'(\Gamma \vdash) & = & 0 \\ \mathrm{Pos}'(\Gamma \vdash T \ni t) & = & \mathrm{dom}(\Gamma) + 1 \\ \mathrm{Pos}'(\Gamma \vdash e \in S) & = & \mathrm{dom}(\Gamma) + 1 \end{array}$$

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. We thus acquire by a straightforward recursive labelling strategy:

LEMMA 28 (representing derivations).

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

PROOF By structural induction on S, it is enough to show that at each node, we have $\{J : \mathcal{J} \mid \lfloor J \rfloor = P\} \cong \mathrm{Pos}'(P) \to \mathcal{R}$.

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 a thoroughly unremarkable $\mathcal{R}$-module. Let us establish something only slightly more remarkable. In fact it is obvious, because when designing the system, I took the trouble to ensure that any nonzero resource demand in the conclusion of each rule is linearly a factor of the demands in the premises.

THEOREM 29 (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* $\{D' : \mathcal{T}(\mathcal{J}) \mid \lfloor D' \rfloor = d, T' \text{ valid}\}$.

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.

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

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

$$\frac{\phi\Delta \vdash \phi\rho\ f \in (\pi\, x\!:\!S) \to T \quad \phi\Delta' \vdash \phi(\rho\pi) = (\phi\rho)\pi\ S \ni s}{\phi\Delta + \phi\Delta' = \phi(\Delta + \Delta') \vdash \phi\rho\ f\ s \in T[s\!:\!S/x]}$$

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

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

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.

$$\frac{\Delta_0 \vdash \rho_0\ f \in (\pi\, x\!:\!S) \to T \quad \Delta_0{}' \vdash \rho_0\pi\ S \ni s}{\Delta_0 + \Delta_0{}' \vdash \rho_0\ f\ s \in T[s\!:\!S/x]} +$$

$$\frac{\Delta_1 \vdash \rho_1\ f \in (\pi\, x\!:\!S) \to T \quad \Delta_1{}' \vdash \rho_1\pi\ S \ni s}{\Delta_1 + \Delta_1{}' \vdash \rho_1\ f\ s \in T[s\!:\!S/x]} =$$

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

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 whenever they are used to make multiples and sums.

LEMMA 30 (factorization). *If* $\Delta \vdash \phi\rho\ T \ni t$ *then for some* 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.

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

The inductive hypotheses give $\Delta_0/\phi \vdash \rho\ f \in (\pi\, x\!:\!S) \to T$, and reassociating, $\Delta_1/\phi \vdash \rho\pi\ S \ni s$, so a little distribution gives us $\phi(\Delta_0/\phi + \Delta_1/\phi) \vdash \phi\rho\ f\ s \in T[s\!:\!S/x]$. □

COROLLARY 31 (nothing from nothing). *If* $\Delta \vdash 0\ T \ni t$ *then* $\Delta = 0\lfloor\Delta\rfloor$.

PROOF Lemma 30 with $\phi = \rho = 0$. □

LEMMA 32 (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*

$\Delta \vdash (\phi + \rho)\ \vdash e \in S$ *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.

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

The inductive hypotheses give $\Delta_0' \vdash \phi\ f \in (\pi\, x : S) \to T$ 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 we obtain $\Delta_0' + \Delta_1' \vdash \phi\pi\ f\ s \in T[s \!:\! S/x]$ and $\Delta_0'' + \Delta_1'' \vdash \rho\pi\ f\ s \in T[s \!:\! S/x]$. □

8. Resourced Stability Under Substitution

In this section, we will establish that basic thinning and substitution operations lift from the syntax of terms and eliminations to the judgments of checking and synthesis. It may seem peculiar to talk of thinning in a precisely resource-accounted setting, but as the *pre*context grows, the context will show that we have *zero* of the extra things.

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

DEFINITION 33 (localization). *Define*

$$\begin{array}{lcl} - \vdash - : \mathrm{Cx}(\Gamma) \times \mathcal{J} & \to & \mathcal{J} \\ \Delta \vdash\ \Delta' \vdash \rho\, T \ni t & = & \Delta, \Delta' \vdash \rho\, T \ni t \\ \Delta \vdash\ \Delta' \vdash \rho\, e \in S & = & \Delta, \Delta' \vdash \rho\, e \in S \end{array}$$

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

PROOF Induction on derivations, with $\mathcal{J}$ absorbing all local extensions to the context, so that the inductive hypothesis applies under binders. It is clear that we can 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 35 (stability under substitution). *Admissibly,*

$$\frac{\Delta, \phi\, x \!:\! S \vdash \mathcal{J} \quad \Delta' \vdash \phi\ e \in S}{\Delta + \Delta' \vdash \mathcal{J}[e/x]}$$

PROOF The proof is by 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 of the cases are purely structural, but the devil is in the detail of the resourcing, so let us pay attention. 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 31 tells us that $\phi = 0$ and hence $\Delta' = 0\Gamma$, allowing the induction to go through. For VAR with variables other than x, again, $\phi = 0$ and the induction goes through. For VAR at x itself, Lemma 34 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 29) exactly none of them to its type. In the APP case, Lemma 32 allows us to share out our es in exactly the quantities demanded in the premises. □

9. Computation Preserves Typing

The design of the rule 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 [18], the result comes out in one delightfully unsubtle dollop exactly because of the uniformity of the information flow in 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 36 (parallel preservation).

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

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

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

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, we must have had

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

$$\frac{\therefore \Gamma' \vdash_0 *_i \ni S' \quad \therefore \Gamma', x \!:\! S' \vdash_0 *_i \ni T'}{\Gamma' \vdash_0\ *_i \ni (\pi\, x \!:\! S') \to T'}$$

For LAM, we must have had

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

$$\frac{\therefore \Delta', \rho\pi\, x \!:\! S' \vdash \rho\, T' \ni t'}{\Delta' \vdash \rho\ (\pi\, x \!:\! S') \to T' \ni \lambda x.\, t'}$$

For ELIM, we have two cases. For the structural case, we will need a little more computation.

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

$$\begin{array}{cccc} & S & \preceq & T \\ & \twoheaddownarrow & & \twoheaddownarrow \\ & S' & & T' \\ \exists & \twoheaddownarrow & & \twoheaddownarrow \\ & S'' & \preceq & T'' \end{array} \qquad \cfrac{\cfrac{\cfrac{\Delta' \vdash \rho\, e' \in S'}{\Delta' \vdash \rho\, e' \in S''}\ S' \twoheadrightarrow S''}{\Delta' \vdash \rho\, T'' \ni \underline{e'}}\ S'' \preceq T''}{\Delta' \vdash \rho\, T' \ni \underline{e'}}\ T' \twoheadrightarrow T''$$

Lemma 21 reestablishes subtyping after computation.

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

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

Then by Theorem 23 (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

$$\dfrac{\Delta \vdash \rho\, e \in S}{\Delta \vdash \rho\, e \in R}\qquad \begin{array}{cccc}\Delta & e & S & \rightsquigarrow R\\ \twoheaddownarrow & \triangledown & \exists\twoheaddownarrow & \exists\twoheaddownarrow\\ \Delta' & e' & S' & \twoheadrightarrow R'\end{array}\qquad \dfrac{\therefore \Delta' \vdash \rho\, e' \in S'}{\Delta' \vdash \rho\, e' \in R'}$$

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

$$\begin{array}{cc}\dfrac{}{0\Gamma_0, \rho\, x\!:\!S, 0\Gamma_1 \vdash \rho\, x \in S} & \begin{array}{ccc}\Gamma_0 & S & \Gamma_1\\ \twoheaddownarrow & \twoheaddownarrow & \twoheaddownarrow\\ \Gamma_0' & S' & \Gamma_1'\end{array}\\ \dfrac{}{0\Gamma_0', \rho\, x\!:\!S', 0\Gamma_1' \vdash \rho\, x \in S'} & \end{array}$$

The case of CUT is just structural.

$$\begin{array}{cc}\dfrac{\lfloor\Delta\rfloor \vdash_0 *_i \ni S \quad \Delta \vdash \rho\, S \ni s}{\Delta \vdash \rho\, s\!:\!S \in S} & \begin{array}{ccc}\Delta & S & s\\ \twoheaddownarrow & \triangledown & \triangledown\\ \Delta' & S' & s'\end{array}\end{array}$$
$$\dfrac{\therefore \lfloor\Delta'\rfloor \vdash_0 *_i \ni S' \quad \therefore \Delta' \vdash \rho\, S' \ni s'}{\Delta' \vdash \rho\, s'\!:\!S' \in S'}$$

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

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

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

$$\begin{array}{cc}S & T\\ \exists\twoheaddownarrow & \exists\twoheaddownarrow\\ S' & T'\end{array}\quad \dfrac{\therefore \Delta_0' \vdash \rho\, f \in (\pi\, x\!:\!S') \to T' \quad \therefore \Delta_1' \vdash \rho\pi\ S' \ni s}{\Delta_0' + \Delta_1' \vdash \rho\, f'\ s' \in T'[s'\!:\!S'/x]}$$

where Corollary 11 tells us that $T[s : S/x] \twoheadrightarrow T'[s' : S'/x]$. This leaves only the case where application performs β-reduction, which is of course the villain of the piece. We have

$$\dfrac{\begin{array}{l}\Delta_0 \vdash \rho\, (\lambda x.\, t : (\pi\, x\!:\!S_0) \to T_0) \in (\pi\, x\!:\!S_1) \to T_1\\ \Delta_1 \vdash \rho\pi\ S_1 \ni s\end{array}}{\Delta_0 + \Delta_1 \vdash \rho\, (\lambda x.\, t : (\pi\, x\!:\!S_0) \to T_0)\ s \in T_1[s\!:\!S_1/x]}$$

$$\begin{array}{ccccccccc}\Delta_0 + & \Delta_1 & t & S_0 & T_0 & s & S_0 & T_0\\ \twoheaddownarrow \quad & \twoheaddownarrow & \triangledown & \triangledown & \triangledown & \triangledown & \twoheaddownarrow & \twoheaddownarrow\\ \Delta_0' + & \Delta_1' & t' & S_0' & T_0' & s' & S_1 & T_1\end{array}$$

noting that the possible use of POST means we apply at a function type computed from that given explicitly. 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:

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

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 35 (stability under substitution) gives us

$$\begin{array}{l}\lfloor\Delta_0' + \Delta_1'\rfloor \vdash_0 *_i \ni T_0'[s'\!:\!S_0'/x]\\ \Delta_0' + \Delta_1' \vdash \rho\, T_0'[s'\!:\!S_0'/x] \ni t'[s'\!:\!S_0'/x]\end{array}$$

from which CUT synthesizes

$$\Delta_0' + \Delta_1' \vdash \rho\ (t'\!:\!T_0')[s'\!:\!S_0'/x] \in T_0'[s'\!:\!S_0'/x]$$

so we may apply POST to compute our target type to a common reduct $T_0 \twoheadrightarrow \{T_0', T_1, T_2\} \twoheadrightarrow T_1'$, and deliver

$$\Delta_0' + \Delta_1' \vdash \rho\ (t'\!:\!T_0')[s'\!:\!S_0'/x] \in T_1'[s'\!:\!S_1'/x]$$

as required. □

COROLLARY 37 (preservation).

$$\begin{array}{ccc}\Delta & T & t\\ \twoheaddownarrow & \twoheaddownarrow & \twoheaddownarrow\\ \Delta' & T' & t'\end{array} \Rightarrow \dfrac{\Delta \vdash \rho\, T \ni t}{\Delta' \vdash \rho\, T' \ni t'}\qquad \begin{array}{cc}\Delta & e\\ \twoheaddownarrow & \twoheaddownarrow\\ \Delta' & e'\end{array} \Rightarrow \begin{array}{c}S\\ \exists\twoheaddownarrow\\ S'\end{array} \wedge \dfrac{\Delta \vdash \rho\, e \in S}{\Delta' \vdash \rho\, e' \in S'}$$

PROOF Iteration of Theorem 36. □

10. Erasure to an Implicit Calculus

Run-time programs live in the good old untyped lambda calculus, which is here also the teletyped lambda calculus.

DEFINITION 38 (programs).

$$\mathtt{p} ::= x \mid \texttt{\textbackslash}x\ \texttt{->}\ \mathtt{p} \mid \mathtt{p}\ \mathtt{p}$$

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

DEFINITION 39 (erasure judgments). *When $\rho \neq 0$, we may form judgments as follows.*

$$\Delta \vdash \rho\, T \ni t\ \blacktriangleright \mathtt{p} \qquad \Delta \vdash \rho\, e \in S\ \blacktriangleright \mathtt{p}$$

That is, programs are nonzero-resourced. Such judgments are derived by an elaborated version of the existing rules which add programs as outputs.

DEFINITION 40 (checking and synthesis with erasure).

PRE+
$$\frac{\Delta \vdash \rho\, R \ni t \ \blacktriangleright \mathtt{p}}{\Delta \vdash \rho\, T \ni t \ \blacktriangleright \mathtt{p}}\ T \leadsto R$$

LAM0
$$\frac{\Delta, 0\,x\!:\!S \vdash \rho\, T \ni t \ \blacktriangleright \mathtt{p}}{\Delta \vdash \rho\ (\phi\, x\!:\!S) \to T \ni \lambda x.\, t \ \blacktriangleright \mathtt{p}}\ \rho\phi = 0$$

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

ELIM+
$$\frac{\Delta \vdash \rho\, e \in S \ \blacktriangleright \mathtt{p}}{\Delta \vdash \rho\, T \ni \underline{e} \ \blacktriangleright \mathtt{p}}\ S \preceq T$$

POST+
$$\frac{\Delta \vdash \rho\, e \in S \ \blacktriangleright \mathtt{p}}{\Delta \vdash \rho\, e \in R \ \blacktriangleright \mathtt{p}}\ S \leadsto R$$

VAR+
$$\frac{}{0\Gamma, \rho\, x\!:\!S, 0\Gamma' \vdash \rho\, x \in S \ \blacktriangleright x}$$

APP0
$$\frac{\Delta \vdash \rho\, f \in (\phi\, x\!:\!S) \to T \ \blacktriangleright \mathtt{p} \quad \lfloor\Delta\rfloor \vdash_0 S \ni s}{\Delta \vdash \rho\, f\ s \in T[s\!:\!S/x] \ \blacktriangleright \mathtt{p}}\ \rho\phi = 0$$

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

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

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.

$$\frac{\rho + \pi = 0}{\rho = \pi = 0}$$

Consequently, 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 the VAR rule can still contemplate it Accordingly, no variable escapes its scope.

We obtain without difficulty that erasure can be performed.

LEMMA 41 (erasures uniquely exist). *If $\rho \neq 0$, then*

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

PROOF Induction on derivations. □

The question, none the less, is whether the resulting programs behave sensibly. We may at least be sure that they come from impeccable sources.

LEMMA 42 (erasure rules elaborate). *Admissibly,*

$$\frac{\Delta \vdash \rho\, T \ni t \ \blacktriangleright \mathtt{p}}{\Delta \vdash \rho\, T \ni t} \qquad \frac{\Delta \vdash \rho\, e \in S \ \blacktriangleright \mathtt{p}}{\Delta \vdash \rho\, e \in S}$$

PROOF Induction on derivations. □

The unerased forms may thus be used to form types.

COROLLARY 43 (erasure rules elaborate). *Admissibly,*

$$\frac{\Delta \vdash \rho\, T \ni t \ \blacktriangleright \mathtt{p}}{\lfloor\Delta\rfloor \vdash_0 T \ni t} \qquad \frac{\Delta \vdash \rho\, e \in S \ \blacktriangleright \mathtt{p}}{\lfloor\Delta\rfloor \vdash_0 e \in S}$$

PROOF Lemma 42, then Theorem 29, multiplying by 0. □

How do programs behave? They may compute by β reduction. I choose to be liberal about where they may do so.

DEFINITION 44 (program computation).

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

$$\frac{\mathtt{p} \leadsto \mathtt{p}'}{\mathtt{p}\ \mathtt{p}_a \leadsto \mathtt{p}'\ \mathtt{p}_a} \qquad \frac{\mathtt{p} \leadsto \mathtt{p}'}{\mathtt{p}_f\ \mathtt{p} \leadsto \mathtt{p}_f\ \mathtt{p}'}$$

The key to understanding the good behaviour of computation is to observe that any term which erases to some $\backslash x \,\texttt{->}\, \mathtt{p}$ 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 45 (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 $\rho\pi = 0$ and T is an n-to-ρ-function type.*

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

LEMMA 46 (applicability). *If $\Delta \vdash \rho\, T \ni t \ \blacktriangleright \ \backslash x \,\texttt{->}\, \mathtt{p}$, then $T \twoheadrightarrow$ some n-to-ρ-function type T' and $t \twoheadrightarrow$ some $\lambda^n\vec{y}.\, \lambda x.\, t'$ such that*

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

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

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

PROOF Proceed by induction on derivations. Rules VAR+ and APP+ are excluded by the requirement to erase to $\backslash x \,\texttt{->}\, \mathtt{p}$. 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 a 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+, the subtyping rule for schemes lets us invoke the inductive hypothesis which delivers a cut, $\underline{e} \twoheadrightarrow \lambda\vec{y}.\, \lambda x.\, t' : S'$ with $S \twoheadrightarrow S'$. Hence $S \ni \underline{\lambda\vec{y}.\, \lambda x.\, t' : S'}$ and then Theorem 36 (preservation) and Theorem 23 (subsumption) allow us the υ-reduction to $T \ni \lambda\vec{y}.\, \lambda x.\, t'$.

For APP0, the inductive hypothesis gives us $f \twoheadrightarrow \lambda y.\, \lambda^n\vec{y}.\, \lambda x.\, t' : (\phi\, x : S') \to T'$ with $S \twoheadrightarrow S'$ and $T \twoheadrightarrow T'$, and T' an n-toρ-function type. Hence $f\ s \twoheadrightarrow (\lambda^n\vec{y}.\, \lambda x.\, t' : T')[s : S'/x]$. 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 47 (step simulation). *The following implications hold.*

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

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

PROOF We proceed by induction on derivations and inversion of program computation. The only interesting case is the APP+ case when the computation takes a β-step.

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

We have some f whose type is a 0-to-ρ function type and which erases to some $\backslash x \,\texttt{->}\, \mathtt{p}$, so we may invoke Lemma 46 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, $f\, s \twoheadrightarrow (t : T')[s : S']$, 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 are convinced that terms are well typed, contemplation has served its purpose and we may safely erase to 0-free programs which will neither go wrong nor violate their resource consumption conditions.

11. Take It Or Leave It

Let us consider making our rig-based resource management a little less rigid. In particular, the $\{0, 1, \omega\}$ rig apparently insists that the argument of a function in $(\omega\, x : S) \to T$ has *at least one* run-time use in some place where ω are required. That is, it 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 way to allow more flexibility is to impose an 'order', $\leq$ on the rig. We can extend it conjunctively pointwise to Γ-contexts, so $\Delta \leq \Delta'$ if Δ' has at least as many of each variable in Γ as Δ has.

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

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

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

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

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

To retain Lemmas 30 and 32 (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 48 (weakening). *If $\rho \leq \rho'$ then*

$$\frac{\Delta' \vdash \rho'\pi\, T \ni t}{\exists \Delta.\ \Delta \leq \Delta' \wedge\ \Delta \vdash \rho\pi\, T \ni t}$$

$$\frac{\Delta' \vdash \rho'\pi\, e \in S}{\exists \Delta.\ \Delta \leq \Delta' \wedge\ \Delta \vdash \rho\pi\, e \in S}$$

PROOF Induction on derivations is sufficient, 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 safety of erasure, we must add a further condition that keeps WEAK from bringing a contemplated variable back into a usable position:

$$\frac{\rho \leq 0}{\rho = 0}$$

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: if you have plenty, you can throw it away, but if you have one, you must use it.

12. Contemplation

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

The issue we encounter when we try to construct a dependent type theory is to identify which things classified by types the variables in types may stand for. It is not always obvious, because there may be more than one kind of thing that the same types are types for, but the things that the variables in types stand for are special: let us call them the *icons* of the theory, for they are contemplated images. We acquire a fully dependent type theory when everything classified by a type *has* an icon, not when everything *is* an icon.

Here, we use the same types to classify terms and eliminations in whatever quantity and also untyped programs after erasure, but it is the eliminations of quantity zero which are the icons, and everything classified by a type in one way or another corresponds to an icon.

Such considerations should warn us to be careful about jumping to conclusions, however enthusiastic we may be feeling. 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 [29], 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 dependent session types because its icons give too much information: they are the contemplated form of 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 thing classified by session types, but it is the signal traffic which must take the iconic role. What I am trying to say is that 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 for programming in the style of Gay and Vasconcelos [12] with dependent session types and linearly managed channels.

What we do have is the basis for a propositions-as-types account of specifying and verifying linearly typed programs, where the idealised behaviour of the program is available for discussion 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' [21, 27]. 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] becomes still more crucial.

Looking to the nearer future, I have already mentioned the desirability of a normalization proof, and its orthogonality to the resource issue in this setting. Of course, we must also look beyond the $\multimap$ and give dependent accounts of other linear connectives: the 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 your money or some goods to the value of your money. It is far from clear how to dualize these notions.

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 is likely to be an optimisation [10].

What we can now bring to all of these possibilities is the idea to separate contemplation from consumption, ensuring that contemplation counts for nothing in the resource analysis and that the contemplated fragment of our calculi 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 is an icon?'. As Gershwin and Heyward might have put it, 'got my song, got heaven the whole day long'.

Acknowledgments

Many acknowledgments are due. I have had a lot of assistance, much inspiration from colleagues and many enlightening conversations. Names will be named and honour bestowed if and when this paper ceases to be anonymous.

References

[1] A. Abel. *Normalization by Evaluation: Dependent Types and Impredicativity*. PhD thesis, Ludwig-Maximilians-Universität München, 2013. Habilitationsschrift.

[2] A. Abel, T. Coquand, and P. Dybjer. Normalization by evaluation for martin-lof 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] R. Adams. Pure type systems with judgemental equality. *J. Funct. Program.*, 16(2):219–246, 2006.

[4] A. Asperti, W. Ricciotti, C. S. Coen, and E. Tassi. A bi-directional refinement algorithm for the calculus of (co)inductive constructions. *Logical Methods in Computer Science*, 8(1), 2012.

[5] J. Bernardy, P. Jansson, and R. Paterson. Proofs for free - parametricity for dependent types. *J. Funct. Program.*, 22(2):107–152, 2012.

[6] E. Brady. Idris, a general-purpose dependently typed programming language: Design and implementation. *J. Funct. Program.*, 23(5):552–593, 2013.

[7] L. Caires and F. Pfenning. Session types as intuitionistic linear propositions. In P. Gastin and F. 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] I. Cervesato and F. Pfenning. A Linear Logical Framework. *Inf. Comput.*, 179(1):19–75, 2002.

[9] E. de Vries, R. Plasmeijer, and D. M. Abrahamson. Uniqueness Typing Simplified. In O. Chitil, Z. Horváth, and V. 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] F. Domínguez and A. 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. URL `http://dl.acm.org/citation.cfm?id=2228095.2228101`.

[11] M. Gaboardi, A. Haeberlen, J. Hsu, A. Narayan, and B. C. Pierce. Linear dependent types for differential privacy. In R. Giacobazzi and R. 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.

[12] S. J. Gay and V. T. Vasconcelos. Linear type theory for asynchronous session types. *J. Funct. Program.*, 20(1):19–50, 2010.

[13] G. Gentzen. Untersuchungen über das logische schließen. *Mathematische Zeitschrift*, 29(2-3):176–210, 405–431, 1935.

[14] J. Girard. Linear logic. *Theor. Comput. Sci.*, 50:1–102, 1987.

[15] N. R. Krishnaswami, P. Pradic, and N. Benton. Integrating Linear and Dependent Types. In S. K. Rajamani and D. 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.

[16] Z. 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.

[17] P. 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.

[18] J. McKinna and R. Pollack. Some lambda calculus and type theory formalized. *J. Autom. Reasoning*, 23(3-4):373–409, 1999.

[19] A. Miquel. The implicit calculus of constructions. In *TLCA*, pages 344–359, 2001.

[20] B. C. Pierce and D. N. Turner. Local type inference. *ACM Trans. Program. Lang. Syst.*, 22(1):1–44, 2000.

[21] J. C. Reynolds. Types, abstraction and parametric polymorphism. In *IFIP Congress*, pages 513–523, 1983.

[22] R. Shi and H. Xi. A linear type system for multicore programming in ATS. *Sci. Comput. Program.*, 78(8):1176–1192, 2013.

[23] N. Swamy, J. Chen, C. Fournet, P. Strub, K. Bhargavan, and J. Yang. Secure distributed programming with value-dependent types. *J. Funct. Program.*, 23(4):402–451, 2013.

[24] M. Takahashi. Parallel Reductions in λ-Calculus (revised version). *Information and Computation*, 118(1):120–127, 1995.

[25] B. Toninho, L. Caires, and F. Pfenning. Dependent session types via intuitionistic linear type theory. In P. Schneider-Kamp and M. 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.

[26] M. Vákár. Syntax and Semantics of Linear Dependent Types. *CoRR*, abs/1405.0033, 2014.

[27] P. Wadler. Theorems for free! In *FPCA*, pages 347–359, 1989.

[28] P. Wadler. Is there a use for linear logic? In C. Consel and O. 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.

[29] P. Wadler. Propositions as sessions. In P. Thiemann and R. B. Findler, editors, *ACM SIGPLAN International Conference on Functional Pro-*

gramming, ICFP'12, Copenhagen, Denmark, September 9-15, 2012, pages 273–286. ACM, 2012.

[30] K. Watkins, I. Cervesato, F. Pfenning, and D. Walker. A concurrent logical framework: The propositional fragment. In S. Berardi, M. Coppo, and F. 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.