BRICS NS-04-1 Aceto et al. (eds.): SOS '04 Preliminary Proceedings

Basic Research in Computer Science

Preliminary Proceedings of the Workshop on

Structural Operational Semantics

SOS '04

London, United Kingdom, August 30, 2004

Luca Aceto
Willem Jan Fokkink
Irek Ulidowski (editors)

BRICS Notes Series **NS-04-1**

ISSN 0909-3206 **August 2004**

See back inner page for a list of recent BRICS Notes Series publications. Copies may be obtained by contacting:

BRICS
Department of Computer Science
University of Aarhus
Ny Munkegade, building 540
DK–8000 Aarhus C
Denmark

Telephone: +45 8942 3360
Telefax: +45 8942 3255
Internet: BRICS@brics.dk

BRICS publications are in general accessible through the World Wide Web and anonymous FTP through these URLs:

`http://www.brics.dk`
`ftp://ftp.brics.dk`
This document in subdirectory `NS/04/1/`

Electronic Notes in Theoretical Computer Science

Structural Operational Semantics

London, UK
August 30, 2004

Guest Editors:
Luca Aceto, BRICS, Aalborg University, Denmark
Wan Fokkink, CWI, Amsterdam, The Netherlands
Irek Ulidowski, University of Leicester, United Kingdom

Table of Contents

Foreword

The workshop on Structural Operational Semantics aimed at being a forum for researchers, students and practitioners interested in new developments, and directions for future investigation, in the field of structural operational semantics. One of the specific goals of the workshop was to establish synergies between the concurrency and programming language communities working on the theory and practice of SOS. Moreover, it aimed at widening the knowledge of SOS among postgraduate students and young researchers from the U.K. and abroad.

This volume contains the *preliminary proceedings* of the workshop, which was held in London (U.K.) on 30 August 2004. It includes the three papers that were selected for presentation by the program committee out of seven submissions. The final proceedings will appear as a volume in the ENTCS series.

We would like to thank the authors of the submitted papers, the invited and tutorial speakers, and the members of the program committee for their contribution to both the meeting and this volume. Many thanks to Philippa Gardner and Nobuko Yoshida (CONCUR 2004 Conference Chairs), and Julian Rathke and Vladimiro Sassone (Satellite Workshops Chairs), for the opportunity they gave us to organize the workshop, and for their continuous support. We would also like to thank Michael Mislove and Uffe Engberg for their help with the editing of the proceedings. Finally, we gratefully acknowledge the support of BRICS (Basic Research in Computer Science), Centre of the Danish National Research Foundation, and of the Engineering and Physical Sciences Research Council (EPSRC).

Luca Aceto, BRICS, Aalborg University, Denmark
Wan Fokkink, CWI, Amsterdam, The Netherlands
Irek Ulidowski, University of Leicester, United Kingdom

Workshop on Structural Operational Semantics - Program Committee

Luca Aceto (DK)	Wan Fokkink (NL)
Rob van Glabbeek (AU)	Ralf Lämmel (NL)
Peter Mosses (DK)	David Sands (SE)
Alex Simpson (UK)	Simone Tini (IT)
Irek Ulidowski (UK)	Erik de Vink (NL)

SOS 2004 Preliminary Version

Toward the concept of backtracking computation

M. Kulaš

FernUniversität Hagen, FB Informatik, D-58084 Hagen, Germany [1]

Abstract

This article proposes a new mathematical definition of the execution of pure Prolog, in the form of axioms in a structural operational semantics. The main advantage of the model is its ease in representing backtracking, due to the functionality of the transition relation and its converse. Thus, *forward and backward* derivation steps are possible. A novel concept of *stages* is introduced, as a refinement of final states, which captures the evolution of a backtracking computation. An advantage over the traditional stack-of-stacks approaches is a *modularity* property. Finally, the model combines the intuition of the traditional 'Byrd box' metaphor with a compact representation of execution state, making it feasible to formulate and prove theorems about the model. In this paper we introduce the model and state some useful properties.

Key words: backtracking, Prolog, operational semantics

1 Motivation, aims and results

In this paper, we introduce S_1:PP, a new operational semantics for pure Prolog, and establish some useful properties, aiming toward an algebraic definition of the concept of Prolog computation. On the way, we obtain some new concepts useful for characterizing backtracking, but possibly also useful for objects which evolve over time. Such an object is in logic programming *the goal*, in its dynamic sense ('this unique, run-time invocation of a Prolog procedure'), as opposed to its static or syntactic sense ('this goal formula').

The goal is a basic concept of logic programming, but nevertheless one which proved hard to grasp in a formal way, even in the case of pure Prolog. The problem is the possible evolution of 'the' goal through slightly different identities, in case of a non-deterministic procedure.

[1] Email: marija.kulas@fernuni-hagen.de

This is a preliminary version. The final version will be published in Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

For example, assume we pose the following query to a Prolog system: $\mathsf{p(X,Y)}, \mathsf{q(Y)}, \mathsf{fail}$. Assume two answers for $\mathsf{p(X,Y)}$, say $\mathsf{p(a,Z)}$ and $\mathsf{p(b,3)}$. In the static sense, we have only one goal $\mathsf{q(Y)}$, but dynamically we can have two different goals: first $\mathsf{q(Z)}$, and if $\mathsf{q(Z)}, \mathsf{fail}$ terminates, then $\mathsf{q(3)}$. Both goals have their own creation, lifetime of forward and backward execution, and possible expiration. It is vital for an operational semantics of backtracking to differentiate between such separate objects.

In the rest of this paper we proceed as follows. First a canonical form of predicates is defined, into which the original pure Prolog program shall be transformed. Then, in section 3, a novel operational semantics of pure Prolog is defined, in a structural operational manner. Throughout the section 4 – section 7 we develop formal tools (concepts and theorems) suitable for characterizing Prolog computation. This obviously includes defining in some way or other the (dynamic) concept of the goal as well. We solved this problem by means of *stages*, through which an initial event (representing the creation of a goal) passes in the course of computation. Stages can be seen as a generalization of the normal form idea, in the sense that stages are independent on the context of computation, as shown in section 7, but organized by macro transitions. Starting from individual transitions as given in the model, simple derivations (forward and backward) are built, which are the basis of simple passes, and simple passes aggregate into composed passes. Finally, we show in section 8 how composed passes model Prolog computations.

Our approach can be seen as a formalization of the original Byrd model. But there is an important detail: we extend the notion of a port, initially conceived by Byrd for selected atoms, to *general goal formulas*. The shifting of attention from atoms to general goal formulas proved to be a key idea and made a very simple model possible. The model in its first version, called S:PP, was proposed in [17]. But the handling of variables turned out to be difficult. The new model S_1:PP improves on this.

2 Preliminaries

Before it can be interpreted in our model, the original Prolog program has to be transformed into a canonical form, the common single-clause representation. This representation is arguably 'near enough' to the original program, the only differences concern the head-unification (which is now delegated to the body) and the choices (which are now uniformly expressed as disjunction).

Definition 2.1 (canonical form) *We say that a predicate P/n is in* canonical form*, if its definition consists of a single clause $P(X_1, ..., X_n) :- B; Bs$. Here B is a canonical body, of the form $X_1{=}T_1, \ldots, X_n{=}T_n, G, Gs$, and $P(X_1, ..., X_n)$ is a canonical head, i. e. $X_1, ..., X_n$ are distinct variables not appearing in $G, Gs, T_1, ..., T_n$. Further, Bs is a disjunction of canonical bodies (possibly empty), Gs is a conjunction of goals (possibly empty), and G is a goal (for facts:* true*).*

For example, the following program

```
q(a,b).
q(Z,c) :- r(Z).
r(c).
```

would be canonically represented as

```
q(X,Y) :- X=a, Y=b, true; X=Z, Y=c, r(Z).
r(X) :- X=c, true.
```

3 The semantics S_1:PP

The model S_1:PP we are proposing fits naturally into a structural operational format. For easy reference, the model is defined in two figures, Figure 1 (syntax) and Figure 2 (rules). Notice that there are no premisses to the transition rules, so the calculus consists solely of axioms.

Definition 3.1 (event) *An* event *is a quadruple* (*Port*, *Goal*, *A-stack*, *B-stack*), *as given by the grammar in Figure 1.*

Intuitively, an event is a state of Prolog computation, and it is determined in our model by four parameters:

port: *call*, *exit*, *fail* or *redo*

current goal: a general goal formula

history of current goal: stack of ancestors, for short: *A-stack*

current environment: stack of bets, for short: *B-stack*

Definition 3.2 (transition) *Let Π be a pure Prolog program in canonical form, as defined by Figure 1 and Definition 2.1.* The transition relation $\rightarrowtriangle_\Pi$ *is defined in Figure 2. The converse relation shall be denoted by* $\triangleleft\!\!-_\Pi$. *If* $E_1 \rightarrowtriangle_\Pi E$, *we say that* E_1 leads to E. *Alternatively, we say that* E_1 *is a* predecessor *to* E, *and* E *is a* successor *to* E_1. *An event* E *can be* entered, *if some event leads to it. An event* E *can be* left, *if it leads to some event.*

The left-hand sides of the transition rules are mutually disjoint, i. e. there are no critical pairs, so we have

Lemma 3.3 (transitions are deterministic) *The relation $\rightarrowtriangle_\Pi$ is functional, i. e. for each event E there can be at most one event E_1 such that $E \rightarrowtriangle_\Pi E_1$.*

Remark 3.4 (converse relation) *The converse of the port transition relation is not functional, since there may be more than one event leading to the same event. For example, call* $T_1{=}T_2\,\langle\frac{nil}{nil}\rangle \rightarrowtriangle_\Pi$ *fail* $T_1{=}T_2\,\langle\frac{nil}{nil}\rangle$, *and redo* $T_1{=}T_2\,\langle\frac{\sigma\bullet nil}{nil}\rangle \rightarrowtriangle_\Pi$ *fail* $T_1{=}T_2\,\langle\frac{nil}{nil}\rangle$. *Further down it will be shown that, for events that are* legal, *the converse relation is functional. In our example, redo* $T_1{=}T_2\,\langle\frac{\sigma\bullet nil}{nil}\rangle$ *is not a legal event.*

Definition 3.5 (derivation) *Let E_0, E be events. A Π-*derivation of E from E_0 in k steps*, written as $E_0 \rightarrowtriangle_\Pi{}^k E$, is a path of length k from E_0 to E in the graph of $\rightarrowtriangle_\Pi$. We say that E can be* reached *from E_0. Derivation of a nonzero length is denoted by $E_0 \rightarrowtriangle_\Pi^+ E$, and derivation of any length by $E_0 \rightarrowtriangle_\Pi^* E$.*

Definition 3.6 (initial event) *An* initial event *is call $Q\,\langle\frac{nil}{nil}\rangle$ for any Q.*

Intuitively, the goal Q of an initial event corresponds in Prolog to a *top-level goal* (*query*).

Definition 3.7 (legal derivation, legal event) *If there is a goal Q such that call $Q\,\langle\frac{nil}{nil}\rangle \rightarrowtriangle_\Pi^* E_0 \rightarrowtriangle_\Pi^* E$, then we say that $E_0 \rightarrowtriangle_\Pi^* E$ is a* legal Π-derivation*, and E is a* legal Π-event*.*

Definition 3.8 (final event) *A legal Π-event E is a* final Π-event*, if there is no transition $E \rightarrowtriangle_\Pi E_1$.*

Notation 1 (impossible event) *As a notational convenience, all the events which are not final and do not lead to any further events by means of $\rightarrowtriangle_\Pi$ are depicted as leading to the* impossible event*, written as $\bot$. Analogously for events that are not initial and cannot be entered.*

In particular, *redo* $\mathsf{fail} \rightarrowtriangle_\Pi \bot$ and *exit* $\mathsf{fail} \leftarrowtriangle_\Pi \bot$ for any Π. Some more examples: *call* $G\,\langle\frac{\sigma \bullet nil}{nil}\rangle \leftarrowtriangle_\Pi \bot$, *redo* $G\,\langle\frac{\Sigma}{nil}\rangle \leftarrowtriangle_\Pi \bot$ (cannot be entered, non-initial), and *redo* $\mathsf{p}\,\langle\frac{nil}{U}\rangle \rightarrowtriangle_\Pi \bot$ (cannot be left, but not legal, p being an atomary goal). The last example is perhaps less obvious, but shall be proven later (Lemma 4.1).

3.1 Remarks on the calculus

(i) In S_1:PP, the word *goal* is used in both its usual senses (section 1): as a syntactic domain, meaning 'goal formula', and as one of the four components of an execution state, meaning 'current goal'.

(ii) SLD-resolution is operating on the *selected atom*, but S_1:PP is operating on the whole *current goal*.

(iii) For the syntactic domains that have not been defined in Figure 1, we refer to [14] (substitutions), [20] (logic programming) and [12] (Prolog).

(iv) The most general unifiers σ shall be chosen to be idempotent, namely $\sigma(\sigma(T)) = \sigma(T)$. This is always possible.

(v) Resolution is modeled by (S_1:atom:1), and it is the only rule actually depending on Π. If the predicate of the atomary goal has a definition in Π, the resolution will succeed, because of canonical form.

(vi) Note the requirement $\sigma(G_A) = G_A$ in (S_1:atom:1). Since the clauses are in canonical form, unifying the head of a clause with a goal could do no more than rename the goal. We prefer the mgu to operate only on the clause.

(vii) A *fresh variable*, at a certain point of a derivation, represented by an event, is a variable not appearing in the previous course of the derivation, represented by the goal and the A-stack of the event.

(viii) It is not necessary to specify the semantics of fail explicitly, as in (S_1:fail). Instead, fail can be regarded as a user's predicate with an empty definition.

Notation 2 (distinguishing two levels) *Object-level terms (i. e. actual Prolog terms) are shown in sans serif, like* true*. Meta-level terms (i. e. anything else in the calculus) are shown in italics, like call,*α*.*

Notation 3 (dropping Π) *In the following we usually drop any reference to Π, since a program in pure Prolog cannot change during a derivation. However, a fixed program Π is always assumed. Observe that all the new relations in this paper, built upon the transition relation, also implicitly depend on Π.*

Notation 4 (auxiliary) *Here is some additional notation used in theorems.*

(i) Anonymous meta-variable*: If some parts of an event are of no interest in a topic at hand, they shall be abstracted away by the underscore "_".*

(ii) *To navigate an ancestor X, two functions are used: $\lfloor X \rfloor$, which is the selected branch of X, as defined in Figure 1, and $\lceil X \rceil$, which is X without tags: $\lceil N/A, B \rceil := A, B$, $\lceil N/A; B \rceil := A; B$, $\lceil G_A \rceil := G_A$.*

(iii) Stack addition and subtraction*: Concatenation to the right of a stack we denote by $+$, and if $U + V = W$, then $W - V := U$. Concatenating to both stacks of an event we denote by $\oplus$, with $\Gamma\ G\ \langle \frac{\Sigma}{U} \rangle \oplus \langle \frac{\Delta}{V} \rangle := \Gamma\ G\ \langle \frac{\Sigma + \Delta}{U + V} \rangle$, and $\Gamma\ G\ \langle \frac{\Sigma + \Delta}{U + V} \rangle \ominus \langle \frac{\Delta}{V} \rangle := \Gamma\ G\ \langle \frac{\Sigma}{U} \rangle$.*

(iv) Stack order*: If there is W such that $U = W + V$, then we say that $U \succeq V$.*

(v) *$\sigma\ |_T$ means substitution σ restricted to the variables of the term T.*

(vi) Macro transitions $\overset{\triangleright}{\rightarrowtriangle}$, $\overset{\triangleleft}{\rightarrowtriangle}$, $\longrightarrow$, $\Longrightarrow$ *will be defined in section 5 and section 6.*

4 Uniqueness claim

First we state a useful property, which we call the pendant lemma. Observe that the formulation is *non-deterministic* in that we only claim the existence of a pendant event (with the identical B-stack), but it is not known whether it is the only one.

Lemma 4.1 (pendant) *If call $G\ \langle \frac{nil}{nil} \rangle \rightarrowtriangle^*$ fail $H\ \langle \frac{\Theta}{W} \rangle$, then*

$$call\ G\ \langle \tfrac{nil}{nil} \rangle \rightarrowtriangle^*\ call\ H\ \langle \tfrac{\Theta}{W} \rangle \rightarrowtriangle^*\ fail\ H\ \langle \tfrac{\Theta}{W} \rangle.$$

If call $G\ \langle \frac{nil}{nil} \rangle \rightarrowtriangle^*$ redo $H\ \langle \frac{\Theta}{W} \rangle$, then

$$call\ G\ \langle \tfrac{nil}{nil} \rangle \rightarrowtriangle^*\ exit\ H\ \langle \tfrac{\Theta}{W} \rangle \rightarrowtriangle^*\ redo\ H\ \langle \tfrac{\Theta}{W} \rangle.$$

The pendant lemma enables us to prove a vital property of our calculus: there can be only one successor to a given event, and moreover, for a legal event

event	::=	port goal $\langle \frac{\text{B-stack}}{\text{A-stack}} \rangle$
program	::=	$\{\text{definition.}\}^+$
definition	::=	atom :− goal
port	::=	push | pop
push	::=	*call* | *redo*
pop	::=	*exit* | *fail*
goal	::=	$\mathsf{true} \mid \mathsf{fail} \mid \text{atom} \mid \text{term} = \text{term} \mid \text{goal}; \text{goal} \mid \text{goal}, \text{goal}$
ancestor	::=	atom | tag/goal; goal | tag/goal, goal
tag	::=	*1* | *2*
memo	::=	$BY(\text{goal}) \mid OR(\text{tag})$
bet	::=	mgu | memo
A-stack	::=	*nil* | ancestor • A-stack
B-stack	::=	*nil* | bet • B-stack

Meta-variables

E	:	event						
Π	:	program						
Γ	:	port,	*Push*	:	push,	*Pop*	:	pop
U, V	:	A-stack,	a, b, $\widehat{G}$	:	ancestor,	N	:	tag
Σ, Θ, Ψ, Ω, Δ	:	B-stack,	α	:	bet			
σ	:	substitution						
A, B, C, G, H	:	goal						
G_A	:	atom						
T	:	term						

Meta-functions

$\lfloor 1/A, B \rfloor := A$, $\lfloor 2/A, B \rfloor := B$, and analogously for disjunction
$\sigma(T)$ = application of σ upon T
$mgu(T_1, T_2)$ = mgu of T_1 and T_2
$subst(\Sigma)$ = current substitution = composition of all mgus from Σ;
$subst(\Sigma)(T)$ shall be abbreviated to $\Sigma(T)$, and is defined as follows:

$$nil(T) := T$$

$$\alpha \bullet \Sigma(T) := \begin{cases} \alpha(\Sigma(T)), & \text{if } \alpha \text{ is an mgu} \\ \Sigma(T), & \text{if } \alpha \text{ is a memo} \end{cases}$$

Syntactic domains taken in their usual sense:

term (taken in the Prolog sense, as a superset of goal);
atom (user-defined predication in logic programming);
substitution, renaming, mgu.

Fig. 1. Language of events

Conjunction

$$\begin{aligned}
\mathit{call}\ A, B\ \langle \tfrac{\Sigma}{U} \rangle &\rightarrowtriangle \mathit{call}\ A\ \langle \tfrac{\Sigma}{1/A,B \bullet U} \rangle && (S_1\text{:conj:1})\\
\mathit{exit}\ A\ \langle \tfrac{\Sigma}{1/A,B \bullet U} \rangle &\rightarrowtriangle \mathit{call}\ B\ \langle \tfrac{\Sigma}{2/A,B \bullet U} \rangle && (S_1\text{:conj:2})\\
\mathit{fail}\ A\ \langle \tfrac{\Sigma}{1/A,B \bullet U} \rangle &\rightarrowtriangle \mathit{fail}\ A, B\ \langle \tfrac{\Sigma}{U} \rangle && (S_1\text{:conj:3})\\
\mathit{exit}\ B\ \langle \tfrac{\Sigma}{2/A,B \bullet U} \rangle &\rightarrowtriangle \mathit{exit}\ A, B\ \langle \tfrac{\Sigma}{U} \rangle && (S_1\text{:conj:4})\\
\mathit{fail}\ B\ \langle \tfrac{\Sigma}{2/A,B \bullet U} \rangle &\rightarrowtriangle \mathit{redo}\ A\ \langle \tfrac{\Sigma}{1/A,B \bullet U} \rangle && (S_1\text{:conj:5})\\
\mathit{redo}\ A, B\ \langle \tfrac{\Sigma}{U} \rangle &\rightarrowtriangle \mathit{redo}\ B\ \langle \tfrac{\Sigma}{2/A,B \bullet U} \rangle && (S_1\text{:conj:6})
\end{aligned}$$

Disjunction

$$\begin{aligned}
\mathit{call}\ A; B\ \langle \tfrac{\Sigma}{U} \rangle &\rightarrowtriangle \mathit{call}\ A\ \langle \tfrac{\Sigma}{1/A;B \bullet U} \rangle && (S_1\text{:disj:1})\\
\mathit{fail}\ A\ \langle \tfrac{\Sigma}{1/A;B \bullet U} \rangle &\rightarrowtriangle \mathit{call}\ B\ \langle \tfrac{\Sigma}{2/A;B \bullet U} \rangle && (S_1\text{:disj:2})\\
\mathit{fail}\ B\ \langle \tfrac{\Sigma}{2/A;B \bullet U} \rangle &\rightarrowtriangle \mathit{fail}\ A; B\ \langle \tfrac{\Sigma}{U} \rangle && (S_1\text{:disj:3})\\
\mathit{exit}\ C\ \langle \tfrac{\Sigma}{N/A;B \bullet U} \rangle &\rightarrowtriangle \mathit{exit}\ A; B\ \langle \tfrac{OR(N) \bullet \Sigma}{U} \rangle,\ \text{with}\ C\ldots^{2} && (S_1\text{:disj:4})\\
\mathit{redo}\ A; B\ \langle \tfrac{OR(N) \bullet \Sigma}{U} \rangle &\rightarrowtriangle \mathit{redo}\ C\ \langle \tfrac{\Sigma}{N/A;B \bullet U} \rangle,\ \text{with}\ C\ldots^{2} && (S_1\text{:disj:5})
\end{aligned}$$

True and Fail

$$\begin{aligned}
\mathit{call}\ \mathsf{true}\ \langle \tfrac{\Sigma}{U} \rangle &\rightarrowtriangle \mathit{exit}\ \mathsf{true}\ \langle \tfrac{\Sigma}{U} \rangle && (S_1\text{:true:1})\\
\mathit{redo}\ \mathsf{true}\ \langle \tfrac{\Sigma}{U} \rangle &\rightarrowtriangle \mathit{fail}\ \mathsf{true}\ \langle \tfrac{\Sigma}{U} \rangle && (S_1\text{:true:2})\\
\mathit{call}\ \mathsf{fail}\ \langle \tfrac{\Sigma}{U} \rangle &\rightarrowtriangle \mathit{fail}\ \mathsf{fail}\ \langle \tfrac{\Sigma}{U} \rangle && (S_1\text{:fail})
\end{aligned}$$

Explicit unification

$$\begin{aligned}
\mathit{call}\ T_1{=}T_2\ \langle \tfrac{\Sigma}{U} \rangle &\rightarrowtriangle \begin{cases} \mathit{exit}\ T_1{=}T_2\ \langle \tfrac{\sigma \bullet \Sigma}{U} \rangle, & \text{if mgu...}^{3} \\ \mathit{fail}\ T_1{=}T_2\ \langle \tfrac{\Sigma}{U} \rangle, & \text{otherwise} \end{cases} && (S_1\text{:unif:1})\\
\mathit{redo}\ T_1{=}T_2\ \langle \tfrac{\sigma \bullet \Sigma}{U} \rangle &\rightarrowtriangle \mathit{fail}\ T_1{=}T_2\ \langle \tfrac{\Sigma}{U} \rangle && (S_1\text{:unif:2})
\end{aligned}$$

User-defined atomary goal G_A

$$\begin{aligned}
\mathit{call}\ G_A\ \langle \tfrac{\Sigma}{U} \rangle &\rightarrowtriangle \begin{cases} \mathit{call}\ \sigma(B)\ \langle \tfrac{\Sigma}{G_A \bullet U} \rangle, & \text{if } H{:-}B\ldots^{4} \\ \mathit{fail}\ G_A\ \langle \tfrac{\Sigma}{U} \rangle, & \text{otherwise} \end{cases} && (S_1\text{:atom:1})\\
\mathit{exit}\ B\ \langle \tfrac{\Sigma}{G_A \bullet U} \rangle &\rightarrowtriangle \mathit{exit}\ G_A\ \langle \tfrac{BY(B) \bullet \Sigma}{U} \rangle && (S_1\text{:atom:2})\\
\mathit{fail}\ B\ \langle \tfrac{\Sigma}{G_A \bullet U} \rangle &\rightarrowtriangle \mathit{fail}\ G_A\ \langle \tfrac{\Sigma}{U} \rangle && (S_1\text{:atom:3})\\
\mathit{redo}\ G_A\ \langle \tfrac{BY(B) \bullet \Sigma}{U} \rangle &\rightarrowtriangle \mathit{redo}\ B\ \langle \tfrac{\Sigma}{G_A \bullet U} \rangle && (S_1\text{:atom:4})
\end{aligned}$$

[2] with $C = \lfloor N/A; B \rfloor$.
[3] if $mgu(\Sigma(T_1), \Sigma(T_2)) = \sigma$.
[4] if $H :- B$ is a fresh renaming of a clause in Π, and $\sigma = mgu(G'_A, H)$ with $G'_A := \Sigma(G_A)$ and $\sigma(G'_A) = G'_A$.

Fig. 2. Operational semantics S_1:PP of pure Prolog

there can be only one predecessor.

Theorem 4.2 (legal transitions are unique) *If E is a legal event, then E can have only one legal predecessor, and only one successor. In case E is non-initial, there is exactly one legal predecessor. In case E is non-final, there is exactly one successor.*

Having established functionality of the transition relation and its converse, we may unfold a legal derivation from each of its endpoints. First let us see how far we can go from an initial event by means of transitions.

Lemma 4.3 (ancestor) *If $\Gamma\ G\,\langle\frac{\Sigma}{a \bullet V}\rangle$ is a legal event, then there are Push and Σ' such that $Push\,\lceil a\rceil\,\langle\frac{\Sigma'}{V}\rangle \rightarrowtriangle^+ \Gamma\ G\,\langle\frac{\Sigma}{a \bullet V}\rangle$ is a legal derivation.*

Lemma 4.4 (tagged parent) *If $\Gamma\ G\,\langle\frac{\Sigma}{a \bullet V}\rangle$ is a legal event, and $a = N/A, B$ or $a = N/A; B$, then $G = \lfloor a \rfloor$.*

Lemma 4.5 (final event) *If E is a legal pop event with a non-empty A-stack, then there is always a transition $E \rightarrowtriangle E_1$.*

Lemma 4.6 *If $call\ G\,\langle\frac{nil}{nil}\rangle \rightarrowtriangle^+ \Gamma\ H\,\langle\frac{\Theta}{W}\rangle$ and $\Gamma\ H\,\langle\frac{\Theta}{W}\rangle \neq Pop\ \_\,\langle\frac{\div}{nil}\rangle$, then $W = V + \widehat{G} \bullet nil$ for some V and an ancestor $\widehat{G}$ such that $\lceil \widehat{G} \rceil = G$.*

Lemma 4.7 *If $call\ G\,\langle\frac{nil}{nil}\rangle \rightarrowtriangle^* Pop\ H\,\langle\frac{\Sigma}{nil}\rangle$, then $H = G$.*

The above lemmas suggest events of the form $Pop\ \_\,\langle\frac{\div}{nil}\rangle$ as natural endpoints of derivation. For this reason we develop a concept of derivation around such events. We start with a concept of simple derivation.

5 Simple derivation and subevent

In this section, we set about defining some new, 'macro' transition relations, by collapsing whole sequences of transition steps into one big step. Arguably, illegal derivations do not make much sense in such a context, therefore we exclude them:

Notation 5 (only legal derivations) *In the transition relations that we shall define from now on, namely $\overset{\triangleright}{\rightarrowtriangle}, \overset{\triangleleft}{\rightarrowtriangle}, \longrightarrow, \Longrightarrow$, it is always assumed that the derivations are legal.*

Definition 5.1 (forward or backward simple derivation) *Consider a legal derivation $Push\ G\,\langle\frac{\Sigma}{U}\rangle \rightarrowtriangle^+ E$ such that there is no $Pop\ \_\,\langle\frac{\div}{U}\rangle$ within this derivation, i. e. $Push\ G\,\langle\frac{\Sigma}{U}\rangle \rightarrowtriangle^+ Pop\ \_\,\langle\frac{\div}{U}\rangle \rightarrowtriangle^+ E$ is not allowed. Such a derivation we call a* forward derivation relative to U, *and denote by $Push\ G\,\langle\frac{\Sigma}{U}\rangle \overset{\triangleright}{\rightarrowtriangle} E$. Analogously, a legal derivation $Pop\ G\,\langle\frac{\Sigma}{U}\rangle \rightarrowtriangle^+ E$ such that there is no $Push\ \_\,\langle\frac{\div}{U}\rangle$ within this derivation, is a* backward derivation relative to U, *denoted by $Pop\ G\,\langle\frac{\Sigma}{U}\rangle \overset{\triangleleft}{\rightarrowtriangle} E$. A forward or a backward derivation relative to U is a* simple derivation relative to U.

Forward derivation gives rise to *subevents*:

Definition 5.2 (subevent) *If* $Push\ G\ \langle \frac{\Sigma}{U} \rangle \overset{\triangleright}{\rightarrow\!\!\triangleright} \Gamma\ H\ \langle \frac{\Theta}{W} \rangle$ *then* $\Gamma\ H\ \langle \frac{\Theta}{W} \rangle$ *is a* subevent *of* $Push\ G\ \langle \frac{\Sigma}{U} \rangle$.

Lemma 4.6 and Lemma 4.7 can be generalized to provide for the case of an arbitrary push event and an arbitrary A-stack, and proven in the same manner as the original lemmas:

Theorem 5.3 (the subevent property) *If* $Push\ G\ \langle \frac{\Sigma}{U} \rangle \overset{\triangleright}{\rightarrow\!\!\triangleright} \Gamma\ H\ \langle \frac{\Theta}{W} \rangle$ *and* $\Gamma\ H\ \langle \frac{\Theta}{W} \rangle \neq Pop\ \_\ \langle \frac{\div}{U} \rangle$, *then* $W = V + \widehat{G} \bullet U$ *for some* V *and an ancestor* $\widehat{G}$ *such that* $\lceil \widehat{G} \rceil = G$.

Lemma 5.4 (endpoint) *If* $Push\ G\ \langle \frac{\Sigma}{U} \rangle \overset{\triangleright}{\rightarrow\!\!\triangleright} Pop\ H\ \langle \frac{\Theta}{U} \rangle$ *then* $H = G$.

Obviously, each push transition is a forward derivation. Some forward derivations can be composed as well. This follows from the subevent property.

Corollary 5.5 *If* $Push_1\ G\ \langle \frac{\Sigma}{U} \rangle \overset{\triangleright}{\rightarrow\!\!\triangleright} Push\ H\ \langle \frac{\Theta}{a \bullet U} \rangle \overset{\triangleright}{\rightarrow\!\!\triangleright} E$, *then* $Push_1\ G\ \langle \frac{\Sigma}{U} \rangle \overset{\triangleright}{\rightarrow\!\!\triangleright} E$. *Also, if* $Push\ G\ \langle \frac{\Sigma}{U} \rangle \overset{\triangleright}{\rightarrow\!\!\triangleright} E \rightarrow\!\!\triangleright E_1$ *with* $E \neq Pop\ \_\ \langle \frac{\div}{U} \rangle$, *then* $Push\ G\ \langle \frac{\Sigma}{U} \rangle \overset{\triangleright}{\rightarrow\!\!\triangleright} E_1$.

Lemma 5.6 (forward pass) *If* $Pop\ G\ \langle \frac{\Sigma}{U} \rangle$ *is a legal event, then* $Pop\ G\ \langle \frac{\Sigma}{U} \rangle \overset{\triangleleft}{\leftarrow\!\!\triangleleft} Push\ G\ \langle \frac{\Sigma^\circ}{U} \rangle$ *for some* $Push$ *and* Σ°.

The next statement follows from the subevent property. Analogous claim with pop and push swapping places, holds due to the simple pass lemma.

Lemma 5.7 (no pop no push) *Let* $Push_0\ G\ \langle \frac{\div}{U} \rangle \rightarrow\!\!\triangleright^*\ E$ *be a legal derivation, such that there is no* $Pop\ \_\ \langle \frac{\div}{U} \rangle$ *within the derivation. Then there is no* $Push\ \_\ \langle \frac{\div}{U} \rangle$ *within the derivation as well. The same holds for derivations of the form* $E \rightarrow\!\!\triangleright^*\ Push_0\ G\ \langle \frac{\div}{U} \rangle$.

Taking into account composition of forward derivations, we can prove a stronger version of the ancestor lemma. The new version has the advantage of determinism, i. e. there is only one place in a derivation where the parent event can be: the most recent past event of the form $Push\ \lceil a \rceil\ \langle \frac{\div}{V} \rangle$.

Lemma 5.8 (ancestor, stronger) *If* $\Gamma\ G\ \langle \frac{\Sigma}{a \bullet V} \rangle$ *is a legal event, then there are* $Push$ *and* Σ' *such that* $Push\ \lceil a \rceil\ \langle \frac{\Sigma'}{V} \rangle \overset{\triangleright}{\rightarrow\!\!\triangleright} \Gamma\ G\ \langle \frac{\Sigma}{a \bullet V} \rangle$ *is a legal derivation.*

Lemma 5.9 (subevent, converse) *If for a legal event* $\Gamma\ H\ \langle \frac{\Theta}{W} \rangle$ *holds that* $W = V + a \bullet U$, *then* $\Gamma\ H\ \langle \frac{\Theta}{W} \rangle$ *is a subevent of* $Push\ \lceil a \rceil\ \langle \frac{\Sigma}{U} \rangle$ *for some* $Push$ *and some* Σ.

As we have seen, forward or backward derivations between events with identical A-stack and identical goal play a special role in the calculus. We abstract such derivations to a new concept:

Definition 5.10 (forward or backward simple pass) *A forward derivation* $Push\ G\ \langle \frac{\div}{U} \rangle \overset{\triangleright}{\rightarrow\!\!\triangleright} Pop\ G\ \langle \frac{\div}{U} \rangle$ *we call a* forward pass relative to G and U. *Analogously, a backward derivation* $Pop\ G\ \langle \frac{\div}{U} \rangle \overset{\triangleleft}{\rightarrow\!\!\triangleleft} Push\ G\ \langle \frac{\div}{U} \rangle$ *we call a* backward pass relative to G and U. *A forward or a backward pass from* E_1 *to* E_2

is a simple pass*, denoted by* $E_1 \longrightarrow E_2$*. The events* E_1, E_2 *are called* stages*. To denote stages we may use the fixed parts (the goal and the A-stack) as superscripts, like this:* $E^{G,U}$*.*

6 Composed derivation

Fortified with the useful results like the pendant and forward pass lemma, we are now in a position to prove stronger results. As with the stronger version of the ancestor lemma, the advantage is in the deterministic specification of the correlated events from the past. For example, for a fail event we now know that its pendant call (with the identical B-stack) is the most recent call event bearing the same goal and the same A-stack.

Theorem 6.1 (pendant, stronger) *Let* $fail\ H\ \langle \frac{\Theta}{W} \rangle$ *be a a legal event. Then* $fail\ H\ \langle \frac{\Theta}{W} \rangle \leftarrow^* call\ H\ \langle \frac{\Theta}{W} \rangle$*, where* $call\ H\ \langle \frac{\_}{W} \rangle$ *does not appear within the derivation. Furthermore, if* $redo\ H\ \langle \frac{\Theta}{W} \rangle$ *is a legal event, then* $redo\ H\ \langle \frac{\Theta}{W} \rangle \longleftarrow exit\ H\ \langle \frac{\Theta}{W} \rangle$*.*

Lemma 6.2 (B-stack) *If* $call\ G\ \langle \frac{\Sigma}{U} \rangle \longrightarrow \Gamma\ H\ \langle \frac{\Sigma'}{U} \rangle$*, then* $\Sigma' \succeq \Sigma$*.*

Lemma 6.3 (no call no fail) *If* $E \rightarrowtriangle^* Pop\ G\ \langle \frac{\_}{U} \rangle$ *is a legal derivation, and* $call\ G\ \langle \frac{\_}{U} \rangle$ *is not within this derivation, then* $fail\ G\ \langle \frac{\_}{U} \rangle$ *is also not within this derivation.*

From Lemma 5.6 and Theorem 6.1 we know that a legal pop event can run through a series of past stages, like $exit\ \_\langle \frac{\_}{U} \rangle \longleftarrow redo\ \_\langle \frac{\_}{U} \rangle \longleftarrow exit\ \_\langle \frac{\_}{U} \rangle \longleftarrow \ldots$ Due to the finiteness of a converse derivation and Lemma 5.6, a $call\ \_\langle \frac{\_}{U} \rangle$ is bound to appear. So we must ultimately reconstruct a derivation $exit\ \_\langle \frac{\_}{U} \rangle \leftarrow^* call\ \_\langle \frac{\_}{U} \rangle$, where no events of the form $call\ \_\langle \frac{\_}{U} \rangle$ intervene. Similarly for a fail event.

Definition 6.4 (composed pass) *Consider a sequence of simple passes* $E_1^{G,U} \longrightarrow^+ E_2^{G,U}$ *such that there is no* $call\ G\ \langle \frac{\_}{U} \rangle$ *within this sequence, i. e.* $E_1^{G,U} \longrightarrow^+ call\ G\ \langle \frac{\_}{U} \rangle \longrightarrow^+ E_2^{G,U}$ *is not allowed. Such a sequence is called* composable*, or a* composed pass relative to G and U*, and denoted by* $E_1^{G,U} \Longrightarrow E_2^{G,U}$*.*

It can be seen that $call\ G\ \langle \frac{\_}{U} \rangle$ cannot appear within a composed pass relative to G and U even if regarded as a derivation, i. e. neither among the stages of the simple passes nor somewhere in between. One important question remains: How do the B-stacks of the particular stages relate to each other? This is the main concern of our next claim, companion to Lemma 5.6.

Theorem 6.5 (composed pass) *The following two relationships hold:*

$$If\ fail\ G\ \langle \tfrac{\Sigma}{U} \rangle\ is\ legal,\ then\ fail\ G\ \langle \tfrac{\Sigma}{U} \rangle \Longleftarrow call\ G\ \langle \tfrac{\Sigma}{U} \rangle. \tag{1}$$

$$If\ exit\ G\ \langle \tfrac{\Sigma}{U} \rangle\ is\ legal,\ then\ exit\ G\ \langle \tfrac{\Sigma}{U} \rangle \Longleftarrow call\ G\ \langle \tfrac{\Sigma^\circ}{U} \rangle,\ with\ \Sigma \succeq \Sigma^\circ \tag{2}$$

For (2) further holds: If G *is a disjunction, then* $\Sigma \succ \Sigma^\circ$*, starting with* $OR(N)$ *for some* N*, and analogously for a unification or an atomary goal.*

We already know that for a legal redo event holds $\mathit{redo}\ G\,\langle\frac{\Sigma}{U}\rangle \longleftarrow \mathit{exit}\ G\,\langle\frac{\Sigma}{U}\rangle$, which adds some more relationships like

$$\text{If } \mathit{fail}\ G\,\langle\tfrac{\Sigma}{U}\rangle \Longleftarrow \mathit{redo}\ G\,\langle\tfrac{\Sigma'}{U}\rangle, \text{ then } \Sigma' \succeq \Sigma \tag{3}$$

$$\text{If } \mathit{redo}\ G\,\langle\tfrac{\Sigma}{U}\rangle \text{ is legal, then } \mathit{redo}\ G\,\langle\tfrac{\Sigma}{U}\rangle \Longleftarrow \mathit{call}\ G\,\langle\tfrac{\Sigma^\circ}{U}\rangle, \text{ with } \Sigma \succeq \Sigma^\circ \tag{4}$$

As a by-product, the following supplement to Lemma 4.5 can be obtained, leading to a conclusion that the only final events are legal pop events with an empty A-stack:

Lemma 6.6 (non-final event) *If E is a legal push event, then there is always a transition $E \rightarrowtriangle E_1$.*

7 Independence claim and modularity

Remark 7.1 (up-to-date calls) *Bearing in mind the canonical form of the clauses, as well as idempotency of the mgus in our model, it can be seen from the rule (S_1:atom:1) that for any event $\mathit{call}\,\sigma(B)\,\langle\frac{\Sigma}{G_A \bullet U}\rangle$ holds: $\Sigma(\sigma(B)) = \sigma(B)$. In other words, the goal part of such an event is up-to-date with respect to the B-stack.*

Theorem 7.2 (adding stacks) *Let $\mathit{call}\ G\,\langle\frac{nil}{nil}\rangle \rightarrowtriangle E_1 \rightarrowtriangle \ldots \rightarrowtriangle E_n \rightarrowtriangle \mathit{Pop}\ G\,\langle\frac{\Omega}{nil}\rangle$ be a legal derivation. Then for every G°, U and Σ such that $\mathit{call}\ G^\circ\,\langle\frac{\Sigma}{U}\rangle$ is a legal event and $\Sigma(G^\circ) = G$, holds:*

$$\mathit{call}\ G^\circ\,\langle\tfrac{\Sigma}{U}\rangle \rightarrowtriangle E_1^\circ \oplus \langle\tfrac{\Sigma}{U}\rangle \rightarrowtriangle \ldots \rightarrowtriangle E_n^\circ \oplus \langle\tfrac{\Sigma}{U}\rangle \rightarrowtriangle \mathit{Pop}\ G^\circ\,\langle\tfrac{\Omega+\Sigma}{U}\rangle$$

is also a legal derivation. Here if $E_i = \Gamma\ H\,\langle\frac{\Theta}{V}\rangle$, then $E_i^\circ := \Gamma\ H^\circ\,\langle\frac{\Theta}{V^\circ}\rangle$, where $\Sigma(H^\circ) = H$ and $\Sigma(V^\circ) = V$.

Theorem 7.3 (subtracting stacks) *Let $\mathit{call}\ G\,\langle\frac{\Sigma}{U}\rangle \rightarrowtriangle E_1 \rightarrowtriangle \ldots \rightarrowtriangle E_n \rightarrowtriangle \mathit{Pop}\ G\,\langle\frac{\Omega}{U}\rangle$ be a legal derivation, and $E_i \neq \mathit{Pop}\ \_\,\langle\frac{-}{U}\rangle$ for every i. Then for $G' := \Sigma(G)$ holds:*

$$\mathit{call}\ G'\,\langle\tfrac{nil}{nil}\rangle \rightarrowtriangle E_1' \ominus \langle\tfrac{\Sigma}{U}\rangle \rightarrowtriangle \ldots \rightarrowtriangle E_n' \ominus \langle\tfrac{\Sigma}{U}\rangle \rightarrowtriangle \mathit{Pop}\ G'\,\langle\tfrac{\Omega-\Sigma}{nil}\rangle$$

is also a legal derivation. Here if $E_i = \Gamma\ H\,\langle\frac{\Theta}{V}\rangle$, then $E_i' := \Gamma\ H'\,\langle\frac{\Theta}{V'}\rangle$, where $\Sigma(H) = H'$ and $\Sigma(V) = V'$.

The previous two claims show that a simple pass, starting with a call event, is independent on the starting contents of the stacks. The stacks represent the *context* of the computation for the goal at hand.

It would be interesting to see whether similar properties hold for a simple pass starting with a redo event. First note that if $\mathit{redo}\ G\,\langle\frac{\Sigma}{U}\rangle$ is a legal event, then, according to Theorem 6.1, $\mathit{redo}\ G\,\langle\frac{\Sigma}{U}\rangle \longleftarrow \mathit{exit}\ G\,\langle\frac{\Sigma}{U}\rangle$. Using Theorem 6.5, we further obtain that $\mathit{redo}\ G\,\langle\frac{\Sigma}{U}\rangle \Longleftarrow \mathit{call}\ G\,\langle\frac{\Sigma^\circ}{U}\rangle$, where $\Sigma \succeq \Sigma^\circ$. This gives us a hint on how much we may cut off of the stacks. Observe that, as opposed to a call event, we do not necessarily arrive at a

legal redo event by means of adding or subtracting stacks. (For example, a redo event with an empty A-stack cannot be reached.) But if we do, then we may know its next stage, due to the following claim:

Lemma 7.4 (starting with redo) *Let* $\mathit{redo}\ G\,\langle\frac{\Sigma}{U}\rangle \rightarrowtriangle E_1 \rightarrowtriangle \ldots \rightarrowtriangle E_n \rightarrowtriangle \mathit{Pop}\ G\,\langle\frac{\Omega}{U}\rangle$ *be a legal derivation, and* $E_i \neq \mathit{Pop}\ \_\,\langle\frac{\div}{U}\rangle$ *for every* i*. Further let* $\mathit{redo}\ G\,\langle\frac{\Sigma}{U}\rangle \Longleftarrow \mathit{call}\ G\,\langle\frac{\Sigma^\circ}{U}\rangle$*. Then for* $G' := \Sigma^\circ(G)$ *and* E'_i *analogous as in Theorem 7.3 holds:*

$$\mathit{redo}\ G'\,\langle\tfrac{\Sigma-\Sigma^\circ}{\mathit{nil}}\rangle \rightarrowtriangle E'_1 \ominus \langle\tfrac{\Sigma^\circ}{U}\rangle \rightarrowtriangle \ldots \rightarrowtriangle E'_n \ominus \langle\tfrac{\Sigma^\circ}{U}\rangle \rightarrowtriangle \mathit{Pop}\ G'\,\langle\tfrac{\Omega-\Sigma^\circ}{\mathit{nil}}\rangle.$$

Moreover, for $\Theta(G^\circ) = G$ *and* E°_i *analogous as in Theorem 7.2 holds:*

$$\mathit{redo}\ G^\circ\,\langle\tfrac{\Sigma+\Theta}{U+V}\rangle \rightarrowtriangle E^\circ_1 \oplus \langle\tfrac{\Theta}{V}\rangle \rightarrowtriangle \ldots \rightarrowtriangle E^\circ_n \oplus \langle\tfrac{\Theta}{V}\rangle \rightarrowtriangle \mathit{Pop}\ G^\circ\,\langle\tfrac{\Omega+\Theta}{U+V}\rangle.$$

Our next aim is to prove: even upon backtracking, the goal shall pass always the same stages, independent on the starting contents of the stacks.

Lemma 7.5 (backward independence) *Consider the sequences*

$$\mathit{call}\ G\,\langle\tfrac{\Sigma}{U}\rangle \longrightarrow \mathit{exit}\ G\,\langle\tfrac{\Sigma'}{U}\rangle \longrightarrow \mathit{redo}\ G\,\langle\tfrac{\Sigma'}{U}\rangle \longrightarrow \mathit{exit}\ G\,\langle\tfrac{\Omega}{U}\rangle$$
$$\mathit{call}\ H\,\langle\tfrac{\Theta}{V}\rangle \longrightarrow \mathit{exit}\ H\,\langle\tfrac{\Theta'}{V}\rangle \longrightarrow \mathit{redo}\ H\,\langle\tfrac{\Theta'}{V}\rangle \longrightarrow \mathit{exit}\ H\,\langle\tfrac{\Psi}{V}\rangle$$

where $\Theta(H) = \Sigma(G)$*. Then* $\Theta' = \Sigma' - \Sigma + \Theta$ *and* $\Psi = \Omega - \Sigma + \Theta$*.*

In other words, the second derivation starts from a different context, but from the same goal, and it passes the same stages as in the first derivation. Notice the presence of backtracking.

This result can be generalized for arbitrary composed passes. Bearing in mind that there can be only one appearance of a fail stage in a composed pass, due to Lemma 6.3, we arrive at a general claim of independence upon the context of derivation.

Theorem 7.6 (independence) *Let the following sequences be composable, where* $m, k \geq 1$*, and let* $\Theta(H) = \Sigma(G)$*.*

$$\mathit{call}\ G\,\langle\tfrac{\Sigma}{U}\rangle \longrightarrow E_1^{G,U} \longrightarrow \ldots \longrightarrow E_m^{G,U}$$
$$\mathit{call}\ H\,\langle\tfrac{\Theta}{V}\rangle \longrightarrow E_1^{H,V} \longrightarrow \ldots \longrightarrow E_k^{H,V}$$

Then for any i *in common* ($i \leq m, k$) *holds:*

$$\mathit{If}\ E_i^{G,U} := \Gamma\ G\,\langle\tfrac{\Sigma'}{U}\rangle \ \ \mathit{then}\ \ E_i^{H,V} = \Gamma\ H\,\langle\tfrac{\Theta'}{V}\rangle, \ \ \mathit{such\ that}\ \Theta' = \Sigma' - \Sigma + \Theta.$$

Independence on the context reveals a form of *modularity* or *compositionality* in S_1:PP, to be illustrated at the end of section 8.

Motivated by the independence result above, we now define a concept of computation intended to mimick SLD-derivations in the style of Prolog, i. e. SLD-derivations with leftmost selection and depth-first search. In the following we call such SLD-derivations *Prolog computations*.

8 Modeling pure Prolog computations

Based on the concept of stages, it is possible to express Prolog computation in a succinct way. Throughout this section, Π denotes again a pure Prolog program in canonical form, as defined in Figure 1 and Definition 2.1.

Theorem 8.1 (existential termination, success, failure) *Prolog computation of a goal G relative to Π terminates existentially if there is a simple pass*

$$call\ G\ \langle \tfrac{nil}{nil} \rangle \longrightarrow_{\Pi} Pop\ G\ \langle \tfrac{\Delta}{nil} \rangle$$

In case $Pop = exit$, the Prolog computation of G is successful, otherwise it is failed.

Theorem 8.2 (the first computed answer) *If $call\ G\ \langle \frac{nil}{nil} \rangle \longrightarrow_{\Pi} exit\ G\ \langle \frac{\Delta}{nil} \rangle$, then $subst(\Delta)\ |_G$ is the first computed answer substitution for G relative to Π in Prolog.*

Theorem 8.3 (all computed answers, universal termination) *In a composable sequence*

$$call\ G\ \langle \tfrac{nil}{1/G,\mathsf{fail} \bullet nil} \rangle \longrightarrow_{\Pi}{}^{2k-1} exit\ G\ \langle \tfrac{\Delta}{1/G,\mathsf{fail} \bullet nil} \rangle$$

is $subst(\Delta)\ |_G$ the kth computed answer substitution for G relative to Π in Prolog. Furthermore, G is universally terminating relative to Π if

$$call\ G\ \langle \tfrac{nil}{1/G,\mathsf{fail} \bullet nil} \rangle \Longrightarrow_{\Pi} fail\ G\ \langle \tfrac{nil}{1/G,\mathsf{fail} \bullet nil} \rangle$$

Actually we can be a bit more precise, by considering a goal G as a subgoal of other goals. Recall that, if $call\ G\ \langle \frac{\Sigma}{U} \rangle$ is legal and $U = a_n \bullet \ldots \bullet a_1 \bullet nil$, then $call\ G\ \langle \frac{\Sigma}{U} \rangle \overset{\triangleleft}{\leftarrow}_{\Pi} Push\ \lceil a_1 \rceil\ \langle \frac{\div}{nil} \rangle$. In other words, $call\ G\ \langle \frac{\Sigma}{U} \rangle$ is a subevent of the most recent $Push\ \lceil a_1 \rceil\ \langle \frac{\div}{nil} \rangle$. Moreover, since a push event with an empty A-stack cannot be reached, we know that ours must be the oldest event of the derivation, of the form $call\ \lceil a_1 \rceil\ \langle \frac{nil}{nil} \rangle$. In analogy to subgoals in Prolog, let us define two new concepts:

Definition 8.4 (S_1-supergoal, S_1-subgoal) *Let $call\ G\ \langle \frac{\Sigma}{U} \rangle$ be legal. If $U = a_n \bullet \ldots \bullet a_1 \bullet nil$, we say that $\lceil a_1 \rceil$ is the* S_1-supergoal *of G', and G' is a* S_1-subgoal *of $\lceil a_1 \rceil$, where $G' := \Sigma(G)$. In case $U = nil$, we define G' to be its own S_1-supergoal.*

Theorem 8.5 (supergoal) *Let $call\ G\ \langle \frac{\Sigma^\circ}{U} \rangle$ be a legal event. Consider the maximal composable sequence $call\ G\ \langle \frac{\Sigma^\circ}{U} \rangle \longrightarrow_{\Pi}{}^{2k-1} exit\ G\ \langle \frac{\Sigma}{U} \rangle$. Then holds: $subst(\Sigma - \Sigma^\circ)\ |_{G'}$ is the kth and last computed answer substitution for G' relative to Π in Prolog, where $G' := \Sigma^\circ(G)$ and the query of the Prolog computation was the S_1-supergoal of G'.*

In conclusion, we show an example of a modular derivation. Let $\mathit{exit}\ A, B\ \langle \frac{\Sigma}{U} \rangle$ be a legal event. How could it have been derived? Luckily, inverse transitions are deterministic for legal events (uniqueness property), so we may unfold a legal derivation from whichever endpoint it seems more promising.

$$
\begin{aligned}
&\mathit{exit}\ A, B\ \langle \tfrac{\Sigma}{U} \rangle \\
&\quad \leftarrow\ \mathit{exit}\ B\ \langle \tfrac{\Sigma}{2/A,B \bullet U} \rangle, \text{ by } (S_1\text{:conj:4}) \\
&\quad \Longleftarrow\ \mathit{call}\ B\ \langle \tfrac{\Sigma^\circ}{2/A,B \bullet U} \rangle, \text{ by Theorem 6.5.(2), with } \Sigma \succeq \Sigma^\circ \\
&\quad \leftarrow\ \mathit{exit}\ A\ \langle \tfrac{\Sigma^\circ}{1/A,B \bullet U} \rangle, \text{ by } (S_1\text{:conj:2}) \\
&\quad \Longleftarrow\ \mathit{call}\ A\ \langle \tfrac{\Sigma^{\circ\circ}}{1/A,B \bullet U} \rangle, \text{ by Theorem 6.5.(2), with } \Sigma^\circ \succeq \Sigma^{\circ\circ} \\
&\quad \leftarrow\ \mathit{call}\ A, B\ \langle \tfrac{\Sigma^{\circ\circ}}{U} \rangle, \text{ by } (S_1\text{:conj:1})
\end{aligned}
$$

Additionally, it can be seen that the whole derivation is a composable sequence. So we obtain

Lemma 8.6 (conjunction) *Let* $\mathit{exit}\ A, B\ \langle \frac{\Sigma}{U} \rangle$ *be a legal event. Then* $\Sigma^\circ, \Sigma^{\circ\circ}$ *exist with* $\Sigma \succeq \Sigma^\circ \succeq \Sigma^{\circ\circ}$ *such that* $\mathit{exit}\ A, B\ \langle \frac{\Sigma}{U} \rangle \Longleftarrow \mathit{call}\ A, B\ \langle \frac{\Sigma^{\circ\circ}}{U} \rangle$*, and also* $\mathit{exit}\ B\ \langle \frac{\Sigma}{2/A,B \bullet U} \rangle \Longleftarrow \mathit{call}\ B\ \langle \frac{\Sigma^\circ}{2/A,B \bullet U} \rangle$ *and* $\mathit{exit}\ A\ \langle \frac{\Sigma^\circ}{1/A,B \bullet U} \rangle \Longleftarrow \mathit{call}\ A\ \langle \frac{\Sigma^{\circ\circ}}{1/A,B \bullet U} \rangle$.

In a similar way we can unfold a disjunction and an atomary goal, thus simulating the vanilla meta-interpreter in our model.

9 Related work and outlook

S_1:PP was inspired by the traditional metaphor of a box with four ports, known as the Byrd model [9], designed to represent the evolution of a procedure call in Prolog. The Byrd model is a seminal work in control flow, but it proved hard to formalize, and variable handling was not tackled in the original work at all. There are very interesting proposals like the graph-based model of Tobermann and Beckstein [23], who formalize the graph traversal idea of Byrd, defining the notion of a *trace* (of a given query with respect to a given program), as a path in a trace graph. The ports are quite lucidly defined as hierarchical nodes of such a graph. However, trace graphs are not very manageable. A finitary algebraic 'translation' of this model would be interesting. Another formal approach in the spirit of Byrd model is a continuation-based approach of Jahier, Ducassé and Ridoux [15]. There is also a stack-based attempt in [18], but although it provides for some parametrizing, it suffers essentially the same problem as the continuation-based approach, or the tracer given in [9], taken as a specification of Prolog execution: In these attempts, the mutable character of a goal in Prolog (as discussed in section 1) has not been captured. Rather, they provide a cumulative view of a derivation ('trace'), sacrificing lots of structure, which has to be reconstructed from the trace by some other device. In contrast to the few specifications of the Byrd box, there are many more general models of pure (or even full) Prolog execution. We

mention here only some models, directly relevant to S_1:PP, and for a more comprehensive discussion see e. g. [18]. Comparable to our work are stack-based approaches. Stärk gives in [22], as a side issue, a simple operational semantics of pure logic programming. A state of execution is a stack of frame stacks, where each frame consists of a goal (ancestor) and an environment. The seminal paper of Jones and Mycroft [16] was the first to present a stack-of-stacks model of execution, applicable to pure Prolog with cut added. Such approaches (including our previous attempt [18]) are in general suffering from the lack of modularity, i. e. it is not possible to abstract the execution of a subgoal.

This paper has given a simple mathematical definition S_1:PP of pure Prolog, especially suited to represent backtracking computation, and fulfilling a modularity claim. Some useful properties of the calculus have been shown. It would be important to see how this model can be extended to accomodate full Standard Prolog language. Also, the potential for specifying other languages, notably in a constraint programming area, seems worth investigating.

Acknowledgement

Many thanks for helpful comments upon a previous draft of this paper are due to C. Beierle and the anonymous referees.

References

[1] J. H. Andrews. *Logic Programming: Operational Semantics and Proof Theory*. Cambridge University Press, 1992.

[2] J. H. Andrews. The witness properties and the semantics of the Prolog cut. *Theory and Practice of Logic Programming*, 3(1):1–59, 2003.

[3] K. R. Apt and M. H. van Emden. Contributions to the theory of logic programming. *J. of the ACM*, 29:841–862, 1982.

[4] B. Arbab and D. M. Berry. Operational and denotational semantics of Prolog. *J. of Logic Programming*, 4(4):309–329, 1987.

[5] M. Baudinet. Proving termination properties of Prolog programs: A semantic approach. *J. of Logic Programming*, 14:1–29, 1992.

[6] M. Billaud. Simple operational and denotational semantics for Prolog with cut. *Theoretical Computer Science*, 71(2):193–208, 1990.

[7] M. Billaud. Axiomatizations of backtracking. In *Proc. of the STACS*, pages 71–82, 1992.

[8] E. Börger and D. Rosenzweig. A mathematical definition of full Prolog. *Science of Computer Programming*, 24(3):249–286, 1995.

[9] Lawrence Byrd. Understanding the control flow of Prolog programs. In S. A. Tärnlund, editor, *Proc. of the 1980 Logic Programming Workshop*, pages 127–138, Debrecen, Hungary, 1980. Also as D. A. I. Research Paper No. 151.

[10] M. H. M. Cheng, M. H. van Emden, R. N. Horspool, and M. Levy. Compositional operational semantics for Prolog programs. *J. New Generation Computing,*, 10(3):315–328, 1992.

[11] E. P. de Vink. Comparative semantics for Prolog with cut. *Science of Computer Programming*, 13(1):237–264, 1989.

[12] P. Deransart, A. Ed-Dbali, and L. Cervoni. *Prolog: The Standard (Reference Manual)*. Springer-Verlag, 1996.

[13] P. Deransart and G. Ferrand. An operational formal definition of Prolog: A specification method an its application. *J. New Generation Computing*, 10:121–171, 1992.

[14] E. Eder. Properties of substitutions and unifications. *J. Symbolic Computation*, 1:31–46, 1985.

[15] E. Jahier, M. Ducassé, and O. Ridoux. Specifying Byrd's box model with a continuation semantics. In *Proc. of the WLPE'99, Las Cruces, NM*, volume 30 of *ENTCS*. Elsevier, 2000. http://www.elsevier.com/locate/entcs/.

[16] N. D. Jones and A. Mycroft. Stepwise development of operational and denotational semantics for Prolog. In *Proc. of the ISLP'84*, pages 281–288, Atlantic City, 1984.

[17] M. Kulaš. Pure Prolog execution in 21 rules. In *Proc. of the 5th Workshop on Rule-Based Constraint Reasoning and Programming (RCoRP'03), Kinsale*, September 2003.

[18] M. Kulaš and C. Beierle. Defining Standard Prolog in rewriting logic. In K. Futatsugi, editor, *Proc. of WRLA 2000, Kanazawa*, volume 36 of *ENTCS*. Elsevier, 2001. http://www.elsevier.com/locate/entcs.

[19] T. Lindgren. A continuation-passing style for Prolog. In *Proc. of the 11th Int. Symposium on Logic Programming (ILPS'94)*, pages 603–617, Ithaca, NY, 1994.

[20] J. W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, 2. edition, 1987.

[21] T. Nicholson and N. Foo. A denotational semantics for Prolog. *ACM Trans. on Prog. Lang. and Systems*, 11(4):650–665, 1989.

[22] Robert F. Stärk. The theoretical foundations of LPTP (a logic program theorem prover). *J. of Logic Programming*, 36(3):241–269, 1998. Source distribution http://www.inf.ethz.ch/~staerk/lptp.html.

[23] G. Tobermann and C. Beckstein. What's in a trace: The box model revisited. In *Proc. of the 1st Int. Workshop on Automated and Algorithmic Debugging (AADEBUG'93), Linköping, Sweden*, volume 749 of *LNCS*. Springer-Verlag, 1993.

A An example proof

As an illustration of S_1:PP, here is a proof of the pendant lemma, and its application in proving uniqueness of legal transitions. Although lengthy, the proof of Lemma 4.1 is rather schematic.

Lemma 4.1 (pendant) *If* $call\ G\ \langle \frac{nil}{nil} \rangle \rightarrowtriangle^*\ fail\ H\ \langle \frac{\Theta}{W} \rangle$, *then*

$$call\ G\ \langle \tfrac{nil}{nil} \rangle \rightarrowtriangle^*\ call\ H\ \langle \tfrac{\Theta}{W} \rangle \rightarrowtriangle^*\ fail\ H\ \langle \tfrac{\Theta}{W} \rangle. \tag{5}$$

If $call\ G\ \langle \frac{nil}{nil} \rangle \rightarrowtriangle^*\ redo\ H\ \langle \frac{\Theta}{W} \rangle$, *then*

$$call\ G\ \langle \tfrac{nil}{nil} \rangle \rightarrowtriangle^*\ exit\ H\ \langle \tfrac{\Theta}{W} \rangle \rightarrowtriangle^*\ redo\ H\ \langle \tfrac{\Theta}{W} \rangle. \tag{6}$$

Proof. We shall use induction on the length n of derivation to prove the two parts of the lemma simultaneously. The inductive assumption for each of the parts shall be denoted as Ind(fail) and Ind(redo) respectively. First let us construct base cases. With Lemma 4.1.(5) we are lucky, since derivations of length $n = 1$ can be failed. There are three such cases, and they directly satisfy Lemma 4.1.(5):

$$call\ \mathsf{fail}\ \langle \tfrac{nil}{nil} \rangle \rightarrowtriangle fail\ \mathsf{fail}\ \langle \tfrac{nil}{nil} \rangle,\ \text{by } (S_1\text{:fail}) \tag{7}$$

$$call\ T_1{=}T_2\ \langle \tfrac{nil}{nil} \rangle \rightarrowtriangle fail\ T_1{=}T_2\ \langle \tfrac{nil}{nil} \rangle,\ \text{by } (S_1\text{:unif:1}),\ \text{if no mgu} \tag{8}$$

$$call\ G_A\ \langle \tfrac{nil}{nil} \rangle \rightarrowtriangle fail\ G_A\ \langle \tfrac{nil}{nil} \rangle,\ \text{by } (S_1\text{:atom:1}),\ \text{if no resolvent} \tag{9}$$

With Lemma 4.1.(6) we have a harder time since no derivation of length $n \leq 2$ can end up in a redo, so with the number of derivations getting out of hand, we turn to reconstructing a minimal legal derivation of a redo. A legal redo event stems from a redo or from a fail, so for a minimal derivation it had to be a fail: $fail\ B\ \langle \frac{\Sigma}{2/A,B \bullet V} \rangle \rightarrowtriangle redo\ A\ \langle \frac{\Sigma}{1/A,B \bullet V} \rangle$. Now we are looking for a minimal derivation of this fail event, which may not include any redo events. The predecessor may be a fail (pushing the A-stack) or a call (not affecting the stacks). In case of a call, we get $exit\ A\ \langle \frac{\Sigma}{1/A,B \bullet V} \rangle \rightarrowtriangle call\ B\ \langle \frac{\Sigma}{2/A,B \bullet V} \rangle \rightarrowtriangle fail\ B\ \langle \frac{\Sigma}{2/A,B \bullet V} \rangle$. In this manner we eventually reconstruct the minimal derivations for redo, and they satisfy Lemma 4.1.(6):

$$call\ A,B\ \langle \tfrac{nil}{nil} \rangle \rightarrowtriangle call\ A\ \langle \tfrac{nil}{1/A,B \bullet nil} \rangle \rightarrowtriangle exit\ A\ \langle \tfrac{nil}{1/A,B \bullet nil} \rangle \rightarrowtriangle call\ B\ \langle \tfrac{nil}{2/A,B \bullet nil} \rangle \rightarrowtriangle$$
$$\rightarrowtriangle fail\ B\ \langle \tfrac{nil}{2/A,B \bullet nil} \rangle \rightarrowtriangle redo\ A\ \langle \tfrac{nil}{1/A,B \bullet nil} \rangle$$

Assume Lemma 4.1 holds for derivations of length $1 \leq n < k$ and consider a derivation of length k. We shall use the following simple observation: If $call\ G\ \langle \frac{nil}{nil} \rangle \rightarrowtriangle^* \Gamma\ A\ \langle \frac{\Theta}{W} \rangle$, and $W \neq nil$ or $\Gamma \neq call$, then also

$$call\ G\ \langle \tfrac{nil}{nil} \rangle \rightarrowtriangle^+ \Gamma\ A\ \langle \tfrac{\Theta}{W} \rangle \tag{10}$$

First we discuss the cases for a legal fail event. There are eight possibilities for the last step of its derivation: (S_1:fail), (S_1:unif:1), (S_1:atom:1), (S_1:conj:3), (S_1:disj:3), (S_1:atom:3), (S_1:true:2) and (S_1:unif:2). The first three cases are again directly

satisfied. Case (S_1:conj:3):

$$call\ G\ \langle \tfrac{nil}{nil} \rangle \rightarrowtriangle^{+} fail\ A\ \langle \tfrac{\Theta}{1/A,B \bullet W} \rangle \rightarrowtriangle fail\ A, B\ \langle \tfrac{\Theta}{W} \rangle,\ \text{the last step} \tag{11}$$

$$call\ G\ \langle \tfrac{nil}{nil} \rangle \rightarrowtriangle^{*} call\ A\ \langle \tfrac{\Theta}{1/A,B \bullet W} \rangle \rightarrowtriangle^{*} fail\ A\ \langle \tfrac{\Theta}{1/A,B \bullet W} \rangle,\ \text{by IND(FAIL)} \tag{12}$$

$$call\ G\ \langle \tfrac{nil}{nil} \rangle \rightarrowtriangle^{*} call\ A, B\ \langle \tfrac{\Theta}{W} \rangle \rightarrowtriangle call\ A\ \langle \tfrac{\Theta}{1/A,B \bullet W} \rangle,\ \text{predecessor \& (10)} \tag{13}$$

$$call\ G\ \langle \tfrac{nil}{nil} \rangle \rightarrowtriangle^{*} call\ A, B\ \langle \tfrac{\Theta}{W} \rangle \rightarrowtriangle^{*} fail\ A, B\ \langle \tfrac{\Theta}{W} \rangle,\ \text{by (11)-(13)},\quad q.e.d.$$

Case (S_1:atom:3) is similar to the previous. Case (S_1:disj:3) can be handled by double use of IND(FAIL):

$$call\ G\ \langle \tfrac{nil}{nil} \rangle \rightarrowtriangle^{+} fail\ B\ \langle \tfrac{\Theta}{2/A;B \bullet W} \rangle \rightarrowtriangle fail\ A; B\ \langle \tfrac{\Theta}{W} \rangle,\ \text{the last step} \tag{14}$$

$$call\ G\ \langle \tfrac{nil}{nil} \rangle \rightarrowtriangle^{*} call\ B\ \langle \tfrac{\Theta}{2/A;B \bullet W} \rangle \rightarrowtriangle^{*} fail\ B\ \langle \tfrac{\Theta}{2/A;B \bullet W} \rangle,\ \text{by IND(FAIL)} \tag{15}$$

$$call\ G\ \langle \tfrac{nil}{nil} \rangle \rightarrowtriangle^{*} fail\ A\ \langle \tfrac{\Theta}{1/A;B \bullet W} \rangle \rightarrowtriangle call\ B\ \langle \tfrac{\Theta}{2/A;B \bullet W} \rangle,\ \text{predecessor \& (10)} \tag{16}$$

$$call\ G\ \langle \tfrac{nil}{nil} \rangle \rightarrowtriangle^{*} call\ A\ \langle \tfrac{\Theta}{1/A;B \bullet W} \rangle \rightarrowtriangle^{*} fail\ A\ \langle \tfrac{\Theta}{1/A;B \bullet W} \rangle,\ \text{by IND(FAIL)} \tag{17}$$

$$call\ G\ \langle \tfrac{nil}{nil} \rangle \rightarrowtriangle^{*} call\ A; B\ \langle \tfrac{\Theta}{W} \rangle \rightarrowtriangle call\ A\ \langle \tfrac{\Theta}{1/A;B \bullet W} \rangle,\ \text{predecessor \& (10)} \tag{18}$$

$$call\ G\ \langle \tfrac{nil}{nil} \rangle \rightarrowtriangle^{*} call\ A; B\ \langle \tfrac{\Theta}{W} \rangle \rightarrowtriangle^{*} fail\ A; B\ \langle \tfrac{\Theta}{W} \rangle,\ \text{by (14)-(18)},\quad q.e.d.$$

For the last two cases (redo-fail transitions) we shall use IND(REDO). Case (S_1:true:2):

$$call\ G\ \langle \tfrac{nil}{nil} \rangle \rightarrowtriangle^{+} redo\ \mathsf{true}\ \langle \tfrac{\Theta}{W} \rangle \rightarrowtriangle fail\ \mathsf{true}\ \langle \tfrac{\Theta}{W} \rangle,\ \text{the last step} \tag{19}$$

$$call\ G\ \langle \tfrac{nil}{nil} \rangle \rightarrowtriangle^{*} exit\ \mathsf{true}\ \langle \tfrac{\Theta}{W} \rangle \rightarrowtriangle^{*} redo\ \mathsf{true}\ \langle \tfrac{\Theta}{W} \rangle,\ \text{by IND(REDO)} \tag{20}$$

$$call\ G\ \langle \tfrac{nil}{nil} \rangle \rightarrowtriangle^{*} call\ \mathsf{true}\ \langle \tfrac{\Theta}{W} \rangle \rightarrowtriangle exit\ \mathsf{true}\ \langle \tfrac{\Theta}{W} \rangle,\ \text{predecessor \& (10)} \tag{21}$$

$$call\ G\ \langle \tfrac{nil}{nil} \rangle \rightarrowtriangle^{*} call\ \mathsf{true}\ \langle \tfrac{\Theta}{W} \rangle \rightarrowtriangle^{*} fail\ \mathsf{true}\ \langle \tfrac{\Theta}{W} \rangle,\ \text{by (19)-(21)},\quad q.e.d.$$

The last case (S_1:unif:2) can be handled in the same manner.

It remains to discuss the cases for a legal redo event. There are four possibilities for the last step of its derivation: (S_1:conj:5), (S_1:conj:6), (S_1:disj:5) and (S_1:atom:4). Here a symmetry between 'entering redo' and 'leaving exit' takes over. Namely, if we take the rules for entering redo, turn the arrow around and replace exit for redo and call for fail, then we obtain the rules for leaving exit. Due to determinacy of such transitions, each 'entering redo' can be simulated with an appropriate 'leaving exit', which reconstructs the stacks in exactly the same manner. The case (S_1:conj:5) is special because it uses IND(FAIL):

$$call\ G\ \langle \tfrac{nil}{nil} \rangle \rightarrowtriangle^{+} fail\ B\ \langle \tfrac{\Theta}{2/A,B \bullet W} \rangle \rightarrowtriangle redo\ A\ \langle \tfrac{\Theta}{1/A,B \bullet W} \rangle,\ \text{the last step} \tag{22}$$

$$call\ G\ \langle \tfrac{nil}{nil} \rangle \rightarrowtriangle^{*} call\ B\ \langle \tfrac{\Theta}{2/A,B \bullet W} \rangle \rightarrowtriangle^{*} fail\ B\ \langle \tfrac{\Theta}{2/A,B \bullet W} \rangle,\ \text{by IND(FAIL)} \tag{23}$$

$$call\ G\ \langle \tfrac{nil}{nil} \rangle \rightarrowtriangle^{*} exit\ A\ \langle \tfrac{\Theta}{1/A,B \bullet W} \rangle \rightarrowtriangle call\ B\ \langle \tfrac{\Theta}{2/A,B \bullet W} \rangle,\ \text{predecessor \& (10)} \tag{24}$$

$$call\ G\ \langle \tfrac{nil}{nil} \rangle \rightarrowtriangle^{*} exit\ A\ \langle \tfrac{\Theta}{1/A,B \bullet W} \rangle \rightarrowtriangle^{*} redo\ A\ \langle \tfrac{\Theta}{1/A,B \bullet W} \rangle,\ \text{by (22)-(24)},\quad q.e.d.$$

Case (S_1:conj:6):

$$call\ G\ \langle \tfrac{nil}{nil} \rangle \rightarrowtriangle^{+} redo\ A, B\ \langle \tfrac{\Theta}{W} \rangle \rightarrowtriangle redo\ B\ \langle \tfrac{\Theta}{2/A,B \bullet W} \rangle \tag{25}$$

$$call\ G\ \langle \tfrac{nil}{nil} \rangle \rightarrowtriangle^{*} exit\ A, B\ \langle \tfrac{\Theta}{W} \rangle \rightarrowtriangle^{*} redo\ A, B\ \langle \tfrac{\Theta}{W} \rangle,\ \text{by IND(REDO)} \tag{26}$$

$$call\ G\ \langle \tfrac{nil}{nil} \rangle \rightarrowtriangle^{*} exit\ B\ \langle \tfrac{\Theta}{2/A,B \bullet W} \rangle \rightarrowtriangle exit\ A, B\ \langle \tfrac{\Theta}{W} \rangle,\ \text{predecessor \& (10)} \tag{27}$$

$$call\ G\ \langle \tfrac{nil}{nil} \rangle \rightarrowtriangle^{*} exit\ B\ \langle \tfrac{\Theta}{2/A,B \bullet W} \rangle \rightarrowtriangle redo\ B\ \langle \tfrac{\Theta}{2/A,B \bullet W} \rangle,\ \text{by (25)-(27)},\quad q.e.d.$$

The last two cases (S_1:disj:5) and (S_1:atom:4) can be handled in the same manner. This concludes the proof of Lemma 4.1. □

Theorem 4.2 (legal transitions are unique) *If E is a legal event, then E can have only one legal predecessor, and only one successor. In case E is non-initial, there is exactly one legal predecessor. In case E is non-final, there is exactly one successor.*

Proof. The successor part follows from the functionality of $\rightarrowtriangle$. Looking at the rules, we note that only two kinds of events may have more than one predecessor: *fail* $G_A\,\langle\frac{\Sigma}{U}\rangle$ and *fail* $T_1{=}T_2\,\langle\frac{\Sigma}{U}\rangle$. Let *fail* $T_1{=}T_2\,\langle\frac{\Sigma}{U}\rangle$ be a legal event. Then it has at least one derivation from an initial event, say *call* $G\,\langle\frac{nil}{nil}\rangle$. Its predecessor in this derivation may have been *call* $T_1{=}T_2\,\langle\frac{\Sigma}{U}\rangle$, on the condition that $\Sigma(T_1)$ and $\Sigma(T_2)$ have no mgu, or it may have been *redo* $T_1{=}T_2\,\langle\frac{\sigma\bullet\Sigma}{U}\rangle$. In the latter case we obtain:

$$\textit{call}\ G\,\langle\tfrac{nil}{nil}\rangle \rightarrowtriangle^*\ \textit{redo}\ T_1{=}T_2\,\langle\tfrac{\sigma\bullet\Sigma}{U}\rangle \rightarrowtriangle\ \textit{fail}\ T_1{=}T_2\,\langle\tfrac{\Sigma}{U}\rangle,\ \text{assumption} \tag{28}$$

$$\textit{call}\ G\,\langle\tfrac{nil}{nil}\rangle \rightarrowtriangle^*\ \textit{exit}\ T_1{=}T_2\,\langle\tfrac{\sigma\bullet\Sigma}{U}\rangle \rightarrowtriangle^*\ \textit{redo}\ T_1{=}T_2\,\langle\tfrac{\sigma\bullet\Sigma}{U}\rangle,\ \text{Lemma 4.1.(6)} \tag{29}$$

$$\textit{call}\ G\,\langle\tfrac{nil}{nil}\rangle \rightarrowtriangle^*\ \textit{call}\ T_1{=}T_2\,\langle\tfrac{\Sigma}{U}\rangle \rightarrowtriangle\ \textit{exit}\ T_1{=}T_2\,\langle\tfrac{\sigma\bullet\Sigma}{U}\rangle,\ \text{by } (S_1\text{:unif:1}) \tag{30}$$

Let us comment a bit on (29)-(30). From (29) we know that *exit* $T_1{=}T_2\,\langle\frac{\sigma\bullet\Sigma}{U}\rangle$ is a legal event, i. e. it is reachable from an initial event. But there is only one possibility to enter *exit* $T_1{=}T_2\,\langle\frac{\sigma\bullet\Sigma}{U}\rangle$, namely via ($S_1$:unif:1), under the condition $mgu(\Sigma(T_1),\Sigma(T_2)) = \sigma$. To sum up: If *redo* $T_1{=}T_2\,\langle\frac{\sigma\bullet\Sigma}{U}\rangle$ is a legal predecessor of *fail* $T_1{=}T_2\,\langle\frac{\Sigma}{U}\rangle$, then $mgu(\Sigma(T_1),\Sigma(T_2)) = \sigma$.

So depending on the existence of this particular mgu, either *redo* $T_1{=}T_2\,\langle\frac{\sigma\bullet\Sigma}{U}\rangle$ or *call* $T_1{=}T_2\,\langle\frac{\Sigma}{U}\rangle$ is a legal predecessor of *fail* $T_1{=}T_2\,\langle\frac{\Sigma}{U}\rangle$, but never both.

By a similar argument, this time using Lemma 4.1.(5), we can prove that *fail* $G_A\,\langle\frac{\Sigma}{U}\rangle$ can have only one legal predecessor. □

Congruence of Bisimulation in a Non-Deterministic Call-By-Need Lambda Calculus

Matthias Mann

Institut für Informatik
Johann Wolfgang Goethe-Universität
Postfach 11 19 32
D-60054 Frankfurt, Germany
mann@cs.uni-frankfurt.de

Abstract

We present a call-by-need $\boldsymbol{\lambda}$-calculus $\boldsymbol{\lambda}_{\mathrm{ND}}$ with an erratic non-deterministic operator `pick` and a non-recursive `let`. A definition of a bisimulation is given, which has to be based on a further calculus named $\boldsymbol{\lambda}_{\approx}$, since the naïve bisimulation definition is useless. The main result is that bisimulation in $\boldsymbol{\lambda}_{\approx}$ is a congruence and coincides with the contextual equivalence.

The proof is a non-trivial extension of Howe's method. This might be a step towards defining useful bisimulation relations and proving them to be congruences in calculi that extend the $\boldsymbol{\lambda}_{\mathrm{ND}}$-calculus.

Key words: Bisimulation, Congruence, Contextual Equivalence, Non-determinism, Call-by-need Lambda Calculus

1 Introduction

Equality plays a prominent role in reasoning about programs. Thus specifically for $\boldsymbol{\lambda}$-calculi, there is a certain range of concepts when two terms should be considered equal. First, there is the notion of *convertibility*, i.e. two terms are equivalent if they could be transformed to each other according to the conversion rules of the calculus. Usually conversion is permitted inside arbitrary contexts, i.e. program fragments, hence convertibility is a congruence.

For deterministic calculi, there is a large number of reasonable equations, e.g. useful program transformations, which neither are provable by, nor stand in contradiction to, convertibility. Hence there is a serious interest in $\boldsymbol{\lambda}$-theories (cf. [5, Part IV]), that is, consistent extensions of the $\boldsymbol{\lambda}$-calculus which are closed under derivation.

This is a preliminary version. The final version will be published in Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

The *contextual equivalence* due to [22], which discriminates terms by their behaviour in all contexts, falls into this category. Typically, termination is observed (cf. [18]). Thus, with $\Downarrow$ denoting termination while C stands for program contexts, contextual equivalence $\simeq_c$ can be expressed as

$$s \simeq_c t \iff (\forall C : C[s] \Downarrow \iff C[t] \Downarrow)$$

This obviously establishes a congruence and since it is based on the observation of some behaviour, it is often called *observational congruence*. We consider contextual equivalence more significant than convertibility for several reasons. First, contextual equivalence does not directly depend on the reduction rules of a calculus and therefore can be used as a separate justification for its design. Secondly, contextual equivalence validates more meaningful equations than convertibility, the latter of which may, e.g., only relate terms of the same asymptotic complexity in some cases (cf. [28]).

In a non-deterministic setting, on the other hand, convertibility in general leads to an inconsistent theory. Hence there are, of course, convertible terms which are *not* contextual equivalent. But in many cases, the deterministic part of a calculus can be proven sound w.r.t. contextual equivalence.

Thus contextual equivalence is of great interest in the important field of correct program transformations, for both, deterministic and non-deterministic calculi. However, proofs of contextual equivalence could turn out to be non-trivial, since all contexts have to be taken into account. Hence it is common to reduce the number of contexts by a *context lemma* (cf. [18]), e.g. in [19] the observation of specific machine configurations is sufficient whereas [20,14,13,30] use evaluation contexts. This is unquestionably a very useful, and for many tasks adequate, approach. But it does not resolve the issue in principle, since still a generally infinite number of contexts has to be considered.

Bisimulation, the origins of which date back to the work of Park [24] and Milner [17], provides a more stepwise approach for proving equations and hence bisimulation techniques have been applied frequently to functional programming (e.g. [2,26,8,9]) since then. In this area, there is some variety of relations, but in general the definition of bisimilarity involves the greatest fixed point of some monotonic operator.

Because of this, a bisimulation proof can be very concise where the corresponding proof of contextual equivalence is subtle, as example 4.9 will show. So it lends itself to a powerful proof instrument, but in order to employ it for proving correctness of program transformations, one has to ensure that it is a congruence. E.g. Groote and Vaandrager, in the introduction of [10], emphasise that proving bisimulation a congruence is vital. But as the effort in [2,11,12] shows, this is in general not a trivial task. Moreover, we demonstrate in example 3.3, that within the scope of non-determinism combined with sharing, bisimulation has to be designed carefully.

The aim of this work now is to establish a sensible definition of bisimulation for a non-deterministic call-by-need $\boldsymbol{\lambda}$-calculus and to show that it complies

with contextual equivalence. Therefore the structure of the paper is as follows: In section 2, after we survey several $\boldsymbol{\lambda}$-calculi with non-determinism and/or sharing, we discuss the major techniques for proving bisimulation a congruence.

The $\boldsymbol{\lambda}_{\mathrm{ND}}$-calculus, the subject of our study, will be introduced in section 3 then. We have intentionally chosen a very basic calculus to act as a starting point for further studies. Since for several reasons we will explain there, a definition of bisimulation in $\boldsymbol{\lambda}_{\mathrm{ND}}$ working directly with `let`-environments is problematic, we develop in section 4 with the $\boldsymbol{\lambda}_{\approx}$-calculus a way to prune the evaluation in environments at an arbitrary finite depth. We accomplish this by adapting the reduction rules, so that bisimulation may be based upon reduction to pure abstractions without a surrounding `let`-environment while recording every possible outcome of the original environment. This empowers us to define bisimulation and eventually prove it a congruence in theorem 5.2 by an extension of Howe's method in [11,12], i.e. that the so-called "precongruence candidate" is preserved under reduction. The section concludes with its main result, namely that in $\boldsymbol{\lambda}_{\approx}$ bisimulation matches contextual equivalence.

In section 6, the link between the $\boldsymbol{\lambda}_{\mathrm{ND}}$- and the $\boldsymbol{\lambda}_{\approx}$-calculus is established. The achievement of theorem 6.3 is, that the contextual equivalence of the $\boldsymbol{\lambda}_{\approx}$-calculus agrees with the one of the $\boldsymbol{\lambda}_{\mathrm{ND}}$-calculus. Its proof relies heavily on the diagram method as e.g. [14,13] use it. Due to space limitations we present sketches for the most important proofs and will refer to [15] for supplying complete evidence.

2 Related Work

Of course, there has been a lot of research on extended $\boldsymbol{\lambda}$-calculi and also a fair amount on how to prove bisimulation a congruence in this area. Since it seems impossible to take all of the publications on this subject into account, we will briefly discuss only some of the related work.

2.1 Non-Determinism and Sharing in $\boldsymbol{\lambda}$-Calculi

When introducing a non-deterministic construct into a programming language or, e.g. a $\boldsymbol{\lambda}$-calculus, a number of questions have to be clarified. Apart from the classification of non-determinism as e.g. in [31], we consider a major topic the decision, what kind of terms should be permitted to be copied.

Non-determinism in languages without sharing, i.e. those that retain a copying (β)-rule like e.g. [23,7,29], is completely different from our work because it will distinguish $\lambda x.(x + x)$ from $\lambda x.(2 * x)$. Likewise is the situation with [6], since, as usual also in explicit substitution-calculi (cf. [1]), substitutions are distributed over applications and hence duplicated.

The deterministic call-by-need calculi of [4,3,16] realise explicit sharing using a `let`-construct (or special syntactic entities, as is the case in [3]) and

restrict copying to abstractions. However, their equational theory is based on convertibility rather than on contextual equivalence.

Thus the calculi in [20,14,13,30] which all provide a non-deterministic choice, sharing and contextual equivalence roughly represent the direction of our investigations. Though there are a few differences. Since these papers do not discuss bisimulation, it seemed sensible to carry out our studies in a rather elementary calculus, as this should increase readability, too.

Hence, like the work of [14,13], the $\boldsymbol{\lambda}_{\mathrm{ND}}$-calculus only has a non-recursive `let`, whereas the calculi in [20,30] provide recursive bindings. Furthermore, like [14,13] but in contrast to [20,30], the $\boldsymbol{\lambda}_{\mathrm{ND}}$-calculus neither has a `case` nor data constructors.

2.2 Proving Bisimilarity a Congruence

As indicated before, for non-deterministic $\boldsymbol{\lambda}$-calculi in combination with sharing there has not been much research on bisimulation in relation to contextual equivalence. The lazy lambda calculus of [2] is a deterministic, and in fact call-by-name $\boldsymbol{\lambda}$-calculus. Denotational approaches of this kind are connected to operational techniques by Pitts, but [25] does not incorporate non-determinism.

Of the purely operational methods, the rule format of [27] is deterministic, while the approach of Howe [11,12] in principle permits non-deterministic evaluation. Also in [10] bisimulation is shown a congruence, but their rule format is too restricted to represent our calculus. As Howe already remarks, the definition of the precongruence candidate in [10] is too weak to be proven stable under substitutions [12, Lemma 3.2]. Like his earlier work [11], the technique of Howe assumes that every term may be copied, and hence has to be adapted in order to cope with sharing.

Even though not dealing with sharing, Sands demonstrates in [26] the extensibility of Howe's approach by applying it to express improvements; and also [8,9] makes use of the method, but for typed programs. So the decision to base our work on [11] looked most promising.

3 $\boldsymbol{\lambda}_{\mathrm{ND}}$ – a Non-Deterministic $\boldsymbol{\lambda}$-Calculus with Sharing

The $\boldsymbol{\lambda}_{\mathrm{ND}}$-calculus closely resembles the one of [14], apart from the difference that the nondeterministic choice is modelled by the syntactic construct `pick` rather than a constant. In the grammar of figure 1, let V denote a non-

$$E ::= V \;\mid\; (\lambda x.E) \;\mid\; (E\ E) \;\mid\; (\texttt{let}\ x = E\ \texttt{in}\ E) \;\mid\; (\texttt{pick}\ E\ E)$$

Fig. 1. Syntax for expressions in the language Λ_{ND}

terminal for variables. Hence the terms of the language, referred to as Λ_{ND}, are variables or formed by application as well as the operators λ, `let` and

`pick`. Since the symbol $=$ is part of the `let`-construct, we use $\equiv$ for syntactic equality up to renaming of bound variables. Furthermore, we write $s[t/x]$ for substituting every free occurrence of x in s by t and adopt the distinct variable convention, i.e. suppose all bound variables to be distinct from each other and the free variables. We implicitly assume this convention to take effect after every reduction step, so e.g. the double occurrence of the term $\lambda y.r$ in the specification of the (cp)-rule below, does not pose a problem.

$$\texttt{let } x = (\texttt{let } y = t_y \texttt{ in } t_x) \texttt{ in } s \xrightarrow{llet} \texttt{let } y = t_y \texttt{ in } (\texttt{let } x = t_x \texttt{ in } s) \quad \text{(llet)}$$

$$(\texttt{let } x = t_x \texttt{ in } s)\, t \xrightarrow{lapp} \texttt{let } x = t_x \texttt{ in } (s\, t) \quad \text{(lapp)}$$

$$(\lambda x.s)\, t \xrightarrow{lbeta} \texttt{let } x = t \texttt{ in } s \quad \text{(lbeta)}$$

$$\texttt{pick } s\ t \xrightarrow{ndl} s \quad \text{(ndl)}$$

$$\texttt{pick } s\ t \xrightarrow{ndr} t \quad \text{(ndr)}$$

$$\xrightarrow{nd} \;=\; \xrightarrow{ndl} \;\cup\; \xrightarrow{ndr} \quad \text{(nd)}$$

$$\texttt{let } x = \lambda y.r \texttt{ in } D[x] \xrightarrow{cp} \texttt{let } x = \lambda y.r \texttt{ in } D[\lambda y.r] \quad \text{(cp)}$$

Fig. 2. The reduction rules of the $\boldsymbol{\lambda}_{\text{ND}}$-calculus

As usual, a *context* is a term with a single hole and with $C[s]$ we denote filling the hole of a context C with the term s. Note, that the distinct variable convention does not apply to variables which become bound in the hole of a context. The $\boldsymbol{\lambda}_{\text{ND}}$-calculus will be equipped with an operational semantics based on a small-step reduction relation. We will give a succinct account of these rules from figure 2. The purpose of (llet) and (lapp) is mainly to rearrange `let`-bindings for subsequent reductions. The ordinary (β)-rule is superseded by (lbeta) which just creates a `let`-binding. The rules (ndl) and (ndr) implement the non-deterministic choice and are combined into (nd).

With (cp) one occurrence of a variable bound to an abstraction may be replaced with a *copy* of this abstraction. Note that in contrast to the $\boldsymbol{\lambda}_{\approx}$-calculus of section 4, the rule (cp) copies an abstraction only to one location at a time. This not only conforms with earlier work on call-by-need $\boldsymbol{\lambda}$-calculi (cf. [4,3,16]) but is also closer to the implementation of lazy functional languages than a simultaneous substitution would be.

In order to obtain call-by-need evaluation, the *normal-order* reduction defined later will always take place in *reduction contexts*. These do not introduce a hole in the argument of an application, nor in the binding of a `let`, nor within a λ-term either. In figure 3 the sets $\mathcal{R}$ and $\mathcal{S}$ of reduction and *surface contexts* are designated by the symbols R and S respectively. Surface contexts do not possess a hole under an abstraction and will become more important in section 4. Note, that every reduction context is also a surface context.

$$
\begin{array}{ll}
A_L ::= [\,]\,e & A_L^* ::= [\,] \mid A_L[A_L^*] \\
L_R ::= \texttt{let}\ x = e\ \texttt{in}\ [\,] & L_R^* ::= [\,] \mid L_R[L_R^*]
\end{array}
$$

$$
\begin{array}{l}
R ::= L_R^*[A_L^*] \mid L_R^*[\texttt{let}\ x = A_L^*\ \texttt{in}\ R[x]] \\
S ::= [\,] \mid S\,e \mid e\,S \mid \texttt{let}\ x = e\ \texttt{in}\ S \mid \\
\qquad \texttt{let}\ x = S\ \texttt{in}\ e \mid \texttt{pick}\ S\ e \mid \texttt{pick}\ e\ S
\end{array}
$$

Fig. 3. Major context classes for Λ_{ND}

Let $a \in \{\mathit{llet}, \mathit{lapp}, \mathit{lbeta}, \mathit{ndl}, \mathit{ndr}, \mathit{cp}\}$ be any of the reduction rules in figure 2. We then denote with $\xrightarrow{\mathcal{R},\ a}$ the application of the rule (a) in any reduction context $R \in \mathcal{R}$ and write $\to^*$ for the reflexive-transitive closure of reduction relations. The normal-order reduction of the following definition uniquely identifies a normal-order *redex* and is, except for the non-deterministic rules, also unique.

Definition 3.1 A reduction $s \xrightarrow{\mathcal{R},\ a} t$ is called *normal-order* and depicted by $s \xrightarrow{n,\ a} t$ if it is one of the following.

(i) If $s \equiv L_R^*[A_L^*[r]]$ and rule (lapp), (lbeta), (ndl) or (ndr) is applied to r.

(ii) If $s \equiv L_R^*[\texttt{let}\ x = A_L^*[r]\ \texttt{in}\ R[x]]$ with some reduction context R such that rule (lapp), (lbeta), (ndl) or (ndr) is applied to r.

(iii) If $s \equiv L_R^*[\texttt{let}\ x = \lambda y.r\ \texttt{in}\ R[x]] \xrightarrow{n,\ cp} L_R^*[\texttt{let}\ x = \lambda y.r\ \texttt{in}\ R[\lambda y.r]] \equiv t$ by rule (cp) for some reduction context R.

(iv) If rule (llet) is applied as follows:

$$
\begin{aligned}
s \equiv L_R^*[\texttt{let}\ x = (\texttt{let}\ y = t_y\ \texttt{in}\ t_x)\ \texttt{in}\ R[x]] &\xrightarrow{n,\ \mathit{llet}} \\
L_R^*[\texttt{let}\ y = t_y\ \texttt{in}\ (\texttt{let}\ x = t_x\ \texttt{in}\ R[x])] &\equiv t
\end{aligned}
$$

The above definition complies with [13] and slightly differs from [4] as discussed in [13, p. 42]. Intuitively, it can be described as follows. Descend into contexts of the form L_R and subsequently A_L, until (nd), (lapp) or (lbeta) becomes applicable, the case (i). If during this process a variable is encountered, follow its binding. Whenever possible, perform (cp) or (llet) for the variable in question, i.e. cases (iii) and (iv) respectively. Otherwise, in case (ii), if the variable is bound to an application, descend into the A_L^*-context as far as possible in order to apply (nd), (lapp) or (lbeta).

The notion of *convergence* is then defined by a normal-order reduction sequence to a term of the form $L_R^*[\lambda x.t]$, i.e. a *weak head normal form*, WHNF for short. So we write $s \Downarrow t$ if and only if $s \xrightarrow{n}{}^* t$ and t is a WHNF, $s \Downarrow$ if there exists such a t and $s \not\Downarrow$ if not. Apparently, the normal-order reduction is neither confluent nor terminating, i.e. a term may reduce to multiple weak head normal forms or none at all.

The procedure to determine the normal-order redex is quite complex, so it is not obvious how to represent the normal-order reduction directly by a structural operational semantics. E.g. the structured evaluation systems of [12], apart from being geared to big-step operational semantics, seem not capable of this. This arises from the fact that both, normal-order reducible terms and weak head normal forms, could be formed with the `let`-operator.

3.1 Contextual Equivalence

Convergence, as defined in the previous section, exhibits the so-called "may convergence", i.e. $s \Downarrow$ holds if there is *any* normal-order reduction sequence starting with s and leading to a WHNF. The notion of "must convergence", i.e. that *all* normal-order reduction sequences starting with s lead to a WHNF, also makes sense for a non-deterministic calculus (cf. [20,13,30]). However, for reason of simplicity, the following definition only regards "may convergence".

Definition 3.2 The *contextual approximation* $\lesssim_{\Lambda_{\mathrm{ND}},c}$ is defined by

$$s \lesssim_{\Lambda_{\mathrm{ND}},c} t \iff \forall C:\ C[s]\Downarrow \Longrightarrow C[t]\Downarrow$$

and *contextual equivalence* $\simeq_{\Lambda_{\mathrm{ND}},c}$ by $s \simeq_{\Lambda_{\mathrm{ND}},c} t \iff s \lesssim_{\Lambda_{\mathrm{ND}},c} t \wedge t \lesssim_{\Lambda_{\mathrm{ND}},c} s$.

The goal is to define bisimulation so that it complies with contextual equivalence. The following example makes clear that it is impossible to employ the usual "reduce to weak head normal form and apply to fresh arguments"-approach like e.g. in [2].

Example 3.3 Let the combinators $\mathbf{K} \equiv \lambda x_1.\lambda x_2.x_1$ and $\mathbf{K2} \equiv \lambda y_1.\lambda y_2.y_2$ as well as the non-converging term $\mathbf{\Omega} \equiv (\lambda z.z\,z)\,(\lambda z.z\,z)$ be as usual.

Then $s \equiv \mathtt{let}\ v = \mathtt{pick}\ \mathbf{K}\ \mathbf{K2}\ \mathtt{in}\ \lambda w.v$ and $t \equiv \lambda w.\mathtt{pick}\ \mathbf{K}\ \mathbf{K2}$ could be distinguished by the context $C \equiv \mathtt{let}\ f = [\,]\ \mathtt{in}\ ((f\,\mathbf{K})\,(f\,\mathbf{K})\,\mathbf{\Omega}\,\mathbf{\Omega}\,\mathbf{K})$ in the following way: Concerning t we may construct a normal-order reduction sequence $C[t] \xrightarrow{n}{}^{*} L^{*}_{R}[\mathbf{K}]$ whereas there is no converging normal-order reduction sequence for $C[s]$ since v is shared.

Obviously, the terms s and t are weak head normal forms and if applied to an arbitrary (dummy) argument both may either yield $\mathbf{K}$ or $\mathbf{K2}$. Hence s and t could not be distinguished by application to an argument.

The previous example also reveals that in the $\boldsymbol{\lambda}_{\mathrm{ND}}$-calculus the transformation $\lambda y.\mathtt{let}\ x = s\ \mathtt{in}\ t \rightsquigarrow \mathtt{let}\ x = s\ \mathtt{in}\ \lambda y.t$, i.e. shifting `let` over λ, in general is not correct w.r.t. contextual equivalence. This is so, because the term $\mathtt{let}\ v = \mathtt{pick}\ \mathbf{K}\ \mathbf{K2}\ \mathtt{in}\ \lambda w.v$ becomes $\lambda w.\mathtt{let}\ v = \mathtt{pick}\ \mathbf{K}\ \mathbf{K2}\ \mathtt{in}\ v$ by a reverse application of this transformation. One could simply play through the example with these two terms or, alternatively, argue that the latter is contextual equivalent (cf. [13, rule (ucp)]) to the term t in the example.

The example suggests, that because of the `let`-environments, weak head normal forms do not carry enough information in order to be distinguished solely by application to arguments. There may be several ways to adjust bisimulation so that examples of the above sort work, but it is not clear which one will really produce a suitable definition of bisimulation.

Our approach eliminating the environments has the additional benefit that proving the precongruence candidate stable under the rule (llet) becomes obsolete, a task which seems to be infeasible for the other variations of a definition we have tried.

So before we introduce the special calculus $\boldsymbol{\lambda}_{\approx}$ which eliminates `let`-environments by collecting all possible outcomes, we illustrate by an example that in the $\boldsymbol{\lambda}_{\mathrm{ND}}$-calculus the rule (llet) in general is necessary to find a WHNF.

Example 3.4 Consider the term $s \equiv \texttt{let}\ x = (\texttt{let}\ y = t_y\ \texttt{in}\ \lambda z.t)\ \texttt{in}\ x$ which obviously has a WHNF by the following normal-order reduction:

$$\begin{aligned} & \texttt{let}\ x = (\texttt{let}\ y = t_y\ \texttt{in}\ \lambda z.t)\ \texttt{in}\ x \\ \xrightarrow{n,\ llet} & \texttt{let}\ y = t_y\ \texttt{in}\ (\texttt{let}\ x = \lambda z.t\ \texttt{in}\ x) \\ \xrightarrow{n,\ cp} & \texttt{let}\ y = t_y\ \texttt{in}\ (\texttt{let}\ x = \lambda z.t\ \texttt{in}\ \lambda z.t) \end{aligned}$$

Apparently, the effect of (llet) cannot be accomplished neither by a different scope nor target for the (cp)-rule. Obviously, making a copy of the whole environment $\texttt{let}\ y = t_y\ \texttt{in}\ \lambda z.t$ is in general no option either, since then e.g. for a term of the form $\texttt{let}\ f = (\texttt{let}\ y = \texttt{pick}\ \mathbf{K}\ \mathbf{K2}\ \texttt{in}\ \lambda x.x\,y)\ \texttt{in}\ f\,(f\,\boldsymbol{\Omega})$ this would alter its value w.r.t. contextual approximation.

4 $\boldsymbol{\lambda}_{\approx}$ – Approximating Expressions of the $\boldsymbol{\lambda}_{\mathrm{ND}}$-Calculus

As figure 4 shows, a special constant $\circledcirc$ is added to the language which is now designated by $\Lambda_{\approx}$. The reduction rules of the $\boldsymbol{\lambda}_{\approx}$-calculus in figure 5 evolve

$$E ::= V \ \mid\ \circledcirc \ \mid\ (\lambda x.E) \ \mid\ (E\ E) \ \mid\ (\texttt{let}\ x = E\ \texttt{in}\ E) \ \mid\ (\texttt{pick}\ E\ E)$$

Fig. 4. Syntax for expressions in the language $\Lambda_{\approx}$

from the ones in $\boldsymbol{\lambda}_{\mathrm{ND}}$ as follows. First, by the rule (stop) which may reduce every non-$\circledcirc$ term to $\circledcirc$, a further level of non-determinism is introduced. As there is no rule for $\circledcirc$, this delimits the reduction, i.e. evaluation is pruned underneath. Along with the existing non-determinism of the calculus, we will utilise rule (stop) in order to represent every term by, so to speak, a set of terms which have been evaluated to varying depth.

Since it is our goal to eliminate top-level environments, it is natural to completely copy terms that could not be reduced further, namely $\circledcirc$ and ab-

stractions, and garbage-collect their binding with the rule (cpa) in parallel. So we are able to show in section 6 that the original (cp)-rule becomes obsolete.

Furthermore, all these reductions will be permitted inside arbitrary surface contexts, which are denoted by the symbol $\mathcal{S}$ as before. Hence there is no

$$(\texttt{let}\ x = t_x\ \texttt{in}\ s)\, t \xrightarrow{lapp}_{\boldsymbol{\lambda}\approx} \texttt{let}\ x = t_x\ \texttt{in}\ (s\, t) \tag{lapp}$$

$$(\lambda x.s)\, t \xrightarrow{lbeta}_{\boldsymbol{\lambda}\approx} \texttt{let}\ x = t\ \texttt{in}\ s \tag{lbeta}$$

$$\texttt{pick}\ s\ t \xrightarrow{ndl}_{\boldsymbol{\lambda}\approx} s \tag{ndl}$$

$$\texttt{pick}\ s\ t \xrightarrow{ndr}_{\boldsymbol{\lambda}\approx} t \tag{ndr}$$

$$\texttt{let}\ x = s\ \texttt{in}\ t \xrightarrow{cpa}_{\boldsymbol{\lambda}\approx} t[s/x] \quad \text{where } s \equiv \lambda z.q \text{ or } s \equiv \circledcirc \tag{cpa}$$

$$s \xrightarrow{stop}_{\boldsymbol{\lambda}\approx} \circledcirc \quad \text{if } s \not\equiv \circledcirc \tag{stop}$$

Fig. 5. The reduction rules of the $\boldsymbol{\lambda}\approx$-calculus

need for the rule (llet) either, since we could first reduce inside the binding of a `let`-environment before collapsing it using (cpa). We will give a more detailed account on this process in section 6 where we show that convergence in $\boldsymbol{\lambda}_{\mathrm{ND}}$ and $\boldsymbol{\lambda}\approx$ coincides.

As indicated above, the reductions of the $\boldsymbol{\lambda}\approx$-calculus may take place in surface contexts; hence $\xrightarrow{\mathcal{S},\, a}_{\boldsymbol{\lambda}\approx}$ stands for an application of the rule (a) inside any surface context $S \in \mathcal{S}$. Since it is possible to evaluate up to an arbitrary depth before cutting off with the rule (stop), we call this an *approximation reduction* and will omit the subscript $\boldsymbol{\lambda}\approx$ for the remainder of this section if no confusion arises. The notion of convergence in the $\boldsymbol{\lambda}\approx$-calculus is then defined by $s \Downarrow \lambda x.t \iff s \xrightarrow{\mathcal{S}\,*}_{\boldsymbol{\lambda}\approx} \lambda x.t$, i.e. if there exists an approximation reduction sequence to an abstraction.

4.1 Transformation on Reduction Sequences

In anticipation of bisimulation proofs, it can be shown that applications of the rules (cpa) and (lbeta) never do any harm to an approximation reduction sequence. In case of the former, this is valid only w.r.t. some (stop)-reductions inside arbitrary contexts.

Definition 4.1 A $\xrightarrow{C,\ stop}$-reduction is called *internal*, depicted by $\xrightarrow{i,\ C,\ stop}$, if $C \in \mathcal{C}$ is a context which is not a surface context, i.e. $C \notin \mathcal{S}$.

These internal (stop)-reductions may always be moved to the end of an approximation reduction sequence.

Lemma 4.2 *Let $s, \lambda x.t \in \Lambda_{\approx}$ be terms with $s\ (\xrightarrow{\mathcal{S}}_{\boldsymbol{\lambda}_{\approx}} \cup \xrightarrow{i,\ stop})^*\ \lambda x.t$. Then there is also a reduction $s \xrightarrow{\mathcal{S}}{}^*_{\boldsymbol{\lambda}_{\approx}} \lambda x.t'$ such that $\lambda x.t' \xrightarrow{i,\ stop}{}^*\ \lambda x.t$ holds.*

Internal (stop)-reductions may become necessary to clean up forking situations as follows. Consider the case that an ordinary (stop)-reduction is applied to an abstraction bound to a variable in a `let`-expression. If the rule (cpa) is used afterwards for this `let`-expression, the ◎-terms previously introduced may be found under abstractions. Therefore a simple commutation of the above (stop)- and (cpa)-reductions cannot achieve the same effect.

Hence we can show that for every converging approximation reduction sequence the preference of reductions by rule (cpa) leads to abstractions with some internal (stop)-reductions delayed.

Lemma 4.3 *Let $s, s', \lambda x.t$ be terms so that $s \xrightarrow{\mathcal{S},\ cpa} s'$ and $s \xrightarrow{\mathcal{S}}{}^*_{\boldsymbol{\lambda}_{\approx}} \lambda x.t$ hold. Then s' has an approximation reduction to an abstraction $\lambda x.t'$ which differs from $\lambda x.t$ only by internal (stop)-reductions, i.e. $\lambda x.t' \xrightarrow{i,\ stop}{}^*\ \lambda x.t$ holds.*

For reductions by rule (lbeta), a stronger statement applies. That is to say, if (lbeta) is applicable in a surface context, it does not matter whether a different reduction is performed first.

Lemma 4.4 *Let s, t be terms such that $s \xrightarrow{\mathcal{S},\ lbeta} t$ holds. Then for all abstractions $\lambda z.q$ we have $s \xrightarrow{\mathcal{S}}{}^*_{\boldsymbol{\lambda}_{\approx}} \lambda z.q$ if and only if $t \xrightarrow{\mathcal{S}}{}^*_{\boldsymbol{\lambda}_{\approx}} \lambda z.q$ holds.*

The proofs for lemma 4.3 and 4.4 use the technique of complete sets of forking and, in the case of lemma 4.2, commuting diagrams (cf. [13,30]). Further details can be found in [15, section 2.3.1].

Another essential result consists in reordering converging approximation reduction sequences so that reduction first takes place inside the `let`-bindings.

Theorem 4.5 *For every reduction $\texttt{let}\ x = s\ \texttt{in}\ t \xrightarrow{\mathcal{S}}{}^*_{\boldsymbol{\lambda}_{\approx}} \lambda z.q$ there is also an approximation reduction sequence of the following form:*

$$\texttt{let}\ x = s\ \texttt{in}\ t \xrightarrow{\texttt{let}\ x=\mathcal{S}\ \texttt{in}\ t}{}^*_{\boldsymbol{\lambda}_{\approx}} \texttt{let}\ x = s'\ \texttt{in}\ t \xrightarrow{[\,],\ cpa} t[s'/x] \xrightarrow{\mathcal{S}}{}^*_{\boldsymbol{\lambda}_{\approx}} \lambda z.q$$

where s' represents ◎ or an abstraction.

Proof. Induction on the length of the approximation reduction sequence. □

4.2 Similarity

Owing to the rules (stop) and (cpa), we now have the potential to equip abstractions with the information about their `let`-environments up to an arbitrary depth. This fact will be exploited through non-determinism, i.e. by considering *all* possible approximation reductions to abstractions.

We use the terms “bisimulation” and “bisimilarity” like e.g. [2,25] and therefore define the notion of a *simulation* first.

Definition 4.6 The operation $[\cdot]_{\approx} : \Lambda^0_{\approx} \times \Lambda^0_{\approx} \to \Lambda^0_{\approx} \times \Lambda^0_{\approx}$ over relations on closed terms is defined by

$$s' \; [\eta]_{\approx} \; t' \iff \forall \lambda x.s : \; (s' \Downarrow \lambda x.s \implies \exists \lambda y.t : \; (t' \Downarrow \lambda y.t \wedge \forall r : \; r \in \Lambda^0_{\approx} \implies (\lambda x.s)\, r \; \eta \; (\lambda y.t)\, r))$$

and called an *experiment*. A relation $\eta \subseteq \Lambda^0_{\approx} \times \Lambda^0_{\approx}$ is a *simulation* if $\eta \subseteq [\eta]_{\approx}$.

It is clear that $[\cdot]_{\approx}$ is monotonic, i.e. $\eta_1 \subseteq \eta_2 \implies [\eta_1]_{\approx} \subseteq [\eta_2]_{\approx}$, hence its greatest fixed point exists.

Definition 4.7 Define the *similarity* $\lesssim_b$ to be the greatest fixed point of $[\cdot]_{\approx}$, i.e. $\lesssim_b = \mathrm{gfp}([\cdot]_{\approx})$, and the *bisimilarity* $\sim_b$ by $s \sim_b t \iff s \lesssim_b t \; \wedge \; t \lesssim_b s$.

So two terms s and t are considered *bisimilar* as long as their approximation reduction leads to sets of abstractions such that there are elements from each set which are bisimilar if applied to arbitrary arguments. The next example underpins that this is exactly what we need to obtain the same capability in distinguishing terms as with contexts.

Example 4.8 As is known, the two terms $s \equiv \texttt{let}\ v = \texttt{pick}\ \mathbf{K}\ \mathbf{K2}\ \texttt{in}\ \lambda w.v$ and $t \equiv \lambda w.\texttt{pick}\ \mathbf{K}\ \mathbf{K2}$ of example 3.3 could be distinguished by contexts.

Now we can show that $t \not\lesssim_b s$ does not hold either. Since t already is an abstraction, we therefore consider all possible approximation reduction sequences for s that lead to an abstraction:

$$s \xrightarrow{\texttt{let}\ v=[\,]\ \texttt{in}\ \lambda w.v,\ ndl} \texttt{let}\ v = \mathbf{K}\ \texttt{in}\ \lambda w.v \xrightarrow{[\,],\ cpa} \lambda w.\mathbf{K}$$

$$s \xrightarrow{\texttt{let}\ v=[\,]\ \texttt{in}\ \lambda w.v,\ ndr} \texttt{let}\ v = \mathbf{K2}\ \texttt{in}\ \lambda w.v \xrightarrow{[\,],\ cpa} \lambda w.\mathbf{K2}$$

Since the non-deterministic choice has been fixed, neither of these abstractions exposes the necessary behaviour. Particularly, t may converge when applied to the argument sequences $\mathbf{\Omega}, \mathbf{\Omega}, \mathbf{K}$ and $\mathbf{\Omega}, \mathbf{K}, \mathbf{\Omega}$, while $\lambda w.\mathbf{K}$ does not converge for the former, nor does $\lambda w.\mathbf{K2}$ for the latter.

What follows is an example of a proof which is straightforward for similarity but seems rather involved using the definition of contextual approximation.

Example 4.9 Let $r, s, t \in \Lambda^0_{\approx}$ be arbitrary closed terms. Then we have

$$r \; \lesssim_b \; t \; \wedge \; s \; \lesssim_b \; t \implies \texttt{pick}\, r\, s \; \lesssim_b \; t$$

i.e. if t behaves “better” than both r and s, then it is immaterial which one is chosen thereof. So assume $\texttt{pick}\, r\, s \Downarrow \lambda y.p$, then $r \Downarrow \lambda y.p$ or $s \Downarrow \lambda y.p$. Since by the premise we have $r \; \lesssim_b \; t$ and $s \; \lesssim_b \; t$, the proposition is shown.

We will now extend similarity to open terms. The motivation for doing so is twofold. First, the notion of a congruence is less meaningful when dealing with closed terms. E.g., inferring $\lambda x.s \sim_b \lambda x.t$ from $s \sim_b t$ for closed s and t does not gain much, since x is only a dummy variable. Secondly, the proof method interacts closely with the extension of $\lesssim_b$ to open terms anyway.

So we have to bear in mind which terms may be copied in the $\boldsymbol{\lambda}_{\approx}$-calculus. Since this is the case for $\circledcirc$ and abstractions only, the technique to use all closing substitutions is not applicable, as the following example substantiates.

Example 4.10 Consider the open terms $f\,f$ and $\texttt{let}\ x = f\ \texttt{in}\ x\,x$ which are contextual equivalent in the $\boldsymbol{\lambda}_{\mathrm{ND}}$-calculus since copying variables is permitted (cf. correctness of rule (lcv) in [13]).

But demanding the terms to be bisimilar for every closing substitution is not possible: $(f\,f)[\texttt{pick}\ \mathbf{K}\ \mathbf{K2}/f]$ may yield $\mathbf{K}\,\mathbf{K2}$ which, along the lines of example 3.3, converges if successively applied to the arguments $\boldsymbol{\Omega}$, $\boldsymbol{\Omega}$ and $\mathbf{K}$, whereas $(\texttt{let}\ x = f\ \texttt{in}\ x\,x)[\texttt{pick}\ \mathbf{K}\ \mathbf{K2}/f]$ clearly does not.

Hence what we need is a restriction of the substitutions such that free variables are mapped only to $\circledcirc$ or closed abstractions.

Definition 4.11 Let $s, t \in \Lambda_{\approx}$ be (possibly open) terms. We then write $s \lesssim_b^o t$ if and only if $\sigma(s) \lesssim_b \sigma(t)$ holds for all closing substitutions σ whose range $\boldsymbol{rng}(\sigma)$ satisfies $\boldsymbol{rng}(\sigma) \subseteq \{\, p \in \Lambda^0_{\approx} \mid p \equiv \circledcirc \vee p \equiv \lambda z.q \,\}$.

In [15, section 4.7], we show that an equivalent notion may be defined by considering all closing $\texttt{let}$-environments.

4.3 The Precongruence Candidate

In this section, let τ stand for any operator of the $\Lambda_{\approx}$-language (i.e. $\circledcirc$, λ, $\texttt{let}$, $\texttt{pick}$ or application) and $\overline{a}_i$ for a sequence of its operands. With $\overline{a}_i\ \eta\ \overline{b}_i$ we denote the condition that $a_i\ \eta\ b_i$ for every i holds. A relation $\eta \subseteq \Lambda_{\approx} \times \Lambda_{\approx}$ is then called *operator-respecting*, or *compatible*, if and only if $\overline{a}_i\ \eta\ \overline{b}_i$ implies $\tau(\overline{a}_i)\ \eta\ \tau(\overline{b}_i)$ for all operators. A *precongruence* is a compatible preorder.

The following defines a relation which is compatible by definition but not necessarily transitive. The intention is to show that it coincides with $\lesssim_b^o$ for which the criteria will be developed in this section.

Definition 4.12 Let $\eta \subseteq \Lambda^0_{\approx} \times \Lambda^0_{\approx}$ be a preorder. Then define its *precongruence candidate* $\widehat{\lesssim_b} \subseteq \Lambda_{\approx} \times \Lambda_{\approx}$ by

- $x\ \widehat{\lesssim_b}\ b$ if $x \in V$ is a variable and $x \lesssim_b^o b$.
- $\tau(\overline{a}_i)\ \widehat{\lesssim_b}\ b$ if there exists $\overline{a}'_i$ such that $\overline{a}_i\ \widehat{\lesssim_b}\ \overline{a}'_i$ and $\tau(\overline{a}'_i) \lesssim_b^o b$.

Howe [11, p. 201] aptly gives the informal account that *a* $\widehat{\lesssim_b}$ *b if b can be obtained from a via one bottom-up pass of replacements of subterms by*

terms that are larger under $\lesssim_b{}^o$. As noted before, in the $\boldsymbol{\lambda}_{\approx}$-calculus only $\circledcirc$ and abstractions may be copied. Hence the following two lemmata reflect our counterpart to [11, Lemma 1] and [12, Lemma 3.2] respectively.

Lemma 4.13 *Let* $b, b' \in \Lambda_{\approx}$ *be terms. Then* $b \mathrel{\widehat{\lesssim_b}} b'$ *implies* $b[\circledcirc/x] \mathrel{\widehat{\lesssim_b}} b'[\circledcirc/x]$.

Lemma 4.14 *For all* $b, b' \in \Lambda_{\approx}$ *and closed abstractions* $\lambda z.r, \lambda z.r' \in \Lambda^0_{\approx}$ *the following holds:* $b \mathrel{\widehat{\lesssim_b}} b' \ \wedge \ \lambda z.r \mathrel{\widehat{\lesssim_b}} \lambda z.r'$ *implies* $b[\lambda z.r/x] \mathrel{\widehat{\lesssim_b}} b'[\lambda z.r'/x]$.

Both lemmata are proven by induction on the definition of the precongruence candidate in which we take advantage of how $\lesssim_b{}^o$ is defined.

Let in the following η_0 stand for the *restriction* of a preorder $\eta \subseteq \Lambda_{\approx} \times \Lambda_{\approx}$ to closed terms, i.e. $\eta_0 = \eta \cap \Lambda^0_{\approx} \times \Lambda^0_{\approx}$, then from the above we can show:

Theorem 4.15 *The relation* $(\widehat{\lesssim_b})_0 \subseteq \lesssim_b$ *holds, if and only if* $\widehat{\lesssim_b} \subseteq \lesssim_b{}^o$, *if and only if* $\lesssim_b{}^o$ *is a precongruence.*

We will establish the first set inclusion $(\widehat{\lesssim_b})_0 \subseteq \lesssim_b$ which, by co-induction, follows from $(\widehat{\lesssim_b})_0 \subseteq [(\widehat{\lesssim_b})_0]_{\approx}$, since $\lesssim_b$ is the greatest fixed point of $[\cdot]_{\approx}$ and contains all simulations. To use induction on the length of converging approximation reductions sequences, we therefore have to show that $(\widehat{\lesssim_b})_0$ is preserved under every single-step reduction.

5 Proving $\lesssim_b{}^o$ a precongruence

Preservation of $(\widehat{\lesssim_b})_0$ under every single-step reduction amounts to the condition that $s \ (\widehat{\lesssim_b})_0 \ t \ \wedge \ s \xrightarrow{\mathcal{S}}_{\boldsymbol{\lambda}_{\approx}} s'$ implies $s' \ (\widehat{\lesssim_b})_0 \ t$ in which the terms s, s', t all are closed. Note that the only *closing surface contexts*, i.e. those with which open terms can be closed, involve a L_R-context somewhere.

It can be shown that for every converging approximation reduction there is also an approximation reduction sequence to the same abstraction, where no approximation reduction takes place in a closing surface context. Hence in the following, it is sufficient to examine top-level reductions on closed terms.

Lemma 5.1 *Let* $r, s \in \Lambda^0_{\approx}$ *be closed terms such that* $r \xrightarrow{[\,],\,a}_{\boldsymbol{\lambda}_{\approx}} s$ *holds. Then for every closed term* $t \in \Lambda^0_{\approx}$ *we have:* $r \ (\widehat{\lesssim_b})_0 \ t$ *implies* $s \ (\widehat{\lesssim_b})_0 \ t$.

Proof. In the case of $a \in \{ndl, ndr, stop\}$ the claim is obvious. The remaining reduction rules are by induction on the definition of the precongruence candidate, where for (lapp) and (lbeta) compatibility of $\lesssim_b$ w.r.t. contexts of the form $([\,]\,e)$ is applied.

The proof of the rule (cpa) is done distinguishing the cases for $\circledcirc$ and abstractions, while exploiting lemma 4.13 and 4.14, respectively. □

It is remarkable, that by virtue of Howe's method we achieve a modular proof. Because it could be done for each of the rules separately, it is easily extensible if new reduction rules are added to the calculus.

Furthermore, using (cpa) instead of (cp) greatly simplifies matters, since by the integrated garbage collection there is no need to keep track of the copied term at its target location w.r.t to its binding in the `let`-environment.

Theorem 5.2 *The similarity $\lesssim_b^o$ is a precongruence.*

Proof. With lemma 5.1, the premises for theorem 4.15 are satisfied. □

Since the language $\Lambda_\approx$ contains strictly more contexts than Λ_{ND} due to the constant $\circledcirc$, we define contextual approximation for $\boldsymbol{\lambda}_\approx$ separately. We will show in section 6 that these contexts do not add any computational power.

Definition 5.3 The *contextual approximation* $\lesssim_{\Lambda_\approx,c}$ for $\boldsymbol{\lambda}_\approx$ is defined by

$$s \lesssim_{\Lambda_\approx,c} t \iff \forall C : C[s]\Downarrow \implies C[t]\Downarrow$$

and *contextual equivalence* $\simeq_{\Lambda_\approx,c}$ by $s \simeq_{\Lambda_\approx,c} t \iff s \lesssim_{\Lambda_\approx,c} t \ \wedge\ t \lesssim_{\Lambda_\approx,c} s$.

By virtue of theorem 5.2 it becomes nearly straightforward to show that similarity $\lesssim_b^o$ coincides with the contextual approximation in the $\boldsymbol{\lambda}_\approx$-calculus.

Theorem 5.4 *Let $s, t \in \Lambda_\approx$ be terms. Then $s \lesssim_b^o t$ iff $s \lesssim_{\Lambda_\approx,c} t$ holds.*

6 Correspondence of Equality in $\boldsymbol{\lambda}_{\mathrm{ND}}$ and $\boldsymbol{\lambda}_\approx$

Since our aims were a method to prove the contextual equivalence $\simeq_{\Lambda_{\mathrm{ND}},c}$ in the $\boldsymbol{\lambda}_{\mathrm{ND}}$-calculus, we have an obligation to show that this is indeed the same as the contextual equivalence $\simeq_{\Lambda_\approx,c}$ in the $\boldsymbol{\lambda}_\approx$-calculus. Hence, we first recall how the contextual approximation is defined:

$$s \lesssim_c t \iff (\forall C : C[s]\Downarrow \implies C[t]\Downarrow)$$

It is quite evident that the correspondence of $\lesssim_{\Lambda_{\mathrm{ND}},c}$ and $\lesssim_{\Lambda_\approx,c}$ requires the following property: For every term $t \in \Lambda_\approx$ there is a normal-order reduction to a weak head normal form if and only if t has an approximation reduction to an abstraction. We therefore understand the notion of normal-order reduction in $\boldsymbol{\lambda}_{\mathrm{ND}}$ as extended to terms from $\Lambda_\approx$ in the obvious way, i.e. regarding $\circledcirc$ as a constant which has no normal-order reduction.

Moreover, it can be shown that for every $\Lambda_\approx$-context C which distinguishes two terms $s, t \in \Lambda_{\mathrm{ND}}$ there is also a Λ_{ND}-context C' such that $C'[s]$ converges but $C'[t]$ does not, or vice versa. For this purpose we may obtain C' from C by simply replacing all occurrences of $\circledcirc$ with $\boldsymbol{\Omega}$, whose normal-order reduction does not terminate.

Thus, the contexts of Λ_{ND} and $\Lambda_\approx$ are equally powerful in distinguishing terms and we also obtain a sufficient condition from the above mentioned property. I.e., we may confine attention to the transformation of converging reduction sequences between the two calculi in the following.

6.1 *Transforming* $\xrightarrow{\mathcal{S}}_{\boldsymbol{\lambda}_{\approx}}$- *into* $\xrightarrow{n}_{\boldsymbol{\lambda}_{\mathrm{ND}}}$-*reduction sequences*

The process of constructing a normal-order reduction to a WHNF is by induction on the length of a converging approximation reduction. Since (cpa) and (nd) are the only approximation reductions to reach an abstraction within a single step, the induction base should be clear.

For the induction step it is then to show that every approximation reduction may be moved to the end of a normal-order reduction sequence. For approximation reductions which are performed inside surface contexts that are not reduction contexts, this is an obvious task, since the corresponding contexts are disjoint. So it turns out that only reductions by rule (cpa) inside reduction contexts are of particular interest. For these, in [15, section 5.1] a complete set of commuting diagrams w.r.t. normal-order reductions is established. These diagrams do not duplicate $\xrightarrow{\mathcal{R},\ cpa}$-reductions and hence could be composed by induction which leads to the following result.

Lemma 6.1 *Let $r, \lambda x.s \in \Lambda_{\approx}$ be terms such that $r \xrightarrow{\mathcal{S}}{}^{*}_{\boldsymbol{\lambda}_{\approx}} \lambda x.s$ holds. Then there is also a normal order reduction $r \xrightarrow{n}{}^{*}_{\boldsymbol{\lambda}_{\mathrm{ND}}} t$ where t is a WHNF.*

Proof. Using the arguments discussed above for an induction on the length of an approximation reduction sequence to an abstraction. □

6.2 *Transforming* $\xrightarrow{n}_{\boldsymbol{\lambda}_{\mathrm{ND}}}$- *into* $\xrightarrow{\mathcal{S}}_{\boldsymbol{\lambda}_{\approx}}$-*reduction sequences*

Since every reduction context is also a surface context, the only normal-order reductions which are no approximation reductions are those by the rules (cp) and (llet). Since with (cpa) the former has a counterpart in the $\boldsymbol{\lambda}_{\approx}$-calculus, its treatment is not difficult.

But in order to make the latter superfluous, the reduction strategy has to be adapted so that for a term like $\mathtt{let}\ x = (\mathtt{let}\ y = t_y\ \mathtt{in}\ t_x)\ \mathtt{in}\ R[x]$ the approximation reduction first proceeds inside $\mathtt{let}\ y = t_y\ \mathtt{in}\ t_x$ until $\circledcirc$ or an abstraction is reached, which could be copied into $R[x]$ using (cpa) then. Because of theorem 4.5, this procedure is always possible and may also be applied recursively to the subterm $\mathtt{let}\ y = t_y\ \mathtt{in}\ t_x$ of the above scenario.

Lemma 6.2 *Let $r, s \in \Lambda_{\approx}$ be terms such that s is a WHNF and $r \xrightarrow{n}{}^{*}_{\boldsymbol{\lambda}_{\mathrm{ND}}} s$ holds. Then there is also an approximation reduction $r \xrightarrow{\mathcal{S}}{}^{*}_{\boldsymbol{\lambda}_{\approx}} \lambda x.t$ to some abstraction.*

Proof. By induction on the length of a normal-order reduction sequence. □

Putting all these parts together we achieve the correspondence of similarity with contextual approximation, both in the $\boldsymbol{\lambda}_{\mathrm{ND}}$- and the $\boldsymbol{\lambda}_{\approx}$-calculus.

Theorem 6.3 *Let $s, t \in \Lambda_{\approx}$ be arbitrary $\boldsymbol{\lambda}_{\approx}$-terms. Then $s \lesssim_b t$ holds, if and only if $s \lesssim_{\Lambda_{\approx},c} t$, if and only if $s \lesssim_{\Lambda_{\mathrm{ND}},c} t$ is valid.*

Our main objective of proving contextual equivalences in the $\boldsymbol{\lambda}_{\mathrm{ND}}$-calculus now becomes a simple consequence.

Corollary 6.4 *For all terms $s, t \in \Lambda_{\mathrm{ND}}$ we have $s \sim_b^o t$ iff $s \simeq_{\Lambda_{\mathrm{ND}},c} t$ holds.*

7 Conclusion and Future Work

To the best of our knowledge, for the first time a sensible bisimulation has been defined for a non-deterministic call-by-need calculus and shown to be equivalent to contextual equivalence. The proof that bisimulation is a congruence extended Howe's method, where two points emerged to be of significance.

First, we have seen that testing terms by just reducing them to weak head normal form and applying these WHNF's to arbitrary arguments is not appropriate. Instead, the terms to be tested have rather be equipped with all the information about which choices have to be shared and which may be copied. We accomplished this by performing evaluation inside surface contexts up to every arbitrary depth, in which also choices in `let`-environments may be forced. Since we non-deterministically collect all these possible outcomes, we therefore have enough potential to discriminate terms.

The other aspect concerns the kind of terms that may be copied. As we have seen, the precongruence candidate or, strictly speaking, the extension of the bisimilarity to open terms had to be adapted such that only $\circledcirc$ and abstractions are considered. This might point out a general way for the proof of the fundamental substitution lemma to go through, i.e. for [11, Lemma 1] and [12, Lemma 3.2] respectively, or lemma 4.13 and 4.14 in our case.

On the basis of these explanations, we feel confident that the technique demonstrated in this paper is powerful enough for the treatment of a language extending the $\boldsymbol{\lambda}_{\mathrm{ND}}$-calculus with a `case` and data constructors. It could also be worth to apply the results of this paper to the design and development of generic and purely syntactic systems of structural operational semantics, e.g. like the structured evaluation systems of [12] but suited for non-determinism combined with sharing.

As remarked earlier, the contextual equivalence does not regard must convergence nor, on a par with it, divergence. Like the work of [20,13,30] suggests, it is quite reasonable in a non-deterministic calculus to regard possibly infinite reduction sequences. Hence as a further extension of the $\boldsymbol{\lambda}_{\mathrm{ND}}$-calculus, also divergent behaviour might be incorporated. So, writing $s \Uparrow$ if s has a non-terminating normal-order reduction, a possible — and sensible — definition of the contextual equivalence might be given by

$$s \simeq_c t \iff ((\forall C : C[s]\Downarrow \iff C[t]\Downarrow) \wedge (\forall C : C[s]\Uparrow \iff C[t]\Uparrow))$$

Using contextual approximation, the above contextual equivalence may be

established in several ways. It may seem appealing to adopt a definition like

$$s \lesssim_c t \iff (\forall C : (C[s]\Downarrow \implies C[t]\Downarrow) \wedge (C[t]\Uparrow \implies C[s]\Uparrow))$$

from [13] for the contextual approximation. But for our method, this will pose technical difficulties in showing that similarity equals contextual approximation. This is, because then e.g. $\mathbf{K} \lesssim_c \texttt{pick}\ \mathbf{\Omega}\ \mathbf{K}$ will not hold anymore and therefore $s \xrightarrow{\mathcal{S}}_{\boldsymbol{\lambda}\approx} t \implies t \lesssim_b s$ neither. We preferably would like to retain this property, since it has turned out to be extremely helpful in the proof. It appears to us that, by the duality of convergence and divergence, it is feasible to define a separate "approximation" relation for divergence. For that relation a method similar to the one presented in this paper seems possible.

There is another aspect concerning the omission of divergence: As we have indicated before, there are equalities in the $\boldsymbol{\lambda}_{\text{ND}}$-calculus which are not true in a calculus regarding divergence, e.g. [13]. These include the following equivalences, where $\bot$ stands for an arbitrary term which does not have a weak head normal form:

$$\texttt{pick}\ s\ \bot \simeq_{\Lambda_{\text{ND}},c} s$$
$$\texttt{pick}\ \bot\ t \simeq_{\Lambda_{\text{ND}},c} t$$

So in the $\boldsymbol{\lambda}_{\text{ND}}$-calculus the operator `pick` behaves *bottom-avoiding* which suggests that it could be worthwhile to apply our results to this kind of calculi.

Further enhancements may be devoted to making bisimulation proofs easier to handle. Since the approximation reduction in the $\boldsymbol{\lambda}_{\approx}$-calculus is highly non-deterministic, a direct definition of bisimulation in $\boldsymbol{\lambda}_{\text{ND}}$ is desirable, which provides more information on how to proceed comparing two terms.

Moreover, because sharing does not change the termination behaviour of terms in a deterministic setting, an application of our results to the improvement theory of [19], where terms could be distinguished if they differ in the number of reductions necessary to reach a weak head normal form, may be of interest for future research.

8 Acknowledgements

I would like to express my gratitude to Manfred Schmidt-Schauß and David Sabel for all their valuable comments and constructive criticism. I am particularly indebted to Manfred Schmidt-Schauß for scrutinising several proofs and having directed my attention to non-deterministic call-by-need $\boldsymbol{\lambda}$-calculi.

References

[1] Abadi, M., L. Cardelli, P.-L. Curien and J.-J. Lévy, *Explicit substitutions*, Journal of Functional Programming **1** (1991), pp. 375–416.

[2] Abramsky, S., *The lazy lambda calculus*, in: D. A. Turner, editor, *Research Topics in Functional Programming*, University of Texas at Austin Year of Programming Series, Addison-Wesley, 1990 pp. 65–116.

[3] Ariola, Z. M. and M. Felleisen, *The call-by-need lambda calculus*, Journal of Functional Programming **7** (1997), pp. 265–301.

[4] Ariola, Z. M., J. Maraist, M. Odersky, M. Felleisen and P. Wadler, *A call-by-need lambda calculus*, in: *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (1995), pp. 233–246.

[5] Barendregt, H. P., "The Lambda Calculus, Its Syntax and Semantics," Elsevier Science Publishers, 1984.

[6] Boudol, G., *Lambda-calculi for (strict) parallel functions*, Information and Computation **108** (1994), pp. 51–127.

[7] de'Liguoro, U. and A. Piperno, *Nondeterministic extensions of untyped λ-calculus*, Information and Computation **122** (1995), pp. 149–177.

[8] Gordon, A. D., "Functional programming and input/output," Distinguished Dissertations in Computer Science, Cambridge University Press, 1994.

[9] Gordon, A. D., *Bisimilarity as a theory of functional programming*, Theoretical Computer Science **228** (1999), pp. 5–47.

[10] Groote, J. F. and F. Vaandrager, *Structured operational semantics and bisimulation as a congruence*, Information and Computation **100** (1992).

[11] Howe, D. J., *Equality in lazy computation systems*, in: *Proceedings, Fourth Annual Symposium on Logic in Computer Science*, 1989, pp. 198–203.

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

[13] Kutzner, A., "Ein nichtdeterministischer call-by-need Lambda-Kalkül mit erratic Choice: Operationale Semantik, Programmtransformationen und Anwendungen," Ph.D. thesis, Johann Wolfgang Goethe-Universität, Frankfurt am Main (1999).

[14] Kutzner, A. and M. Schmidt-Schauß, *A nondeterministic call-by-need lambda calculus*, in: *International Conference on Functional Programming 1998* (1998), pp. 324–335.

[15] Mann, M., *Towards Sharing in Lazy Computation Systems*, Frank report 18, Institut für Informatik, J.W. Goethe-Universität Frankfurt am Main (2004), http://www.ki.informatik.uni-frankfurt.de/papers.

[16] Maraist, J., M. Odersky and P. Wadler, *The call-by-need lambda calculus*, Journal of Functional Programming **8** (1998).

[17] Milner, R., *An algebraic definition of simulation between programs*, in: D. C. Cooper, editor, *Proceedings of the 2nd International Joint Conference on Artificial Intelligence* (1971), pp. 481–489.

[18] Milner, R., *Fully Abstract Models of Typed lambda-Calculi*, Theoretical Computer Science **4** (1977), pp. 1–22.

[19] Moran, A. K. and D. Sands, *Improvement in a lazy context: An operational theory for call-by-need*, in: *Proc. POPL'99, the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (1999), pp. 43–56.

[20] Moran, A. K., D. Sands and M. Carlsson, *Erratic Fudgets: A semantic theory for an embedded coordination language*, in: *the Third International Conference on Coordination Languages and Models; COODINATION'99*, number 1594 in Lecture Notes in Computer Science, Springer-Verlag, 1999 pp. 85–102, extended available: [21].

[21] Moran, A. K., D. Sands and M. Carlsson, *Erratic Fudgets: A semantic theory for an embedded coordination language (extended version)* (1999).

[22] Morris, J., "Lambda-Calculus Models of Programming Languages," Ph.D. thesis, MIT (1968).

[23] Ong, C.-H. L., *Non-determinism in a functional setting*, in: *Logic in Computer Science*, 1993, pp. 275–286.

[24] Park, D., *Concurrency and automata on infinite sequences*, in: P. Deussen, editor, *Theoretical Computer Science*, Lecture Notes in Computer Science **104** (1981), pp. 167–183.

[25] Pitts, A. M., *A note on logical relations between semantics and syntax*, Logic Journal of the IGPL **5** (1997), pp. 589–601.

[26] Sands, D., *Operational theories of improvement in functional languages (extended abstract)*, in: *Proceedings of the Fourth Glasgow Workshop on Functional Programming*, Workshops in Computing Series (1991), pp. 298–311.

[27] Sands, D., *From SOS rules to proof principles*, Technical report, Chalmers University of Technology and Göteborg University (1997).

[28] Sands, D., J. Gustavsson and A. Moran, *Lambda calculi and linear speedups*, in: T. Æ. Mogensen, D. Schmidt and I. H. Sudborough, editors, *The essence of computation: complexity, analysis, transformation*, number 2566 in Lecture Notes in Computer Science, Springer-Verlag New York, Inc., 2002 pp. 60–82.

[29] Sangiorgi, D., *The lazy lambda calculus in a concurrency scenario*, Information and Computation **111** (1994), pp. 120–153.

[30] Schmidt-Schauß, M., *FUNDIO: A Lambda-Calculus with a* `letrec`, `case`, *Constructors, and an IO-Interface: Approaching a Theory of* `unsafePerformIO`, Frank report 16, Institut für Informatik, J.W. Goethe-Universität Frankfurt am Main (2003).

[31] Søndergard, H. and P. Sestoft, *Non-determinism in Functional Languages*, The Computer Journal **35** (1992), pp. 514–523.

A Deterministic Logical Semantics for Esterel

Olivier Tardieu [1]

INRIA
Sophia Antipolis, France

Abstract

Esterel is a synchronous design language for the specification of reactive systems. There exist two main semantics for Esterel. On the one hand, the logical behavioral semantics provides a simple and compact formalization of the behavior of programs using SOS rules. But it does not ensure deterministic executions for all programs and all inputs. As non-deterministic programs have to be rejected as incorrect, this means it defines behaviors for incorrect programs, which is not convenient. On the other hand, the constructive semantics is deterministic (amongst other properties) but at the expense of a much more complex formalism. In this work, we construct and thoroughly analyze a new deterministic semantics for Esterel that retains the simplicity of the logical behavioral semantics, from which it derives. In our view, it provides a much better framework for formal reasoning about Esterel programs.

Key words: synchronous languages, concurrency theory, structural operational semantics.

1 Introduction

Esterel [7,8] is a high-level imperative parallel programming language for the specification of reactive systems [9,13]. It was born in the eighties [6], and evolved since then. In this work, we consider the Esterel v5 dialect [4,5] endorsed by current academic compilers [1,10]. Pure Esterel is the subset of the full Esterel language where data variables and data-handling primitives are abstracted away. As the issues we are interested in in this work are not related to data in any way, we shall concentrate on the pure Esterel language.

Esterel is a synchronous language [2]. Primitives constructs execute in zero time except for one `pause` instruction. Hence, time flows as a sequence of logical instants separated by explicit pauses. In each instant, several elementary instantaneous computations take place simultaneously.

Esterel deals with signals. Signals have a Boolean status, which obeys the

[1] Email: olivier.tardieu@sophia.inria.fr

This is a preliminary version. The final version will be published in Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

signal coherence law: in each instant, a signal is absent by default, present if emitted in this instant. In "`present A then emit B end`" for instance, `B` is emitted, thus present, if `A` is present.

Both absence and presence are instantly broadcast, and simultaneously available to all threads of execution. This perfect synchrony hypothesis may result in causality cycles [4,14], as for example in the parallel composition:

```
present A then emit B end || present B then emit A end
```

which admits two possible executions conforming to the signal coherence law:

- both `A` and `B` are present and emitted;
- both `A` and `B` are absent and not emitted.

This program is said to be non-deterministic. Similarly, there exist non-reactive programs with no possible execution, for example:

```
present A then emit B end || present B else emit A end
```

In Esterel, we want programs to have deadlock-free deterministic executions. Therefore, non-reactive and non-deterministic programs have to be rejected as incorrect. Two main semantics have been formalized for Esterel:

- The logical behavioral semantics [3] simply formalizes the signal coherence law. It defines no execution for a non-reactive program, and several distinct executions for a non-deterministic program [2].
- The constructive semantics [4] is inspired from digital circuits and three-valued logic. It only defines a subset of the executions defined by the logical behavioral semantics. By rejecting more "unreasonable" programs than just non-reactive and non-deterministic programs, it ensures that executions can be "causally" computed. As a result, it defines no execution for non-reactive as well as non-deterministic programs.

These two semantics handle non-determinism in opposite manners. Neither is truly convenient.

- On the one hand, an execution defined by the logical behavioral semantics is not necessarily correct, as it may be the execution of a non-deterministic, thus incorrect program. Moreover, non-determinism sometimes compensates for non-reactivity making a program reactive and deterministic although it contains non-reactive or non-deterministic pieces of code.
- On the other hand, the constructive semantics only defines correct executions, but at the expense of a much more complex formalism.

Therefore, we introduce in this work a third alternative semantics that we derive from the logical behavioral semantics. It retains the simple formalism of the logical behavioral semantics, while only defining correct executions. In particular, it makes sure errors do not cancel one another.

[2] In general, determinism and reactivity depend on inputs (cf. Section 4).

The paper is organized as the following. In Section 2, we describe the pure Esterel language. We formalize its logical behavioral semantics in Section 3, and discuss reactivity and determinism in Section 4. We build our deterministic semantics in Section 5. In Section 6, we thoroughly compare the two semantics. We briefly discuss the constructive semantics of Esterel in Section 7, and conclude in Section 8.

2 Syntax and Intuitive Semantics

$p, q ::=$	`nothing`	does nothing, terminates instantly
	`pause`	stops the execution till next instant
	p `;` q	runs p, then q if/when p terminates
	p `||` q	runs p in parallel with q
	`loop` p `end`	repeats p forever
	`signal` S `in` p `end`	declares signal S in p
	`emit` S	emits signal S
	`present` S `then` p `else` q `end`	runs p if S is present, q otherwise
	`trap` T `in` p `end`	declares, catches exception T in p
	`exit` T_d	raises exception T of depth d

Fig. 1. Primitive Pure Esterel Constructs

Without loss of generality, we focus in this work on a kernel language inspired from Berry [4], which retains just enough of the pure Esterel language to attain its full expressive power. Figure 1 describes the grammar of our kernel language, as well as the intuitive behavior of its constructs.

The non-terminals p and q denote *statements* (i.e. *programs*), S *signals* and T *exceptions*. Signals and exceptions are identifiers lexically scoped and respectively declared within statements by the constructs "`signal` S `in` p `end`" and "`trap` T `in` p `end`".

The infix "`;`" operator binds tighter than "`||`". Brackets "`[`" and "`]`" may be used to group statements in arbitrary ways. In a `present` statement, `then` or `else` branches may be omitted. For example, "`present` S `then` p `end`" is a shortcut for "`present` S `then` p `else nothing end`".

2.1 Instants and Reactions

An Esterel statement runs in steps called *reactions* in response to the *ticks* of a *global clock*. Each reaction takes one *instant*. Primitive constructs execute in zero time except for the `pause` instruction. When the clock ticks, a reaction occurs, which computes the *output signals* and the *new state* of the program, from the *input signals* and the *current state* of the program. It may either finish the execution instantly or delay part of it till the next instant, because it reached at least one `pause` instruction. In the latter case, the execution is resumed when the clock ticks again from the locations of the `pause` instructions

reached in the previous instant. And so on.

"`emit A; pause; emit B; emit C; pause; emit D`" emits the signal `A` in the first instant of its execution, then emits `B` and `C` in the second instant, finally emits `D` and terminates in the third instant. It takes three instants to complete, that is to say proceeds by three reactions. The signals `B` and `C` are emitted *simultaneously*, as their emissions occur in the same instant of execution. In particular, "`emit B; emit C`" and "`emit C; emit B`" cannot be distinguished in Esterel.

2.2 Synchronous Concurrency

Concurrency in Esterel is synchronous. One reaction of the parallel composition "p `||` q" is made of exactly one reaction of each non-terminated branch, until the termination of all branches. For example,

```
[
  pause; emit A; pause; emit B
||
  emit C; pause; emit D
];
emit E
```

emits `C` in the first instant of its execution, then emits `A` and `D` in the second instant, then emits `B` and `E` and terminates in the third instant.

2.3 Exceptions

Exceptions are lexically scoped, declared and caught by the "`trap` T `in` p `end`" construct, raised by the "`exit` T_d" instruction. The integer d encodes the *depth* of "`exit` T":

- if "`exit` T_d" is enclosed in a declaration of T then d *must* be the number of exception declarations that have to be traversed before reaching that of T;
- if "`exit` T_d" is not enclosed in a declaration of T then d *must* be greater or equal to the number of exception declarations enclosing this `exit` statement.

For example,

```
trap T in
  trap U in
    exit T1   has depth 1 because of the declaration of U
  ||
    exit U0   has depth 0
  ||
    exit V3   could have any depth greater or equal to 2
  end;
  exit T0     has depth 0
end
```

Such a "De Bruijn" encoding of exceptions for Esterel was first advocated for by Gonthier [11]. As usual, we shall only make depths explicit when necessary.

In sequential code, the `exit` statement behaves as a "goto" to the end of the matching `trap` block. For example,

```
trap T in
  emit A; pause; emit B; exit T; emit C
end;
emit D
```

emits `A` in the first instant, then `B` and `D` and terminates in the second instant. Signal `C` is never emitted.

An exception raised in a parallel context causes all parallel branches to terminate instantly. In the example below, `A` and `E` are emitted in the first instant, then `B`, `F`, and `D` in the second and final one. Neither `C` nor `G` is emitted.

```
trap T in
  emit A; pause; emit B; exit T; emit C
||
  emit E; pause; emit F; pause; emit G
end;
emit D
```

Remark exceptions implement *weak preemption*: "`exit T`" in the first branch does not prevent `F` to be simultaneously emitted in the second one.

Exception declarations may be nested. In the following example, `A` is not emitted, as the outermost exception `T` has priority over inner ones, `U` here.

```
trap T in
  trap U in
    exit T1 || exit U0
  end;
  emit A
end
```

In other words, the exception of greater depth has always priority.

2.4 Loops

The statement "`loop emit S; pause end`" emits `S` at each instant and never terminates. Finitely iterated loops may be obtained by combining `loop`, `trap` and `exit` statements, as for instance in the kernel expansions of "`await` S":

`trap T in loop pause; present` S `then exit T end end end`

Loop bodies may not be *instantaneous* [17]. For example, "`loop emit S end`" is not a correct program. Such a pattern would prevent the reaction to reach completion. Therefore, loop bodies are required to raise an exception or retain the control for at least one instant, that is to say execute a `pause` or an `exit` statement in each iteration.

2.5 Signals

The instruction "`signal` S `in` p `end`" declares the *local* signal S in p. The free signals of a statement are said to be *interface* signals for this statement.

In an instant, a signal S is *emitted* iff at least one "`emit` S" statement is executed in this instant. In an instant, the *status* of a signal S is either *present* or *absent*. If S is present then all "`present` S `then` p `else` q `end`" statements executed in this instant, execute their "`then` p" branch in this instant; if S is absent they all execute their "`else` q" branch.

- A local signal is present iff it is emitted.
- An interface signal is present iff it is provided by the *environment*.

Remark an interface signal may be both absent and emitted. For example,

- In "`signal S in emit S; pause; present S then emit O end end`", `S` is present in the first instant of execution only, thus `O` is not emitted by this statement, as `S` is absent at the time of the "`present S`" test.
- In "`signal S in present S then emit O end || emit S end`", both `S` and `O` are emitted, `S` is present.
- In "`emit X; present X then emit O end`", the status of `X` depends on the environment, hence `O` is emitted iff `X` is provided by the environment.

3 Logical Behavioral Semantics

The *logical behavioral semantics* of Esterel [4,11] formalizes the informal semantics of the previous section. It describes the reactions of a statement p via a labeled transition system:

$$p \xrightarrow[I]{O,k} p'$$

where:

- the set I is the set of *present signals*,
- the set O is the set of *emitted signals*,
- the integer k is the *completion of the reaction*,
- the statement p' is the *residual of the reaction*.

Figure 2 expresses the logical behavioral semantics of Esterel as a set of facts and deduction rules in a structural operational style [16].

3.1 Completion Code and Residual

The completion code k and the residual p' encode the status of the execution:

- If $k = 1$ then this reaction does not complete the execution of p.
  It has to be continued by the execution of p' in the next instant.

- If $k \neq 1$ then this reaction ends the execution of p (p' does not matter):
  - $k = 0$ if the execution completes *normally* (without exception).
  - $k = d + 2$ if an exception of depth d *escapes* from p.

In particular, the completion code of "`exit` T_d" is "$2+d$". In order to compute the completion code "$\downarrow k$" of "`trap` T `in` p `end`" from the completion k of p, we define:

$$\downarrow k = \begin{cases} 0 & \text{if } k = 0 \text{ or } k = 2 \\ 1 & \text{if } k = 1 \\ k - 1 & \text{if } k > 2 \end{cases}$$

Conveniently, if p terminates with completion code k and q with completion code l then "p `||` q" terminates with code "$\max(k, l)$". For example,

$$\texttt{trap T in exit T}_0 \texttt{ || exit V}_4 \texttt{ end} \xrightarrow[I]{\emptyset, 3} \texttt{trap T in nothing end}$$

3.2 Present and Emitted Signals

The set I, written below the arrow, lists the signals provided by the environment. It drives the reactions of `present` statements:

- if $S \in I$ and $p \xrightarrow[I]{O,k} p'$ then `present` S `then` p `else` q `end` $\xrightarrow[I]{O,k} p'$.
- if $S \notin I$ and $q \xrightarrow[I]{O,k} q'$ then `present` S `then` p `else` q `end` $\xrightarrow[I]{O,k} q'$.

The set O, written above the arrow, lists the emitted interface signals. In particular,

$$\texttt{emit } S \xrightarrow[I]{\{S\}, 0} \texttt{nothing}$$

The *signal coherence law* – a local signal is present iff emitted – is enforced for the statement "`signal` S `in` p `end`" by the rules:

(signal+) if S is supposed present in p then it is emitted by p;
(signal−) if S is supposed absent in p then it is not emitted by p.

For instance, for inputs $I = \{\texttt{A}\}$,

$$\frac{\texttt{emit S} \xrightarrow[\{\texttt{A,S}\}]{\{\texttt{S}\}, 0} \texttt{nothing} \quad \texttt{S} \in \{\texttt{S}\}}{\texttt{signal S in emit S end} \xrightarrow[\{\texttt{A}\}]{\emptyset, 0} \texttt{signal S in nothing end}} \text{ using (signal+)}$$

$$\frac{\texttt{pause} \xrightarrow[\{\texttt{A}\}]{\emptyset, 1} \texttt{nothing} \quad \texttt{S} \notin \emptyset}{\texttt{signal S in pause end} \xrightarrow[\{\texttt{A}\}]{\emptyset, 1} \texttt{signal S in nothing end}} \text{ using (signal−)}$$

We shall further discuss these rules later.

$$\texttt{nothing} \xrightarrow[I]{\emptyset, 0} \texttt{nothing} \qquad \text{(nothing)}$$

$$\texttt{pause} \xrightarrow[I]{\emptyset, 1} \texttt{nothing} \qquad \text{(pause)}$$

$$\texttt{exit}\ T_d \xrightarrow[I]{\emptyset, d+2} \texttt{nothing} \qquad \text{(exit)}$$

$$\texttt{emit}\ S \xrightarrow[I]{\{S\}, 0} \texttt{nothing} \qquad \text{(emit)}$$

$$\frac{p \xrightarrow[I]{O, k} p' \quad k \neq 0}{\texttt{loop}\ p\ \texttt{end} \xrightarrow[I]{O, k} p'\texttt{;}\ \texttt{loop}\ p\ \texttt{end}} \qquad \text{(loop)}$$

$$\frac{p \xrightarrow[I]{O, k} p' \quad q \xrightarrow[I]{O', l} q'}{p\ \texttt{||}\ q \xrightarrow[I]{O \cup O', \max(k, l)} p'\ \texttt{||}\ q'} \qquad \text{(parallel)}$$

$$\frac{S \in I \quad p \xrightarrow[I]{O, k} p'}{\texttt{present}\ S\ \texttt{then}\ p\ \texttt{else}\ q\ \texttt{end} \xrightarrow[I]{O, k} p'} \qquad \text{(present+)}$$

$$\frac{S \notin I \quad q \xrightarrow[I]{O, k} q'}{\texttt{present}\ S\ \texttt{then}\ p\ \texttt{else}\ q\ \texttt{end} \xrightarrow[I]{O, k} q'} \qquad \text{(present−)}$$

$$\frac{p \xrightarrow[I]{O, 2} p'}{\texttt{trap}\ T\ \texttt{in}\ p\ \texttt{end} \xrightarrow[I]{O, 0} \texttt{nothing}} \qquad \text{(trap-catch)}$$

$$\frac{p \xrightarrow[I]{O, k} p' \quad k \neq 2}{\texttt{trap}\ T\ \texttt{in}\ p\ \texttt{end} \xrightarrow[I]{O, \downarrow k} \texttt{trap}\ T\ \texttt{in}\ p'\ \texttt{end}} \qquad \text{(trap-through)}$$

$$\frac{p \xrightarrow[I]{O, k} p' \quad k \neq 0}{p\texttt{;}\ q \xrightarrow[I]{O, k} p'\texttt{;}\ q} \qquad \text{(sequence-p)}$$

$$\frac{p \xrightarrow[I]{O, 0} p' \quad q \xrightarrow[I]{O', k} q'}{p\texttt{;}\ q \xrightarrow[I]{O \cup O', k} q'} \qquad \text{(sequence-q)}$$

$$\frac{p \xrightarrow[I \cup \{S\}]{O, k} p' \quad S \in O}{\texttt{signal}\ S\ \texttt{in}\ p\ \texttt{end} \xrightarrow[I]{O \setminus \{S\}, k} \texttt{signal}\ S\ \texttt{in}\ p'\ \texttt{end}} \qquad \text{(signal+)}$$

$$\frac{p \xrightarrow[I \setminus \{S\}]{O, k} p' \quad S \notin O}{\texttt{signal}\ S\ \texttt{in}\ p\ \texttt{end} \xrightarrow[I]{O, k} \texttt{signal}\ S\ \texttt{in}\ p'\ \texttt{end}} \qquad \text{(signal−)}$$

Fig. 2. Logical Behavioral Semantics

3.3 Execution

An *execution* of the statement p is a potentially infinite *chain* of reactions, such that all completion codes are equal to 1, but the last one in the finite case:

- finite execution: $p \xrightarrow[I_0]{O_0,1} p_1 \xrightarrow[I_1]{O_1,1} ... \xrightarrow[I_n]{O_n,k} p_{n+1}$, with $k \neq 1$, for some $n \in \mathbb{N}$.
- infinite execution: $p \xrightarrow[I_0]{O_0,1} p_1 \xrightarrow[I_1]{O_1,1} ... \xrightarrow[I_n]{O_n,1} ...$

We say that $\boldsymbol{I} = (I_0, I_1, ..., I_n)$ in the finite case and $\boldsymbol{I} = (I_n)_{n\in\mathbb{N}}$ in the infinite case is the *sequence of inputs* of the execution. Similarly, $\boldsymbol{O}$ is the *sequence of outputs*.

For example, the statement "`emit A; pause; emit B`" emits `A` and does not terminate instantly, with the residual "`nothing; emit B`" remaining to be executed. In the second and final instant of execution, `B` is emitted.

$$\texttt{emit A; pause; emit B} \xrightarrow[I_0]{\{\texttt{A}\},1} \texttt{nothing; emit B} \xrightarrow[I_1]{\{\texttt{B}\},0} \texttt{nothing}$$

We note $p \to p'$ iff there exists I and O such that $p \xrightarrow[I]{O,1} p'$. We say that q is *reachable* from p iff $p \xrightarrow{*} q$ where $\xrightarrow{*}$ is the *reflexive transitive closure* of $\to$.

4 Logical Correctness

Depending on the statement p and inputs I, the logical behavioral semantics may define zero, one or several reactions. Moreover, a given reaction may admit more than one proof, that is to say result from more than one composition of the rules of the semantics. For example, for $I = \{\texttt{A}\}$,

	reaction	proof
`nothing`	1	1
`loop nothing end`	0	0
`signal S in present S else emit S end end`	0	0
`signal S in present S then emit S end end`	1	2
`signal S in present S then emit S else pause end end`	2	2

In particular, for "`signal S in present S then emit S end end`", the semantics defines exactly one reaction, but with two possible proofs, obtained by using either the (signal−) or the (signal+) rule:

$$\dfrac{\dfrac{\texttt{S} \notin \{\texttt{A}\} \qquad \texttt{nothing} \xrightarrow[\{\texttt{A}\}]{\emptyset,0} \texttt{nothing}}{\texttt{present S then emit S end} \xrightarrow[\{\texttt{A}\}]{\emptyset,0} \texttt{nothing}} \qquad \texttt{S} \notin \emptyset}{\texttt{signal S in present S then emit S end end} \xrightarrow[\{\texttt{A}\}]{\emptyset,0} \texttt{signal S in nothing end}}$$

$$\dfrac{\dfrac{\texttt{S} \in \{\texttt{A},\texttt{S}\} \qquad \texttt{emit S} \xrightarrow[\{\texttt{A},\texttt{S}\}]{\{\texttt{S}\},0} \texttt{nothing}}{\texttt{present S then emit S end} \xrightarrow[\{\texttt{A},\texttt{S}\}]{\{\texttt{S}\},0} \texttt{nothing} \qquad \texttt{S} \in \{\texttt{S}\}}}{\texttt{signal S in present S then emit S end end} \xrightarrow[\{\texttt{A}\}]{\emptyset,0} \texttt{signal S in nothing end}}$$

The *internal behavior* of "`signal S in present S then emit S end end`" is not deterministic, since the local signal `S` can be both present or absent. Its *observed behavior* is nevertheless deterministic.

We expect programs to have deterministic deadlock-free executions. So, we have to discard as "incorrect" programs with no or too many possible behaviors. In this section, we formalize such a correctness criterion.

We define:

- p is *reactive* iff for all I, there exists at least one tuple (O, k, p') s.t. $p \xrightarrow[I]{O,k} p'$.
- p is *deterministic* iff for all I there is at most one tuple (O, k, p') s.t. $p \xrightarrow[I]{O,k} p'$.
- p is *strongly deterministic* iff p is deterministic and for all (I, O, k, p') the proof of $p \xrightarrow[I]{O,k} p'$ is unique if it exists.
- p is *logically correct* iff for all q reachable from p, q is reactive and deterministic.
- p is *strongly correct* iff for all q reachable from p, q is reactive and strongly deterministic.

Determinism ensures that the observed behavior of a statement is deterministic. Strong determinism guarantees that its internal behavior is deterministic, too. Reactivity combined with (strong) determinism ensures that there exists a unique reaction (with a unique proof) for this statement, whatever the inputs.

Logical correctness characterizes statements that have deterministic deadlock-free executions for any sequence of inputs. In addition, strong correctness ensures strong determinism. Strong correctness becomes a concern as soon as side effects or debugging have to be taken into account, as both may expose the internal behavior of a program. Of course, strong correctness implies logical correctness.

5 Deterministic Semantics

The logical behavioral semantics provides a very compact, structural formalization of the behavior of Esterel programs, which makes formal reasoning about the language tractable. Moreover, it defines reactivity and determinism, which are the agreed minimal correctness criteria for Esterel programs.

However, working with these criteria can be tedious. While, reactivity may be attested with a simple proof tree, establishing (strong-)determinism is more

complex and formally requires a proof about proof trees (proof of uniqueness).

Moreover, defining first many (proofs of) reactions for non-(strongly)-deterministic statements, which we then discard because there are too many, seems utterly inefficient.

Therefore, we propose to rewrite the rules for local signal declarations:

$$\frac{p \xrightarrow[I\cup\{S\}]{O,k} p' \qquad S \in O}{\texttt{signal}\ S\ \texttt{in}\ p\ \texttt{end} \xrightarrow[I]{O\setminus\{S\},k} \texttt{signal}\ S\ \texttt{in}\ p'\ \texttt{end}} \quad \text{(signal+)}$$

$$\frac{p \xrightarrow[I\setminus\{S\}]{O,k} p' \qquad S \notin O}{\texttt{signal}\ S\ \texttt{in}\ p\ \texttt{end} \xrightarrow[I]{O,k} \texttt{signal}\ S\ \texttt{in}\ p'\ \texttt{end}} \quad \text{(signal−)}$$

as the following (where k^+, k^-, etc. are nothing but convenient names):

$$\frac{p \xmapsto[I\setminus\{S\}]{O^-,k^-} p^- \qquad S \in O^- \qquad p \xmapsto[I\cup\{S\}]{O^+,k^+} p^+ \qquad S \in O^+}{\texttt{signal}\ S\ \texttt{in}\ p\ \texttt{end} \xmapsto[I]{O^+\setminus\{S\},k^+} \texttt{signal}\ S\ \texttt{in}\ p^+\ \texttt{end}} \quad \text{(signal++)}$$

$$\frac{p \xmapsto[I\setminus\{S\}]{O^-,k^-} p^- \qquad S \notin O^- \qquad p \xmapsto[I\cup\{S\}]{O^+,k^+} p^+ \qquad S \notin O^+}{\texttt{signal}\ S\ \texttt{in}\ p\ \texttt{end} \xmapsto[I]{O^-,k^-} \texttt{signal}\ S\ \texttt{in}\ p^-\ \texttt{end}} \quad \text{(signal−−)}$$

We call the resulting semantics the *deterministic semantics*, and denote the corresponding reactions by the transition symbol "$\mapsto$".

Intuitively, it consists in enforcing in each signal rule that the other one does not apply, without introducing negative premises [12] such as:

$$S, p, I, O, k, \text{ and } p' \text{ are } \textit{not} \text{ such that } p \xrightarrow[I\cup\{S\}]{O,k} p' \text{ and } S \in O$$

Rather than negating the whole precondition, we only swap the binary decision $S \in O$ for $S \notin O$, and vice versa. In the logical behavioral semantics, we had:

- (signal+): if S is supposed present in p then it is emitted by p.
- (signal−): if S is supposed absent in p then it is not emitted by p.

In our deterministic semantics, the rules for the `signal` construct become:

- (signal++):
  - if S is supposed present in p then it is emitted.
  - if S is supposed absent in p then it is *still* emitted.
- (signal−−):
  - if S is supposed absent in p then it is not emitted.
  - if S is supposed present in p then it is not emitted *either*.

5.1 Examples

For example, the deterministic semantics produces the same reactions as the logical behavioral semantics, in the following two cases (cf. Section 3):

$$\frac{\texttt{pause} \overset{\emptyset,1}{\underset{\{\texttt{A}\}}{\longmapsto}} \texttt{nothing} \quad \texttt{S} \notin \emptyset \qquad \texttt{pause} \overset{\emptyset,1}{\underset{\{\texttt{A},\texttt{S}\}}{\longmapsto}} \texttt{nothing} \quad \texttt{S} \notin \emptyset}{\texttt{signal S in pause end} \overset{\emptyset,1}{\underset{\{\texttt{A}\}}{\longmapsto}} \texttt{signal S in nothing end}}$$

$$\frac{\texttt{emit S} \overset{\{\texttt{S}\},0}{\underset{\{\texttt{A}\}}{\longmapsto}} \texttt{nothing} \quad \texttt{S} \in \{\texttt{S}\} \qquad \texttt{emit S} \overset{\{\texttt{S}\},0}{\underset{\{\texttt{A},\texttt{S}\}}{\longmapsto}} \texttt{nothing} \quad \texttt{S} \in \{\texttt{S}\}}{\texttt{signal S in emit S end} \overset{\emptyset,0}{\underset{\{\texttt{A}\}}{\longmapsto}} \texttt{signal S in nothing end}}$$

The deterministic semantics defines no reaction for:

- the non-reactive statement:
  "`signal S in present S else emit S end end`"
- the non-deterministic statement:
  "`signal S in present S then emit S else pause end end`"
- the non-strongly-deterministic statement:
  "`signal S in present S then emit S end end`"

5.2 Determinism

The new semantics is *globally deterministic*:

Theorem 5.1 *For all p and I, there exists at most one (O, k, p') s.t. $p \overset{O,k}{\underset{I}{\longmapsto}} p'$.*

Theorem 5.2 *For all p, I, O, k, p', the proof of $p \overset{O,k}{\underset{I}{\longmapsto}} p'$ is unique if it exists.*

Proof. Simple structural induction on p. □

There is no need to count proofs and reactions in the deterministic semantics.

5.3 Properness

Since, the uniqueness of proofs and reactions is ensured, we shall say that the statement p is correct with respect to the deterministic semantics, i.e. *proper*, iff the deterministic semantics defines at least one reaction at any stage of the execution of p for any sequence of inputs. Formally, we define:

- p is *initially proper* iff for all I, there exists (O, k, p') such that $p \overset{O,k}{\underset{I}{\longmapsto}} p'$.
- $p \mapsto p'$ iff there exists I and O such that $p \overset{O,1}{\underset{I}{\longmapsto}} p'$.
- $\overset{*}{\mapsto}$ is the reflexive transitive closure of $\mapsto$.
- p is *proper* iff for all q such that $p \overset{*}{\mapsto} q$, q is initially proper.

6 Comparison

We now precisely relate the logical behavioral and the deterministic semantics.

6.1 Properness implies strong correctness

Theorem 6.1 *If* $p \overset{O,k}{\underset{I}{\longmapsto}} p'$ *then* $p \overset{O,k}{\underset{I}{\longrightarrow}} p'$.

Theorem 6.2 *If* $p \overset{O_0,k_0}{\underset{I}{\longmapsto}} p'_0$ *and* $p \overset{O_1,k_1}{\underset{I}{\longmapsto}} p'_1$ *then* $O_0 = O_1$, $k_0 = k_1$, $p'_0 = p'_1$.

Theorem 6.3 *If* $p \overset{O,k}{\underset{I}{\longmapsto}} p'$ *then the proof of* $p \overset{O,k}{\underset{I}{\longrightarrow}} p$ *is unique.*

Proof. cf. Appendix A. □

By writing

$$p \overset{O,k}{\underset{I}{\longmapsto}} p'$$

we not only express that p *may* react to inputs I, with outputs O, completion code k, and residual p' in the deterministic semantics, thus in the logical behavioral semantics as well (Th. 6.1), but also that it *must* react this way in both semantics (Th. 5.1 and 6.2), and that its internal behavior is deterministic (Th. 5.2 and 6.3). As a consequence,

Corollary 6.4 *If* p *is proper then* p *is strongly correct.*

6.2 Strong correctness does not imply properness

Reciprocally, a strongly correct statement is not necessarily proper, as reactivity combined with strong determinism does not imply initial properness. Let's consider two examples:

- ```
  signal S in
    present S then loop nothing end end
  end
  ```

  For all inputs I, the logical behavioral semantics defines the following unique proof tree for this program:

$$\dfrac{\dfrac{\texttt{S} \notin I\backslash\{\texttt{S}\} \qquad \texttt{nothing} \overset{\emptyset,0}{\underset{I\backslash\{\texttt{S}\}}{\longrightarrow}} \texttt{nothing}}{\texttt{present S then loop nothing end end} \overset{\emptyset,0}{\underset{I\backslash\{\texttt{S}\}}{\longrightarrow}} \texttt{nothing}} \qquad \texttt{S} \notin \emptyset}{\texttt{signal S in present S then ... end end} \overset{\emptyset,0}{\underset{I}{\longrightarrow}} \texttt{signal S in nothing end}}$$

  The deterministic semantics however defines no reaction for this statement, whatever I. Neither the rule (signal++), nor the rule (signal−−) applies, as "`loop nothing end`", thus "`present S then loop nothing end end`" are not initially proper.
  ```

- ```
  loop
    signal S in
      present S then emit S else pause end
    end
  end
  ```

  The body "`signal S in present S then emit S else pause end end`" of the loop may react in two possible ways in the logical behavioral semantics, whatever I, with respective completion codes 0 and 1:

  $$\texttt{signal S in present S then emit S else pause end end} \xrightarrow[I]{\emptyset,0} \ldots$$

  $$\texttt{signal S in present S then emit S else pause end end} \xrightarrow[I]{\emptyset,1} \ldots$$

  Since exactly one of these two reactions admits a non-zero completion code, the whole loop statement is both reactive and strongly deterministic. On the other hand, the deterministic semantics defines no reaction for the body, hence no reaction for the loop.

6.3 Strongly correct non-proper statements

In the logical behavioral semantics, non-determinism may compensate for non-reactivity, or the other way around, so that a piece of incorrect code may be embedded into a strongly correct program. More precisely,

Theorem 6.5 *If p is reactive and strongly deterministic but not initially proper then there exists a subterm q of p such that q is not reactive or not strongly deterministic.*

Proof. cf. Appendix B. □

Intuitively, q behaves *well* in p only because of its context of occurrence, which constrains the execution of q from the outside, making sure the non-reactive or non-strongly-deterministic behaviors of q are never triggered. In other words, q could be simplified while preserving the behavior of p. Let's consider again our two examples in this new light:

- ```
  signal S in
    present S then loop nothing end end
  end
  ```

  The subterm "`present S then loop nothing end end`" is not reactive because of its `then` branch, but never used with `S` present. Therefore, it can be replaced by its implicit `else` branch, that is to say `nothing`, leading to the equivalent [3] program "`signal S in nothing end`", which is proper.

[3] Technically, they are strongly bisimilar [15] w.r.t. the logical behavioral semantics.

- ```
  loop
    signal S in
      present S then emit S else pause end
    end
  end
  ```

  The body "`signal S in present S then emit S else pause end end`" is not deterministic, but the enclosing loop enforces `S` to be absent. Again, the "`present S then emit S else pause end`" statement can simplified. The resulting program "`loop signal S in pause end end`" is proper and logically equivalent.

Therefore, there is something *wrong* with these programs, even if neither logical correctness nor strong correctness are sensitive to it. In any case, they are intricate constructions with no practical purpose.

7 Constructive Semantics

The constructive semantics of Esterel [4] ensures that behaviors can be effectively computed, that is to say *causally* computed. For instance, although the following program is logically correct, even strongly correct as `S` can only be present, it is rejected by the constructive semantics:

```
signal S in present S then emit S else emit S end end
```

Intuitively, this program is not *constructive* because the status of `S` must be "guessed" prior to its emission. Such an argument however is not relevant to the deterministic semantics, which considers this program to be proper.

On the other hand, the deterministic semantics sometimes rejects constructive programs, such as:

```
signal S in
  present S then
    signal T in present T else emit T end end
  end
end
```

Since `S` cannot be emitted – there is no "`emit S`" statement – the `then` branch of the `present` statement is never "visited" by the constructive semantics. As a result, this program is constructive. On the other hand, the deterministic semantics does explore this branch, so that the program is not proper.

Executions in the constructive semantics being defined by a (complex) monotonous information propagation process, there is at most one reaction defined for each program and each set of inputs. In other words, the constructive semantics is globally deterministic in the sense of Section 5.

In summary, even if both semantics are globally deterministic, the reasons for this property are very different, and the corresponding correctness criteria are unrelated. They both make sense and could be combined.
```

8 Conclusion

In contrast with the logical behavioral semantics of Esterel, the deterministic semantics we introduce in this work, defines at most one execution for all programs and all inputs. In particular, if the deterministic semantics defines the execution of a program, then this execution is unique, thus correct.

Importantly, the deterministic semantics does not change the semantics of "reasonable" programs. If the deterministic semantics of a program is defined then it matches its logical behavioral semantics. Reciprocally, if the deterministic semantics of a program is not defined then the program or some subterm of the program is incorrect w.r.t. the logical behavioral semantics.

Moreover, our new semantics achieves determinism at a much lower cost than the constructive semantics of Berry. As a result, we claim that the deterministic semantics provides a much better starting point for formal reasoning about Esterel programs than both the logical behavioral semantics and the constructive semantics.

A Proof of Theorems 6.1 to 6.3

By structural induction on p, we prove that if $p \xmapsto[I]{O,k} p'$ then:

- $p \xrightarrow[I]{O,k} p'$ with a unique proof;
- if $p \xrightarrow[I]{O_0,k_0} p'_0$ then $O = O_0$, $k = k_0$, $p' = p'_0$.

Proof. Let's consider the case $p =$ "`signal` S `in` q `end`", and choose a set I. By hypothesis, there exists (O, k, p') such that:

$$\texttt{signal}\ S\ \texttt{in}\ q\ \texttt{end} \xmapsto[I]{O,k} p'$$

Either rule (signal++) or (signal−−) has to be used to define this reaction. Let's for instance consider the case (signal−−). The case (signal++) is similar. There exists $(O^-, k^-, q^-, O^+, k^+, q^+)$ such that:

- $q \xmapsto[I\setminus\{S\}]{O^-,k^-} q^-$ and $q \xmapsto[I\cup\{S\}]{O^+,k^+} q^+$
- $S \notin O^-, S \notin O^+, O = O^-, k = k^-, p' =$ "`signal` S `in` q^- `end`".

so that the following deduction holds in the deterministic semantics:

$$\frac{q \xmapsto[I\setminus\{S\}]{O^-,k^-} q^- \quad S \notin O^- \quad q \xmapsto[I\cup\{S\}]{O^+,k^+} q^+ \quad S \notin O^+}{p = \texttt{signal}\ S\ \texttt{in}\ q\ \texttt{end} \xmapsto[I]{O,k} \texttt{signal}\ S\ \texttt{in}\ q^-\ \texttt{end} = p'}$$

By induction hypothesis:

- $q \xrightarrow[I\setminus\{S\}]{O^-,k^-} q^-$ with a unique proof.
- if $q \xrightarrow[I\setminus\{S\}]{O_0^-,k_0^-} q_0^-$ then $O^- = O_0^-$, $k^- = k_0^-$, $q^- = q_0^-$.
- $q \xrightarrow[I\cup\{S\}]{O^+,k^+} q^+$ with a unique proof.
- if $q \xrightarrow[I\cup\{S\}]{O_0^+,k_0^+} q_0^+$ then $O^+ = O_0^+$, $k^+ = k_0^+$, $q^+ = q_0^+$.

On the one hand, as $S \notin O^+$, no reaction for p can be defined using (signal+). On the other hand, by rule (signal−),

- $p \xrightarrow[I]{O^-,k^-} \texttt{signal}\ S\ \texttt{in}\ q^-\ \texttt{end}$ with a unique proof.
- if $p \xrightarrow[I]{O_0,k_0} p_0'$ then $O_0 = O^-$, $k_0 = k^-$, $p_0' = \texttt{signal}\ S\ \texttt{in}\ q^-\ \texttt{end}$.

And similarly for all other cases. □

B Proof of Theorem 6.5

By structural induction on p, we prove that if p and all its subterms are reactive and strongly deterministic then p is initially proper.

Proof. Let's consider the case p = "$\texttt{signal}\ S\ \texttt{in}\ q\ \texttt{end}$", and choose a set I. By hypothesis, q and all its subterms are reactive and strongly deterministic. By induction hypothesis, q is initially proper. Thus, there exists $(k^-, O^-, q^-, k^+, O^+, q^+)$ such that:

$$q \xmapsto[I\setminus\{S\}]{O^-,k^-} q^- \quad \text{and} \quad q \xmapsto[I\cup\{S\}]{O^+,k^+} q^+$$

There are four cases:

- $S \in O^-$, $S \in O^+$, then by rule (signal++), $p \xmapsto[I]{O^+\setminus\{S\},k^+} \texttt{signal}\ S\ \texttt{in}\ q^+\ \texttt{end}$.
- $S \notin O^-$, $S \notin O^+$, then by rule (signal−−), $p \xmapsto[I]{O^-,k^-} \texttt{signal}\ S\ \texttt{in}\ q^-\ \texttt{end}$.
- $S \in O^+$, $S \notin O^-$:
  · by rule (signal+), $p \xrightarrow[I]{O^+\setminus\{S\},k^+} \texttt{signal}\ S\ \texttt{in}\ q^+\ \texttt{end}$
  · by rule (signal−), $p \xrightarrow[I]{O^-,k^-} \texttt{signal}\ S\ \texttt{in}\ q^-\ \texttt{end}$
  Therefore, p is not strongly deterministic. Contradiction.
- $S \notin O^+$, $S \in O^-$, then neither (signal+) nor (signal−) is applicable. Therefore, p is not reactive. Contradiction.

Similarly, in all other cases, the deterministic semantics defines a reaction for p, whatever I. As a consequence, p is initially proper. □

References

[1] *The Esterel v5_92 Compiler*, http://www-sop.inria.fr/esterel.org/.

[2] Benveniste, A., P. Caspi, S. Edwards, N. Halbwachs, P. Le Guernic and R. de Simone, *The Synchronous Languages Twelve Years Later*, Proceedings of the IEEE **91** (2003), pp. 64–83.

[3] Berry, G., *The semantics of pure Esterel*, in: M. Broy, editor, *Program Design Calculi*, Series F: Computer and System Sciences **118** (1993), pp. 361–409.

[4] Berry, G., *The constructive semantics of pure Esterel. Draft version 3* (1999), http://www-sop.inria.fr/esterel.org/.

[5] Berry, G., *The Esterel v5 language primer. Version 5_91* (2000), http://www-sop.inria.fr/esterel.org/.

[6] Berry, G. and L. Cosserat, *The synchronous programming language Esterel and its mathematical semantics*, LNCS **197** (1984), pp. 389–448.

[7] Berry, G. and G. Gonthier, *The Esterel synchronous programming language: Design, semantics, implementation*, Science of Computer Programming **19** (1992), pp. 87–152.

[8] Boussinot, F. and R. de Simone, *The Esterel language*, Another Look at Real Time Programming, Proceedings of the IEEE **79** (1991), pp. 1293–1304.

[9] Edwards, S., "Languages for Digital Embedded Systems," Kluwer, 2000.

[10] Edwards, S. A., V. Kapadia and M. Halas, *Compiling Esterel into Static Discrete-Event Code*, in: *Synchronous Languages Applications and Programming*, 2004.

[11] Gonthier, G., "Sémantique et modèles d'exécution des langages réactifs synchrones: application à Esterel," Thèse d'informatique, Université d'Orsay, Paris, France (1988).

[12] Groote, J. F., *Transition system specifications with negative premises*, in: *CONCUR'90*, Springer, 1990 pp. 332–341.

[13] Halbwachs, N., "Synchronous Programming of Reactive Systems," Kluwer, 1993.

[14] Malik, S., *Analysis of Cyclic Combinational Circuits*, in: *IEEE/ACM International Conference on CAD*, ACM/IEEE (1993), pp. 618–627.

[15] Milner, R., "Communication and Concurrency," Series in Computer Science, Prentice Hall, 1989.

[16] Plotkin, G., *A structural approach to operational semantics*, Report DAIMI FN-19, Aarhus University (1981), to be published in the JLAP special issue on SOS, 2004.

[17] Tardieu, O. and R. de Simone, *Instantaneous termination in pure Esterel*, in: *SAS'03*, LNCS **2694**, 2003, pp. 91–108.

Recent BRICS Notes Series Publications

NS-04-1 **Luca Aceto, Willem Jan Fokkink, and Irek Ulidowski, editors.** ***Preliminary Proceedings of the Workshop on Structural Operational Semantics, SOS '04,*** **(London, United Kingdom, August 30, 2004), August 2004. vi+56.**

NS-03-4 **Michael I. Schwartzbach, editor.** ***PLAN-X 2004 Informal Proceedings,*** **(Venice, Italy, 13 January, 2004), December 2003. ii+95.**

NS-03-3 **Luca Aceto, Zoltán Ésik, Willem Jan Fokkink, and Anna Ingólfsdóttir, editors.** ***Slide Reprints from the Workshop on Process Algebra: Open Problems and Future Directions, PA '03,*** **(Bologna, Italy, 21–25 July, 2003), November 2003. vi+138.**

NS-03-2 **Luca Aceto.** ***Some of My Favourite Results in Classic Process Algebra*****. September 2003. 21 pp. Appears in the** ***Bulletin of the EATCS*****, volume 81, pp. 89–108, October 2003.**

NS-03-1 **Patrick Cousot, Lisbeth Fajstrup, Eric Goubault, Maurice Herlihy, Kurtz Alexander, Martin Raußen, and Vladimiro Sassone, editors.** ***Preliminary Proceedings of the Workshop on Geometry and Topology in Concurrency Theory, GETCO '03,*** **(Marseille, France, September 6, 2003), August 2003. vi+54.**

NS-02-8 **Peter D. Mosses, editor.** ***Proceedings of the Fourth International Workshop on Action Semantics, AS 2002,*** **(Copenhagen, Denmark, July 21, 2002), December 2002. vi+133 pp.**

NS-02-7 **Anders Møller.** ***Document Structure Description 2.0*****. December 2002. 29 pp.**

NS-02-6 **Aske Simon Christensen and Anders Møller.** ***JWIG User Manual*****. October 2002. 35 pp.**

NS-02-5 **Patrick Cousot, Lisbeth Fajstrup, Eric Goubault, Maurice Herlihy, Martin Raußen, and Vladimiro Sassone, editors.** ***Preliminary Proceedings of the Workshop on Geometry and Topology in Concurrency Theory, GETCO '02,*** **(Toulouse, France, October 30–31, 2002), October 2002. vi+97.**

NS-02-4 **Daniel Gudbjartsson, Anna Ingólfsdóttir, and Augustin Kong.** ***An BDD-Based Implementation of the Allegro Software*****. August 2002. 2 pp.**