Electronic Notes in Theoretical Computer Science 41 No. 3 (2000)
URL: http://www.elsevier.nl/locate/entcs/volume41.html pages 70–88

Operational Properties of `Lily`, a Polymorphic Linear Lambda Calculus with Recursion[⋆]

G.M. Bierman[a] A.M. Pitts[b] C.V. Russo[b]

[a] *Department of Computer Science, Warwick University, Coventry CV4 7AL, UK*

[b] *Computer Laboratory, Cambridge University, Cambridge CB2 3QG, UK*

Abstract

Plotkin has advocated the combination of linear lambda calculus, polymorphism and fixed point recursion as an expressive semantic metalanguage. We study its expressive power from an operational point of view. We show that the naturally call-by-value operators of linear lambda calculus can be given a call-by-name semantics without affecting termination at exponential types and hence without affecting ground contextual equivalence. This result is used to prove properties of a logical relation that provides a new extensional characterisation of ground contextual equivalence and relational parametricity properties of polymorphic types.

1 Introduction

When giving denotational semantics of programming languages using domain theory, use is often made of metalanguages in which to phrase the semantic descriptions [16,14]. The attraction of such an approach is that it allows the semantically relevant constructs and proof principles inherent in the domain theory to be abstracted from the often quite complicated mathematical details. Restricting attention to the denotational semantics of deterministic languages (i.e. excluding the use of various kinds of powerdomain), Plotkin [23] makes a convincing case for *polymorphic linear lambda calculus with recursion* as an expressive denotational metalanguage. In particular, the powerful nature of impredicative polymorphism permits the plethora of domain-theoretic constructs to be defined in terms of remarkably few primitive type-forming operations, namely ∀-types, $\forall\alpha.\,\tau$, linear function types, $\tau \multimap \tau'$, and exponential types, $!\tau$: see Fig. 1, which uses the domain-theoretic terminology of [8]. (Plotkin [23] chooses to make intuitionistic function types primitive rather than exponential types; we prefer to express the former in terms of !

⋆ Research supported by UK EPSRC grant GR/M04716.

Lifting	$\tau_\bot \triangleq\ !\tau$	
Functions	$\tau \to \tau' \triangleq\ !\tau \multimap \tau'$	
Strict functions	$\tau \circ\!\!\to \tau' \triangleq \tau \multimap \tau'$	
Smash product	$\tau \otimes \tau' \triangleq \forall\alpha.(\tau \multimap \tau' \multimap \alpha) \multimap \alpha$	
Coalesced sum	$\tau \oplus \tau' \triangleq \forall\alpha.\,!(\tau \multimap \alpha) \multimap\ !(\tau' \multimap \alpha) \multimap \alpha$	
Product	$\tau \times \tau' \triangleq \forall\alpha.((\tau \multimap \alpha) \oplus (\tau' \multimap \alpha)) \multimap \alpha$	
Separated sum	$\tau + \tau' \triangleq\ !\tau \oplus\ !\tau'$	
Existential	$\exists\ \alpha.\,\tau(\alpha) \triangleq \forall\beta.(\forall\alpha.\,\tau(\alpha) \multimap \beta) \multimap \beta$	
Truth values	$\mathsf{T} \triangleq \forall\alpha.\,!\alpha \multimap\ !\alpha \multimap \alpha$	
Flat naturals	$\mathsf{N}_\bot \triangleq \forall\alpha.\,!\alpha \multimap\ !(!\alpha \multimap \alpha) \multimap \alpha$	
Inductive	$\mu\alpha.\,\tau(\alpha) \triangleq \forall\alpha.\,!(\tau(\alpha) \multimap \alpha) \multimap \alpha$	$(\alpha$ +ve in $\tau(\alpha))$
Co-inductive	$\nu\alpha.\,\tau(\alpha) \triangleq \exists\ \alpha.\,!(\alpha \multimap \tau(\alpha)) \otimes \alpha$	$(\alpha$ +ve in $\tau(\alpha))$
Recursive	$\mathtt{rec}\,\alpha.\,\tau(\alpha,\alpha) \triangleq \nu\alpha.\,\tau(\mu\beta.\,\tau(\alpha,\beta),\alpha)$	$(\alpha$ +ve in $\tau(\alpha,\beta)$, β −ve in $\tau(\alpha,\beta))$

Fig. 1. `Lily` as a denotational metalanguage

and $\multimap$ using Girard's famous decomposition: $\tau \to \tau' =\ !\tau \multimap \tau'$.) The definitions in Fig. 1 only have weak properties if one works up to β-convertibility of terms. To get stronger properties, such as category-theoretic universal properties, it should suffice to work with a notion of equality of terms that makes $\forall$-types *relationally parametric* in the sense of Reynolds [24]. In theory, one way to generate such a notion of equality is via a suitable model: Plotkin [23] sketches one using strict, inductive partial equivalence relations on a domain model of the untyped lambda calculus. However, in practice, as far as we know, the details of this relationally parametric model of polymorphic linear lambda calculus with recursion have not been worked out in detail. We take a different[1] and more computational approach: we make polymorphic linear lambda calculus with recursion into a programming language (we call it `Lily`) by endowing it with an operational semantics; we choose a particular notion of contextual equivalence derived from the operational semantics; and we prove that this notion of term equality is relationally parametric with respect to a suitable notion of binary relation. This strategy has been applied successfully by the second author to the combination of polymorphism with PCF [21] and with call-by-value PCF [20]. However, it is not so easy to apply the strategy

[1] It should also be noted that `Lily`'s exponential types give a more refined treatment of usage and strictness properties than does lifting $(-)_\bot$ in the domain model, because the latter happens to have an extra contraction property that we do not assume for !-types in `Lily`.

to linear lambda calculus, as we now explain.

Recall that two terms of a programming language are *contextually equivalent* if interchanging them in any complete program does not affect the observable results of evaluating the program. Even if we fix the operational semantics of the language, we may or may not get different notions of contextual equivalence depending upon what we decide constitutes a 'complete program' and what we observe of evaluation. Consider the classic example of call-by-name PCF. Taking programs to be closed terms of ground type and observing convergence to ground values, we get the notion of *ground contextual equivalence* studied in the seminal paper by Plotkin [22]; whereas if we observe termination of evaluation of closed terms of any type we get a different, *lazy contextual equivalence* analogous to that for the 'lazy' lambda calculus of Abramsky [1]. Lazy contextual equivalence for linear lambda calculi is studied in [5,4]. For `Lily` we work with ground, rather than lazy, contextual equivalence, in order for the definitions in Fig. 1 to be correct; for example, with lazy contextual equivalence the linear function type $\tau \multimap \tau'$ would not represent a domain of strict continuous functions $\tau \circ\!\!\!\rightarrow \tau'$, but rather its lift $(\tau \circ\!\!\!\rightarrow \tau')_\perp$.

Before the work presented here, very little was known about the properties of ground contextual equivalence for linear lambda calculus: see [4, Sect. 7]. For example it was not even known whether it is different from lazy contextual equivalence (we show that it is—see Example 3.7). Roughly speaking, what was lacking was a sufficiently powerful analysis of the properties of linear functions whose codomains are 'observable' types (ones at which ground contextual equivalence distinguishes divergent terms from ones in canonical form). We provide such an analysis, and much more besides. The contributions of this paper are as follows.

A Strictness Theorem. The conventional wisdom [2, p 16] is that operational semantics of linear lambda calculus is 'naturally' call-by-value in some parts (e.g. for $\multimap$, $\otimes$, and $\oplus$) and call-by-name in others (e.g. for $!$, $\rightarrow$, and $\times$). We show that the naturally call-by-value operators can be given a call-by-name semantics without affecting termination *at exponential types* (Theorem 2.3) and hence without affecting ground contextual equivalence. This technical result turns out to be the key to developing a rich operational theory for `Lily`.

A Parametric Logical Relation. Using the Strictness Theorem, we show that `Lily` ground contextual equivalence coincides with a particular logical relation (Theorem 3.2), involving parameters that are 'admissible' term-relations similar to those used previously by the second author [20,21]. As a corollary we obtain new extensionality results for linear lambda calculus (Corollary 3.5). The logical relation also allows us to prove results that validate the definitions in Fig. 1 and indicate the expressive power of `Lily`'s combination of linear lambda calculus with impredicative polymorphism and fixpoint recursion (see Remark 3.8, Example 3.9 and Remark 3.10).

A Common Intermediate Language for Strict and Lazy. From a programming language perspective `Lily` has the potential to be a common intermediate language for both strict and lazy functional programming. To realise that potential one would need to work at a more intensional level than we do in most of this paper (cf. [27]). However we do indicate how to replace the call-by-name semantics of terms involving exponential types in `Lily` (which subsumes fixpoint recursion) with a call-by-need semantics without affecting ground contextual equivalence (see Sect. 4). Details will appear elsewhere, along with an exploration of translations of the pure core of ML and Haskell into `Lily`.

2 A Strictness Theorem

`Lily` combines a term calculus for Plotkin's dual intuitionistic linear logic [3] with fixpoint recursion and impredicative polymorphism. In the presence of the latter it turns out that we can cut down to just linear function and exponential types without losing expressive power. The types and terms of the `Lily` language are given in Fig. 2, together with its type assignment relation. We find it convenient to employ a syntactic distinction between *intuitionistic variables* $x \in \mathit{IVar}$, that may be duplicated and discarded, and *linear variables* $a \in \mathit{LVar}$, that must be used exactly once. $\forall\alpha.(\_)$, $\lambda a : \tau.(\_)$, $\Lambda\alpha.(\_)$, $!(x = (\_) : \tau)$, and $\texttt{let}\ !x = M\ \texttt{in}\ (\_)$ are variable-binding constructs and we identify types and terms up to renaming of bound variables. The notations $\mathit{ftv}(\_)$, $\mathit{fiv}(\_)$, $\mathit{flv}(\_)$ are used to denote the sets of free type variables, free intuitionistic variables and free linear variables of an expression. We use the notation $-[-/-]$ to indicate capture-avoiding substitution.

Recursively defined thunks

A syntactic novelty of `Lily` is the absorption of fixpoint recursion into the terms of exponential type. A *recursively defined thunk* $!(x = M : \tau)$ is a canonical form introducing a recursively defined non-linear term as a suspended computation. The corresponding eliminator $\texttt{let}\ !x = M\ \texttt{in}\ M'$ evaluates the term M to such a recursive thunk, substitutes an unfolding of the thunk's body for the intuitionistic variable x and evaluates the body M' (see Fig. 3 below; and see Sect. 4 for an alternative, call-by-need semantics). Since a thunk's body may be duplicated during the evaluation of the `let` construct, potentially duplicating linear variables, the typing rules ensure that thunks contain no free linear variables. We can get conventional fixpoint terms by defining

(1) $$\texttt{fix}\ x : \tau.\, M \triangleq \texttt{let}\ !x = !(x = M : \tau)\ \texttt{in}\ M$$

and non-recursive thunks (of type $!\tau$) by defining

(2) $$!M \triangleq\ !(x = M : \tau) \qquad (x \notin \mathit{fiv}(M)).$$

Types $\tau ::=$	α	type variable ($\alpha \in \mathit{TyVar}$)
	$\tau \multimap \tau$	linear function type
	$\forall \alpha.\, \tau$	$\forall$-type
	$!\tau$	exponential type
Terms $M ::=$	a	linear variable ($a \in \mathit{LVar}$)
	x	intuitionistic variable ($x \in \mathit{IVar}$)
	$\lambda a : \tau.\, M$	abstraction
	$M\, M$	application
	$\Lambda \alpha.\, M$	generalisation
	$M\, \tau$	specialisation
	$!(x = M : \tau)$	recursively defined thunk
	$\texttt{let } !x = M_1 \texttt{ in } M_2$	exponential eliminator

Type assignment relation $\Gamma\,;\Delta \vdash_{\overline{\alpha}} M : \tau$ is inductively generated by the following rules. (Γ is a finite function from intuitionistic variables to types with domain $\mathit{dom}(\Gamma)$; Δ is a finite function from linear variables to types with domain $\mathit{dom}(\Delta)$, and $\overline{\alpha}$ is a finite set of type variables.)

$$\frac{\mathit{ftv}(\Gamma,\tau) \subseteq \overline{\alpha} \quad x \notin \mathit{dom}(\Gamma)}{\Gamma, x : \tau\,; \emptyset \vdash_{\overline{\alpha}} x : \tau} \qquad \frac{\mathit{ftv}(\Gamma,\tau) \subseteq \overline{\alpha}}{\Gamma\,; a : \tau \vdash_{\overline{\alpha}} a : \tau}$$

$$\frac{\Gamma\,; \Delta, a : \tau \vdash_{\overline{\alpha}} M : \tau' \quad a \notin \mathit{dom}(\Delta)}{\Gamma\,; \Delta \vdash_{\overline{\alpha}} \lambda a : \tau.\, M : \tau \multimap \tau'}$$

$$\frac{\Gamma\,; \Delta_1 \vdash_{\overline{\alpha}} M_1 : \tau \multimap \tau' \quad \Gamma\,; \Delta_2 \vdash_{\overline{\alpha}} M_2 : \tau \quad \mathit{dom}(\Delta_1) \cap \mathit{dom}(\Delta_2) = \emptyset}{\Gamma\,; \Delta_1, \Delta_2 \vdash_{\overline{\alpha}} M_1\, M_2 : \tau'}$$

$$\frac{\Gamma\,; \Delta \vdash_{\overline{\alpha},\alpha} M : \tau \quad \alpha \notin \overline{\alpha} \cup \mathit{ftv}(\Gamma,\Delta)}{\Gamma\,; \Delta \vdash_{\overline{\alpha}} \Lambda\alpha.\, M : \forall\alpha.\,\tau} \qquad \frac{\Gamma\,; \Delta \vdash_{\overline{\alpha}} M : \forall\alpha.\,\tau \quad \mathit{ftv}(\tau') \subseteq \overline{\alpha}}{\Gamma\,; \Delta \vdash_{\overline{\alpha}} M\,\tau' : \tau[\tau'/\alpha]}$$

$$\frac{\Gamma, x : \tau\,; \emptyset \vdash_{\overline{\alpha}} M : \tau \quad x \notin \mathit{dom}(\Gamma)}{\Gamma\,; \emptyset \vdash_{\overline{\alpha}} !(x = M : \tau) : !\tau}$$

$$\frac{\begin{array}{ll} \Gamma\,; \Delta_1 \vdash_{\overline{\alpha}} M_1 : !\tau & \Gamma, x : \tau\,; \Delta_2 \vdash_{\overline{\alpha}} M_2 : \tau' \\ x \notin \mathit{dom}(\Gamma) & \mathit{dom}(\Delta_1) \cap \mathit{dom}(\Delta_2) = \emptyset \end{array}}{\Gamma\,; \Delta_1, \Delta_2 \vdash_{\overline{\alpha}} \texttt{let } !x = M_1 \texttt{ in } M_2 : \tau'}$$

Fig. 2. `Lily` syntax and type assignment

Conversely, recursively defined thunks could have been expressed in terms of non-recursive thunks and fixpoints, by taking $!(x = M : \tau)$ to be $!(\texttt{fix}\ x : \tau.\, M)$ (see Corollary 3.6). There is perhaps not much to choose between the two formulations. We prefer the compactness of the one we have presented. It easily generalises to *mutually recursive* thunks $!(x_1 = M_1 : \tau_1, \ldots, x_n = M_n : \tau_n)$ whose type $!(\tau_1, \ldots, \tau_n)$ is equivalent to $!\tau_1 \otimes \cdots \otimes !\tau_n$.

Ground contextual equivalence

Recall from the Introduction that we wish to identify terms if they give the same evaluation behaviour in all contexts of ground type, such as types of booleans and natural numbers. But there are no such types in Lily as we have defined it! In fact this *ground contextual equivalence* is the same as the contextual equivalence determined by observing the termination properties of evaluation at exponential types $!\tau$ (and then the ground types are definable, modulo ground contextual equivalence, as in Fig. 1). To see this, consider adding to Lily a type bool together with truth-values and conditionals

$$\frac{}{\Gamma\,;\emptyset \vdash_{\overline{\alpha}} \texttt{true} : \texttt{bool}} \qquad \frac{}{\Gamma\,;\emptyset \vdash_{\overline{\alpha}} \texttt{false} : \texttt{bool}}$$

$$\frac{\begin{array}{ll}\Gamma\,;\Delta_1 \vdash_{\overline{\alpha}} M_1 : \texttt{bool} & \mathit{dom}(\Delta_1) \cap \mathit{dom}(\Delta_2) = \emptyset \\ \Gamma\,;\Delta_2 \vdash_{\overline{\alpha}} M_2 : \tau & \Gamma\,;\Delta_2 \vdash_{\overline{\alpha}} M_3 : \tau\end{array}}{\Gamma\,;\Delta_1, \Delta_2 \vdash_{\overline{\alpha}} \texttt{if}\ M_1\ \texttt{then}\ M_2\ \texttt{else}\ M_3 : \tau}$$

Then for any closed term M of exponential type $!\tau$, $\texttt{let}\ !x = M\ \texttt{in}\ \texttt{true}$ is a closed term of type bool that evaluates to true if and only if M evaluates (in the operational semantics to be described below) to some thunk. Conversely, for any closed term B of type bool, B evaluates to true if and only if the closed term $\texttt{if}\ B\ \texttt{then}\ !(x = x : \tau)\ \texttt{else}\ \Omega\,(!\tau)$ of type $!\tau$ evaluates to some thunk. Here Ω is a generic divergent term

$$\Omega \triangleq \texttt{let}\ !x = !(x = x : \forall \alpha.\, \alpha)\ \texttt{in}\ x \tag{3}$$

whose type is $\forall \alpha.\, \alpha$. (Using the macro in equation (1) for fixpoint expressions, this takes on the more familiar form $\Omega = \texttt{fix}\ x : (\forall \alpha.\, \alpha).\, x$.) Thus contextual equivalence based upon observing convergence to true at type bool is the same as that based upon observing convergence to canonical form at exponential types. (This is different from observing convergence at *all* types: see Example 3.7 below.)

When defining a contextual equivalence for *linear* lambda calculi one has to refine the traditional formulation, in which 'holes' in contexts have implicit parameters (namely the binding variables within whose scope the hole lies), since it matters whether these parameters are linear or not. The first author discusses one such refinement in [4, Sect. 3.1] using second-order variables to give a more explicit treatment of holes (see also [19,5,15]). An attractive alternative (because it doesn't require the introduction of extra syntactic machinery) is to avoid the use of contexts completely and define the equivalence to be the largest substitutive congruence relation on well-typed open terms having the required convergence property for closed terms of exponential type. This 'relational' approach to contextual equivalence is taken in [10,21] and will be used in the full version of this paper. However, in order to simplify the exposition in this extended abstract, we will restrict attention to ground contextual equivalence of *closed* terms (of closed types), for which we can side-step these issues about contexts and use the following definition. It depends upon the notion of a closed term M of closed exponential type converging to (some) canonical from, which we write as $M \Downarrow_!$ and define below (see Corollary 2.4).

Definition 2.1 (Ground contextual equivalence for closed terms) Let *Typ* denote the set $\{\tau \mid \mathit{ftv}(\tau) = \emptyset\}$ of *closed* types; and for $\tau \in \mathit{Typ}$, let $\mathit{Term}(\tau)$ denote the set $\{M \mid \emptyset\,;\emptyset \vdash_\emptyset M : \tau\}$ of *closed* terms of type τ. Given $M, M' \in \mathit{Term}(\tau)$, we write $M =_{\text{gnd}} M' : \tau$ to mean:

- for any x, N, τ' satisfying $x : \tau\,; \emptyset \vdash_\emptyset N : !\tau'$, it is the case that $N[M/x] \Downarrow_!$ if and only if $N[M'/x] \Downarrow_!$; and
- for any a, N, τ' satisfying $\emptyset\,; a : \tau \vdash_\emptyset N : !\tau'$, it is the case that $N[M/a] \Downarrow_!$ if and only if $N[M'/a] \Downarrow_!$.

Operational semantics

Figure 3 gives two possible evaluation relations for closed `Lily` terms, differing in their treatment of the application of a linear function to an argument. The strict (or call-by-value) relation $\Downarrow_s$ corresponds to the 'natural' operational interpretation of intuitionistic linear logic advocated by Abramsky [2, p 16] and used by others (such as [27,4]); whereas the other relation, $\Downarrow_n$, used for a linear lambda calculus by Crole in [5], gives *all* constructs a non-strict (or call-by-name) semantics. The two relations give rise to two termination relations

$$\begin{array}{ll} \textit{strict} \text{ termination relation:} & M \Downarrow_s \triangleq \exists\, V.\, M \Downarrow_s V \\ \textit{non-strict} \text{ termination relation:} & M \Downarrow_n \triangleq \exists\, V.\, M \Downarrow_n V. \end{array}$$

These are different relations, as the following simple example shows.

Example 2.2 Choose any closed types $\tau, \tau' \in \mathit{Typ}$. Define

$$E \triangleq \lambda a : \tau.\, \lambda f : \tau \multimap \tau'.\, f\, a.$$

Then $E\,(\Omega\,\tau) \in \mathit{Term}((\tau \multimap \tau') \multimap \tau')$ and $E\,(\Omega\,\tau) \Downarrow_n$ but $E\,(\Omega\,\tau) \not\Downarrow_s$. (Here Ω is the generic divergent term defined at (3).)

However, as the following slightly surprising theorem shows, the two termination relations *do* coincide *if we restrict our attention to terms of exponential type.*

Theorem 2.3 (Strictness Theorem) *For all $\tau, \tau' \in \mathit{Typ}$, $M \in \mathit{Term}(\tau)$, and open terms of exponential type, $\emptyset\,; a : \tau \vdash_\emptyset N : !\tau'$*

$$N[M/a] \Downarrow_n \Leftrightarrow \exists\, V.\, M \Downarrow_s V \;\&\; N[V/a] \Downarrow_n.$$

Common rules:

$$\frac{}{\lambda a : \tau.\, M \Downarrow \lambda a : \tau.\, M} \qquad \frac{}{\Lambda \alpha.\, M \Downarrow \Lambda \alpha.\, M} \qquad \frac{}{!(x = M : \tau) \Downarrow !(x = M : \tau)}$$

$$\frac{M \Downarrow \Lambda \alpha.\, M' \quad M'[\tau/\alpha] \Downarrow V}{M\,\tau \Downarrow V}$$

$$\frac{M_1 \Downarrow !(x = M : \tau) \quad M_2[(\texttt{let}\ !x = !(x = M : \tau)\ \texttt{in}\ M)/y] \Downarrow V}{\texttt{let}\ !y = M_1\ \texttt{in}\ M_2 \Downarrow V}$$

Strict evaluation, $M \Downarrow_s V$, is inductively generated by the common rules plus

$$\frac{M_1 \Downarrow_s \lambda a : \tau.\, M \quad M_2 \Downarrow_s V \quad M[V/a] \Downarrow_s V'}{M_1\, M_2 \Downarrow_s V'}$$

Non-strict evaluation, $M \Downarrow_n V$, is inductively generated by the common rules plus

$$\frac{M_1 \Downarrow_n \lambda a : \tau.\, M \quad M[M_2/a] \Downarrow_n V}{M_1\, M_2 \Downarrow_n V}$$

Fig. 3. Evaluating closed `Lily` terms

Hence in particular for $M \in Term(!\tau)$, $M \Downarrow_n \Leftrightarrow M \Downarrow_s$.

Proof (sketch) The intuition for why the theorem holds is that if $N[M/a]$ converges (under either semantics), then its canonical form must be a thunk of type $!\tau'$, and this thunk must be a residual of one of the original thunks in $N[M/a]$. Since none of those thunks can mention a linear variable (by the typing rule for thunks), the residual thunk cannot suspend any linear arguments, so each linear argument within $N[M/a]$ must have been evaluated before reaching the canonical form. Thus, when evaluating to a thunk, it makes no difference to termination behaviour whether we choose to postpone or force the evaluation of function arguments, since, in either case, the arguments must be evaluated before reaching the canonical form. That's the intuition, but we found it surprisingly hard to give a formal proof. In this extended abstract we merely sketch the structure of our proof. Among the terms N having a free linear variable a, we single out those for which a occurs in a position where it will be immediately evaluated in the non-strict semantics. The structure of these *evaluation contexts* can be analysed as a nested stack $F[a] = F_1[F_2[\cdots F_n[a] \cdots]]$ of 'frames' $F_i[-]$ of the form $(-\, M)$, $(-\, \tau)$, or $(\texttt{let}\ !x = -\ \texttt{in}\ M)$. The advantage of this *frame stack* formulation is that it permits us to give a direct inductive definition of the non-strict termination relation $F[M] \Downarrow_n$ that follows the syntactical structure of the frame stack F and the term M. Arguing by structural induction for this relation, we prove that for $F[N]$ of exponential type and containing a free linear variable a it is the case that

$$(F[N])[M/a] \Downarrow_n \Rightarrow \exists\, V.\, M \Downarrow_s V \ \&\ (F[N])[V/a] \Downarrow_n .$$

(There are two subcases, proved simultaneously, according to whether a occurs in F, or in N. It is of course crucial that we are restricting attention to terms $F[N]$ of exponential type; for example, a base case of the induction is when F is the empty frame stack and N is in canonical form, of exponential type, and hence necessarily not involving a, rendering this case trivial.) Taking F to be the empty frame stack in the above implication, we obtain the left-to-right half of the theorem. The other half of the theorem we deduce from a result of independent interest, namely a version of the Mason-Talcott 'ciu' theorem [12] for `Lily` ground contextual equivalence. For $\tau \in \mathit{Typ}$ and $M, M' \in \mathit{Term}(\tau)$ define *ciu-equivalence*, $M =_{\mathrm{ciu}} M' : \tau$, to mean that for all closed frame stacks F mapping from τ to an exponential type, we have $F[M] \Downarrow_n \Leftrightarrow F[M'] \Downarrow_n$. Then it is the case that $=_{\mathrm{ciu}}$ *coincides with* $=_{\mathrm{gnd}}$ *and hence in particular is a congruence.* (There are by now a number of means for establishing this kind of result; we prefer one that is an adaptation of the method of proving congruence due to Howe [9].) Using the congruence property of $=_{\mathrm{ciu}}$ it is simple enough to prove

$$M \Downarrow_s V \Rightarrow N[M/a] =_{\mathrm{ciu}} N[V/a]$$

by induction on the derivation of $M \Downarrow_s V$. The right-to-left half of the theorem follows from this and the definition of $=_{\mathrm{ciu}}$. □

Corollary 2.4 *If we take the notion of convergence at exponential type,* $M \Downarrow_!$, *used in the definition of ground contextual equivalence (Definition 2.1) to be either* $M \Downarrow_s$ *or* $M \Downarrow_n$, *we get the same equivalence relation on* `Lily` *terms.*

Remark 2.5 ('Computational' types) Do other types apart from exponentials enjoy the strictness property of Theorem 2.3? Call a type κ 'computational' if for all $\tau \in \mathit{Typ}$, $M \in \mathit{Term}(\tau)$ and open terms $\emptyset\,; a : \tau \vdash_{\emptyset} N : \kappa$ it is the case that

$$N[M/a] \Downarrow_n \Leftrightarrow \exists\, V.\, M \Downarrow_s V \;\&\; N[V/a] \Downarrow_n.$$

Thus the theorem says that $!\tau'$ is computational, for any $\tau' \in \mathit{Typ}$. Using the definitions in Fig. 1, we conjecture that all closed types in the grammar given by

$$\begin{aligned} \kappa \ ::= \ & \alpha \mid !\tau \mid \kappa \otimes \kappa \mid \kappa \oplus \kappa \mid \tau + \tau \mid \mathsf{T} \mid \mathsf{N}_{\perp} \\ \mid \ & \mu\alpha.\, \kappa(\alpha) \mid \nu\alpha.\, \kappa(\alpha) \mid \mathtt{rec}\,\alpha.\, \kappa(\alpha, \alpha) \end{aligned}$$

are computational.

3 A Parametric Logical Relation

We are going to show that ground contextual equivalence of `Lily` terms, $=_{\mathrm{gnd}}$ (Definition 2.1), coincides with a certain logical relation that, by construction, has various extensionality and parametricity properties that we wish to establish for $=_{\mathrm{gnd}}$. The following definition gives some operations on binary relations between `Lily` terms that we need to achieve this.

Definition 3.1 For each closed type $\tau \in \mathit{Typ}$, let $\mathit{Test}(\tau)$ denote the set of closed linear function abstractions $\lambda a : \tau.\, M$ of type $\tau \multimap\, !\tau'$ for some $\tau' \in \mathit{Typ}$. For $\tau, \tau' \in \mathit{Typ}$, define the set of *term-relations* to be

$$\mathit{Rel}(\tau, \tau') \triangleq \{\, r \mid r \subseteq \mathit{Term}(\tau) \times \mathit{Term}(\tau')\,\}$$

and the set of *test-relations* to be

$$\mathit{Rel}^*(\tau, \tau') \triangleq \{\, s \mid s \subseteq \mathit{Test}(\tau) \times \mathit{Test}(\tau')\,\}\,.$$

We define the following operations on term-relations and test-relations:

- Given $r \in \mathit{Rel}(\tau, \tau')$, define $r^\top \in \mathit{Rel}^*(\tau, \tau')$ to be
$$r^\top \triangleq \{\, (V, V') \mid \forall (M, M') \in r.\, V\, M \Downarrow_! \Leftrightarrow V'\, M' \Downarrow_! \,\}\,.$$
- Given $s \in \mathit{Rel}^*(\tau, \tau')$, define $s^\top \in \mathit{Rel}(\tau, \tau')$ to be
$$s^\top \triangleq \{\, (M, M') \mid \forall (V, V') \in s.\, V\, M \Downarrow_! \Leftrightarrow V'\, M' \Downarrow_! \,\}\,.$$
- Given $r_1 \in \mathit{Rel}(\tau_1, \tau_1')$ and $r_2 \in \mathit{Rel}(\tau_2, \tau_2')$, define $r_1 \multimap r_2 \in \mathit{Rel}(\tau_1 \multimap \tau_2, \tau_1' \multimap \tau_2')$ to be
$$r_1 \multimap r_2 \triangleq \{\, (M, M') \mid \forall (M_1, M_1') \in r_1.\, (M\, M_1, M'\, M_1') \in r_2 \,\}\,.$$
- Given a family $(R(r) \in \mathit{Rel}(\tau[\sigma/\alpha], \tau'[\sigma'/\alpha']) \mid \sigma, \sigma' \in \mathit{Typ}, r \in \mathit{Rel}(\sigma, \sigma'))$ of term-relations, define $\forall r.\, R(r) \in \mathit{Rel}(\forall \alpha.\, \tau, \forall \alpha'.\, \tau')$ to be
$$\forall r.\, R(r) \triangleq \{\, (M, M') \mid \forall \sigma, \sigma' \in \mathit{Typ}, r \in \mathit{Rel}(\sigma, \sigma').\, (M\, \sigma, M'\, \sigma') \in R(r) \,\}\,.$$
- Given $r \in \mathit{Rel}(\tau, \tau')$, define $!r \in \mathit{Rel}(!\tau, !\tau')$ to be
$$!r \triangleq \{\, (!(x = M : \tau), !(x' = M' : \tau')) \mid (\mathtt{fix}\ x : \tau.\, M, \mathtt{fix}\ x' : \tau'.\, M') \in r \,\}\,.$$
(Fixpoint terms such as $\mathtt{fix}\ x : \tau.\, M$ were defined in (1).)

The operation $r \mapsto r^{\top\top}$ derived from the above definition is a closure operation on term-relations whose fixed points $r = r^{\top\top}$ turn out to have good properties (they respect $=_{\mathrm{gnd}}$ and are suitable for a syntactic version of fixpoint induction) that we exploit to get the following theorem. We omit its proof in this extended abstract, because it is quite involved; although the structure of the proof is similar to [21, Sect. 4] the details are different. [2]

Theorem 3.2 (Relational parametricity for $=_{\mathrm{gnd}}$) *For each* `Lily` *type τ and each list $\vec{\alpha} = \alpha_1, \ldots, \alpha_n$ of distinct type variables containing the free type variables of τ, we define a function from tuples of term-relations, $\vec{r} = r_1, \ldots, r_n$, to term-relations*

$$r_1 \in \mathit{Rel}(\tau_1, \tau_1'), \ldots, r_n \in \mathit{Rel}(\tau_n, \tau_n') \mapsto \Delta_\tau(\vec{r}/\vec{\alpha}) \in \mathit{Rel}(\tau[\vec{\tau}/\vec{\alpha}], \tau[\vec{\tau}'/\vec{\alpha}'])$$

by induction on the structure of τ using the operations of Definition 3.1, as follows:

$$\Delta_{\alpha_i}(\vec{r}/\vec{\alpha}) \triangleq r_i$$

[2] For one thing, the Strictness Theorem 2.3 is needed in several places; for another, the use of linear function abstractions rather than evaluation contexts as 'tests' in Definition 3.1 means we have to work harder to prove the theorem—the reward being a richer collection of $\top\top$-closed relations and hence a better ability to prove properties of $=_{\mathrm{gnd}}$.

$$\begin{aligned}
\Delta_{\tau_1 \multimap \tau_2}(\vec{r}/\vec{\alpha}) &\triangleq \Delta_{\tau_1}(\vec{r}/\vec{\alpha}) \multimap \Delta_{\tau_2}(\vec{r}/\vec{\alpha})\\
\Delta_{\forall\alpha.\,\tau}(\vec{r}/\vec{\alpha}) &\triangleq \forall r.\, \Delta_\tau(\vec{r}, r^{\top\top}/\vec{\alpha}, \alpha)\\
\Delta_{!\tau}(\vec{r}/\vec{\alpha}) &\triangleq (!\Delta_\tau(\vec{r}/\vec{\alpha}))^{\top\top}.
\end{aligned}$$

When τ is closed, we can take $\vec{\alpha}$ and $\vec{r}$ to be empty and get $\Delta_\tau \triangleq \Delta_\tau(\emptyset/\emptyset) \in Rel(\tau,\tau)$. Then for all $M, M' \in Term(\tau)$

$$(M, M') \in \Delta_\tau \Leftrightarrow M =_{\text{gnd}} M' : \tau.$$

As part of the proof of the above theorem, one needs to establish the following technical property of the parametric logical relation which we state separately because it is useful in its own right.

Lemma 3.3 *For each* `Lily` *type τ, with free type variables in $\vec{\alpha}$ say, if each term-relation r in $\vec{r}$ satisfies $r = r^{\top\top}$, then so does $\Delta_\tau(\vec{r}/\vec{\alpha})$.*

Corollary 3.4 (Kleene equivalences) *For $\tau \in Typ$, write $Val(\tau)$ for the subset of $Term(\tau)$ consisting of the closed terms in canonical form—the terms which appear of the right-hand side of the evaluation relations in Fig. 3, namely abstractions, generalisations and recursively defined thunks (cf. Fig. 2). If $M, M' \in Term(\tau)$ are* Kleene equivalent, *i.e. satisfy $\forall V \in Val(\tau).\, M \Downarrow_n V \Leftrightarrow M' \Downarrow_n V$, then $M =_{\text{gnd}} M' : \tau$. (Similarly for $\Downarrow_s$.) Hence*

$$\begin{aligned}
(\lambda a : \tau.\, M)\, N &=_{\text{gnd}} M[N/a] : \tau'\\
(\Lambda\alpha.\, M)\, \sigma &=_{\text{gnd}} M[\sigma/\alpha] : \tau[\sigma/\alpha]\\
\texttt{let } !y \texttt{ = } !(x \texttt{ = } N : \tau) \texttt{ in } M &=_{\text{gnd}} M[(\texttt{fix } x : \tau.\, N)/y] : \tau'.
\end{aligned}$$

Moreover, if $M, M' \in Term(\tau)$ are doth divergent (say $M \not\Downarrow_s$ and $M' \not\Downarrow_s$), then $M =_{\text{gnd}} M' : \tau$.

Proof. It follows from Lemma 3.3 and the definition of $(-)^\top$ that $\Delta_\tau(\vec{r}/\vec{\alpha})$ respects Kleene equivalence. Hence so does $=_{\text{gnd}}$; and being reflexive, this implies that it also contains Kleene equivalence. □

Corollary 3.5 (Extensionality properties of $=_{\text{gnd}}$)

(4) $M =_{\text{gnd}} M' : \tau \multimap \tau' \Leftrightarrow \forall V \in Val(\tau).\, M\, V =_{\text{gnd}} M'\, V : \tau'$

(5) $M =_{\text{gnd}} M' : \forall\alpha.\,\tau \Leftrightarrow \forall \sigma \in Typ.\, M\,\sigma =_{\text{gnd}} M'\,\sigma : \tau[\sigma/\alpha]$

(6) $$\begin{aligned} M =_{\text{gnd}} M' : !\tau \Leftrightarrow\ & (M \not\Downarrow_! \,\&\, M' \not\Downarrow_!) \vee \exists\, x, N, x', N'.\\ & M \Downarrow_s !(x \texttt{ = } N : \tau)\ \&\\ & M' \Downarrow_s !(x' \texttt{ = } N' : \tau)\ \&\\ & \texttt{fix } x : \tau.\, N =_{\text{gnd}} \texttt{fix } x' : \tau.\, N' : \tau \end{aligned}$$

Proof. These properties follow by combining Theorem 3.2 with Lemma 3.3, the definition of Δ, and the Strictness Theorem 2.3. For example, to prove (4) first observe that

(7) $M =_{\text{gnd}} M' : \tau \multimap \tau' \Leftrightarrow \forall N \in Term(\tau).(M\,N, M'\,N) \in \Delta_{\tau'}$

holds by a standard argument for such logical relations, using the facts that $=_{\mathrm{gnd}}$ coincides with Δ and that $\Delta_{\tau \multimap \tau'} = \Delta_\tau \multimap \Delta_{\tau'}$. By Lemma 3.3, $\Delta_{\tau'} = (\Delta_{\tau'})^{\top\top}$ and so the right-hand side of equation (7) is equivalent to

$$\forall N \in \mathit{Term}(\tau).\, \forall (F, F') \in (\Delta_{\tau'})^{\top}.\, F\,(M\,N) \Downarrow_! \Leftrightarrow F'\,(M'\,N) \Downarrow_! \ . \tag{8}$$

By the Strictness Theorem 2.3 (and Corollary 2.4)

$$F\,(M\,N) \Downarrow_! \ \Leftrightarrow \ \exists\, V.\, N \Downarrow_s V \ \&\ F\,(M\,V) \Downarrow_!$$

and similarly for F'. Therefore (8) is equivalent to

$$\forall V \in \mathit{Val}(\tau).\, \forall (F, F') \in (\Delta_{\tau'})^{\top}.\, F\,(M\,V) \Downarrow_! \Leftrightarrow F'\,(M'\,V) \Downarrow_!$$

i.e. to $\forall V \in \mathit{Val}(\tau).(M\,V, M'\,V) \in (\Delta_{\tau'})^{\top\top}$, i.e. to $\forall V \in \mathit{Val}(\tau).(M\,V, M'\,V) \in \Delta_{\tau'}$. So replacing the right-hand side of equation (7) with this and applying Theorem 3.2 again, we get the desired extensionality property (4) of terms of linear function type. □

Corollary 3.6 *Any recursively defined thunk, $!(x = M : \tau) \in \mathit{Val}(!\tau)$, can be expressed as a non-recursive thunk (2) of a fixpoint term (1) up to ground contextual equivalence:*

$$!(x = M : \tau) =_{\mathrm{gnd}} !(\mathtt{fix}\ x : \tau.\, M) : !\tau. \tag{9}$$

So in particular every element of $\mathit{Val}(!\tau)$ is ground contextually equivalent to $!N$ for some $N \in \mathit{Term}(\tau)$.

Proof. By the extensionality property (6) in Corollary 3.5, equation (9) holds if

$$\mathtt{fix}\ x' : \tau.\, \mathtt{fix}\ x : \tau.\, M =_{\mathrm{gnd}} \mathtt{fix}\ x : \tau.\, M : \tau$$

where $x' \notin \mathit{fiv}(\mathtt{fix}\ x : \tau.\, M)$; but this does hold, by unfolding the left-hand side using the last Kleene equivalence in Corollary 3.4:

$$\begin{aligned} \mathtt{fix}\ x' : \tau.\, \mathtt{fix}\ x : \tau.\, M &=_{\mathrm{gnd}} (\mathtt{fix}\ x : \tau.\, M)[(\mathtt{fix}\ x' : \tau.\, \mathtt{fix}\ x : \tau.\, M)/x'] \\ &= \quad \mathtt{fix}\ x : \tau.\, M. \end{aligned}$$

□

Example 3.7 (Ground and lazy contextual equivalences differ) Consider the generic divergent term Ω (see equation (3)) and the generic family of divergent terms Ω':

$$\Omega \triangleq \mathtt{fix}\ x : (\forall \alpha.\, \alpha).\, x \qquad \Omega' \triangleq \Lambda \alpha.\, \mathtt{fix}\ x : \alpha.\, x.$$

These are both closed terms of type $\forall \alpha.\, \alpha$. For any $\tau \in \mathit{Typ}$ it is not hard to see that $\Omega\,\tau \not\Downarrow_s$ and $\Omega'\,\tau \not\Downarrow_s$. Hence $\Omega\,\tau =_{\mathrm{gnd}} \Omega'\,\tau : \tau$, by Corollary 3.4. Therefore by the extensionality property (5) in Corollary 3.5 we have $\Omega =_{\mathrm{gnd}} \Omega' : \forall \alpha.\, \alpha$. However, these two terms are evidently not equated by *lazy* contextual equivalence, where one observes convergence in contexts of *all* types, not just of exponential types; for Ω' is in canonical form whereas Ω diverges, so that we can observe a difference between them using the identity context. Similar examples can be given using function types and property (4) rather than using $\forall$-types, thus settling an open problem in [4, Sect. 7].

Remark 3.8 (Relational parametricity for $\forall$-types) As a consequence of Theorem 3.2, terms of $\forall$-types enjoy relational parametricity properties modulo ground contextual equivalence. For since any $M \in \mathit{Term}(\forall\alpha.\,\tau)$ satisfies $M =_{\mathrm{gnd}} M : \forall\alpha.\,\tau$, by the theorem we have $(M, M) \in \Delta_{\forall\alpha.\,\tau}$. Hence from Definition 3.1 we get that for any $\sigma, \sigma' \in \mathit{Typ}$ and $r \in \mathit{Rel}(\sigma, \sigma')$ satisfying $r = r^{\top\top}$, it is the case that $(M\,\sigma, M\,\sigma') \in \Delta_\tau(r/\alpha)$. Then one can use the definition of $\Delta_\tau(r/\alpha)$ to infer properties of M. Of course, to use this method one needs a rich source of term-relations satisfying $r = r^{\top\top}$. Such a source arises from the fact that the graph $\{(M, M') \mid F\,M =_{\mathrm{gnd}} M' : \sigma'\}$ of any linear function $F \in \mathit{Term}(\sigma \multimap \sigma')$ is such a term-relation. This allows one to establish many 'free theorems' [28] to do with (di)naturality properties of the `Lily` type constructors with respect to linear functions (which play the role in this theory that strict continuous functions do in domain theory). Indeed, if we make definitions of types as in Fig. 1, the expected category-theoretic properties of these types can be established up to ground contextual equivalence. Example 3.9 shows this for coalesced sums; the categorical properties of the other type constructors in Fig. 1 will be treated in the full version of this paper.

Example 3.9 (Categorical coproducts) Consider the category whose objects are closed `Lily` types, $\tau \in \mathit{Typ}$, and whose morphisms from τ to τ' are ground contextual equivalence classes of closed `Lily` terms of type $\tau \multimap \tau'$. The composition of morphisms represented by $M \in \mathit{Term}(\tau \multimap \tau')$ and $M' \in \mathit{Term}(\tau' \multimap \tau'')$ is the morphism represented by $M' \circ M \in \mathit{Term}(\tau \multimap \tau'')$, where

$$M' \circ M \triangleq \lambda a : \tau.\, M'\,(M\,a).$$

The identity morphism for τ is represented by $\mathit{Id}_\tau \in \mathit{Term}(\tau \multimap \tau)$, where

$$\mathit{Id}_\tau \triangleq \lambda a : \tau.\, a.$$

(The validity of β-conversion for ground contextual equivalence (Corollary 3.4) and the extensionality property (4) in Corollary 3.5 are needed to see that these definitions do yield a category.) For closed types $\tau_1, \tau_2 \in \mathit{Typ}$, we claim that

$$\tau_1 \oplus \tau_2 \triangleq \forall\alpha.\, !(\tau_1 \multimap \alpha) \multimap !(\tau_2 \multimap \alpha) \multimap \alpha \tag{10}$$

is the coproduct of τ_1 *and* τ_2 *in this category*, with coproduct injections represented by $\mathit{Inl} \in \mathit{Term}(\tau_1 \multimap \tau_1 \oplus \tau_2)$ and $\mathit{Inr} \in \mathit{Term}(\tau_2 \multimap \tau_1 \oplus \tau_2)$, where

$$\begin{aligned}\mathit{Inl} \triangleq\ & \lambda a_1 : \tau_1.\, \Lambda\alpha.\, \lambda b_1 : !(\tau_1 \multimap \alpha).\, \lambda b_2 : !(\tau_2 \multimap \alpha).\\ & \qquad \texttt{let } !x_1 \texttt{ = } b_1 \texttt{ in let } !x_2 \texttt{ = } b_2 \texttt{ in } x_1\,a_1\end{aligned}$$

$$\begin{aligned}\mathit{Inr} \triangleq\ & \lambda a_2 : \tau_2.\, \Lambda\alpha.\, \lambda b_1 : !(\tau_1 \multimap \alpha).\, \lambda b_2 : !(\tau_2 \multimap \alpha).\\ & \qquad \texttt{let } !x_1 \texttt{ = } b_1 \texttt{ in let } !x_2 \texttt{ = } b_2 \texttt{ in } x_2\,a_2.\end{aligned}$$

To establish this claim, first note that given any object $\tau \in \mathit{Typ}$ and any morphisms represented by $F_i \in \mathit{Term}(\tau_i \multimap \tau)$ $(i = 1, 2)$, then

$$[F_1, F_2] \triangleq \lambda a : \tau_1 \oplus \tau_2.\, a\,\tau\,(!F_1)(!F_2)$$

represents a morphism from $\tau_1 \oplus \tau_2$ to τ satisfying

$$\begin{cases} [F_1, F_2] \circ \mathit{Inl} =_{\mathrm{gnd}} F_1 : \tau_1 \multimap \tau \\ [F_1, F_2] \circ \mathit{Inr} =_{\mathrm{gnd}} F_2 : \tau_2 \multimap \tau \end{cases} \tag{11}$$

(using Corollaries 3.4 and 3.5). So we just have to see that $[F_1, F_2]$ is the unique such morphism. It is now that we use the relational parametricity properties of $\forall$-types mentioned in Remark 3.8. We can show, for any $G \in \mathit{Term}(\tau_1 \oplus \tau_2 \multimap \tau)$ and $M \in \mathit{Term}(\tau_1 \oplus \tau_2)$, that

$$G(M(\tau_1 \oplus \tau_2)(!\mathit{Inl})(!\mathit{Inr})) =_{\mathrm{gnd}} M\,\tau\,!(G \circ \mathit{Inl})\,!(G \circ \mathit{Inr}) : \tau \tag{12}$$

(where we are using the notation for non-recursive thunks introduced in equation (2)). Postponing the proof of this naturality property for a moment, let us see how it yields the required uniqueness property of $\tau_1 \oplus \tau_2$, namely

$$(G_1 \circ \mathit{Inl} =_{\mathrm{gnd}} G_2 \circ \mathit{Inl} : \tau_1 \multimap \tau) \,\&\, (G_1 \circ \mathit{Inr} =_{\mathrm{gnd}} G_2 \circ \mathit{Inr} : \tau_2 \multimap \tau) \\ \Rightarrow (G_1 =_{\mathrm{gnd}} G_2 : \tau_1 \oplus \tau_2 \multimap \tau). \tag{13}$$

Given any $F_i \in \mathit{Term}(\tau_i \multimap \tau)$ $(i = 1, 2)$, taking $G = [F_1, F_2]$ in equation (12) and using equations (11), we get

$$[F_1, F_2](M(\tau_1 \oplus \tau_2)(!\mathit{Inl})(!\mathit{Inr})) =_{\mathrm{gnd}} M\,\tau\,(!F_1)\,(!F_2) : \tau$$

and hence by definition of $[F_1, F_2]$ (and validity of β-conversion for $=_{\mathrm{gnd}}$)

$$(M(\tau_1 \oplus \tau_2)(!\mathit{Inl})(!\mathit{Inr}))\,\tau\,(!F_1)\,(!F_2) =_{\mathrm{gnd}} M\,\tau\,(!F_1)\,(!F_2) : \tau.$$

So for any $\tau \in \mathit{Typ}$ and $V_i \in \mathit{Val}(!(\tau_i \multimap \tau))$, using Corollary 3.6 to express V_i as $!F_i$ for suitable (fixpoint) expressions F_i, we deduce that

$$(M(\tau_1 \oplus \tau_2)(!\mathit{Inl})(!\mathit{Inr}))\,\tau\,V_1\,V_2 =_{\mathrm{gnd}} M\,\tau\,V_1\,V_2 : \tau.$$

Therefore by Corollary 3.5 we have

$$M =_{\mathrm{gnd}} M(\tau_1 \oplus \tau_2)(!\mathit{Inl})(!\mathit{Inr}) : \tau_1 \oplus \tau_2 \tag{14}$$

for any $M \in \mathit{Term}(\tau_1 \oplus \tau_2)$. So given any $G_1, G_2 \in \mathit{Term}(\tau_1 \oplus \tau_2 \multimap \tau)$ and any $V \in \mathit{Val}(\tau_1 \oplus \tau_2)$, from equations (14) and (12) we get

$$G_i\,V =_{\mathrm{gnd}} G_i\,(V(\tau_1 \oplus \tau_2)(!\mathit{Inl})(!\mathit{Inr})) =_{\mathrm{gnd}} V\,\tau\,!(G_i \circ \mathit{Inl})\,!(G_i \circ \mathit{Inr}) : \tau.$$

So if G_1 and G_2 satisfy the antecedent of equation (13), we get $G_1\,V =_{\mathrm{gnd}} G_2\,V : \tau$ for all V, and so the conclusion of equation (13) holds by Corollary 3.5 (together with the congruence properties of $=_{\mathrm{gnd}}$ that are inherent in its definition).

Thus the coproduct property of $\tau_1 \oplus \tau_2$ is a consequence of property (12). To prove (12), consider the term-relation

$$r \triangleq \{\, (M, M') \mid G\,M =_{\mathrm{gnd}} M' : \tau \,\} \in \mathit{Rel}(\tau_1 \oplus \tau_2, \tau).$$

As in Remark 3.8, applied to the particular $\forall$-type that defines $\tau_1 \oplus \tau_2$ in (10), each $M \in \mathit{Term}(\tau_1 \oplus \tau_2)$ satisfies

$$(M(\tau_1 \oplus \tau_2), M\,\tau) \in (!(\Delta_{\tau_1} \multimap r))^{\top\top} \multimap (!(\Delta_{\tau_2} \multimap r))^{\top\top} \multimap r. \tag{15}$$

It is not hard to see that $(\mathit{Inl}, G \circ \mathit{Inl}) \in \Delta_{\tau_1} \multimap r$ and that $\Delta_{\tau_1} \multimap r = (\Delta_{\tau_1} \multimap r)^{\top\top}$ (the latter because $r = r^{\top\top}$); from these facts it follows that $(!\mathit{Inl}, !(G \circ \mathit{Inl})) \in !(\Delta_{\tau_1} \multimap r)$ and hence $(!\mathit{Inl}, !(G \circ \mathit{Inl})) \in (!(\Delta_{\tau_1} \multimap r))^{\top\top}$. Similarly, we have $(!\mathit{Inr}, !(G \circ \mathit{Inr})) \in (!(\Delta_{\tau_2} \multimap r))^{\top\top}$. So from (15) we get

$$(M(\tau_1 \oplus \tau_2)(!\mathit{Inl})(!\mathit{Inr})\ ,\ M\,\tau\,!(G \circ \mathit{Inl})\,!(G \circ \mathit{Inr})) \in r$$

from which (12) follows by definition of r.

Remark 3.10 (Defining types up to ground contextual isomorphism) Instead of making a definitional extension of `Lily` as in Fig. 1, one can consider extending the syntax and semantics of `Lily` with term-formers, typing and evaluation rules for tensor product, sum, product, existential, inductive, co-inductive, and recursive types. One can prove the key Theorems 2.3 and 3.2 for this extended version of `Lily` (and in doing so, one sees for which of the term-formers is it the case that strict and non-strict operational semantics are equivalent for $=_{\text{gnd}}$: for example tensor products $M \otimes M'$ can be evaluated strictly, but pairs (M, M') cannot). Then one can use the relational parametricity property mentioned in Remark 3.8 to prove the following definability result for types.

We say that two types τ and τ' are ground contextually isomorphic *if there are functions $I \in \mathit{Term}(\tau \multimap \tau')$ and $J \in \mathit{Term}(\tau' \multimap \tau)$ whose compositions $J \circ I$ and $I \circ J$ are ground contextually equivalent to the identity functions for τ and τ' respectively. Then in the extended version of* `Lily`*, the new types are all definable in terms of $\multimap$, $\forall$ and $!$ up to ground contextual isomorphism, using the formulas on the right-hand side in Fig. 1.*

Details of the proof (which is quite involved, especially when it comes to recursive types, where ideas due to Freyd [7] and Plotkin [6] are needed) will appear in the full version of this paper.

4 A Lazy Version of `Lily`

In Sect. 2, we investigated the equivalence of call-by-value and call-by-name evaluation strategies for arguments to linear functions. Nothing is gained by adopting a call-by-need (or lazy) strategy for linear function application, since a linear argument is either suspended, or evaluated exactly once. However, in the operational semantics in Fig. 3, when evaluating `let` $!y$ `=` M_1 `in` M_2, the value of M_1 is eliminated by substituting the same unfolding of its body for each occurrence of y in M_2. As there is no restriction on the number of occurrences, this duplicates computations that could be shared. For `Lily` to merit serious consideration as an intermediate language for both strict and lazy source languages, we should provide a call-by-need operational semantics for such terms. Furthermore, to apply the results of this paper to this `Lazy Lily` we must show that ground contextual equivalence is unaffected by the switch from call-by-name to call-by-need semantics.

Environments $A ::= []$ empty

$\quad [x = M]\, A$ binding.

Call-by-need evaluation, $\langle M, A\rangle \downarrow \langle V, A'\rangle$, is inductively generated by the following rules:

$$\frac{}{\langle \lambda a : \tau.\, M, A\rangle \downarrow \langle \lambda a : \tau.\, M, A\rangle}$$

$$\frac{}{\langle \Lambda\alpha.\, M, A\rangle \downarrow \langle \Lambda\alpha.\, M, A\rangle}$$

$$\frac{}{\langle !(x = M : \tau), A\rangle \downarrow \langle !(x = M : \tau), A\rangle}$$

$$\frac{\langle M_1, A_1\rangle \downarrow \langle \lambda a : \tau.\, M, A_2\rangle \quad \langle M[M_2/a], A_2\rangle \downarrow \langle V, A_3\rangle}{\langle M_1\, M_2, A_1\rangle \downarrow \langle V, A_3\rangle}$$

$$\frac{\langle M, A_1\rangle \downarrow \langle \Lambda\alpha.\, M', A_2\rangle \quad \langle M'[\tau/\alpha], A_2\rangle \downarrow \langle V, A_3\rangle}{\langle M\, \tau, A_1\rangle \downarrow \langle V, A_3\rangle}$$

$$\frac{\langle M, A_1\, [x = M]\, A_2\rangle \downarrow \langle V, A_3\, [x = M']\, A_4\rangle}{\langle x, A_1\, [x = M]\, A_2\rangle \downarrow \langle V, A_3\, [x = V]\, A_4\rangle}$$

$$\frac{\begin{array}{ll}\langle M_1, A_1\rangle \downarrow \langle !(x = M : \tau), A_2\rangle & x = \mathit{fresh}(\mathit{dom}(A_2)) \\ \langle M_2[x/y], [x = M]\, A_2\rangle \downarrow \langle V, A_3\rangle & \end{array}}{\langle \mathtt{let}\ !y = M_1\ \mathtt{in}\ M_2, A_1\rangle \downarrow \langle V, A_3\rangle}$$

Fig. 4. Lazy Evaluation of closed `Lily` terms

Fig. 4 defines an evaluation relation for `Lazy Lily` in the style of [11]. An *environment* (or heap) is an association list mapping (distinct) intuitionistic identifiers to suspended terms, similar to an explicit substitution. (The notation $A_1\, [x = M]\, A_2$ denotes the (list) concatenation of A_1 and $[x = M]\, A_2$.) The lazy evaluation relation $\_ \downarrow \_$ relates *configurations* pairing a term and an initial environment to configurations pairing a canonical form and a final environment. The variable x is evaluated by looking up its suspended term M in the initial environment, evaluating that term in the same environment, and then returning its value V along with the final environment, updated to record the value of x. Since the environment is threaded through a derivation, the result of the first computation of M is cached and recalled in subsequent references to x: the computation is shared. Evaluating a `let` $!y = M_1$ `in` M_2 term evaluates M_1 to a canonical form $!(x = M : \tau)$, binds the body of this thunk to a fresh renaming of y (creating a cycle in the environment), and continues with the evaluation of the renamed body $M_2[x/y]$. Assuming that the initial configuration is closed with respect to the domain of the environment—a property that is preserved by evaluation—choosing an x that is fresh for $dom(A_2)$ avoids the capture of free variables.

Formulating the lazy semantics is easy, but proving it correct for the non-strict semantics is not. The correctness results in [11] are with respect to a denotational model. To get a more direct operational proof, we hoped to make use of the 'small-step' abstract machine semantics formulated by Sestoft [26] and the operational techniques in [15] that are based on it. As it turned out, the 'big-step' style of [11] proved more amenable. Seaman and Iyer [25] give an operational proof of correctness for Lazy PCF using this style of operational semantics, but their semantics only shares the evaluation of function arguments, not recursive terms, whose evaluations are duplicated by unfolding (as for the call-by-name relation $\Downarrow_n$ in Fig. 3). Moreover, Seaman and Iyer report that they were unable to extend their proof technique to a semantics that shares evaluations of recursive terms. The problem is that sharing these computations creates cycles in the environment, violating an otherwise decreasing measure that they use in their inductive proof. Fortunately, we have identified a more robust measure that allows us to extend the proof technique to the setting of both `Lazy Lily` and Lazy PCF with shared recursion. Since linearity is not the issue, and the result for Lazy PCF is of wider interest, this result will be reported elsewhere. (The rules in Fig. 4 are very similar to those of Seaman and Iyer [25] for Lazy PCF; to cater for cycles, our variable rule evaluates the suspended term with respect to the entire environment $A_1\,[x = M]\,A_2$, not just the remainder A_2: this is just the alternative semantics of fixpoints proposed, but not proved correct, in [25].)

5 Conclusion

The material presented in this paper establishes some powerful techniques and results for exploring the surprisingly great expressive power that arises from the combination of linear lambda calculus, polymorphism and recursion. In principal, one can give semantics to a wide range of programming languages via compositional translations into the `Lily` language, using its versions of the standard constructs of Scott-Strachey denotational semantics (Fig. 1) and using the results presented here as the basis for proofs of correctness properties of the translations. In fact `Lily` terms modulo ground contextual equivalence give a more refined treatment of strictness and usage properties than does the model based on domains and strict continuous functions—for lifting $(-)_\perp$ in the latter has a contraction property that we have not built into `Lily`'s !-types. Accordingly, much remains to be done to explore the properties of such translations. For example, it would be interesting to explore properties of `Lily` versions of the semantics of Idealised Algol given by O'Hearn [17]; or of the lazy state threads of Peyton Jones and Launchbury [18] (to make computational effects implicit in translations to `Lily`, one could consider combining it with the calculus of monads of Moggi [13]).

References

[1] S. Abramsky. The lazy λ-calculus. In D. A. Turner, editor, *Research Topics in Functional Programming*, chapter 4, pages 65–117. Addison Wesley, 1990.

[2] S. Abramsky. Computational interpretations of linear logic. *Theoretical computer Science*, 111:3–57, 1993.

[3] A. Barber. *Linear Type Theories, Semantics and Action Calculi.* PhD thesis, Department of Computer Science, University of Edinburgh, 1997.

[4] G. M. Bierman. Program equivalence in a linear functional language. *J. Functional Programming*, 10:167–190, 2000.

[5] R.L. Crole. How linear is Howe? In G. McCusker, A. Edalat, and S. Jourdan, editors, *Advances in Theory and Formal Methods*, pages 60–72. Imperial College Press, 1996.

[6] M. Fiore and G. D. Plotkin. An axiomatization of computationally adequate domain theoretic models of FPC. In *9th Annual Symposium on Logic in Computer Science*, pages 92–102. IEEE Computer Society Press, Washington, 1994.

[7] P. J. Freyd. Remarks on algebraically compact categories. In M. P. Fourman, P. T. Johnstone, and A. M. Pitts, editors, *Applications of Categories in Computer Science, Proceedings LMS Symposium, Durham, UK, 1991*, volume 177 of *LMS Lecture Note Series*, pages 95–106. Cambridge University Press, 1992.

[8] C. A. Gunter and D. S. Scott. Semantic domains. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, pages 633–674. North-Holland, 1990.

[9] D. J. Howe. Proving congruence of bisimulation in functional programming languages. *Information and Computation*, 124(2):103–112, 1996.

[10] S. B. Lassen. Relational reasoning about contexts. In A. D. Gordon and A. M. Pitts, editors, *Higher Order Operational Techniques in Semantics*, Publications of the Newton Institute, pages 91–135. Cambridge University Press, 1998.

[11] J. Launchbury. A natural semantics for lazy evaluation. In *20th Symp. on Principles of Programming Languages*, pages 144–154, Charleston, South Carolina, 1993. ACM.

[12] I. A. Mason and C. L. Talcott. Equivalence in functional languages with effects. *Journal of Functional Programming*, 1:287–327, 1991.

[13] E. Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55–92, 1991.

[14] E. Moggi. Metalanguages and applications. In A. M. Pitts and P. Dybjer, editors, *Semantics and Logics of Computation*, Publications of the Newton Institute, pages 185–239. Cambridge University Press, 1997.

[15] A. K. Moran and D. Sands. Improvement in a lazy context: An operational theory for call-by-need. In *Proc. 26th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 43–56. ACM Press, 1999.

[16] P. D. Mosses. Denotational semantics. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, pages 577–632. North-Holland, 1990.

[17] P. W. O'Hearn and J. C. Reynolds. From Algol to polymorphic linear lambda-calculus. *Journal of the ACM*, 47(1):167–223, 2000.

[18] S. L Peyton Jones and J. Launchbury. State in Haskell. *Lisp and Symbolic Computation*, 8:293–341, 1995.

[19] A. M. Pitts. Some notes on inductive and co-inductive techniques in the semantics of functional programs. Notes Series BRICS-NS-94-5, BRICS, Department of Computer Science, University of Aarhus, 1994.

[20] A. M. Pitts. Existential types: Logical relations and operational equivalence. In K. G. Larsen, S. Skyum, and G. Winskel, editors, *Automata, Languages and Programming, 25th International Colloquium, ICALP'98, Aalborg, Denmark, July 1998, Proceedings*, volume 1443 of *Lecture Notes in Computer Science*, pages 309–326. Springer-Verlag, Berlin, 1998.

[21] A. M. Pitts. Parametric polymorphism and operational equivalence. *Mathematical Structures in Computer Science*, 10:321–359, 2000.

[22] G. D. Plotkin. LCF considered as a programming language. *Theoretical Computer Science*, 5:223–255, 1977.

[23] G. D. Plotkin. Second order type theory and recursion. Notes for a talk at the Scott Fest, February 1993.

[24] J. C. Reynolds. Types, abstraction and parametric polymorphism. In R. E. A. Mason, editor, *Information Processing 83*, pages 513–523. North-Holland, Amsterdam, 1983.

[25] J. Seaman and S. Purushothaman Iyer. An operational semantics of sharing in lazy evaluation. *Science of Computer Programming*, 27(3):289–322, 1996.

[26] P. Sestoft. Deriving a lazy abstract machine. *Journal of Functional Programming*, 7:231–264, 1997.

[27] D. N. Turner and P. Wadler. Operational interpretations of linear logic. *Theoretical Computer Science*, 227:231–248, 1999.

[28] P. Wadler. Theorems for free! In *Fourth International Conference on Functional Programming Languages and Computer Architecture*, London, UK, 1989.