A Generic Operational Metatheory for Algebraic Effects

Citation for published version:

Digital Object Identifier (DOI):
10.1109/LICS.2010.29

Link:
Link to publication record in Edinburgh Research Explorer

Document Version:
Peer reviewed version

Published In:
2010 25th Annual IEEE Symposium on Logic in Computer Science

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.
A Generic Operational Metatheory for Algebraic Effects

Patricia Johann
Dep. of Comput. and Inf. Sci.
University of Strathclyde
Glasgow G1 1XH, UK
patricia@cis.strath.ac.uk

Alex Simpson
School of Informatics
University of Edinburgh
Edinburgh EH8 9AB, UK
Alex.Simpson@ed.ac.uk

Janis Voigtlander
University of Bonn
53012 Bonn, Germany
jv@iai.uni-bonn.de

Abstract—We provide a syntactic analysis of contextual preorder and equivalence for a polymorphic programming language with effects. Our approach applies uniformly across a range of algebraic effects, and incorporates, as instances: errors, input/output, global state, nondeterminism, probabilistic choice, and combinations thereof. Our approach is to extend Plotkin and Power’s structural operational semantics for algebraic effects (FoSSaCS 2001) with a primitive “basic preorder” on ground type computation trees. The basic preorder is used to derive notions of contextual preorder and equivalence on program terms. Under mild assumptions on this relation, we prove fundamental properties of contextual equivalence on program terms. Under mild assumptions on this relation, we prove fundamental properties of contextual preorder (hence equivalence) including extensionality properties and a characterisation via applicative contexts, and we provide machinery for reasoning about polymorphism using relational parametricity.

I. INTRODUCTION

The goal of this paper is to give a systematic account of contextual equivalence for extensions of a functional language with non-functional effects, including errors, input/output, global state, (finite) nondeterminism, and combinations thereof. It thus fits into a tradition that includes studies by Mason and Talcott [1], Pitts and Stark [2], and Lassen [3], who respectively consider functional languages with mutable S-expressions, local state, and countable nondeterminism. The main novelty here is that, rather than considering each different kind of effect individually, we provide a general operational framework that can cater for a whole range of effects uniformly. Using this, we obtain generic operational metatheorems stating properties of contextual equivalence that hold for every instantiation of our framework to effects of interest.

The programming language we use as a vehicle for this is an extension of Plotkin’s PCF [4] with polymorphism and effects. Although, in the present paper, we consider a call-by-name version only, we believe our methods adapt also to call-by-value (see Section IX for further discussion).

The first difficulty we need to overcome is that the operational semantics of effects are not usually presented in a uniform manner. For example, in structural operational semantics [5]: pure functional languages are formalised using deterministic transitions $M \rightarrow M'$; nondeterminism introduces branching; languages with store require transitions $(M, s) \rightarrow (M', s')$ between state-carrying configurations; languages with input/output use labelled transitions, e.g., $M \xrightarrow{\text{f3}} M'$ (“output 3”); etc. To exaggerate only slightly, every new feature requires its own format.

To address this problem, we build on ideas of Plotkin and Power. In a major programme of research, see, e.g., [6], [7], [8], [9], they identify the class of algebraic effects, which are effects defined via well-behaved effect-triggering operations, and which include all the examples mentioned above. Algebraic effects can be given a uniform operational presentation, based on reducing a program to a computation tree (as defined in Section III below) that records all the effects (potentially) encountered during execution [6], [7]. If taken in isolation, the computation-tree presentation treats effect operations merely as uninterpreted syntax. Indeed, it is determined by the signature of effect operations alone. To furnish the effect operations with meaning, Plotkin and Power [6], [7] require a denotational semantics, with a suitable computational monad [10]. Their main result is then a generic computational adequacy theorem, relating the denotation of a term to that of its computation tree.

Our goal is a systematic study of contextual equivalence $\leq_{\text{ctx}}$, which is an operational notion. Accordingly, we propose a syntactic method of ascribing meaning to effect operations. We augment computation trees by identifying the minimum machinery needed to define $\leq_{\text{ctx}}$: an assumed equivalence relation $\leq_{\text{basic}}$ on ground type trees. In fact, for greater generality, we assume a preorder $\leq_{\text{basic}}$ on ground type trees, and use it to define contextual preorder $\leq_{\text{ctx}}$.

Our aim is to obtain generic operational metatheorems for $\leq_{\text{ctx}}$ (and $\leq_{\text{cte}}$), applying to any signature of effect operations together with a specified preorder $\leq_{\text{basic}}$. In order to do so, we need to assume that $\leq_{\text{basic}}$ satisfies two properties: it must be admissible and compositional, as defined in Section IV. These properties arise naturally when, as is frequently the case, the $\leq_{\text{basic}}$ relation can be described as an observational preorder determined by a
decomposable family of open observations on ground type trees, see Section V. The main examples of algebraic effects all have observational preorders induced in such a way. Thus our assumptions, and hence our theorems, are indeed applicable.

We prove our results using the powerful machinery of \(\sqsubseteq\)-closed relations, developed by Pitts and Stark [11], [12], [2]. This adapts naturally to our setting, by using the basic relation \(\sqsubseteq_{\text{basic}}\) to induce the required closure operations. The definitions are given in Section VI, whose culmination is the characterisation of contextual preorder as a logical relation (Theorem VI.6). The proof has the usual structure [2], [11], [12], but makes essential use of the admissibility and (Theorem VI.6). The proof has the usual structure [2], [11], [12], but makes essential use of the admissibility and compositional property of commuting with evaluation contexts (cf. Milner [13]. For call-by-value languages with effects, such operations as being specified by a signature \(\Sigma\) that assigns an arity to each effect operations \(\sigma\). We allow four different formats. The simplest case is that of a finite arity operation \(\sigma\), specified as \(\sigma : \alpha^n \rightarrow \alpha\) for some \(n \in \mathbb{N}\). Infinite arity operations are specified as \(\sigma : \alpha^\mathbb{N} \rightarrow \alpha\).

The remaining two cases are parameterised versions of the above. A parameterised finite arity operation is specified as \(\sigma : \mathbb{N} \times \alpha^n \rightarrow \alpha\), and a parameterised infinitary operation is specified as \(\sigma : \mathbb{N} \times \alpha^\mathbb{N} \rightarrow \alpha\). Computationally interesting examples of different cases are given below.

Types are assigned to terms according to the axioms and rules in Figure 2, where \(\Gamma\) ranges over typing environments of the form \(\vec{\alpha}, x_1 : \tau_1, \ldots, x_m : \tau_m\) for a finite list \(\vec{\alpha}\) of distinct type variables, \(m \in \mathbb{N}\) (where \(\mathbb{N}\) is the set of natural numbers including 0), a list \(\vec{x} = x_1, \ldots, x_m\) of distinct term variables, and types \(\tau_1, \ldots, \tau_m\) whose free variables are in \(\vec{\alpha}\). The typing is relative to the signature \(\Sigma\). In a typing judgement of the form \(\Gamma \vdash M : \tau\), with \(\Gamma\) as above, we require that the free variables of the term \(M\) are in \(\vec{\alpha}, \vec{x}\) and that the free variables of the type \(\tau\) are in \(\vec{\alpha}\).

Our main operational metatheorems about contextual preorder are in Section VII. For example, we prove extensionality properties, as well as a “context lemma” (Theorem VII.2) characterising contextual equivalence at arbitrary type via ground type applicative contexts, cf. Milner [13]. For call-by-value languages with effects, such results are known to hold only in restricted form, e.g., the “ciu” theorem of [1]. In contrast, in our call-by-name setting, the theorems remain valid in their classic form. We also establish various results about contextual equivalence between effect operations. For example, effect operations commute with evaluation contexts. This again connects with the work of Plotkin and Power, but whereas for them, the (denotational) property of commuting with evaluation contexts is taken as the definition of an algebraic operation, in our setting the operational analogue holds as a consequence of our operational semantics.

The inclusion of polymorphism in our language allows us to exploit one of the strengths of the logical-relation machinery: it yields principles for reasoning about relational parametricity [11], [12]. In Section VIII, we present one consequence of relational parametricity in our call-by-name effectful setting. The type \(\forall X. (\tau \rightarrow X) \rightarrow X\) is characterised as Moggi’s type \(T(\tau)\) [10], where \(T\) is a computational monad encapsulating the effects present in the language. This provides an operational vindication of a type identity that has previously been demonstrated denotationally for general monadic effects [14].

This paper is intended to be the first in a sequence, which will further extend the generic approach to wider classes of effects and languages. Some specific ideas in this direction and a discussion of other related work appear in Section IX.

For space reasons, all proofs are omitted from this conference version of the paper.

II. THE LANGUAGE

Our language is a polymorphic version of Plotkin’s PCF [4] with the addition of operations for manipulating effects. The syntax of types and terms is given in Figure 1, where \(\alpha\) and \(x\) range over disjoint countably infinite sets of type variables and term variables, respectively. The syntax is standard except for the effect operations. We shall consider such operations as being specified by a signature \(\Sigma\) that assigns an arity to each effect operations \(\sigma\). We allow four different formats. The simplest case is that of a finite arity operation \(\sigma\), specified as \(\sigma : \alpha^n \rightarrow \alpha\) for some \(n \in \mathbb{N}\). Infinite arity operations are specified as \(\sigma : \alpha^\mathbb{N} \rightarrow \alpha\).

The remaining two cases are parameterised versions of the above. A parameterised finite arity operation is specified as \(\sigma : \mathbb{N} \times \alpha^n \rightarrow \alpha\), and a parameterised infinitary operation is specified as \(\sigma : \mathbb{N} \times \alpha^\mathbb{N} \rightarrow \alpha\). Computationally interesting examples of different cases are given below.

Types are assigned to terms according to the axioms and rules in Figure 2, where \(\Gamma\) ranges over typing environments of the form \(\vec{\alpha}, x_1 : \tau_1, \ldots, x_m : \tau_m\) for a finite list \(\vec{\alpha}\) of distinct type variables, \(m \in \mathbb{N}\) (where \(\mathbb{N}\) is the set of natural numbers including 0), a list \(\vec{x} = x_1, \ldots, x_m\) of distinct term variables, and types \(\tau_1, \ldots, \tau_m\) whose free variables are in \(\vec{\alpha}\). The typing is relative to the signature \(\Sigma\). In a typing judgement of the form \(\Gamma \vdash M : \tau\), with \(\Gamma\) as above, we require that the free variables of the term \(M\) are in \(\vec{\alpha}, \vec{x}\) and that the free variables of the type \(\tau\) are in \(\vec{\alpha}\). We write \(\text{Typ}\) for the set of closed types. Given \(\tau \in \text{Typ}\), we write \(\text{Term}(\tau)\) for the set of terms \(M\) for which \(\emptyset \vdash M : \tau\) is derivable, where \(\emptyset\) is the empty typing environment. Further, we set \(\text{Term} = \bigcup_{\tau \in \text{Typ}} \text{Term}(\tau)\).

Example 1 (error). Let \(\text{Err}\) be a set of error labels. For each \(e \in \text{Err}\), the signature contains a zero-arity operation (i.e., constant) \(\text{raise}_e : \alpha^0 \rightarrow \alpha\).

Example 2 (nondeterminism). The signature contains a single binary choice operation or \(\alpha^2 \rightarrow \alpha\).

Example 3 (global state). Let \(\text{Loc}\) be a set of locations. For each \(l \in \text{Loc}\), we include operations \(\text{lookup}_l : \alpha^\mathbb{N} \rightarrow \alpha\) and \(\text{update}_l : \mathbb{N} \times \alpha \rightarrow \alpha\). These denote actions that can be performed on \(l\) considered as a storage cell: \(\text{lookup}_l(\lambda n :: \mathbb{N} : \text{Nat}. M)\) assigns the content of location \(l\) to variable \(n\) and proceeds as \(M\); \(\text{update}_l(n; M)\) evaluates \(N\), assigns the result to \(l\), and proceeds as \(M\).

Example 4 (input/output). The signature contains effect operations \(\text{read} : \alpha^\mathbb{N} \rightarrow \alpha\) and \(\text{write} : \mathbb{N} \times \alpha \rightarrow \alpha\). Intuitively, \(\text{read}(\lambda n :: \mathbb{N} : \text{Nat}. M)\) reads a number from input, assigns it to \(n\) and proceeds as \(M\); and \(\text{write}(N; M)\) evaluates \(N\) to a number, outputs it, and proceeds as \(M\).

The examples above do not involve any parameterised infinitary operations, i.e., operations of arity \(\mathbb{N} \times \alpha^\mathbb{N} \rightarrow \alpha\). Our main reason for including such operations is that they provide us with a single “hardest case” to consider in proofs. In addition, the formulation of some of the above operations may seem cumbersome. For example, in place of the polymorphic \(\text{update}_l\), one might prefer to use an assignment command \(l := M\) of unit type. We adopt the
former because it fits nicely into our polymorphic type theory, and does not require a unit type. The equivalence of the two styles is a general property of algebraic effects [9].

III. OPERATIONAL SEMANTICS

The subset of Term whose elements (called values) respect the following grammar is denoted by Value:

\[ V ::= \lambda x :: \tau. M \mid \Lambda \alpha. M \mid Z \mid S(V). \]

We write Value(\(\tau\)) for the values of type \(\tau\). The elements of Value(Nat) are the numerals \(Z, S(Z), S^2(Z), \ldots\), for which we use the shorthand \(\overline{0}, \overline{1}, \overline{2}, \ldots\).

The operational semantics of our language needs to specify both how closed terms evaluate to values and also any effect behaviour that may occur during execution. Following Plotkin and Power [7], we achieve the latter by abstracting away from the nature of the effects performed. We use a syntactically defined computation tree to capture every effect operation that could potentially occur during the given execution, irrespective of how the effect operations are themselves interpreted. The operational semantics will determine a function \(\cdot\) mapping every closed term \(M \in \text{Term}(\tau)\) to a computation tree \([M]\) of type \(\tau\). We first define the notion of computation tree, and then the machinery via which the computation tree of a term is determined.

**Definition III.1.** A computation tree of closed type \(\tau\) is a labelled tree each node of which is one of the following.

- A leaf node labelled \(\perp\).
- A leaf node labelled \(V\), where \(V \in \text{Value}(\tau)\).
- A node labelled \(\sigma\) with children \(t_1, \ldots, t_n\) indexed by \(\{1, \ldots, n\}\), where \(\sigma\) is an effect operation \(\sigma : \alpha^n \rightarrow \alpha\).
- A node labelled \((\sigma, m)\) with children \(t_1, \ldots, t_n\) indexed by \(\{1, \ldots, n\}\), where \(\sigma : \text{Nat} \times \alpha^n \rightarrow \alpha\).
- A node labelled \(\sigma\) with children \(t_0, t_1, t_2, \ldots\) indexed by \(\mathbb{N}\), where \(\sigma : \alpha^{\mathbb{N}} \rightarrow \alpha\).
- A node labelled \((\sigma, m)\) with children \(t_0, t_1, t_2, \ldots\) indexed by \(\mathbb{N}\), where \(\sigma : \text{Nat} \times \alpha^{\mathbb{N}} \rightarrow \alpha\).

It is immediate that every subtree of a computation tree is itself a computation tree of the same type. We define a partial ordering on computation trees (of the same type) by \(t_1 \sqsubseteq t_2\) if \(t_1\) can be obtained from \(t_2\) by pruning (possibly infinitely many) subtrees and labelling the pruning points with \(\perp\). It is standard that this order endows the set Trees(\(\tau\)) of trees of type \(\tau\) with the structure of an \(\omega\)-complete partial order (cpo), with least element \(\perp\).

The mechanism for evaluating a term to its computation tree uses reduce/reduct-pairs and a notion of reduction in context. We define these appropriately for call-by-name evaluation. The reduce/reduct-pairs are written \(R \leadsto R'\) (with \(R, R' \in \text{Term}\)) and listed in the following table:

<table>
<thead>
<tr>
<th>R</th>
<th>R'</th>
</tr>
</thead>
<tbody>
<tr>
<td>((\lambda x :: \tau. M))</td>
<td>(M[A/x])</td>
</tr>
<tr>
<td>((\Lambda \alpha. M))</td>
<td>(M[\tau/\alpha])</td>
</tr>
<tr>
<td>case (\overline{0}) of ({Z \Rightarrow M_1; S(x) \Rightarrow M_2})</td>
<td>(M_1)</td>
</tr>
<tr>
<td>case (n + \overline{1}) of ({Z \Rightarrow M_1; S(x) \Rightarrow M_2})</td>
<td>(M_2[\overline{n}/x])</td>
</tr>
<tr>
<td>fix((F))</td>
<td>(F \text{ fix}(F))</td>
</tr>
</tbody>
</table>

To describe reduction in context, we use the notions of evaluation frames and evaluation stacks, given by the grammars

\[ E ::= (- M) \mid - \tau \mid S(\cdot) \mid \sigma(\cdot; M_1, \ldots, M_\nu) \mid (\text{case } - \text{ of } \{Z \Rightarrow M_1; S(x) \Rightarrow M_2\}) \]

\[ S ::= \text{Id} \mid S \circ E \]

\[ S ::= \text{Id} \mid S \circ E \]

\[ S ::= \text{Id} \mid S \circ E \]

\[ S ::= \text{Id} \mid S \circ E \]

\[ S ::= \text{Id} \mid S \circ E \]

\[ S ::= \text{Id} \mid S \circ E \]
where the evaluation frame \( \sigma(\cdot; M, \ldots, M) \) is only present for \( \sigma \in \Sigma \) with \( \sigma : \text{Nat} \times \alpha^n \rightarrow \alpha \) or \( \sigma : \text{Nat} \times \alpha^\text{Nat} \rightarrow \alpha \). If an evaluation stack comprises a single evaluation frame \( E \), we denote it by \( E \) rather than \( \text{Id} \circ E \). Given an evaluation frame \( E \) and a term \( M \), we write \( E(M) \) for the term that results from replacing "\( \tau \)" by \( M \) in \( E \). Similarly, given a stack \( S \) and term \( M \), the application \( S(M) \) is defined by induction on the structure of \( S \) via the equations \( \text{Id}(M) = M \) and \( (S' \circ E)(M) = S'(E(M)) \).

A type assignment relation \( \Gamma \vdash S : \tau \rightarrow \tau' \) (where \( \Gamma \) again ranges over typing environments) assigns argument and result types to certain evaluation stacks. We define \( \Gamma \vdash S : \tau \rightarrow \tau' \) to hold if \( \Gamma, x : \tau \vdash S(x) : \tau' \), where \( x \) is fresh for \( \Gamma \). Given \( \tau, \tau' \in \text{Typ} \), we write \( \text{Stack}(\tau, \tau') \) for the set of stacks \( S \) for which \( \emptyset \vdash S : \tau \rightarrow \tau' \) is derivable. We set, for every \( \tau \in \text{Typ} \), \( \text{Stack}(\tau) = \text{Stack}(\tau, \text{Nat}) \).

Now a transition \((S_1, M_1) \rightarrow (S_2, M_2)\) (with \( M_1, M_2 \in \text{Term}(\tau) \) and \( S_1, S_2 \in \text{Stack}(\tau, \tau') \), for some \( \tau \) and \( \tau' \)) is possible for exactly the following combinations:

\[
\begin{array}{c|c}
(S, E(N)) & N \notin \text{Value} \\
(S, E(V)) & V \in \text{Value} \\
(S, R) & R \sim R'
\end{array}
\]

To define the computation tree associated with a term \( M \), one evaluates the term as the configuration \((\text{Id}, M)\). If this evaluation terminates with a value, that value is the tree. If it stops at an effect operation, then that operation is used to label the current node in the tree, and the procedure continues with appropriate stack/term configurations for each of the children nodes. If ever one gets to a nonterminating configuration, then the current node gets labelled with \( \bot \). The definition below implements this rigorously.

**Definition III.2.** We define a family of functions \( \cdot |_{\tau, \tau'} \), indexed by \( \tau, \tau' \in \text{Typ} \), mapping \( S \in \text{Stack}(\tau, \tau') \), and \( M \in \text{Term}(\tau) \) to a computation tree \( [S, M]_{\tau, \tau'} \in \text{Trees}(\tau') \). These functions are given by the canonical (i.e., least in the pointwise order on functions into \( \text{Trees}(\tau') \)) solution of the following recursive equation (omitting subscripts):

\[
[S, M] =
\begin{cases}
M & \text{if } S = \text{Id} \text{ and } M \in \text{Value} \\
[S', M'] & \text{if } (S, M) \rightarrow (S', M') \\
\sigma([S, M_1], \ldots, [S, M_n]) & \text{if } M = \sigma(M_1, \ldots, M_n), \sigma : \alpha^n \rightarrow \alpha \\
\sigma([S, M', \text{[]}], [S, M', \text{[]}], [S, M', \text{[]}], \ldots) & \text{if } M = \sigma(M'), \sigma : \alpha^\text{Nat} \rightarrow \alpha \\
\sigma([S, M_1], \ldots, [S, M_n]) & \text{if } M = \sigma(M_1, \ldots, M_n), \sigma : \text{Nat} \times \alpha^n \rightarrow \alpha \\
\sigma([S, M', \text{[]}], [S, M', \text{[]}], [S, M', \text{[]}], \ldots) & \text{if } M = \sigma(M', \text{[]}), \sigma : \text{Nat} \times \alpha^\text{Nat} \rightarrow \alpha
\end{cases}
\]

Finally, for \( M \in \text{Term}(\tau) \), we define \( |M| = |\text{Id}, M| \).

Note that the different cases in the above definition of \( |S, M| \) are exhaustive and mutually exclusive. Also, because we demand the canonical solution, we obtain \( |S, M| = \bot \) when \( (S, M) \) diverges under \( \rightarrow \). In particular, \( |S, \Omega_x| = \bot \), where \( \Omega_x = \text{fix}(\lambda x : \tau \cdot x) \).

**Example 3 (continued).** In the signature for global state, define \( M = \text{fix}(\lambda f : \text{Nat} \rightarrow \text{Nat}, \lambda n : \text{Nat} \cdot B) \) where \( B \) is

\[
case \text{n of } \{ \begin{array}{l}
\text{Z} \\
\text{S(n')}
\end{array} \Rightarrow \text{lookup}_1(\lambda c : \text{Nat} \cdot c); \text{S(n')} \Rightarrow \text{lookup}_1(\lambda c : \text{Nat} \cdot \text{update}_1(S(c); fn')) \}
\]

Then \( M \in \text{Term}(\text{Nat} \rightarrow \text{Nat}) \) and:

\[
[M \top] =
\begin{array}{c}
\text{lookup}_1 \\
\text{lookup}_1 \\
\text{lookup}_1 \\
\text{lookup}_1
\end{array}
\]

In this example, the computation tree contains branches that could not possibly arise according to the intended interpretation of the operations. For instance, the subtree \((\text{update}_1, 1)(\text{lookup}_1(0, \text{T}, 2, \ldots))\) represents the action: first, store the value 1 in the location \( l \); next, read the content of \( l \); if this is 0, return \( 0 \); if it is 1, return \( T \); if it is 2, return \( 2 \); etc. In this case, only the second branch from the choice point is possible, and the value returned will be \( T \). The reason for including such redundancy is that computation trees treat operations as syntax. Without specific interpretation, it is possible that all branches might be relevant. Indeed, if the example is transcribed to the signature for input/output then, in the corresponding subtree \((\text{write}, 1)(\text{read}(0, \text{T}, 2, \ldots))\), all branches are relevant.

**IV. CONTEXTUAL PREORDER**

Our operational semantics treats effect operations merely as uninterpreted syntax. In this section we show that we can uniformly account for the properties of specific effects not by directly defining how they behave, but instead by using computation trees to define a notion of contextual equivalence (and, more generally, preorder) between programs.

To define contextual preorder on programs, we require, as given, a basic preorder relation \( \sqsubseteq_{\text{basic}}^\text{cntxt} \) on ground type computation trees \( \text{Trees}(\text{Nat}) \). We illustrate some natural choices for \( \sqsubseteq_{\text{basic}}^\text{cntxt} \) in the context of our running examples.

**Example 1 (continued).** The only ground type computation trees are \( \bot \), numerals \( \pi \), and errors \( \text{raise}_e \). The natural preorder is the expected flat partial ordering: \( x \sqsubseteq_{\text{basic}} y \) iff \( x = \bot \) or \( x = y \).

**Example 2 (continued).** Different choices for \( \sqsubseteq_{\text{basic}}^\text{cntxt} \) correspond to different paradigms for nonterminating computation. We consider just two.
Ordinary nondeterminism.: Define the set results(t) of possible results of a computation tree t by:

\[ \text{results}(t) = \{ n \mid t \text{ has a leaf labelled with } \pi \} \cup \{ \bot \mid t \text{ has a leaf labelled with } \bot \text{ or } t \text{ has an infinite branch} \} \]

Define: \( t \sqsubseteq^\text{nd} t' \) iff either (i) \( \bot \in \text{results}(t) \) and \( \text{results}(t) - \{ \bot \} \subseteq \text{results}(t') \), or (ii) \( \text{results}(t) = \text{results}(t') \).

This is the well-known Egli-Milner ordering.

Probabilistic choice.: We consider the or operation as performing a fair probabilistic choice (coin toss) between the two alternatives. Given \( t \in \text{Trees}(\text{Nat}) \) and \( n \in \text{Nat} \), define \( P(t = \pi) \) to be the probability, within the tree \( t \) (considered as a Markov chain), of reaching a leaf labelled with \( \pi \). Define \( t \sqsubseteq^\text{pr} t' \) iff, for all \( n \), \( P(t = \pi) \leq P(t' = \pi) \).

**Example 3 (continued).** Define States = Loc \( \rightarrow \) \( \text{Nat} \). We define a partial function exec from \( \text{Trees}(\text{Nat}) \times \text{States} \) to \( \text{Nat} \times \text{States} \), where \( \text{exec}(t, s) \) represents the result of “executing” \( t \) starting from state \( s \). The idea is that \( \text{exec}(t, s) = (n, s') \) iff the execution returns value \( \pi \) with \( s' \) the state at termination. Formally, \( \text{exec} \) is the least-defined partial function satisfying the recursive equations:

\[
\text{exec}(\pi, s) = (n, s) \\
\text{exec}(\text{lookup}_i(t_0, t_1, \ldots), s) = \text{exec}(t_i(s), s) \\
\text{exec}(\text{update}_i, m)(t), s) = \text{exec}(t, s[l := m]),
\]

where \( s[l := m] \) is the evident state obtained from \( s \) by modifying \( l \). One sees that the definition of \( \text{exec} \) is such that redundant branches of the computation tree, as discussed at the end of Section III, are ignored. Define \( t \sqsubseteq^\text{bs} t' \) iff: for every \( s \in \text{States} \), if \( \text{exec}(t, s) \) is defined then so is \( \text{exec}(t', s) \) and \( \text{exec}(t, s) = \text{exec}(t', s) \).

**Example 4 (continued).** An i/o trace is a finite sequence of elements of three kinds: an input \( ?n \), an output \( !n \), or a return \( -n \), where, in each case, \( n \in \text{Nat} \). The sets \( \text{io}(t) \) of (not necessarily completed) i/o traces of computation trees \( t \) are defined as the smallest satisfying:

\[
\text{io}(\pi) = \{ n, \varepsilon \} \\
\text{io}(\text{read}(t_0, t_1, \ldots)) = \{ !\alpha \mid \alpha \in \text{io}(t_n) \} \cup \{ \varepsilon \} \\
\text{io}(\text{write}(n, t)) = \{ !\alpha \mid \alpha \in \text{io}(t) \} \cup \{ \varepsilon \},
\]

where \( \varepsilon \) is the empty sequence, and we concatenate by juxtaposition. Define \( t \sqsubseteq^\text{io} t' \) iff \( \text{io}(t) \subseteq \text{io}(t') \).

Note that in Examples 1 and 4, the basic preorders coincide with the partial order on \( \text{Trees}(\text{Nat}) \) defined in Section III.

The above definitions of the \( \sqsubseteq^\text{bs} \) preorder make use of the standard devices one meets in the usual presentations of operational semantics (e.g., the set States of global states). What we have gained, with respect to the standard presentations, is that each example now fits into the same uniform format; its definition consists of two components: a signature, and a preorder \( \sqsubseteq^\text{bs} \) on Trees(Nat).

We henceforth assume given a signature and \( \sqsubseteq^\text{bs} \) preorder, and we use this data to define contextual preorder for our language. We expect that the contextual preorder should satisfy natural compatibility (precongruence) conditions, and it should be the largest such relation consistent with the relation \( \sqsubseteq^\text{bs} \) at type Nat.

**Definition IV.1.** A well-typed relation \( \mathcal{E} \) comprises 4-tuples of the form \( (\Gamma, M, M', \tau) \) with \( \Gamma \vdash M : \tau \) and \( \Gamma \vdash M' : \tau \). We write \( \Gamma \vdash M \mathcal{E} M' : \tau \) when the tuple \( (\Gamma, M, M', \tau) \) is in \( \mathcal{E} \), and we abbreviate this to \( M \mathcal{E} M' : \tau \) if \( \Gamma = \emptyset \) (and we often omit \( \tau \), which is uniquely determined). The notions of reflexivity, transitivity and symmetry apply to well-typed relations in the obvious way.

**Definition IV.2.** A well-typed relation \( \mathcal{E} \) is said to be compatible if it is closed under the axioms and rules in Figure 3. It is said to be \( \sqsubseteq^\text{bs}-adapte \) if for every \( M, M' \in \text{Term}(\text{Nat}) \), we have that \( M \mathcal{E} M' \implies [M] \sqsubseteq^\text{bs} [M'] \).

Note that every compatible relation is reflexive.

The following result is standard (see, e.g., [3], [11], [12]).

**Proposition IV.3.** There exists a largest \( \sqsubseteq^\text{bs}-adapte \) compatible relation. This relation is a preorder. It is an equivalence relation if \( \sqsubseteq^\text{bs} \) is.

**Contextual preorder \( \sqsubseteq^\text{ctx} \) is defined to be the largest \( \sqsubseteq^\text{bs}-adapte \) admissible compatible relation, as given by the proposition above. Contextual equivalence \( =^\text{ctx} \) is then defined as \( \langle \sqsubseteq^\text{bs} \rangle \cap \langle \sqsubseteq^\text{ctx} \rangle \). In fact, contextual equivalence itself arises as a special case of a contextual preorder, namely \( =^\text{ctx} \) is the contextual preorder obtained by taking the equivalence relation \( \langle \sqsubseteq^\text{bs} \rangle \cap \langle \sqsubseteq^\text{bs} \rangle \) as the basic preorder.

The main goal of this paper is to establish fundamental properties of contextual preorder and equivalence in the general context we have developed thus far: a signature of effect operations together with a primitive preorder \( \sqsubseteq^\text{bs} \) on ground type computation trees. However, we will need to impose two further conditions on the preorder \( \sqsubseteq^\text{bs} \) for the results to hold. The first exploits the \( \omega \)-cpo structure on Trees(Nat), discussed in Section III.

**Definition IV.4.** We say that \( \sqsubseteq^\text{bs} \) is admissible if, for all ascending chains \( t_n, t'_n \) in Trees(Nat), if \( t_n \sqsubseteq^\text{bs} t'_n \) for all \( n \), then \( \bigcup_{n \geq 0} t_n \sqsubseteq^\text{bs} \bigcup_{n \geq 0} t'_n \).

The second condition requires a substitution operation on ground type computation trees. Given \( t, t_0, t_1, \ldots \in \text{Trees}(\text{Nat}) \), define \( t(t_n / \pi)_n \) to be the computation tree obtained by replacing, for every \( n \in \text{Nat} \), each node labelled \( \pi \) in t with the tree \( t_n \) as a subtree rooted at that node.

**Definition IV.5.** We say that \( \sqsubseteq^\text{bs} \) is compositional if,
Admissible and compositional preorders enjoy useful closure properties. If \( \sqsubseteq \) is admissible and compositional, then so is its converse \( \sqsupseteq \). Also, the intersection of any family of admissible and compositional preorders is again admissible and compositional. Thus, when \( \sqsubseteq \) is admissible and compositional, so is \( \equiv \).

In the next section we shall see that all the instances of \( \sqsubseteq \) relations considered above, for the various running examples, are indeed admissible and compositional.

V. Observational Preorders

In this section we describe a useful and rather general method of specifying \( \sqsubseteq \) relations and establishing their admissibility and compositionality. This defines \( \sqsubseteq \) via a collection of possible “observations” on ground type computation, and thus implements the contextual preorder \( \sqsubseteq_{ctx} \) as an “observational preorder”.

**Definition V.1.** An observation \( O \) is a subset of \( Trees(Nat) \).

The idea is that the chosen subset of \( Trees(Nat) \) represents those computations that we observe to satisfy some property being tested for.

**Definition V.2.** Given a family \( O \) of observations, the preorder \( \sqsubseteq_{O} \) is defined by: \( t \sqsubseteq_{O} t' \) iff, for all \( O \in O \), it holds that \( t \in O \) implies \( t' \in O \).

The basic preorders defined in the previous section, for our running examples, all arise from families of observations.

**Example 1 (continued).** The family of observations is \( O^{err} = \{ \{ \overline{n} \} \mid n \in \mathbb{N} \} \cup \{ \{ \text{raise} \_ e \} \mid e \in \text{Err} \} \).

**Example 2 (continued).**

...the evident similar rules for other arities of \( \sigma \)

![Figure 3. Compatibility properties.](image-url)
The following example of an interesting preorder violating this condition was suggested by Paul Levy.

**Example 2 (continued).** The inclusion preorder for nondeterminism is defined by \( t \sqsubseteq_{\text{inclusion}} t' \) iff \( \text{results}(t) \subseteq \text{results}(t') \). Then \( \sqsubseteq_{\text{inclusion}} \) is not determined by a family of open observations since \( \bot \sqsubseteq_{\text{inclusion}} \top \). Nevertheless, it is admissible and compositional.

**Definition V.6.** A family of observations, \( O \), is called decomposable if, whenever \( t(t_n/\pi)_n \in O \in \mathcal{O} \), there exist subfamilies \( O' \) and \( (O'_n) \) of \( O \) such that:

(i) \( t \in \bigcap O' \) and \( t_n \in \bigcap O'_n \) for all \( n \); and

(ii) whenever \( t' \) and \( (t'_n) \) are such that \( t' \in \bigcap O' \) and \( t'_n \in \bigcap O'_n \) for all \( n \), then it holds that \( t'(t'_n/\pi)_n \in O \).

**Proposition V.7.** The relation \( \sqsubseteq_{\text{basic}} \) is compositional if and only if \( O \) is a decomposable family of observations.

Propositions V.4 and V.7 provide a uniform means for specifying admissible and compositional \( \sqsubseteq_{\text{basic}} \) relations: such relations are obtained as \( \sqsubseteq_{\text{basic}} \) whenever \( O \) is a decomposable family of open observations. One gain here is that openness and decomposability often turn out to be easy to establish in practice. Indeed, Propositions V.4 and V.7 serve the simplification of reducing properties of binary relations between trees to properties of sets of trees.

All the families of observations given for the examples above are easily shown to satisfy the openness and decomposability conditions. Thus the \( \sqsubseteq_{\text{basic}} \) relations for our running examples are all both admissible and compositional.

**VI. A LOGICAL CHARACTERISATION OF \( \sqsubseteq_{\text{ctx}} \)**

We now take up our main task of establishing properties of contextual preorder. For the remainder of the paper, we assume that \( \sqsubseteq_{\text{basic}} \) is admissible and compositional.

This section develops the main technical tool: a characterisation of \( \sqsubseteq_{\text{ctx}} \) via a logical relation. The presence of recursion and effect operations in our language means we need to restrict to a well-behaved class of term relations. For this, we use the closure operator approach of Pitts and Stark [2], [11], [12]. This treats terms and stacks as complementary ingredients which combine to reduce relations to a universal “test” relation. This adapts naturally to our setting, by using \( \sqsubseteq_{\text{basic}} \) as the test relation.

Given \( \tau, \tau' \in \text{Typ} \), we define \( \text{Rel}(\tau, \tau') = \mathcal{P}(\text{Term}(\tau) \times \text{Term}(\tau')) \) and \( \text{Rel}^\uparrow(\tau, \tau') = \mathcal{P}(\text{Stack}(\tau) \times \text{Stack}(\tau')) \). For \( r \in \text{Rel}(\tau, \tau') \) and \( s \in \text{Rel}^\uparrow(\tau, \tau') \) we define \( r^\uparrow \in \text{Rel}^\uparrow(\tau, \tau') \) and \( s^\downarrow \in \text{Rel}(\tau, \tau') \) by:

\[
(S, S') \in r^\uparrow \iff \forall (M, M') \in r \cdot |S, M| \sqsubseteq_{\text{basic}} |S', M'| \\
(M, M') \in s^\downarrow \iff \forall (S, S') \in s \cdot |S, M| \sqsubseteq_{\text{basic}} |S', M'|.
\]

As for any order-preserving Galois connection, we always have \( r \subseteq r^\uparrow \cup (r^\uparrow)^\uparrow = r^\uparrow \), and \( r_1 \subseteq r_2 \Rightarrow r_1^\uparrow \subseteq r_2^\uparrow \).

We say that \( r \) is \( \top\top \)-closed if \( r^\uparrow = r \). An important property of \( \top\top \)-closed relations is that they respect contextual preorder.

**Lemma VI.1.** Suppose \( r \in \text{Rel}(\tau_1, \tau_2) \) is \( \top\top \)-closed. For all \( M_1, M_1' \in \text{Term}(\tau_1) \) and \( M_2, M_2' \in \text{Term}(\tau_2) \), if \( M_1 \sqsubseteq_{\text{ctx}} M_1, (M_1, M_2) \in r \) and \( M_2 \sqsubseteq_{\text{ctx}} M_2' \), then \( (M_1', M_2') \in r \).

We now define the logical relation, which maps a type and a list containing relations as interpretations for the type’s free variables to a new relation.

**Definition VI.2.** For every type \( \tau, n \in \mathbb{N} \), list \( \tilde{\alpha} = \alpha_1, \ldots, \alpha_n \) of distinct type variables containing the free variables of \( \tau \), lists \( \tilde{r} = r_1, \ldots, r_n \) of \( \top\top \)-closed types, and list \( \tilde{r}' = r_1', \ldots, r_n' \) of closed types, and list \( r_i = r_1, \ldots, r_n \) with \( r_i \in \text{Rel}(\tau_1, \tau_1') \) for every \( 1 \leq i \leq n \), we define \( \Delta_n(\tilde{r}/\tilde{\alpha}) \in \text{Rel}(\tau[\tilde{r}/\tilde{\alpha}], \tau'[\tilde{r}'/\tilde{\alpha}]) \) by induction on the structure of \( \tilde{r} \) by:

\[
\Delta_{\alpha_i}(\tilde{r}/\tilde{\alpha}) = r_i \\
\Delta_{\alpha_i \rightarrow \alpha}(\tilde{r}/\tilde{\alpha}) = \{ (F, F') \mid \forall (A, A') \in \Delta_{\alpha}(\tilde{r}/\tilde{\alpha}) \cdot (F \downarrow A, F' \downarrow A') \in \Delta_{\alpha}(\tilde{r}/\tilde{\alpha}) \} \\
\Delta_{\text{Nat}}(\tilde{r}/\tilde{\alpha}) = \{ (\text{Id}, \text{Id}) \}^T
\]

**Lemma VI.3.** Let \( \tau, \tilde{\alpha}, \) and \( \tilde{r} \) be as in Definition VI.2. If every relation in \( \tilde{r} \) is \( \top\top \)-closed, then so is \( \Delta_n(\tilde{r}/\tilde{\alpha}) \).

To characterise \( \sqsubseteq_{\text{ctx}} \) via the logical relation from Definition VI.2, we first have to lift the latter from closed terms to a relation on terms containing possibly free variables, that is, to a well-typed relation in the sense of Definition IV.1.

**Definition VI.4.** Given \( \Gamma = \tilde{\alpha}, x_1 : \tau_1, \ldots, x_m : \tau_m \) such that \( \Gamma \vdash M : \tau \) and \( \Gamma \vdash M' : \tau \), we write \( \Gamma \vdash M \Delta M' : \tau \) if \( M[\tilde{\alpha}/\tilde{\alpha}], N[N'/N'] \in \Delta_n(\tilde{r}/\tilde{\alpha}) \) for every \( \tilde{\alpha} = \alpha_1, \ldots, \alpha_n \) and \( \tilde{\alpha}' = \alpha_1', \ldots, \alpha_n' \), with all types closed, every \( r_i \in \text{Rel}(\sigma_i, \sigma_i') \) is \( \top\top \)-closed, and every \( N_1 = N_1, \ldots, N_m \) and \( N' = N_1', \ldots, N_m' \) where each \( (N_j, N_j') \in \Delta_n(\tilde{r}/\tilde{\alpha}) \).

The main step in proving that \( \Delta \) and \( \sqsubseteq_{\text{ctx}} \) coincide is to establish that \( \Delta \) enjoys the fundamental property of logical relations, namely that it is reflexive. As usual, one proves, by induction on terms, the stronger statement that \( \Delta \) is compatible. The new features in our setting are: the admissibility of \( \sqsubseteq_{\text{basic}} \) is used in the case of the fixpoint operator, and the compositional property of \( \sqsubseteq_{\text{basic}} \) is needed to prove compatibility for the effect operations.

**Theorem VI.5.** The relation \( \Delta \) is compatible.

**Theorem VI.6.** The relations \( \Delta \) and \( \sqsubseteq_{\text{ctx}} \) coincide.

**VII. PROPERTIES OF CONTEXTUAL PREORDER**

We now present our main operational metatheorems about contextual preorder. Since, by remarks in Sections IV and V,
contextual equivalence arises as the contextual preorder derived from $=_{\text{basic}}$, which is admissible and compositional whenever $=_{\text{basic}}$ holds. All results apply equally to contextual equivalence. The first result reduces $\subseteq_{\text{ctx}}$ on open terms and types to its restriction to closed terms and types.

**Theorem VII.1.** \(\alpha_1, \ldots, \alpha_m, x_1 :: \tau_1, \ldots, x_n :: \tau_n \vdash M \subseteq_{\text{ctx}} M' :: \tau \text{ iff for all } \sigma_i \in \text{Typ} \text{ (} i = 1, \ldots, m \text{) and all } N_j \in \text{Term} \text{ (} \tau_j[\sigma/\alpha] \text{) (} j = 1, \ldots, n \text{), it is the case that } M[\sigma/\alpha, N/\alpha] \subseteq_{\text{ctx}} M'[\sigma/\alpha, N/\alpha].\)

The next result is our “context lemma”. Contextual preorder at closed terms and types can be reduced to the ground type relation $\subseteq_{\text{basic}}$ merely by considering “applicative” stacks.

**Theorem VII.2.** Let \(M, M' \in \text{Term} (\tau)\). Then \(M \subseteq_{\text{ctx}} M'\) iff for every applicative stack \(S \in \text{Stack} (\tau)\), it holds that \([S, M] \subseteq_{\text{basic}} [S, M']\).

The proof is in two halves. First, it is shown that the result holds if \(S\) ranges over all stacks, rather than just applicative ones. This weaker result, which is a “ciu” theorem, in the sense of [1], follows by a standard application of the logical relation \( \Delta \), cf. [11], [12]. Second, making crucial use of the compositionality of $\subseteq_{\text{basic}}$, it is shown that applicative stacks are just as discriminating as arbitrary stacks.

Since the only applicative stack in \(\text{Stack} (\text{Nat})\) is \(\text{Id}\), it is immediate from Theorem VII.2 that contextual preorder at ground type coincides with $\subseteq_{\text{basic}}$.

**Corollary VII.3.** Let \(M, M' \in \text{Term} (\text{Nat})\). Then \(M \subseteq_{\text{ctx}} M'\) iff \([M] \subseteq_{\text{basic}} [M']\).

We also have that functions behave extensionally at both type and term levels.

**Theorem VII.4.**

\(\text{a) For every } G, G' \in \text{Term} (\forall \alpha. \tau) \text{ we have } G \subseteq_{\text{ctx}} G' \text{ if and only if for all } \tau' \in \text{Typ} \text{ it holds that } G_{\tau'} \subseteq_{\text{ctx}} G'_{\tau'}\).

\(\text{b) For every } F, F' \in \text{Term} (\tau_1 \rightarrow \tau_2) \text{ we have } F \subseteq_{\text{ctx}} F' \text{ if and only if for every } A \in \text{Term} (\tau_1) \text{ it holds that } F A \subseteq_{\text{ctx}} F' A.\)

The reduction pairs in the table in Section III preserve contextual equivalence.

**Theorem VII.5.** \(\equiv_{\text{ctx}} R\), for any reduction-pair \( (L,R) \).

Using the results above, one can show that the usual $\beta$- and $\eta$-equalities, for both types and terms, hold up to $=_{\text{ctx}}$.

The next result shows that effect operations commute with stacks, which is the fundamental property that characterises effect operations as being algebraic in the work of Plotkin and Power [7]. We state the commutativity result for just one of the four possible arities for \(\sigma\).

**Theorem VII.6.** Let \(\sigma : \text{Nat} \times \alpha^{\text{Nat}} \rightarrow \alpha, M \in \text{Term}(\text{Nat}), M_1 \in \text{Term}(\text{Nat} \rightarrow \tau), \text{ and } S \in \text{Stack}(\tau).\)

Then \[S\{\sigma(M; M_1)\} =_{\text{ctx}} \sigma(M; \lambda x :: \text{Nat}. S\{M_1; x\})\]

A further useful result is that inequations between effect operations hold at all types if and only if they hold at ground type. For space reasons, we omit this result from this conference paper, since its formulation requires introducing a syntax for algebraic terms over infinitary operations. One consequence of the result is that inequations between effect operations, such as the equational axioms considered by Plotkin and Power, see, e.g., [8], can be read off from their ground type instances, as specified by $\subseteq_{\text{basic}}$.

To end the section, we make a few remarks about the necessity or otherwise of the assumed conditions on $\subseteq_{\text{basic}}$. As observed by a reviewer, results that assert specific contextual equivalences, such as Theorems VII.5 and VII.6, hold for an arbitrary preorder $\subseteq_{\text{basic}}$, even in the absence of admissibility and compositionality. This is because, by the results of this section, they hold if $\subseteq_{\text{basic}}$ is taken to be tree equality, which is obviously admissible and compositional. Since all other preorders are coarser than equality, so are the corresponding contextual equivalences.

In contrast, we believe that, under mild assumptions, the statement of Corollary VII.3 can be shown to be equivalent to the compositionality of $\subseteq_{\text{basic}}$. In any case, Corollary VII.3 does require some conditions to be imposed on $\subseteq_{\text{basic}}$, hence so does Theorem VII.2.

**VIII. REASONING ABOUT RELATIONAL PARAMETRICITY**

A major application of the $\tau\Gamma$-closed logical relations in the literature is to reason about parametricity properties of polymorphism [11], [12], [15], [16], [17]. In this section, we use the logical relation of Section VI to obtain one example of a consequence of parametricity for our language. We show that, in our effectful setting, the type $\forall \alpha. (\tau \rightarrow \alpha) \rightarrow \alpha$ acts like Moggi’s “monadic” type $T(\tau)$. This phenomenon has previously been demonstrated in a denotational setting in [14]. Operationally, in our call-by-name language, it means that this type represents effectful computations which return, as values, suspended terms of type $\tau$.

Define $T(\tau)$ as an abbreviation for $\forall \alpha. (\tau \rightarrow \alpha) \rightarrow \alpha$. Given a term $\Gamma \vdash M :: \tau$, we define $[M]$ to be $\Lambda \alpha. \lambda f :: \tau \rightarrow \alpha. (f M)$. Note that $\Gamma \vdash [M] :: T(\tau)$. Intuitively, the term $[M]$ is the “thunk” of type $T(\tau)$ that suspends the evaluation of the term $M$. Also, given terms $\Gamma \vdash M :: T(\tau)$ and $\Gamma, x :: \tau \vdash N :: \tau'$, define $\text{let } x \leftarrow M \text{ in } N$ to be $M_\nu (\lambda x :: \tau . N)$. Clearly $\Gamma \vdash \text{let } x \leftarrow M \text{ in } N :: \tau'$ (In the context of our call-by-name language, it is correct to allow an arbitrary type $\tau'$ here rather than restrict to types of the form $T(\tau')$.) Operationally, the term $[M]$, for closed $M$, is a value. Also, the context $\text{let } x \leftarrow - \text{ in } N$ is given by the (applicative)
Theorem VII.2.1. Given \( \Gamma \vdash M :: \tau \) and \( \Gamma, x :: \tau \vdash N :: \tau' \), there is:

\[
\Gamma \vdash \text{let } x = \sigma(M) \text{ in } N =_{ctx} \sigma(\lambda n :: \text{Nat} . \text{let } x = (M n) \text{ in } N).
\]

The theorem below states the two fundamental contextual equivalences relating the above derived constructs. Taken together, these properties assert the correct universal property of Moggi’s “let” operation; for example, for \( \sigma : a^{\text{Nat}} \rightarrow a \), we have:

\[
\text{let } x = \sigma(M) \text{ in } N =_{ctx} \sigma(\lambda n :: \text{Nat} . \text{let } x = (M n) \text{ in } N).
\]

Theorem VIII.1. Given \( \Gamma \vdash M :: \tau \) and \( \Gamma, x :: \tau \vdash N :: \tau' \), we have \( \Gamma \vdash \text{let } x = [M] \text{ in } N =_{ctx} N[M/x] \).

a) Given \( \Gamma \vdash M :: \tau \) and \( \Gamma, x :: \tau \vdash N :: \tau' \), we have \( \Gamma \vdash \text{let } x = M \text{ in } S[[x]] =_{ctx} S\{M\} \).

b) Given \( \Gamma \vdash M :: T(\tau) \) and \( \Gamma \vdash S :: T(\tau) \rightarrow \tau' \), we have \( \Gamma \vdash \text{let } x = M \text{ in } S\{x\} =_{ctx} S\{M\} \).

The proof, which follows the lines of the denotational proof in [14], requires a lemma, which we highlight because it is useful in other applications of relational parametricity: the graphs, up to \( \subseteq_{ctx} \), of functions given by stacks are \( \tau \tau' \)-closed. (The necessity of working up to \( \subseteq_{ctx} \) is explained by Lemma VI.1.) For every \( S \in \text{Stack}(\tau, \tau') \), define \( \text{left-graph} \_S \in \text{Rel}(\tau, \tau') \) and \( \text{right-graph} \_S \in \text{Rel}(\tau', \tau) \) by:

\[
(M, M') \in \text{left-graph} \_S \iff S\{M\} \subseteq_{ctx} M',
\]

\[
(M', M) \in \text{right-graph} \_S \iff M' \subseteq_{ctx} S\{M\}.
\]

Lemma VIII.2. Let \( \tau, \tau' \in \text{Typ} \) and \( S \in \text{Stack}(\tau, \tau') \). Then \( \text{left-graph} \_S \) and \( \text{right-graph} \_S \) are \( \tau \tau' \)-closed.

Finally, again following [14], we give a simple description of the logical relation \( \Delta \) at types \( T(\tau) \). This indicates how Definition VI.2 can be extended in the case of languages with a primitive \( T(\cdot) \) type constructor.

\[
\Delta_{T(\tau)}(\vec{r}/\vec{a}) = \{ ([M], [M']) \mid (M, M') \in \Delta_{T(\tau)}(\vec{r}/\vec{a}) \}^{TT}
\]

IX. Related and Future Work

The idea of providing a uniform operational metatheory for a family of language features has been popular in concurrency theory, where, e.g., general operational rule formats have been studied (see [18] for a survey), and principled methods of deriving behavioural equivalence from reduction rules have been identified [19], [20]. For higher-order languages, possibly involving concurrency, state, and polymorphism, the theory of \textit{environmental bisimulations}, has emerged as a general tool for analysing contextual equivalence [21], [22]. However, the empirical robustness of this theory has not yet been explained via any uniform operational metatheory. Indeed, as far as we know, our work is the first to attempt a systematic operational metatheory that applies to the setting of functional languages with effects. To this end, we have combined two previously distinct research areas: Plotkin and Power’s theory of algebraic effects [6], [7], [8], [9], and Pitts and Stark’s logical-relation-based approach to operational reasoning [2], [11], [12]. By adapting the latter to a range of effects, we provide further evidence of its versatility, see also [15], [16], [17]. Our results have been obtained for a call-by-name language. For call-by-value, the statements of the metatheorems need adjusting. For example, the context lemma does not hold as formulated in Theorem VII.2, since \( \Omega_{\text{Nat}} \rightarrow_{ctx} \text{Nat} \) and \( \lambda x :: \text{Nat} . \Omega_{\text{Nat}} \) are not call-by-value contextually equivalent. Nevertheless, we believe our methods will adapt straightforwardly to establish the correct metatheorems in a call-by-value setting, simply by adjusting the definition of the logical relation in the appropriate way, cf. [12].

Plotkin and Power originally considered computation trees in tandem with a denotational semantics given via a computational monad \( T \) on \( \omega \text{cpos} \) [7]. Plotkin [6] uses such a denotational semantics to define contextual preorder for a (call-by-value) language with effects. His definition does not adapt directly to our polymorphic language, because it is not known how to model the combination of polymorphism and effects in \( \omega \text{cpos} \). Nevertheless, one can instead use the monad \( T \) to give a denotational definition of a \( \subseteq_{\text{basic}} \) relation on computation trees, and thereby derive a contextual preorder as in Section IV. It can be shown that any such denotationally determined \( \subseteq_{\text{basic}} \) relation is automatically admissible and compositional. Thus all our operational metatheorems apply. Plotkin raises the question of whether context-lemma-like results are available for such denotationally determined contextual preorder over his (monomorphic) call-by-value language [6]. Our Theorem VII.2 answers the analogous question affirmatively, for our (polymorphic) call-by-name language.

Our existing theory has certain limitations in its applicability. Indeed, our running examples pretty much cover the range of effects catered for, except for one aspect: we can also deal with combinations of effects. A combination of instances of our framework can itself be considered an instance by amalgamating the signatures of effect operations, and then specifying a suitable admissible and compositional \( \subseteq_{\text{basic}} \) relation on the resulting trees. While this is easily done in individual cases, a more compelling story would be to show that the “sum” and “tensor” operations, in [23], for combining Lawvere theories have analogues in our setting for combining \( \subseteq_{\text{basic}} \) relations, and that these operations correctly account for the principal examples of combinations of algebraic effects. A further interesting development would show that a similar methodological approach to combination is available via constructions on the families of observations of Section V. This might be possible by viewing observations as being generated by modal operators, and by considering methods of combining modal logics. It is also plausible that viewing observations as modalities might lead to a uniform method of deriving logical characterizations of
contextual equivalence.

Our current theory does not cover all effects that have been described as "algebraic" in the literature. For example, the admissibility requirement on $\sqsubseteq_{\text{basic}}$ rules out countable nondeterminism [3]. Also, local state [8] is not incorporated because it requires a type of scopable locations to be used as a parameter and arity in effect operations. While our permitted arities could potentially be generalised beyond finite and Nat, cf. [6], the inclusion of more general arities would raise tricky issues about how to index the branching in computation trees, and how to ensure the compatibility of $\sqsubseteq_{\text{basic}}$ with respect to such indices. While such issues should be solvable in the case of ground-type local store, they are likely to present significant problems for more complex effects such as higher-order store.

A promising direction for further research is to extend our theory to allow effect operations that are blatantly non-algebraic, such as exception handlers and other control primitives. These can be added to our operational semantics by specifying new evaluation frames and reduction rules for them, but leaving the computation trees and the crucial $\sqsubseteq_{\text{basic}}$ relations unchanged. Thus we envisage a general theory in which effect operations are separated into two groups: the algebraic operations, which are the building blocks of the computation tree, and which give rise to the "observables" of behaviour; and the control primitives, which contribute to the reduction rules. While the new reduction rules for control will invalidate our operational metatheorems in their present form, we believe that the methodology of using a logical relation to analyse the contextual preorder derived from a $\sqsubseteq_{\text{basic}}$ relation on computation trees will still be applicable to obtain the correct metatheorems at this generality.

ACKNOWLEDGMENTS

We thank Paul Levy, Gordon Plotkin and the anonymous reviewers for helpful suggestions.

REFERENCES


