Using Synthetic Domain Theory to Prove Operational Properties of a Polymorphic Programming Language Based on Strictness

Citation for published version:
Simpson, A & Rosolini, G 2004 'Using Synthetic Domain Theory to Prove Operational Properties of a Polymorphic Programming Language Based on Strictness'.

General rights
Copyright for the publications made accessible via the Edinburgh Research Explorer is retained by the author(s) and/or other copyright owners and it is a condition of accessing these publications that users recognise and abide by the legal requirements associated with these rights.

Take down policy
The University of Edinburgh has made every reasonable effort to ensure that Edinburgh Research Explorer content complies with UK legislation. If you believe that the public display of this file breaches copyright please contact openaccess@ed.ac.uk providing details, and we will remove access to the work immediately and investigate your claim.
Using Synthetic Domain Theory to Prove Operational Properties of a Polymorphic Programming Language Based on Strictness

Giuseppe Rosolini
Dipartimento di Informatica e Scienze dell’Informazione
Università di Genova, Italy

Alex Simpson
LFCS, School of Informatics
University of Edinburgh, UK

Abstract

We present a simple and workable axiomatization of domain theory within intuitionistic set theory, in which predomains are (special) sets, and domains are algebras for a simple equational theory. We use the axioms to construct a relationally parametric set-theoretic model for a compact but powerful polymorphic programming language, given by a novel extension of intuitionistic linear type theory based on strictness. By applying the model, we establish the fundamental operational properties of the language.

1. Introduction

The idea of synthetic domain theory (SDT) was proposed by Dana Scott around 1980. He suggested that some of the order-theoretic and topological complexities of domain theory might be avoided by simply taking domains to be special sets, and morphisms of domains to be arbitrary set-theoretic functions. Although such an idea is incompatible with ordinary classical set theory, Scott indicated that it should be consistent with intuitionistic set theory [20]. Subsequently, in a long line of research including [18, 9, 6, 21, 14, 22], a substantial theory has been developed, incorporating the full range of domain-theoretic constructions as set-theoretic constructions within intuitionistic set theory. To some extent, this work has fulfilled Scott’s original hope of avoiding the order-theoretic and topological aspects of classical domain theory, yet only at the expense of introducing significant new difficulties of a logical and category-theoretic nature. Even for researchers in semantics, there has hitherto been little incentive to get to grips with the technical demands of SDT, as it has not been clear what the pay-off might be in terms of applications.

The two main goals of the present paper are: (i) to make synthetic domain theory accessible to a wider audience by presenting a notably simple axiomatization within intuitionistic set theory; and, more significantly, (ii) to demonstrate the applicability of this theory by using it to establish non-trivial operational properties of a compact yet powerful polymorphic programming language.

Concerning (i), our axiomatization differs from existing accounts in two main ways. In SDT, “domains” are traditionally defined as algebras for a lifting monad acting on a category of “predomains”. We use the axioms to construct a relationally parametric set-theoretic model for a compact but powerful polymorphic programming language, given by a novel extension of intuitionistic linear type theory based on strictness. By applying the model, we establish the fundamental operational properties of the language.

Concerning (ii), our application is to establish operational properties of a polymorphic language (loosely) based on linear type theory. It was Plotkin who first realised the surprising power of combining linear type theory, polymorphism and recursion [12]. This observation influenced Bierman, Pitts and Russo to design a simple programming language, called Lily, based on only three type constructors: ∀α. σ for polymorphism, !σ for “thunks” and σ → τ for (linear) functions [2]. Following Plotkin’s ideas, this compact language is capable of encoding a rich variety type constructs, including recursive types. The main contribution of [2] was to develop techniques based on operational semantics for reasoning about operational properties of Lily.

In this paper we show that synthetic domain theory of-
fers a denotational alternative to the operational methods of Lily, which we introduce in Section 4. Our language shares the same type structure as Lily. The difference is that we interpret $\sigma \rightarrow \tau$ more generously as a type of strict rather than linear functions. This leads to a more powerful language than Lily in the sense that our language assigns types to more programs. Our language is possible of interest in its own right, as a novel and natural extension of linear type theory, and as a potential intermediate language for use in compilation as a target language for strictness analysis.

In Section 5, working within intuitionistic set theory, we build a relationally parametric model of our language. The existence of such a model is, in itself, already an advantage of our synthetic framework, as models of parametric polymorphism have not been forthcoming within the context of ordinary domain theory. Relational parametricity has many applications. For example, it can be used to justify the correctness of Plotkin’s datatype encodings. Our aim in this paper, however, is to use the model to prove operational properties. In Section 6, we prove a computational adequacy result for our language showing that our model is sound for establishing operational equivalences determined by a strict (call-by-value) operational semantics. In Section 7, we prove a second computational adequacy result for a non-strict (call-by-name) variant of the operational semantics. The fact that computational adequacy holds for the same model with respect to both strict and non-strict operational semantics means that the operational equivalences induced by the two semantics coincide. We thus obtain a denotational proof that the strictness theorem of [2] extends to our language. In addition, we exploit the approximation relation defined in the proof of computational adequacy to establish an operational extensionality result, once again showing that properties for Lily, established in op. cit., extend to our language. Finally, in Section 8, we give a brief justification that our methods, which are based on classically inconsistent axioms about sets, are nonetheless sound for establishing operational properties of programs.

This paper reveals synthetic domain theory to be a serious competitor to state-of-the-art techniques in operational semantics. Although our results exactly mirror those for Lily presented in [2], we establish them for a slightly richer language. While it seems likely that the syntactic methods of [2] should extend to our language,1 such methods are themselves quite involved, and it is certainly worth showing that a complementary approach based on denotational semantics is possible.

Acknowledgements We thank Dana Scott and Phil Wadler for helpful suggestions.

2. Pointed Sets and Strict Functions

In traditional domain theory, a “domain” is a directed-complete partial order (a “predomain”) that also has a least element (i.e. is “pointed”). Thus one has the equation:

$$\text{domain} = \text{pointed predomain}. \quad (1)$$

This equation is the model for our synthetic development. In the synthetic account, predomains are just special sets; we consider their properties in Section 3. First, in this section, we address the pointedness condition, which will be implemented using a notion of pointed set.

To assist the reader, it is convenient to review the classical notion of pointed set. Traditionally, a “pointed set” is a structure $(A, a)$ where $A$ is a set and $a \in A$ is any element, the “point”. A “strict function” $f : (A, a) \rightarrow (B, b)$ is simply a function $f : A \rightarrow B$ such that $f(a) = b$. Equivalently, a pointed set is an algebra $(A, r_\bot)$ for a signature consisting of a single 0-ary operator (i.e. constant) $r_\bot$ and no equations, and a strict function is simply a homomorphism. Again equivalently, a pointed set is an algebra $(A, r_\bot, r_\top)$ for a signature consisting of two operators: a 0-ary operator $r_\bot$ and a unary operator $r_\top$ satisfying the equation

$$r_\top(x) = x. \quad (2)$$

Although seemingly a trivial reformulation, it is this latter presentation that adapts best to our purposes.

It is standard mathematical practice to work informally within classical set theory. As discussed in the introduction, the development of synthetic domain theory has to be carried out using a set theory based on intuitionistic logic. In this paper, we simply modify standard practice and work informally within intuitionistic set theory. To follow the development at an intuitive level, the reader is required merely to have some feeling for intuitionistic reasoning. Formally, the background set theory can be taken to be IZF [19].

Let 1 be the singleton set $\{ \emptyset \}$. Its powerset $P(1)$ is isomorphic to (or may be taken as the definition of) the set $\Omega$ of truth values, under the bijection mapping any subset $e \in P(1)$ to the truth value of the proposition $\emptyset \in e$, and conversely mapping any truth value $p \in \Omega$ to the singleton set $\{ \emptyset \mid p \}$ that contains $\emptyset$ iff $p$ is true. (Equality on $\Omega$ is logical equivalence, i.e. $p = q$ iff $p \iff q$.) Of course $\{ \top, \bot \} \subseteq \Omega$, where $\top$ is “true” and $\bot$ is “false”. It will follow from later axioms that $\Omega \neq \{ \top, \bot \}$, i.e. our logic is forced to be non-classical.

In the last of the accounts of classical pointed sets above, a pointed set was an algebra for a signature consisting of two operators $r_\bot$ and $r_\top$, i.e. for a family of operators $\{ r_p \}_{p \in \{ \bot, \top \}}$. Intutionistically, we shall analogously consider families $\{ r_p \}_{p \in \Sigma}$ indexed by a subset $\Sigma \subseteq \Omega$. Intuitively, $\Sigma$ is to be understood as the set of truth values of
all propositions of the form “the execution of $P$ terminates” where $P$ ranges over all possible “computations”. Rather than formalizing the notion of computation, we leave it abstract and instead axiomatize the properties we need of $\Sigma$.

**Definition 2.1 (Dominance [18])** A subset $\Sigma \subseteq \Omega$ is a dominance if:

1. $\top \in \Sigma$, and
2. for all $p \in \Sigma$, $q \in \Omega$, if $p \rightarrow (q \in \Sigma)$ then $(p \land q) \in \Sigma$.

The set $\Omega$ and its subsets $\{\top, \bot\}$ and $\{\top\}$ are all easily seen to be dominances. Our first axiom asserts that our assumed set $\Sigma$ of termination properties is also a dominance.

**Axiom 1** The distinguished subset $\Sigma \subseteq \Omega$ is a dominance.

**Remark 2.2** Axiom 1 does not imply that $\bot \in \Sigma$. Thus we are not yet asserting the existence of nonterminating computations. This will rather follow in Section 3 as a consequence of Axiom 4, which imposes the closure under computations. This will rather follow in Section 3 as a consequence of Axiom 4, which imposes the closure under computations.

In the classical account of pointed sets, the arity of $r_\bot$ was 0 and that of $r_\top$ was 1. For a pointed set $(X, \{r_p\}_{p \in \Sigma})$, we generalize this by giving $r_p$ the “arity” $\{0 \mid p\}$ in the sense that $r_p$ is a function from $X^{\{0 \mid p\}}$ to $X$. In practice, it is convenient to work instead with an isomorphic description of $X^{\{0 \mid p\}}$. Define

$$X^p = \{e \in \mathcal{P}(X) \mid (\forall x, y \in e, x = y) \land (\exists e \in e \Rightarrow p)\},$$

where we write $(\exists e \in e)$ as shorthand for the proposition $(\exists x \in e, \top)$ stating that $e$ is inhabited. It is easily shown that $X^p$ is isomorphic to $X^{\{0 \mid p\}}$.

**Definition 2.3 (Pointed set)** A pointed set is a structure $(X, \{r_p: X^p \rightarrow X\}_{p \in \Sigma})$ satisfying, for all $x \in X$, all $p, q \in \Sigma$ and $e \in X^{p\lor q}$,

$$r_\top \{x\} = x,$$

$$r_{p\lor q}(e) = r_p\{r_{p\land q}(e) \mid p\}.$$  \hfill(3)

$$r_{p\lor q}(e) = r_p\{r_{p\land q}(e) \mid p\}.$$  \hfill(4)

Here, equation (3) is just equation (2) from before. Equation (4) is derivable for $\Sigma = \{\top, \bot\}$, and is thus redundant classically. The above equations are motivated by being just what is needed for Lemma 2.6 below to hold.

As in the classical case, pointed sets are algebras. Similarly, strict functions are just homomorphisms.

**Definition 2.4 (Strict function)** A strict function from a pointed set $(X, \{r_p^X\}_p)$ to another $(Y, \{r_p^Y\}_p)$ is a function $f: X \rightarrow Y$ satisfying, for all $e \in X^p$,

$$f(r_p^X(e)) = r_p^Y(f(x) \mid x \in e).$$  \hfill(5)

When convenient, we shall leave the operator structure implicit when working with pointed sets, writing $X$ rather than $(X, \{r_p^X\}_p)$. We write $X \rightharpoonup Y$ for the set of strict functions between pointed sets $X, Y$, and we write $X \cong Y$ to mean that $X$ and $Y$ are isomorphic via strict functions.

The category of pointed sets and strict functions enjoys all the usual properties of categories of algebra homomorphisms. For example, for any family $\{(Y^x, \{r_p^Y\}_p)\}_{x \in X}$ of pointed sets, the product $\prod_{x \in X} Y^x, \{r_p^{\prod_{x \in X} Y^x}\}_p$ is pointed, where

$$r_p^{\prod_{x \in X} Y^x}(e) = \{r_p^X(\pi_x(x) \mid e) \mid x \in X\}.$$ 

As the set of functions $X \rightarrow Y$ is isomorphic to $\prod_{x \in X} Y^x$, it follows that, for any pointed set $(Y, \{r_p^Y\}_p)$, the function space $(X \rightarrow Y, \{r_p^{X \rightarrow Y}\}_p)$ is pointed, where

$$r_p^{X \rightarrow Y}(e) = (x \rightarrow r_p^Y(f(x) \mid f \in e)).$$

Given a pointed set $(X, \{r_p^X\}_p)$, we say that a subset $Z \subseteq X$ is subpointed if the operators on $X$ restrict to operators on $Z$, i.e., if, for all $p \in \Sigma$ and $e \in Z^p$, it holds that $r_p^Z(e) \in Z$ (note that indeed $Z^p \subseteq X^p$). If $X, Y$ are pointed and $f, g: X \rightarrow Y$ are strict, then the equalizer $\{x \in X \mid f(x) = g(x)\}$ is a subpointed subset of $X$.

**Lemma 2.5** If $X, Y$ are pointed then $X \rightharpoonup Y$ is a subpointed subset of $X \rightarrow Y$.

**Lemma 2.6 (Lifting)** For any set $X$, the structure $(LX, \{\mu_p\}_p)$ defined below is pointed.

$$LX = \bigcup_{p \in \Sigma} X^p$$

$$\mu_p(E) = \bigcup E$$

Moreover, for any pointed set $(Y, \{r_p\}_p)$, and $f: X \rightarrow Y$, there exists a unique strict $f^*: (LX, \{\mu_p\}_p) \rightarrow (Y, \{r_p\}_p)$ satisfying $f^*(\{x\}) = f(x)$ for all $x \in X$.

In other words, $LX$ is the free pointed set generated by $X$. It also follows from the lemma that $\Sigma \cong L\bot$ is pointed.

### 3. Predomains and Domains

In this section we address the remaining components of equation (1). We separate the properties we require of predomains into two axioms. The first expresses closure under useful constructions on sets. The second is the key axiom for constructing models of polymorphism in Section 5.

**Axiom 2** There is a class, Predom, of special sets, called predomains, that satisfies:

1. If $A$ is a predomain and $A \cong B$ then $B$ is a predomain.
2. If \( \{A_\alpha\}_{\alpha \in X} \) is a set-indexed family of predomains, then their product \( \prod_{\alpha \in X} A_\alpha \) is a predomain.

3. For functions \( f, g : A \to B \) between predomains, the equalizer \( \{a \in A \mid f(a) = g(a)\} \) is a predomain.

4. The set of natural numbers \( \mathbb{N} \) is a predomain.

5. If \( A \) is a predomain then so is its lifting \( L A \).

**Axiom 3** There exists a set \( P \) of predomains such that, for every predomain \( A \), there exists \( B \in P \) such that \( B \cong A \).

**Remark 3.1** Given Axiom 2, Axiom 3 is inconsistent with classical logic. It implies that the complete category of predomains is weakly equivalent to the small category whose objects are sets in \( P \) and whose morphisms are arbitrary functions. This small category is thus itself complete, but only in the weakest of the senses discussed in [16, 7].

**Remark 3.2** By Freyd’s adjoint functor theorem, Axiom 3 implies that the category of predomains is a full reflective subcategory of the category of sets, hence it is cocomplete.

**Definition 3.3 (Domain)** A domain is a pointed set \( (A, \{r_p\}_p) \), where \( A \) is a predomain.

By the closure properties discussed after Definition 2.3, the category of strict functions between domains is complete. We write \( \text{Dom} \) for the class of domains. Define the set:

\[ D = \{(B, \{r_p\}_p) \mid B \in P \text{ and } (B, \{r_p\}_p) \text{ is a pointed set}\} \]

**Lemma 3.4** For every domain \( A \) there exists \( D \in D \) such that \( D \cong A \).

**Remark 3.5** The lemma shows that the complete category of strict maps between domains is weakly equivalent to a small category. It follows that it is also cocomplete.

**Axiom 4** For every domain \( (A, \{r_p\}_p) \) there is a function \( \text{fix}_A : (A \to A) \to A \) satisfying:

1. (Fixed point property) For all \( f : A \to A \) it holds that \( f(\text{fix}_A(f)) = \text{fix}_A(f) \).

2. (Uniformity) For any domain \( (B, \{r_p\}_p) \), and functions \( f : A \to A, g : B \to B \) and strict \( h : A \to B \), if \( g \circ h = h \circ f \) then \( \text{fix}_B(g) = h(\text{fix}_A(f)) \).

**Remark 3.6** Given Axiom 2, Axiom 4 is again inconsistent with classical logic as it implies the existence of nontrivial sets for which every endofunction has a fixed point.

**Remark 3.7** It can be shown, see e.g. [23], that \( \text{fix}_A \) is uniquely determined by the property of uniformity. Moreover, by the dinaturality property of \( \text{loc. cit.} \), \( \text{fix}_A \) does not depend on the algebra structure \( \{r_p\}_p \).

4. A Polymorphic Language

In this section, we introduce our programming language, which is strongly influenced by the language Lily of [2]. It shares the same types as Lily, and has an equivalent language of raw terms. The important difference from Lily lies in the formulation of the typing rules. Our rules are more permissive. They correspond to interpreting \( \sigma \to \tau \) as a type of strict as opposed to linear functions.

We use \( \alpha, \beta, \ldots \) to range over type variables, and \( \sigma, \tau, \ldots \) to range over types, which are given by:

\[ \sigma ::= \alpha \mid \sigma \to \tau \mid !\tau \mid \forall \alpha. \sigma \]

As usual, \( \forall \alpha \) binds \( \alpha \), and we identify types up to renamings of bound variables. We write \( \text{ftv}(\sigma) \) for the set of free type variables of \( \sigma \), if \( \Theta \) is a finite set of type variables then we write \( \sigma(\Theta) \) to mean that \( \text{ftv}(\sigma) \subseteq \Theta \).

We use \( x, y, z, \ldots \) to range over term variables, and \( s, t, \ldots \) to range over raw terms, which are given by:

\[ t ::= x \mid \lambda x : \sigma. t \mid s(t) \mid !t \mid \text{let } lx = s \text{ in } t \mid \Lambda \alpha. t \mid t(\sigma) \mid \text{rec } \alpha. t : \tau \]

Here \( x \) is bound in \( \lambda x : \sigma. t \) and in \( \text{rec } \alpha. t : \tau \), and occurrences of \( x \) in \( t \) are bound in \( \text{let } lx = s \text{ in } t \). We again identify terms up to renamings of bound variables, and we write \( \text{fv}(t) \) for the set of free variables in a term \( t \).

The typing rules make use of labelled contexts, which we define first and then motivate below. As usual, a context is a function \( \Gamma \) mapping a finite set of variables, \( \text{dom}(\Gamma) \), to types. A labelling of \( \Gamma \) is a function from \( \text{dom}(\Gamma) \) to \( \{0, 1\} \). The structure of typing judgments is

\[ \Gamma \vdash_\Theta t : \sigma, \]

where \( \Gamma \vdash_\Theta t : \sigma \) is a labelled context consisting of a type assignment \( \Gamma \) together with a labeling \( \delta \) of \( \Gamma \). A typing judgment is well formed if: \( \text{ftv}(\Gamma, \sigma) \subseteq \Theta \) and \( \text{fv}(t) \subseteq \text{dom}(\Gamma) \).

**Remark 4.1** Even though it refers to concepts yet to be defined, it is helpful to give the intuitive motivation behind the labelling. If \( \Gamma \vdash_\Theta t : \sigma \) is derivable and \( x \in \text{dom}(\Gamma) \) then \( \delta(x) = 1 \) implies that the term \( t \) is strict in \( x \) in the following sense. Given closed terms \( \{s_\Theta\}_{\Theta \in \text{dom}(\Gamma)} \) of appropriate types, then, in any evaluation of a term of the form \( E[i][s_\Theta/x]_{\Theta \in \text{dom}(\Gamma)} \), where \( E[\cdot] \) is a ground evaluation context expecting an argument of type \( \sigma \), it must hold that at least one occurrence of \( s_\Theta \) is evaluated. If instead \( \delta(x) = 0 \) then no guarantees are available about the operational behaviour of \( t \) with respect to terms substituted for \( x \).

The typing rules are given in Figure 1, and are to be read as applying only when the premises and conclusion are all
well formed. The rules make use of the following notation. We write $\Theta, \alpha$ to mean $\Theta \cup \{\alpha\}$, where we assume that $\alpha \notin \Theta$. Under the assumption that $x \notin \text{dom}(\Gamma)$, we write $\Gamma \vdash \delta$, $x : \sigma$ for the context consisting of the type assignment $\Gamma$, $x : \sigma$ and the labelling $\delta[0/x]$. Similarly, we write $\Gamma \vdash \delta$, $x : \sigma$ for the context consisting of the type assignment $\Gamma$, $x : \sigma$ and the labelling $\delta[1/x]$. The notation $\Gamma \vdash \delta$, $x : \sigma$ is used to represent either of the contexts $\Gamma \vdash \delta$, $x : \sigma$ and $\Gamma \vdash \delta$, $x : \sigma$. We also make use of the join semilattice operations on labellings of $\Gamma$ under the pointwise ordering (where the ordering on $\{0, 1\}$ is $0 \leq 1$). We write: $0$ for the least labelling of $\Gamma$ (the everywhere 0 labelling); and $\delta \lor \delta'$ for the join of $\delta$ and $\delta'$.

Remark 4.2 Our language relates to the language Lily of [2] as follows. Given a term $\Gamma; \Delta \vdash \Theta \ t : \sigma$ of Lily, where $\Gamma; \Delta$ is a “dual” intuitionistic/linear context, there exists a (unique) labelling $\delta$ of the concatenated context $\Gamma; \Delta$ such that $\delta(x) = 1$ for all $x \in \text{dom}(\Delta)$ and $\Gamma; \Delta \vdash \Theta \ t : \sigma$.2 Thus every term typable in Lily has a type in our language. The converse fails. In fact, our language is equivalent to extending Lily with a contraction principle that contracts $\Gamma; x : \sigma; y : \sigma; \Delta \vdash \Theta \ t : \sigma$ to $\Gamma; y : \sigma; \Delta \vdash \Theta \ t[y/x] : \sigma$. Such contraction across

---

2 We are ignoring the inessential syntactic difference between the recursively defined thunks of [2] and our explicit recursion operator, as the two are interdefinable, see op. cit.

the intuitionistic/linear divide is stronger than the contraction (within the linear context itself) of relevant logic. We believe that our form of contraction is the natural one for dealing with strictness issues. Given that strictness (as opposed to linearity) is often the important issue in determining an efficient evaluation strategy, it is plausible that languages based on our contraction principle might serve as useful intermediate languages in compilation, e.g. as target languages for strictness analysis.

The reason for formulating the typing rules as we have (rather than, e.g., using an explicit contraction rule) is so that the rules are syntax directed. In fact, labellings as well as types are uniquely determined by terms and contexts.

Lemma 4.3 If $\Gamma \vdash \delta \ t : \sigma$ and $\Gamma \vdash \delta' \ t : \sigma'$ then $\delta = \delta'$ and $\sigma = \sigma'$.

As in [2], we give two operational semantics to our language. In both, values are closed terms of the form:

$$v ::= \lambda x : \sigma. \ t \mid \! t \mid \Lambda \alpha. \ t$$

Figure 2 defines evaluation relations $t \Downarrow^s v$ and $t \Downarrow^n v$ between closed terms $t$ and values $v$. The strict (or call-by-value) relation $t \Downarrow^s v$ is inductively defined by the specific $\Downarrow^s$ rule for application together with all rules written using the neutral $\Downarrow$ notation. Similarly, the non-strict (or call-by-name) relation $t \Downarrow^n v$ is defined by the $\Downarrow^n$ application rule.
together with the neutral rules. We write \( t \vdash^* \sigma \) (resp. \( t \vdash^n \sigma \)) to mean that there exists \( v \) such that \( t \vdash^* v \) (resp. \( t \vdash^n v \)). We write \( t \models^* t' \) for strict Kleene equality: \( t \vdash^* v \) iff \( t' \vdash^* v \).

Both semantics are easily seen to be deterministic: if \( t \vdash^* v \) and \( t \vdash^* v' \) then \( v = v' \) (and similarly for \( \vdash^n \)). Also, both are type sound: if \( t : \sigma \) and \( t \vdash^* v \) then \( v : \sigma \) (and similarly for \( \vdash^n \)), where we write \( t : \sigma \) to mean that \( t \) is a closed term of closed type \( \sigma \).

As in [2], we define contextual (approximation and) equivalence using termination at types of the form \( \sigma \) as observations. In natural extensions of the language with additional type primitives, this turns out to be equivalent to observing termination at ground types only. This choice has many benefits, including: inducing extensionality properties for function and universal types, and the operational correctness of Plotkin’s polymorphic encodings of type constructors. See [2] for a thorough discussion of these issues for the language \( \text{Lily} \).

For closed \( \sigma \), a ground \( \sigma \)-context is a term of the form \( x: \sigma \vdash \Gamma \vdash \tau \). We usually denote such a context by \( C[\cdot] \), and write \( C[t] \) for \( C[t/x] \), where \( t : \sigma \).

**Definition 4.4 (Contextual approximation/equivalence)**

For \( t, t' : \sigma \), we write \( t \equiv_{\text{gnd}} t' \) to mean that, for all ground \( \sigma \)-contexts it holds that \( C[t] \vdash^* \sigma \) implies \( C[t'] \vdash^* \sigma \). We write \( t \equiv_{\text{gnd}} t' \) to mean that both \( t \equiv_{\text{gnd}} t' \) and \( t' \equiv_{\text{gnd}} t \) hold.

We do not introduce a non-strict variant of contextual approximation, because it follows from the theorem below that this coincides with the strict version above.

**Theorem 4.5 (Strictness)**

If \( t : \sigma \) then \( t \vdash^* \sigma \) iff \( t \vdash^n \sigma \).

We shall prove this theorem in Section 7. Until we have the proof, the reader must keep in mind that \( \equiv_{\text{gnd}} \) and \( \equiv_{\text{gnd}} \) are defined using the strict evaluation relation.

**Remark 4.6** Because \( \text{Lily} \) is included in our language, the strictness theorem for \( \text{Lily} \) [2, Theorem 2.3] is a consequence of Theorem 4.5 above.

Contextual approximation and equivalence can be hard to reason with directly. To address this, we introduce a complementary applicative (bisimulation) relation, which is more amenable to certain forms of argument, and we prove operational extensionality: the coincidence of ground contextual approximation and applicative simulation.

**Definition 4.7 (Strict applicative simulation)**

The relation \( \sqsubseteq_{\text{app}} \) is the largest relation between closed terms of identical closed type satisfying:

1. If \( t \sqsubseteq_{\text{app}} t' : \sigma \rightarrow^* \tau \) then, for all values \( v : \sigma \), it holds that \( t(v) \sqsubseteq_{\text{app}} t'(v) : \tau \);
2. If \( t \sqsubseteq_{\text{app}} t' \) then \( t \vdash^* s \) implies \( t' \vdash^* s' \) where \( s \equiv_{\text{app}} s' : \sigma \); and
3. If \( t \sqsubseteq_{\text{app}} t' ; \forall \alpha. \sigma \) then, for all closed \( \tau \), it holds that \( t(\tau) \sqsubseteq_{\text{app}} t'(\tau) : \sigma[\tau/\alpha] \).

Strict applicative simulation has an alternative, more elementary, characterization. A ground evaluation context is a ground context generated by the grammar below:

\[
E[\cdot] ::= \cdot | E[(\cdot)(V)] | E[(\cdot)(\tau)] \quad \text{let } x = (\cdot) \text{ in } E[x].
\]

**Proposition 4.8**

If \( t, t' : \sigma \) then \( t \sqsubseteq_{\text{app}} t' \) iff \( E[t] \vdash^* \sigma \) implies \( E[t'] \vdash^* \sigma \) for all ground evaluation \( \sigma \)-contexts \( E[\cdot] \).

**Theorem 4.9 (Operational extensionality)**

If \( t, t' : \sigma \) then \( t \equiv_{\text{gnd}} t' \) if and only if \( t \equiv_{\text{app}} t' \).

Again, we prove this theorem in Section 7.

5. A Relationally Parametric Model

In this section, we construct a relationally parametric model of our language. To do this, we give two interpretations of types: as domains, and as relations.

For \( D \) a domain, a subdomain of \( D \) is any subpointed subset \( D' \subseteq D \) that is also a predomain. If \( D, E \) are domains then an admissible relation between \( D \) and \( E \) is a subdomain of the domain \( D 	imes E \). We write \( R(D, E) \) for the set of all admissible relations.

**Lemma 5.1**

If \( D, E \) are domains and \( f : D \rightarrow E \) then \( \text{graph}(f) = \{(d, e) | f(d) = e \} \) is an admissible relation between \( D \) and \( E \).

In particular, for any domain \( D \), the diagonal relation \( \Delta_D = \{(x, x) | x \in D \} = \text{graph}(\text{id}_D) \) is admissible.

Given a set of type variables \( \Theta \), a \( \Theta \)-environment is a function \( \gamma : \Theta \rightarrow \text{Dom} \), and a relational \( \Theta \)-environment is a tuple \( \gamma^R = (\gamma_1, \gamma_2, \gamma_R) \), where \( \gamma_1, \gamma_2 \) are \( \Theta \)-environments, and \( \gamma_R \subseteq \prod_{\alpha \in \Theta} \mathcal{R}(\gamma_1(\alpha), \gamma_2(\alpha)) \). For each type \( \sigma(\Theta) \) and \( \Theta \)-environment \( \gamma \), we define a domain \( [\sigma]_{\gamma} \), and, for each relational \( \Theta \)-environment \( \gamma^R \), we define an admissible relation \( [\sigma]^R_{\gamma^R} \subseteq \mathcal{R}(\sigma_{\gamma_1}, \sigma_{\gamma_2}) \). The interdependent definitions are given in Figure 3. In them, we write \( \Delta^R_{\alpha} \) for the relational \( \Theta \)-environment \( (\gamma_1, \gamma_2, \alpha \mapsto \Delta^R_{\gamma(\alpha)}) \) determined by a \( \Theta \)-environment \( \gamma \). That these definitions are good can be seen using Axiom 2.

The relational interpretation of types implies that open types act functionally on the (large) groupoid of strict isomorphisms between domains, cf. [10, 3].

**Lemma 5.2**

**Groupoid (Action)**

If \( \sigma(\Theta, \alpha) \) then, for any \( \Theta \)-environment \( \gamma \), domains \( D_1, D_2 \) and isomorphism \( i : D_1 \rightarrow D_2 \), there exists a unique isomorphism \( \text{gpd}(\sigma, \gamma, i) : [\sigma]_{\gamma[D_1/\alpha]} \rightarrow [\sigma]_{\gamma[D_2/\alpha]} \) such that

\[
[\sigma]^R_{\Delta^R_{\gamma[D_1,D_2,\text{graph}(i)/\alpha]}} \equiv \text{graph}(\text{gpd}(\sigma, \gamma, i)).
\]

Moreover, the mapping \( i \mapsto \text{gpd}(\sigma, \gamma, i) \) is functorial.
\[ \begin{align*}
\llbracket \alpha \rrbracket_\gamma &= \gamma(\alpha) \\
\llbracket \sigma \oslash \tau \rrbracket_\gamma &= \llbracket \sigma \rrbracket_\gamma \oslash \llbracket \tau \rrbracket_\gamma \\
\llbracket \lambda \sigma \rrbracket_\gamma, \rho &= \lambda \llbracket \sigma \rrbracket_\gamma, \rho \\
\llbracket s(t) \rrbracket_\gamma, \rho &= \llbracket s \rrbracket_\gamma \llbracket t \rrbracket_\gamma, \rho \\
\llbracket \text{let} \ x = s \text{ in } t \rrbracket_\gamma, \rho &= (d: \llbracket s \rrbracket_\gamma, \rho) \mapsto \llbracket t \rrbracket_\gamma, \rho(d/x) \\
\llbracket \Delta \alpha \cdot t \rrbracket_\gamma, \rho &= \{ \llbracket t \rrbracket_\gamma, \rho \}_{\llbracket \alpha \rrbracket_\gamma} \\
\llbracket \text{let} \ x: \sigma \cdot t \rrbracket_\gamma, \rho &= \text{fix}(d: \llbracket \sigma \rrbracket_\gamma \mapsto \llbracket t \rrbracket_\gamma, \rho(d/x)) \\
\llbracket \text{rec} \ x: \sigma \cdot t \rrbracket_\gamma, \rho &= \text{fix}(d: \llbracket \sigma \rrbracket_\gamma \mapsto \llbracket t \rrbracket_\gamma, \rho(d/x)) \\
\llbracket \text{let} \ x: \sigma \cdot t \rrbracket_\gamma, \rho &= \text{fix}(d: \llbracket \sigma \rrbracket_\gamma \mapsto \llbracket t \rrbracket_\gamma, \rho(d/x))
\end{align*} \]

**Figure 3. Interpretation of Types**

**Corollary 5.3 (Identity extension)** For any type \( \sigma(\Theta) \) and \( \Theta \)-environment \( \gamma \), \( [\sigma]^{\Delta_{\gamma}}_R = \Delta_{\llbracket \sigma \rrbracket_\gamma} \).

Identity extension implies that all elements in the interpretation of polymorphic types \( \forall \alpha \). \( \sigma \) are relationally parametric.

Next, we define the interpretation of terms. Given a \( \Theta \)-context \( \Gamma \) and a \( \Theta \)-environment \( \gamma \), then a \( \Gamma \cdot \gamma \)-environment is any \( \rho \in \prod_{x \in \text{dom}(\Gamma)} [\Gamma(x)]_\gamma \). A term \( \Gamma \vdash \delta \vdash t: \sigma \) is interpreted as an element \( \llbracket t \rrbracket_{\gamma, \rho} \in [\sigma]_\gamma \), relative to any \( \Theta \)-environment \( \gamma \) and \( \Gamma \cdot \gamma \)-environment \( \rho \). The definition of \( \llbracket t \rrbracket_{\gamma, \rho} \) is given in Figure 4, where terms and subtrees are assumed to have the types given in Figure 1.

**Lemma 5.4 (Interpretation of terms)** If \( \Gamma \vdash \delta \vdash t: \sigma \):

1. (Well-definedness) For any \( \Theta \)-environment \( \gamma \) and \( \Gamma \cdot \gamma \)-environment \( \rho \), the value \( \llbracket t \rrbracket_{\gamma, \rho} \in [\sigma]_\gamma \) is well-defined.

2. (Strictness) If \( x: \tau \in \Gamma \) and \( \delta(x) = 1 \) then for any \( \Theta \)-environment \( \gamma \) and \( \Gamma \cdot \gamma \)-environment \( \rho \), the function \( d \mapsto \llbracket t \rrbracket_{\gamma, \rho(d/x)}: \llbracket \tau \rrbracket_\gamma \to [\sigma]_\gamma \) is strict.

3. (Relational parametricity) For any relational \( \Theta \)-environment \( \gamma_R \), any \( \Gamma \cdot \gamma \)-environment \( \rho_1 \) and \( \Gamma \cdot \gamma \)-environment \( \rho_2 \), define

\[ [\Gamma]_{\gamma_R} \rho_1, \rho_2 \] implies \( [\sigma]_{\gamma_R} ([\Gamma]_{\gamma_R} \rho_1 (x), \rho_2 (x)) \).

**Remark 5.5** The form of completeness that holds for the small category \( D \) (see Remark 3.1) is too weak for interpreting polymorphism in \( D \), see [16]. We side-step the problem by, instead, using the non-small (but properly complete) category \( \text{Dom} \) for the interpretation. The one difficulty that arises lies in interpreting type specialization. For this, the invariance of the interpretation of universal types under groupoid action, cf. [10, 3], which follows from parametricity, is crucial to showing that the definition of \( [\tau]_{\gamma_R} \) given in Figure 4 is independent of the choice of \( i \) and \( D \).

**Remark 5.6** Birkedal, Mogelberg and Petersen have been
studying a domain-theoretic version of the “parametric completion” process of [17]. It would be interesting to compare the category-theoretic model they obtain with the environment model constructed here.

Having constructed a relationally parametric model, we could, at this point, go on to prove useful consequences of parametricity. One of the most important consequences is the correctness of Plotkin’s impredicative encodings of the domain-theoretic type constructors (see [2] for details of the encodings, and for a proof of correctness for coproducts). However, for lack of space, and because the techniques needed for such applications of parametricity are known, we omit reworking the expected verifications in our setting. Instead, for the remainder of the paper, we concentrate on showing how our model can be used to prove operational properties of our language. In particular, we obtain denotational proofs of Theorems 4.5 and 4.9.

6. Computational Adequacy

In order to use our model to prove operational properties, it is necessary to prove computational adequacy.

For \( x, y \) in a set \( X \), we write \( x \sqsubseteq_\Sigma y \) to mean that, for all \( t : X \to \Sigma \), it holds that \( t(x) \) implies \( t(y) \). The \( \sqsubseteq_\Sigma \) relation is a preorder (but not necessarily a partial order).

**Theorem 6.1 (Computational adequacy)** The following equivalent properties all hold.

1. If \( t : ! \sigma \) then \( t \downarrow^s \) if and only if \( \exists d \in [\sigma], [t] = \{d\} \).
2. If \( s, t : \sigma \) then \( [s] \sqsubseteq_\Sigma [t] \) implies \( s \sqsubseteq_{\text{gnd}} t \).
3. If \( s, t : \sigma \) then \( [s] = [t] \) implies \( s \equiv_{\text{gnd}} t \).

We prove statement 1. The left-to-right implication is easily shown by proving that \( t \downarrow^s \) implies \( [t] = [s] \), by induction on the evaluation relation. The converse implication is proved by constructing an “approximation relation” between syntax and semantics. The construction is reminiscent of Girard’s method of proving strong normalization for the polymorphic \( \lambda \)-calculus, see e.g. [4]. A similar approach to computational adequacy for a polymorphic language was previously taken by Amadio [1], who worked concretely with PER models. Our language, with its treatment of strictness, is more refined than Amadio’s (e.g., ours supports Plotkin’s encodings of datatypes), and our proof works within the purely axiomatic setting of this paper.

The crucial point in formulating a usable notion of approximation relation is the identification of a suitable notion of “definedness” for an arbitrary domain \( D \). For \( d \in D \), we write \( d \downarrow \) to mean that there exists a strict function \( t : D \to \Sigma \) such that \( t(d) \).

For a domain \( D \) and closed type \( \sigma \), a **strict approximation relation** between \( D \) and \( \sigma \) is a relation \( \preceq \) between elements of \( D \) and closed terms of type \( \sigma \) satisfying:

- \((\text{sa}1)\) \( \{d \mid d \preceq t\} \) is a subdomain of \( D \).
- \((\text{sa}2)\) If \( d \downarrow \) implies \( d \preceq t \) then \( d \preceq t \).
- \((\text{sa}3)\) If \( d \preceq t \) and \( t \downarrow \) then \( d \preceq t \).
- \((\text{sa}4)\) If \( d \preceq t \) and \( d \downarrow \) then \( d \downarrow^s \).

We write \( A^s(D, \sigma) \) for the set of all strict approximation relations between \( D \) and \( \sigma \).

Given a set of type variables \( \Theta \), a **\( \Theta \)-substitution** is a family \( \vec{\tau} = \{\tau_\alpha\}_{\alpha \in \Theta} \) of closed types. For a type \( \sigma(\Theta) \), we write \( \sigma(\vec{\tau}/\Theta) \) for the evident closed type resulting from the substitution. A strict approximation \( \Theta \)-environment is a tuple \( \vec{\zeta} = (\gamma, \vec{\tau}, \{\preceq_{\alpha}\}_{\alpha \in \Theta}) \), where \( \gamma \) is a \( \Theta \)-environment, \( \vec{\tau} \) is a \( \Theta \)-substitution and \( \preceq_{\alpha} \in A^s(\gamma(\alpha), \tau_\alpha) \). For any type \( \sigma(\Theta) \) and strict approximation \( \Theta \)-environment \( \vec{\zeta} = (\gamma, \vec{\tau}, \{\preceq_{\alpha}\}_{\alpha \in \Theta}) \), Figure 5 defines a strict approximation relation \( \preceq_{\vec{\zeta}} \in A^s(\vec{\gamma}[\vec{\tau}/\Theta]) \).

**Remark 6.2** It takes some work to verify that \( \preceq_{\vec{\zeta}} \), is indeed a strict approximation relation. In fact, this is the first place in the paper that property 4 of Axiom 2 is used.

**Remark 6.3** To the reader acquainted with proofs of computational adequacy, the use of substitutions \( s'[t/x] \) for arbitrary terms \( t \) rather than just values, in the definition of \( \preceq_{\vec{\zeta}} \), may appear to conflict with the strict operational semantics. However, for our language, the proof of computational adequacy is insensitive to this issue: one could indeed restrict to values, but there is no need to do so.

Given a \( \Theta \)-context \( \Gamma \) and a \( \Theta \)-substitution \( \vec{\tau} \), a \( \Gamma, \vec{\tau} \)-substitution is a family \( t = \{t_x : \Gamma(x)[\vec{\tau}/\Theta]\}_{x \in \text{dom}(\Gamma)} \).
Lemma 6.4 Suppose $\Gamma | \delta \vdash_\Theta s : \sigma$. For any strict approximation $\Theta$-environment $\zeta = (\gamma, \bar{\tau}, \{\zeta_\alpha\}_\alpha)$, for any $\Gamma$-environment $\rho$ and $\Gamma$-$\bar{\tau}$-substitution $\bar{\iota}$, define

$$\rho \preceq^\Gamma_\zeta \bar{\iota} \iff \forall x \in \text{dom}(\Gamma), \ \rho(x) \preceq^\Gamma(x) t_x.$$  

Then $\rho \preceq^\Gamma_\zeta \bar{\iota}$ implies $[s]_{\gamma, \rho} \preceq^\zeta s[\bar{\tau}/\Theta][\bar{\iota}/\Gamma]$.

As usual, the lemma is proved by induction on $s$.

By the lemma, if $s : \sigma$ then $[s] \preceq^\sigma s$. The right-to-left implication of Theorem 6.1.1 follows.

7. Further Operational Properties

The goal of this section is to prove the strictness and operational extensionality theorems claimed in Section 4. For this, we define a second approximation relation between syntax and semantics. If the strictness theorem alone were our goal then it would be possible to simply modify the definitions of Section 6 by systematically replacing strict evaluation with non-strict evaluation. However, in order to prove operational extensionality, we need a more substantial alteration. Our proof adapts the technique of [11], where (in essence) operational extensionality for recursively typed languages is addressed, to the more expressive setting of our polymorphic language.

Definition 7.1 (Non-strict applicative simulation) The relation $\preceq^n_{\text{app}}$ is the largest relation between closed terms of identical closed type satisfying:

1. if $t \preceq^n_{\text{app}} t' : \sigma \rightarrow \tau$ then, for all terms $s : \sigma$, it holds that $t(s) \preceq^n_{\text{app}} t'(s) : \tau$;

2. if $t \preceq^n_{\text{app}} t' : ! \sigma$ then $t \vdash \eta^n ! s$ implies $t' \vdash \eta^n ! s'$ where $s \preceq^n_{\text{app}} s' : \sigma$; and

3. if $t \preceq^n_{\text{app}} t' : \forall \alpha. \sigma$ then, for all closed $\tau$, it holds that $t(\tau) \preceq^n_{\text{app}} t'(\tau) : \sigma[\tau/\alpha]$.

For a domain $D$ and closed type $\sigma$, a non-strict approximation relation between $D$ and $\sigma$ is a relation $\preceq^\sigma$ between elements of $D$ and closed terms of type $\sigma$ satisfying:

- (na1) $\{ d \mid d \not\preceq^\sigma \}$ is a subdomain of $D$.
- (na2) if $d \not\preceq^\sigma s$ then $d \not\preceq^\sigma t$.
- (na3) if $d \not\preceq^\sigma t$ and $t \preceq^n_{\text{app}} t'$ then $d \not\preceq^\sigma t'$.

We write $A^n(D, \sigma)$ for the set of all non-strict approximation relations between $D$ and $\sigma$.

A non-strict approximation $\Theta$-environment is a triple $(\gamma, \bar{\tau}, \{\zeta_\alpha\}_\alpha, \bar{\iota})$, where $\gamma$ is an $\Theta$-environment, $\bar{\tau}$ is a $\Theta$-substitution and $\zeta_\alpha \in A^n(\gamma(\alpha), \tau_\alpha)$. For any type $\sigma(\Theta)$ and non-strict approximation $\Theta$-environment $\zeta = (\gamma, \bar{\tau}, \{\zeta_\alpha\}_\alpha)$, Figure 6 defines a relation $\preceq^\zeta_\sigma \subseteq A^n([\sigma]_\gamma, [\bar{\tau}/\Theta])$. (Again, it takes some work to show that the definition is good.)

Lemma 7.2 Suppose $\Gamma | \delta \vdash_\Theta s : \sigma$. For any non-strict approximation $\Theta$-environment $\zeta = (\gamma, \bar{\tau}, \{\zeta_\alpha\}_\alpha)$, for any $\Gamma$-$\bar{\tau}$-environment $\rho$ and $\Gamma$-$\bar{\tau}$-substitution $\bar{\iota}$, define

$$\rho \preceq^\Gamma_\zeta \bar{\iota} \iff \forall x \in \text{dom}(\Gamma), \ \rho(x) \preceq^\Gamma(x) t_x.$$  

Then $\rho \preceq^\Gamma_\zeta \bar{\iota}$ implies $[s]_{\gamma, \rho} \preceq^\zeta s[\bar{\tau}/\Theta][\bar{\iota}/\Gamma]$.

Again, the proof is by induction on $s$. The lemma establishes, in particular, that if $s : \sigma$ then $[s] \preceq^\sigma s$.

Corollary 7.3 (Non-strict computational adequacy) If $t : ! \sigma$ then $t \vdash \eta^n ! t'$ and only if $\exists d \in [\sigma]$, $\{ [t] \} = \{ d \}$.

Theorem 4.5 (the strictness theorem) is an immediate consequence of the above corollary and Theorem 6.1.1.

Lemma 7.4 If $t, t' : \sigma$ then $[t] \preceq^\sigma t' \iff t \preceq^n_{\text{app}} t'$.

Proof (sketch). Using Lemma 7.2, one shows that the relation $[t] \preceq^\sigma t'$ satisfies implications 1–3 of Definition 7.1. So $[t] \preceq^\sigma t'$ implies $t \preceq^n_{\text{app}} t'$.

Conversely, if $t \preceq^n_{\text{app}} t'$ then, by Lemma 7.2, $[t] \preceq^\sigma t$. So, by (na3), $[t] \preceq^\sigma t'$.

Proposition 7.5 If $t, t' : \sigma$ then $t \preceq^{\text{and}} t' \iff t \preceq^n_{\text{app}} t'$.
The statement of the strictness theorem (Theorem 4.5) is a

Π

universal bi-implication between statements.

Π

Theorem 4.5 is an immediate consequence.

8. Justification

In this section, we briefly justify the correctness of our methods for establishing operational properties. All our axioms and arguments can be interpreted within any realizability topos \( \mathcal{E} \) satisfying the strong completeness axiom of [8], by taking predomains to be the well-complete objects. In *op. cit.* it is explicitly shown that Axioms 1, 2 and 4 of this paper are consequences of strong completeness. Moreover, the validity of Axiom 3 follows from the results of [7]. Thus all results of this paper are true when interpreted within \( \mathcal{E} \). It remains to argue that all results with operational content are true in reality.

For this, we observe the following. The evaluation relations \( t : \Sigma^n \), \( t : \Sigma^n v \), \( t : \Sigma^n v \) and \( t : \Sigma^n v \) are all \( \Sigma^0 \). Consequently, the statements \( t \subseteq_{gnd} t' \) and \( \ni_{gnd} t' \) are \( \Pi^0 \). Similarly, \( t \subseteq_{app} t' \) is \( \Pi^0 \), via the characterization of Proposition 4.8. The statement of the strictness theorem (Theorem 4.5) is a bi-implication between \( \Sigma^0 \) formulas, and thus also \( \Pi^0 \). The statement of operational extensionality (Theorem 4.9) is a universally quantified bi-implication between \( \Pi^0 \) formulas.

It is easily shown that a \( \Pi^0 \) sentence holds internally in \( \mathcal{E} \) if and only if it is true in reality (for the right-to-left implication, an unbounded search provides the realizator). It follows that implications between \( \Pi^0 \) statements that are valid in \( \mathcal{E} \) are also true in reality. Therefore, any operational conclusion of one of the forms discussed above, obtained as a result of the methods of this paper, is indeed true.

References


