Continuations for Prototyping Concurrent Languages*

Eneia Todoran[1]
(Eneia.Todoran@cs.utcluj.ro)

Nikolaos S. Papaspyrou[2]
(nickie@softlab.ntua.gr)

[1] Technical University of Cluj-Napoca
Faculty of Automation and Computer Science
Department of Computer Science
Baritiu Str. 28, 400027, Cluj-Napoca, Romania.

[2] National Technical University of Athens
School of Electrical and Computer Engineering
Software Engineering Laboratory
Polytechnioupoli, 15780 Zografou, Greece.

Abstract

We have recently introduced the "continuation semantics for concurrency" (CSC) technique in an attempt to exploit the benefits of using continuations in concurrent systems development. In the CSC approach, a continuation is an application-dependent configuration of computations (partially evaluated denotations). Every computation or group of computations contained in a continuation can be accessed and manipulated separately by the denotational semantic function. The CSC technique provides excellent flexibility and a "pure" continuation-based approach to communication and concurrency, in which all control concepts are modeled as operations manipulating continuations.

In this paper, we present a methodology for concurrent language development, based on denotational semantics. We show that, by using the CSC technique, denotational semantics can be used both as a method for formal specification and design and as a general method for implementing compositional prototypes of concurrent programming languages. We provide continuation structures for various traditional concurrent control concepts. We also present compositional semantic models for the following advanced control concepts that have not been modeled until now without CSC: remote object (process) destruction and cloning and nondeterministic promotion in Andorra-like languages.

1 Introduction

In software engineering, a *prototype* is an initial version of a system which is used to demonstrate concepts, try out design options and, generally, to find out more about the problem and its possible solutions [Somm06]. A prototype can serve as a mechanism for identifying the software requirements, which may be expressed in a semi-formal or a formal notation. This paper is developed around the core idea of using the "continuation semantics for concurrency"

*This paper is based on work partially funded by the joint research and technology program of bilateral cooperation between the Hellenic Republic and Romania. Project title: "Continuations and monads for parallel and distributed computing" (2003–2005).

(CSC) technique — recently introduced by us [Todo00a, Todo00c] — as a prototyping tool in a language development methodology based on denotational semantics.

It is easy to use a functional language, such as Scheme [Kels98] or Haskell [Peyt99], and classic denotational semantic techniques to produce prototype implementations for various aspects of sequential languages. To support this statement it is probably enough to mention the early work of Peter Mosses on the use of denotational descriptions in compiler generation [Moss75, Moss79], or the work of Mitchell Wand on semantic prototyping [Wand84]. However, classic denotational techniques seem inappropriate for producing executable prototypes of concurrent programming languages; in this paper, by *classic technique* we mean either resumption models for concurrency [Plot76, dBak96] or Stratchey and Wadsworth continuations [Stra74]. For a general discussion on the pragmatic issues related to the use of denotational semantics in the development of (complex and / or) concurrent systems the reader should consult [Moss90, Moss96, Abra96]. From the particular perspective of semantic prototyping, it is easy to see that power domains [Plot76] — used extensively in the denotational treatment of concurrency — do not provide an adequate support for empirical testing and evaluation. An element of a power domain is exponential in the length of execution traces[1] and therefore a direct implementation can lead to intractable solutions; also, formal reasoning upon such structures is impractical in most non-trivial applications.

In this paper we present a methodology for the development of concurrent languages based on denotational semantics. We give a number of carefully designed continuation structures for communication and concurrency. We show that, in the CSC approach, the language designer can establish a simple relation between a general notion of *structured continuation* and the control concepts of the (concurrent) language under study. We also show that, by using the CSC technique, denotational semantics can be used not only as a method for formal specification and design, but also as a general method for implementing *tractable* compositional prototypes of concurrent programming languages. The methodology considered in this paper is inspired from Queinnec's work [Quei90, Quei92a, Quei92b], where a similar approach was employed for developing concurrent and distributed extensions of Scheme. In Queinnec's work, the denotational models for concurrent behavior are implemented in Scheme and are based on domains of multisets of computations which can be executed in an interleaved fashion; such domains can be seen as a particular case of the semantic domains for CSC. In this article, we present a number of continuation-based prototype interpreters, that we call *semantic interpreters* or *semantic prototypes*. The semantic interpreters can be seen as prototype implementations of corresponding denotational (mathematical) models; as an implementation tool we prefer the lazy functional programming language Haskell.

In the approach that we employ in this paper, a denotational (compositional) mapping can use CSC continuations to produce a single stream of observables, i.e. a single execution trace. By using a random number generator, the semantic mapping can choose an arbitrary execution trace, thus simulating the nondeterministic behavior of a "real" concurrent system. Alternatively, CSC continuations can be used to model the non-determinism by employing power domains. Our semantic interpreters are parameterized by a notion of program behavior monad. Monads were proposed as a tool for structuring denotational semantics [Mogg90, Mogg91]. They have become quite popular both in the denotational semantics and the functional programming community [Wadl92, Lian95, Lian96, Lian98] and are di-

[1] An element of a power domain is a tree-like structure, or a collection of "traces" essentially equivalent to an unfolding of such a tree.

rectly supported in Haskell [Peyt99]. The program behavior monad can be designed in two ways, roughly corresponding to the two perspectives on non-determinism described in this paragraph. To implement the "single trace" semantics, the monad is parameterized by an *oracle* that decides the alternative to be selected in nondeterministic choices; being given different oracles, *any* possible trace can be obtained. The "all possible traces" semantics is implemented by using an appropriate power domain monad.

In sections 2, 3, 4 and 5.1 we offer semantic interpreters for nine imperative languages, out of which only one is sequential. Each of these imperative languages provides assignment, conditional selection, (parameterless) recursive procedures, and a primitive for producing intermediate results (observables) at the standard output file. The following control concepts are modeled denotationally in a imperative setting by using the CSC technique: sequential composition, parallel composition, process creation, CSP-like [Hoar78, Hoar85] synchronous communication, suspension and the `await` statement [Owic76] (including atomization), the rendez-vous concept (a key notion in languages such as Ada [Ada83] or POOL [Amer89]), remote object (process) destruction, and remote object (process) cloning. The last two operations are studied in section 5.1; they can be encountered at operating system level, in some coordination languages [Holz96], or in distributed object oriented and multi agent systems such as Obliq [Card95] and IBM Java Aglets [Lang98, Aglets]. The former operation kills a parallel running object and is similar to the "`kill -9`" system call in Unix. The latter operation creates an identical copy of a (parallel) running object[2].

In section 5.2.1 and in appendix B.4 we present semantic interpreters designed with CSC for the control flow kernel of three logic programming languages. In section 5.2.1 we study two concurrent languages based on the Andorra model [Warr88]; in appendix B.4 we treat the control flow kernel of sequential Prolog. In our semantic investigation we adopt the "logic programming without logic" approach [dBak91] and we model compositionally the following control concepts: backtracking, AND-OR parallelism, failure, and nondeterministic promotion in Andorra-like languages. The Andorra model was designed to exploit both AND parallelism and search, i.e. don't know nondeterminism. The model gives priority to determinate goals (for which it is known that at most one clause succeeds) over nondeterminate goals, as the nondeterminate steps are likely to multiply work. When only nondeterminate goals remain in a parallel conjunction, the system selects one such goal and performs a so-called *nondeterministic promotion* by replicating all the AND-parallel goals for each non failing alternative of the selected one.

In total, this paper offers semantic interpreters designed with CSC-continuations for twelve languages. To the best of our knowledge the control concepts that we study in section 5 — i.e. the operations for remote object (process) control and the nondeterministic promotion in Andorra — have never been modeled denotationally by using only classic (compositional) techniques.

For each concurrent language under study, we present both a "single trace" and an "all possible traces" semantics; the choice between the two kinds of semantics depends *only* on the selection of the *monad* that is used for modeling the program behavior. The relation is always *simple* because all control concepts (including parallel composition and process synchronization) are modeled as operations manipulating continuations, and only the final yield[3] of the semantic mapping distinguishes between the "single trace" and the "all possible

[2]In this paper, an *object* is a thread (sequence) of computations with a local state.

[3]The final yield of the denotational function is encapsulated in the program behavior monad.

traces" behaviors. Of course, for the sequential languages (which are also deterministic) the two behaviors coincide. For all other languages, the CSC technique is an essential ingredient in the semantic design.

In "all possible traces" semantics, a semantic interpreter can only be tested on toy concurrent programs; in this case the final yield of the interpreter models an element of a power domain, rather than a single execution trace. This gives raise to non-tractable solutions, because an element of a power domain is exponential in the length of execution traces. When the monad for "single trace" semantics is selected, our semantic interpreters are *tractable* and can be tested with "real life" concurrent programs. For example, in section 4.2, we present a simple concurrent generator of prime numbers based on the sieve of Erathostenes. It is not difficult to check that all operations manipulating CSC continuations are polynomial in all parameters involved in computations, including the number of parallel processes. Therefore, in "single trace" semantics, our semantic interpreters are reasonably efficient and can be used without difficulty to test relatively complex concurrent algorithms.[4]

In this paper, instead of using mathematical notation for the definition of the denotational models, we use the lazy functional programming language Haskell [Peyt99]. Haskell provides an excellent support for the specific techniques used in denotational semantics: continuations, monads and fixed point semantics. In addition, a natural correspondence can be established between the mathematical domains used in denotational semantics and the types in Haskell. In this paper, functions, products, and sums — as used in denotational semantics, are implemented in Haskell as functions, tuples and algebraic data types, respectively; also, power domains, are implemented by using Haskell lists. In this way we avoid the unnecessary complexities accompanying the use of (classic) domain theory or the theory of metric spaces, which could have been adopted alternatively. At the same time, we allow our denotational models to be directly implementable, in the form of semantic interpreters for the (concurrent) languages under study, and thus to be easily tested and evaluated.

The CSC technique was first introduced in [Todo00a] by using the mathematical methodology of metric semantics. Also, in [Todo04] a classic (cpo-based [Plot78]) denotational model was designed for CSC. Although we do not develop the idea technically, we work on the assumption that a CSC-based semantic interpreter implemented in Haskell is a *prototype* system, which can easily be synchronized with (or can be the basis of) a corresponding mathematical (denotational) *specification*. This assumption is reasonable if we take into consideration the ideas and the experiments given in [Moss75, Moss79, Wand84, Watt86].

1.1 Continuation Semantics for Concurrency

Continuations are well-known in denotational semantics for the flexibility they provide as a language design tool. Traditional continuations were first introduced in denotational semantics to model the behavior of the `go to` statement [Stra74]. As it is well-known, they can easily capture the semantics of sequential composition, and can be used to model a variety of advanced control concepts, including non-local jumps [Stra74], exceptions, coroutines [Frie86] and even multitasking [Wand80, Dybv89]. However, traditional continuations do not work well enough in the presence of concurrency [Hieb94]. The CSC technique [Todo00a, Todo00c] was introduced in an attempt to exploit the benefits of using continuations in the development

[4]The concurrent version of the sieve of Erathostenes algorithm given in section 4.2 creates a new process for each prime number. Therefore, in this particular case, performance degrades continuously.

of concurrent languages. It can model both sequential and parallel composition in interleaving semantics, as well as various synchronization and communication mechanisms in a "pure" continuation style.

CSC is a tool for denotational semantics.[5] The mathematical domains for CSC can be defined by recursive domain equations [Todo00a, Todo04]. The technique provides an *excellent flexibility* in the compositional design of concurrent control flow concepts. There are control concepts that can be modeled compositionally with CSC, but which seem to be beyond the expressive power of the classic denotational techniques. For example, in [Todo00c] CSC was applied in designing the first compositional semantics for nondeterministic promotion in Andorra-like languages [Warr88].

The central characteristic of the CSC technique is the modeling of continuations as complex structures of computations, where by *computation* we understand a partially evaluated denotation (meaning function).[6] The semantics of each statement is defined with respect to a continuation, which is a representation of the behavior of the *rest* of the program.[7] The space of computations is divided into one *active* computation and the rest of the computations which are encapsulated in a CSC continuation; conceptually, the continuation behaves as an evaluation context [Fell06, Danv04] for the active computation. Intuitively, the CSC technique is a semantic formalization of a process scheduler simulated on a sequential machine. Each computation remains active only until it performs an elementary action; subsequently, another computation taken from the continuation is planned for execution. Depending upon the structure of the continuations and the particular scheduling strategy, the computations contained in a CSC continuation can be evaluated either in some specific order or in an interleaved fashion.

1.2 Contribution

In this paper we describe the basic evaluation mechanism of CSC continuations, we propose a number of carefully designed continuation structures for communication and concurrency, and provide new insights in understanding the excellent flexibility provided by CSC.

CSC continuations are divided in this paper into two categories: closed continuations and open continuations. A closed continuation is a self-contained structure of computations. An open continuation is an evaluation context for the active computation; it is a structure of computations which contains a *hole* (indicating the conceptual position of the active computation). A CSC-based semantic interpreter consists of three components: an *evaluator*, a (continuation-completion mapping together with a) *normalization procedure*, and a *scheduler*. The evaluator maps open continuations to program behaviors. It comprises the (compositional) definition of the semantic mapping, together with language-specific control operators. The functions of the evaluator have one thing in common: they manipulate open continuations. The normalization procedure transforms an open continuation into a corresponding closed continuation. The scheduler maps closed continuations to program behaviors. It acti-

[5] It can also be used for operational semantic design [Todo00a], but in this paper we focus on denotational semantics.

[6] We do not present any operational model in this paper, but it may help to mention that such a *computation* is simply a statement in the case of operational semantics, and respectively the partially evaluated denotation (or meaning) of a statement in the case of denotational semantics [Todo00a].

[7] According to the initial definition [Stra74], a continuation is a representation of the rest of the computation. However, in the CSC approach a continuation is not simply a function to some answer type, but rather a structured configuration of partially evaluated denotations.

vates one computation, by decomposing a closed continuation into an (activable) computation and a corresponding open continuation. In general, the selection of the activable computation is nondeterministic, and it may follow after a (finite) number of synchronization steps. A CSC-based semantic interpretor implements an evaluate-normalize-schedule loop [Danv04].

In this paper we show that the CSC technique gives the language designer the ability of establishing a simple relation between a general notion of structured continuation and the control concepts of the (concurrent) language under study. CSC continuations are language-specific configurations of partially evaluated meaning functions (denotations), which can be accessed and manipulated separately. In a mathematical model, a CSC continuation is an element of a semantic domain defined as the solution of a domain equation where the domain variable occurs in the left-hand side of a function space construction [Todo00a]; in this sense the computation model is rather complex. However, CSC continuations can be designed in terms of simple structures which can provide operational intuition.

All continuation structures given in this article are designed by using two abstract concepts, which seem to provide a basic framework for control flow semantics: the *stack* to model sequential composition, and the *multiset* to model parallel composition. In the technical sections of the paper we present semantic interpreters for twelve languages. The CSC continuation structure is language specific. For a simple language providing only sequential composition the continuation is a stack of computations. In the case of a language with parallel composition and action prefixing[8] the continuation is a multiset of computations. For more complex behaviors we introduce the *ps*-tree — a CSC continuation structure in which *parallel* levels (multisets) alternate with *sequential* levels (stacks). The *ps*-tree is inspired from the the structure of a *cactus stack* [Bobr73]. For a language that combines parallel composition with a general operator for sequential composition (rather than just action prefixing) the structure of the CSC continuation takes the form of a multiset of *ps*-trees (a *ps*-forest).

As a language design tool the CSC technique provides an *excellent flexibility*. Each computation or group of computations contained in a CSC continuation can be accessed and manipulated separately *at evaluation time*. To prove this facility, in section 5 we present compositional semantic models for the following concepts which, as far as we know, have never been modeled denotationally by using only classic techniques: remote object (process) destruction and cloning, and nondeterministic promotion in Andorra-like languages. Based on our experiments, we believe that these control concepts are beyond the expressive power of classic compositional techniques. In the CSC approach their semantics can easily be modeled by appropriate manipulations of the computations contained in continuations.

In this paper we also show that a CSC-based semantic interpreter can generate exactly the observables produced by a "real" implementation of a concurrent programming language. Our semantic interpreters produce no silent steps or communication attempts; the synchronization and the communication information is completely encapsulated in continuations. Due to the excellent flexibility provided by CSC, branching domains are never needed to obtain compositionality. A CSC-based semantic interpreter can always produce exactly the desired observables assembled in a linear model.[9]

[8] Action prefixing is a particular form of sequential composition, in which the first component is an elementary action. It takes the form $a; x$, where a may be an assignment statement or an expression evaluation, and x is an arbitrary statement. For example, CCS [Miln89] and the π-calculus [Miln99] use action prefixing instead of a general operator for sequential composition.

[9] An element of a linear domain is a collection of sequences. An element of a branching domain is a tree-like structure. For a more elaborated explanation the reader may consult, e.g., [dBak96].

Finally, the ability to design compositional prototypes (in "single trace" semantics) for concurrent programming languages is another proof of the flexibility provided by CSC. To the best of our knowledge, denotational semantics has never been used systematically for concurrent languages prototyping and all our attempts to get a general solution to this problem by using only classic compositional techniques have failed.

1.3 Related work

Continuations constitute a classical tool in denotational semantics [Stra74]. For a historical overview of the discoveries of continuations in a variety of settings the reader may consult [Reyn93]. Traditional continuations are frequently used in denotational semantics to capture the behavior of sequential composition. They can also model a variety of more advanced control concepts, including nonlocal jumps [Stra74], coroutines [Frie86] and even multitasking [Wand80, Dybv89]. However, traditional continuations provide only a limited support for concurrency semantics [Hieb94].

Programming languages such as Scheme [Kels98] or SML/NJ [SMLNJ], support continuations through a `call/cc` primitive. In the uniprocessor implementation of Concurrent ML [Repp92, Repp99], threads are implemented with SML/NJ continuations. The behavior of continuations in the presence of concurrency has been investigated, e.g., in the MultiLisp and the MultiScheme projects [Hals85, Mill87, Katz90, Feel93]. One of the first (informal) uses of the continuation concept in a concurrent setting was in the actor model of Carl Hewitt [Hewi77a, Hewi77b].

Traditional (undelimited) continuations are used to represent the entire "rest of the computation" [Stra74]. Delimited continuations [Fell88a, Fell88b, Sita90, Hieb94, Quei91, Danv92, More94] seem to provide a finer control than the traditional continuations. Delimited continuations can be used to represent only a part of the remainder of the computation; also, they support the composition of continuations. The CSC continuation structures presented in this paper are (partly) inspired from [Hieb94], where (stacks of stacks or) trees of stacks are employed in the implementation of *subcontinuations*, a variant of delimited continuations which can be used to control tree-structured concurrency. Tree-structured concurrency can also be controlled with CSC continuations. However, as far as we know, delimited continuations have never been used to model some of the advanced control concepts handled compositionally with CSC continuations in this paper, such as synchronous CSP-like communication or remote object (process) control.

The "continuation semantics for concurrency" (CSC) technique was introduced in [Todo00a] by using the mathematical methodology of metric semantics. In [Todo00a], we defined and related operational and denotational semantic models for a language providing CSP-like synchronous communication and a language that incorporates the asynchronous communication mechanism studied in [dBoe93]. It seems that the metric framework enforces the introduction of some artificial silent steps (with no operational counterpart) in the denotational models designed with CSC for synchronous communication. Subsequently, in [Todo04], we showed that this drawback can be eliminated if classic (cpo-based [Plot78]) domains are employed instead. The use of Haskell as an implementation tool for the denotational models designed with CSC was first experienced in [Todo00c]. Also, in [Todo00c] we showed that CSC continuations are a valuable tool in the analysis and design of semantic models for parallel logic programming, and we designed the first compositional semantics for nondeterministic promotion in Andorra-like languages [Warr88].

The language development methodology considered in this paper is based on Queinnec's work [Quei90, Quei92a, Quei92b], where a number of executable denotational models for concurrent and distributed extensions of Scheme are presented. The denotational models given in Queinnec's work use a choice operator, called *oneof*, to mimic a scheduler by selecting an arbitrary element from a multiset of computations; an alternative definition for *oneof* results in a classical powerdomain semantics. This operator is not defined in the presence of more advanced control notions (modeled with CSC in this paper) such as: tree-structured concurrency, atomization, or CSP-like synchronous communication. The domains of multisets of computations used in Queinnec's work represent only a (very) particular device for concurrency semantics, which can not be extended in a straightforward way to handle arbitrary control flow. Moreover, the concept of a continuation semantics for concurrency is not articulated in [Quei90, Quei92a, Quei92b]. In this paper, we hope to convince the reader that CSC represents a general new approach to concurrency semantics.

It is well-known that classic denotational techniques can be used to produce prototype implementations for (various aspects of) sequential programming languages [Moss75, Moss79, Wand84, Schm86, Watt86]. In this article we show that, by using the CSC technique, denotational semantics can also be used to produce prototype implementations for concurrent programming languages. As far as we know denotational semantics has never been used systematically for concurrent languages prototyping and all our attempts to get a general solution to this problem by using only classic compositional techniques have failed. Moreover, the remote object control operations and the nondeterministic promotion in Andorra that we explore in section 5 seem to be beyond the expressive power of the classic compositional techniques.

1.4 Overview

In section 2 we compare the CSC technique with the classic technique of continuations. We consider a simple sequential language, for which we develop a semantic interpreter designed with classic continuations and one designed with CSC. Section 3 presents semantic interpreters designed with CSC continuations for five concurrent languages of progressive complexity. The concurrent languages studied in section 3 have one thing in common: a global state shared by all processes. We present CSC continuation structures for various control concepts, including parallel composition, process creation, synchronous communication, rendez-vous, atomization and suspension. In section 4 we present semantic interpreters for two languages based on distributed objects, where an *object* is a sequence of computations acting on a local state. In section 5 we offer compositional models for remote object destruction and cloning and for nondeterministic promotion in Andorra-like languages. The section on logic programming (5.2.1) is accompanied by a substantial appendix that includes a semantic interpreter for an abstract language capturing the control flow kernel of pure sequential Prolog. In section 6 we present some concluding remarks and directions for future research.

2 Classic Continuations versus CSC

In this section we define a simple sequential imperative language $\mathcal{L}_{seq}$, which gives us the opportunity to compare the classic technique of continuations with the CSC technique. $\mathcal{L}_{seq}$ comprises a simple language of expressions supporting basic operators on integer numbers and boolean values. Throughout this paper, the notation $(x \in)X$ introduces the set X with

a typical element x ranging over X. In the grammar that follows, $(z \in)Z$ denotes the set of integer constants and $(v \in)V$ the set of (numeric) *variables*. The set of (numeric or) *integer expressions* is $(n \in)N$ and the set of *booleean expressions* is $(b \in)B$. The operators have the obvious intended meaning.

$$
\begin{array}{lcl}
n & ::= & z \mid v \mid n\ op\ n \\
b & ::= & n\ rel\ n \mid \mathbf{not}\ b \mid b\ \mathbf{and}\ b \mid b\ \mathbf{or}\ b \\
op & ::= & + \mid - \mid * \mid / \mid \% \\
rel & ::= & == \mid != \mid < \mid > \mid <= \mid >=
\end{array}
$$

We consider the following set $(a \in)A$ of elementary (or *atomic*) actions:

$$a\ ::=\ v := n \mid \mathbf{write}\ n$$

The statement $v := n$ *assigns* an integer value to a variable; the statement $\mathbf{write}\ n$ *outputs* an integer value as an observable result. Only the values of numeric expressions can be assigned or output; expressions of boolean type will only be used as conditions.

Let $(y \in)Y$ be a set of *procedure variables*. The class $(x \in)X$ of *statements* for $\mathcal{L}_{seq}$ is defined as follows:

$$x\ ::=\ \mathbf{skip} \mid a \mid \mathbf{if}\ b\ \mathbf{then}\ x\ \mathbf{else}\ x \mid x\ ;\ x \mid \mathbf{letrec}\ y = x\ \mathbf{in}\ x \mid \mathbf{call}\ y$$

The language $\mathcal{L}_{seq}$ provides an empty statement, a statement that performs a single elementary action, a conditional statement, sequential composition, and standard constructs for defining and calling (parameterless) recursive procedures.[10]

The syntax of $\mathcal{L}_{seq}$ can be implemented in Haskell by using the following set of data types:

```
type V = String
type Y = String

data N = Z Int | V V | Plus N N | Minus N N | Times N N | Div N N | Mod N N
data B = Eq N N | Ne N N | Lt N N | Le N N | Gt N N | Ge N N
       | Not B | And B B | Or B B

data A = Assign V N | Write N
data X = Skip | Action A | If B X X | Seq X X | LetRec Y X X | Call Y
```

The semantics of (numeric or booleean) expressions is defined with respect to a domain $(s \in)S$ of *states*. A state is a function from variables to numeric (integer) *values* in the set $(u \in)Val$.

```
type Val = Int
type S   = V → Val
```

The semantics of expressions can easily be defined as follows. It is independent of the choice of technique that will be used for defining the rest of the language, and will not change in the rest of the paper.

[10] Throughout this paper, syntactic definitions will be treated as abstract syntax. Parentheses will be used freely to group syntactic constructs, such as expressions or statements. Composition of statements will be left-associative.

$$
\begin{array}{lll}
evN :: N \to S \to Val & & \\
evN\ (Z\ n) & s = n & \\
evN\ (V\ v) & s = s\ v & \\
evN\ (Plus\ n_1\ n_2) & s = evN\ n_1\ s & + \quad evN\ n_2\ s \\
evN\ (Minus\ n_1\ n_2) & s = evN\ n_1\ s & - \quad evN\ n_2\ s \\
evN\ (Times\ n_1\ n_2) & s = evN\ n_1\ s & * \quad evN\ n_2\ s \\
evN\ (Div\ n_1\ n_2) & s = evN\ n_1\ s & `div` \quad evN\ n_2\ s \\
evN\ (Mod\ n_1\ n_2) & s = evN\ n_1\ s & `mod` \quad evN\ n_2\ s
\end{array}
$$

$$
\begin{array}{lll}
evB :: B \to S \to Bool & & \\
evB\ (Eq\ n_1\ n_2) & s = evN\ n_1\ s & == \quad evN\ n_2\ s \\
evB\ (Ne\ n_1\ n_2) & s = evN\ n_1\ s & /= \quad evN\ n_2\ s \\
evB\ (Lt\ n_1\ n_2) & s = evN\ n_1\ s & < \quad evN\ n_2\ s \\
evB\ (Le\ n_1\ n_2) & s = evN\ n_1\ s & <= \quad evN\ n_2\ s \\
evB\ (Gt\ n_1\ n_2) & s = evN\ n_1\ s & > \quad evN\ n_2\ s \\
evB\ (Ge\ n_1\ n_2) & s = evN\ n_1\ s & >= \quad evN\ n_2\ s \\
evB\ (Not\ b) & s = not\ (evB\ b\ s) & \\
evB\ (And\ b_1\ b_2) & s = evB\ b_1\ s & \&\& \quad evB\ b_2\ s \\
evB\ (Or\ b_1\ b_2) & s = evB\ b_1\ s & || \quad evB\ b_2\ s
\end{array}
$$

We also define the domain $(q \in) Q$ of sequences of intermediate observable values. The constant *Epsilon* denotes *normal termination*. To model non-termination we rely on the laziness of Haskell. For all the imperative languages that we study in this paper the type *Obs* of observables is just a synonym of the type *Val* of values.[11]

$$
\begin{array}{l}
\textbf{type}\ Obs = Val \\
\textbf{data}\ Q \quad = Epsilon \mid Observe\ Obs\ Q
\end{array}
$$

The use of monads in denotational semantic descriptions can improve their modularity. To facilitate the definition of a modular and elegant semantics, we introduce a program behavior monad M parameterized by the type of program result. In the rest of the paper, program behaviors will be elements of $M\ Q$. In this section, M will be the standard environment monad, with the domain of states S as the environment, i.e. $M\ Q = S \to Q$. We also define the function *put* which facilitates the output of observable values.

$$
\begin{array}{l}
\textbf{data}\ M\ a = InM\{\, unM :: S \to a \,\} \\
\\
\textbf{instance}\ Monad\ M\ \textbf{where} \\
\quad return\ a \qquad = InM\ (\backslash s \to a) \\
\quad InM\ m >>= f = InM\ (\backslash s \to unM\ (f\ (m\ s))\ s)
\end{array}
$$

$$
\begin{array}{l}
put :: Obs \to M\ Q \to M\ Q \\
put\ u\ (InM\ m) = InM\ (\backslash s \to Observe\ u\ (m\ s))
\end{array}
$$

To conclude the preparations for the technical sections that follow, we introduce the (polymorphic) operator *upd* which "perturbs" a function at a point; it will be used in the semantic definitions for the assignment statement and the recursive definition.

$$
\begin{array}{l}
upd :: Eq\ a \Rightarrow a \to b \to (a \to b) \to a \to b \\
upd\ a\ b\ f\ a' = \textbf{if}\ a' == a\ \textbf{then}\ b\ \textbf{else}\ f\ a'
\end{array}
$$

[11] For the time being, it may seem strange that we used a special data type instead of the equivalent type of lists of values [*Val*]. The purpose of this choice will become clear in later sections.

In section 2.1 we present a semantic interpreter for $\mathcal{L}_{seq}$ designed with traditional continuations. Next, in section 2.2 the semantic interpreter is re-designed with CSC structured continuations. This gives us the opportunity to compare the two techniques.

2.1 Classic Continuation Semantics

The denotational models given in this paper use the technique of continuations, therefore it is reasonable to assume that a *denotation* is a function from the current continuation to a corresponding program behavior. Also, we employ a classic notion of a *semantic environment* which is a mapping from procedure variables to denotations. The domains $(d \in)D$ of denotations and $(e \in)Env$ of (semantic) environments are introduced by the following Haskell declarations:

$$\textbf{type } D \quad = Cont \rightarrow M\ Q$$
$$\textbf{type } Env = Y \rightarrow D$$

A continuation is a representation of the rest of the computation [Stra74]. Using the traditional approach, both continuations and program behaviors are modeled as functions $S \rightarrow Q$ from states to observables.

$$\textbf{type } Cont = M\ Q$$

Functions *rdState* and *inState* provide the interface between states and types constructed by monad M. As a provision for sections 4 and 5.1 where CSC continuations are employed in the semantic representation of distributed objects with local states, the types of *rdState* and *inState* presume that the state may be part of the current continuation (a feature that is actually not needed in sections 2 or 3).[12]

$$rdState :: Cont \rightarrow M\ S$$
$$rdState\ c = InM\ (\backslash s \rightarrow s)$$

$$inState :: S \rightarrow (Cont \rightarrow M\ Q) \rightarrow Cont \rightarrow M\ Q$$
$$inState\ s\ f\ c = InM\ (\backslash s' \rightarrow unM\ (f\ c)\ s)$$

After these preparations, we introduce the equations that define a general continuation semantics for $\mathcal{L}_{seq}$. The functions *semA* and *sem* define the semantics of elementary actions and the (fixed-point) semantics of statements, respectively. The definitions given below are straightforward if we state that *cc* is the *continuation completion* mapping that converts a continuation into a program behavior, and *addc* is the control operator for sequential composition. The following equations will remain unchanged for all the imperative languages that we study in the sequel.

$$semA :: A \rightarrow Cont \rightarrow M\ Q$$
$$semA\ (Assign\ v\ n)\ c = rdState\ c >>= \backslash s \rightarrow$$

[12] In the particular case of a global state the definitions correspond to the *rdEnv* and *inEnv* operations of the environment monad:

$$rdState :: M\ S$$
$$rdState = InM\ (\backslash s \rightarrow s)$$

$$rdState :: S \rightarrow M\ Q \rightarrow M\ Q$$
$$rdState\ s\ m = InM\ (\backslash s' \rightarrow unM\ m\ s)$$

```
                        inState (upd v (evN n s)) s) cc c
semA (Write n)    c = rdState c >>= \s →
                        put (evN n s) (cc c)

sem :: X → Env → D
sem Skip              e c = cc c
sem (Action a)        e c = semA a c
sem (Seq x1 x2)       e c = sem x1 e (addc (Den (sem x2 e)) c)
sem (If b x1 x2)      e c = rdState c >>= \s →
                              if evB b s then sem x1 e c
                                         else sem x2 e c
sem (Call y)          e c = e y c
sem (LetRec y x1 x2) e c = sem x2 e' c
  where e' = upd y (fix (\d → sem x1 (upd y d e))) e
        fix   :: (a → a) → a
        fix f  = f (fix f)
```

We get a classic continuation semantics for $\mathcal{L}_{seq}$ if we implement the operators *cc* and *addc* as below. The following definitions are redundant in the case of a classic continuation semantics, but they smoothen the transition to the following sections where we present a number of CSC-based semantic interpreters.

```
data Comp = Den D

cc :: Cont → M Q
cc = id

addc :: Comp → Cont → Cont
addc (Den d) c = d c
```

Comp is the domain of *computations*. In section 2 a computation is just a denotation, but more complex definitions are employed in the sections that follow. Also, in the classic continuation semantics given in this section *cc* is the identity function and the control operator *addc* models sequential composition as a simple application. More elaborated definitions however are required in the semantic models designed with CSC.

The above semantics of $\mathcal{L}_{seq}$ defines a semantic interpreter implemented in Haskell. To test our semantic interpreter we need initial values for the state, the semantic environment, and the continuation. In the initial state s_0 all variables have indeterminate values. The initial (empty) environment e_0 simply raises an exception whenever it is used. Also, the initial continuation c_0 is the behavior of an empty program; the definition given below is appropriate for the classic continuation semantics of $\mathcal{L}_{seq}$.

```
s0 :: S
s0 v = error "variable not initialized"

e0 :: Env
e0 y = error "unbound procedure variable"

c0 :: Cont
c0 = return Epsilon
```

The execution of a program can be performed by using the function *test*; the auxiliary function *display* facilitates the execution of programs in the initial state.

$display :: Show\ a \Rightarrow M\ a \rightarrow IO\ ()$
$display\ (InM\ m) = print\ (m\ s_0)$

$test :: X \rightarrow IO\ ()$
$test\ x = display\ (sem\ x\ e_0\ c_0)$

To test the semantic interpreter we consider the $\mathcal{L}_{seq}$ program given below. The Haskell implementation of syntactic definitions is straightforward but less readable. In the rest of the paper we only present the abstract syntax of the test programs.

letrec $y =$ **if** $0 < v$ **then** **write** v; $v := v - 1$; **call** y
else **skip**
in $v := 10$; **call** y

Running this program with *test* produces the following output:

$[10, 9, 8, 7, 6, 5, 4, 3, 2, 1]$

2.2 Continuation Semantics for Concurrency

The "continuation semantics for concurrency" (CSC) technique [Todo00a, Todo00c] was introduced in an attempt to exploit the benefits provided by continuations [dBru86] in concurrent systems development. It is a general language design tool, providing *excellent flexibility* in the compositional modeling of parallel and distributed systems. A CSC continuation is a language-specific structure of computations (partially evaluated meaning functions), rather than just a program behavior. In a mathematical model, a CSC continuation is an element of a semantic domain defined as solution of a domain equation where the domain variable occurs in the left-hand side of a function space construction [Todo00a, Todo04]; in this sense the computation model is rather complex. However, CSC continuations can be designed in terms of simple structures which can provide operational intuition. The structure of a CSC continuation reflects the execution order of the computations it contains. In the case of a simple language providing only sequential composition a continuation is a *stack* of computations. To model the interleaved execution of the parallel computations contained in a CSC continuation we employ the concept of a *multiset*. Both stacks and multisets of computations are implemented in this paper using Haskell lists.

According to the CSC evaluation mechanism, the space of computations is divided into one *active* computation and the rest of the computations which are encapsulated in the continuation [Todo00a]. Conceptually, a CSC continuation behaves as an evaluation context [Danv04, Fell06] for the active computation. In this paper CSC continuations are divided into two categories: *closed continuations* and *open continuations*. A closed continuation is a self-contained structure of computations. An open continuation is a structure of computations which contains a *hole* (indicating the conceptual position of the active computation). Both closed and open continuations are semantic representations of what remains to be computed from the program [Stra74]. At the same time, an open continuation is an *evaluation context* [Danv04, Fell06] for the active computation, but a closed continuation cannot be interpreted by using the notion of an evaluation context. As explained below, CSC continuations are only closed for scheduling purposes.

The functions of a CSC-based semantic interpreter can be grouped into the following three components: an evaluator, a continuation-completion mapping together with a normalization

type $SC = [Comp]$

type $Kont = SC$
type $Cont = Kont$

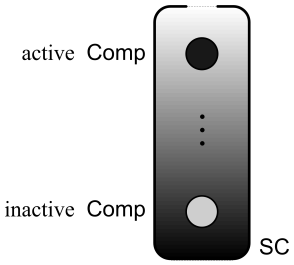


Figure 1: Structure of continuations for $\mathcal{L}_{seq}$: a stack of computations.

procedure, and a scheduler. The evaluator maps open continuations to program behaviors. It comprises the (compositional) definition of the semantic mapping together with language-specific control operators. *The functions of the evaluator have one thing in common: they manipulate open continuations, i.e. evaluation contexts.* The continuation-completion function is called by the evaluator to map an open continuation to the program answer that would result if the continuation alone was left to execute. First, it calls *the normalization procedure, which transforms the open continuation into a corresponding closed continuation*; next it calls the scheduler. Intuitively, the normalization procedure computes the closed continuation that results by removing the 'hole' from an open continuation. *The scheduler maps closed continuations to program behaviors.* It activates a computation by decomposing a closed continuation into an (activable) computation and a corresponding open continuation. In the particular case of a deterministic language like $\mathcal{L}_{seq}$, there is always at most one possible such activation. However, in general, the selection of the activable computation is nondeterministic, and it may follow after a (finite) number of synchronization steps. A CSC-based semantic interpretor implements an "evaluate-normalize-schedule" loop [Danv04].

In the sequel we use the type *Kont* to implement the domain of closed continuations, and the type *Cont* to implement the domain of open continuations. The difference between closed and open continuations can be encoded explicitly in the semantic domains, but this is only an implementation decision. In this paper we prefer to set $Cont = Kont$ and to distinguish between open and closed continuations only at a conceptual level. We believe the intuitive behavior of CSC continuations is not obscured by this decision; at the same time the resulted definitions are a bit simpler. For example, in the case of $\mathcal{L}_{seq}$ both open and closed continuations are stacks of computations executing in sequence, but the reader has to imagine that an open stack contains a *hole* at its top (indicating the conceptual position of the active computation). The definitions of *Cont* and *Kont* for $\mathcal{L}_{seq}$ are given below in terms of a domain *SC* of stacks of computations. The intuitive behavior of an open stack of computations is depicted in figure 1. We emphasize that, with the exception of figure 11, all figures given in this paper depict open continuations.

To conclude the CSC semantics of $\mathcal{L}_{seq}$ it suffices to redefine the special functions on continuations, but it may be appropriate to state clearly the components of the CSC-based semantic interpreter for this case. The domain *Comp* of computations remains as in section 2.1, but the functions *cc* and *addc* have to be redefined.

- The *evaluator* comprises the (compositional) definitions of the semantic functions *sem* and *semA* given in section 2.1, together with a specific new definition of *addc*. Notice that the domain D of denotations (used in the declarations of *semA* and *sem*) is given

in terms of *open* continuations: $D = Cont \rightarrow M\ Q$. The control operator *addc* adds a computation to an open continuation for sequential composition. In the case of $\mathcal{L}_{seq}$, *addc* simply prepends a computation to a stack of computations.

$$addc :: Comp \rightarrow Cont \rightarrow Cont$$
$$addc\ p\ sc = p : sc$$

- The *continuation-completion mapping cc* is called by the evaluator to map an open continuation to a program answer. For this purpose, it normalizes the continuation and next it calls the scheduler *kc*. Function *cc* stops if the normalized continuation is empty. The following definition of *cc* will remain unchanged in the rest of the paper (with the exception of appendix B.2). The *normalization procedure re* transforms an open continuation into a corresponding closed continuation; here, *re* is just the identity function.

$$cc :: Cont \rightarrow M\ Q$$
$$cc\ c = \textbf{case}\ re\ c\ \textbf{of}$$
$$\qquad [\,] \rightarrow return\ Epsilon$$
$$\qquad k \rightarrow kc\ k$$

$$re :: Cont \rightarrow Kont$$
$$re = id$$

- The *scheduler* function *kc* is again very simple for $\mathcal{L}_{seq}$; it activates the computation at the top of the stack, giving it as a continuation the rest of the stack.

$$kc :: Kont \rightarrow M\ Q$$
$$kc\ (Den\ d : sc) = d\ sc$$

The domain of continuations has been modified in this section. If we want to test the CSC-based semantic interpreter we need to re-define the initial continuation c_0. We put:

$$c_0 :: Cont$$
$$c_0 = [\,]$$

All our experiments show that the CSC-based semantic interpreter and the semantic interpreter designed with classic continuations behave the same. In particular, the example program of section 2.1 produces the same execution trace in the CSC semantics.

3 Continuations for Communication and Concurrency

The CSC technique was introduced as a general tool for designing concurrency semantics [Todo00a, Todo00b, Todo00c]. One of our present aims is to show that it can provide finer control than the classic denotational techniques. More importantly, the CSC technique gives the language designer the ability to model compositionally both sequential and concurrent control flow concepts simply by adapting the structure of continuations. In this section we present CSC continuation structures for five concurrent languages. All continuation structures are designed by combining two simple concepts: the *stack* to model sequential composition and the *multiset* to model parallel composition.

Both stacks and multisets are implemented in this paper using Haskell lists. The operations on stacks and multisets are used by the schedulers of our CSC-based semantic interpreters. A multiset is an unordered collection which may contain duplicate elements. The behavior of a multiset is given by the following general multiset scheduling algorithm:

$$
\begin{array}{l}
ms :: [a] \rightarrow [(a, [a])] \\
ms\ xs = aux\ xs\ [\,] \\
\quad \textbf{where}\ aux\ [\,] \qquad ys = [\,] \\
\qquad\qquad aux\ (x : xs)\ ys = (x, xs +\!\!+ ys) : aux\ xs\ (x : ys)
\end{array}
$$

Function *ms* takes as parameter a list implementing a multiset. We use this algorithm to decompose a closed continuation into pairs consisting of an activable computation and a corresponding open continuation. In the case of a multiset of computations such a decomposition is not unique, and the selection of the activable computation is nondeterministic.[13] Suppose that a closed continuation structure is represented as the multiset $[\,d_1, d_2, d_3]$. The *ms* algorithm computes three possible decompositions.

$$ms\ [\,d_1, d_2, d_3] = [(d_1, [\bullet, d_2, d_3]), (d_2, [\bullet, d_3, d_1]), (d_3, [\bullet, d_2, d_1])]$$

The above picture gives the intuition behind the *ms* algorithm. The bullet ($\bullet$) represents the conceptual position of the activable computation, i.e. the *hole* of each open continuation. To simplify the implementation of our schedulers, we find convenient to consider that the hole in a list implementing an open multiset is always at the head of the list. This implementation decision is easily justified by the fact that the reordering of the elements of a multiset is a semantics preserving transformation.[14]

In this section we offer CSC-based semantic interpreters for five concurrent languages: $\mathcal{L}_{ap}$, $\mathcal{L}_{new}$, $\mathcal{L}_{ps}$, $\mathcal{L}_{rv}$ and $\mathcal{L}_{aw}$. Their computation model is based on the concept of a *global state* shared by all concurrent processes. The concept of a local state is studied in section 4. Apart from various combinations of sequential and parallel composition operators, $\mathcal{L}_{ap}$, $\mathcal{L}_{new}$ and $\mathcal{L}_{ps}$ provide CSP-like synchronous communication, $\mathcal{L}_{rv}$ provides a simplified form of the rendez-vous mechanism of Ada [Ada83] or POOL [Amer89], and $\mathcal{L}_{aw}$ incorporates the suspension mechanism of the `await` statement [Owic76]. The control, synchronization and communication concepts are designed by using specific CSC continuation structures.

Unlike the sequential language of section 2, the concurrent languages that we study in the present section allow for the possibility of *deadlock*, i.e. the possibility that two or more concurrent processes wait for each other to do something before either can proceed. A concurrent program can terminate normally or it can terminate in a deadlock state. We redefine the domain Q of sequences of observables to account for the possibility of a deadlock.

$$\textbf{data}\ Q = Epsilon \mid Deadlock \mid Observe\ Obs\ Q$$

We also need to redefine the program behavior monad to deal with the nondeterministic behavior of a concurrent language. As already explained in the introduction, the nondeterminism can be modeled either in "single trace" or in "all possible traces" semantics. We define a specific monad for each of the two interpretations of non-determinism. The two monads can be used interchangeably without other modifications of our semantic interpreters.

[13] In the case of a stack of computations such a decomposition is unique: the activable computation is always selected from the top of the stack.

[14] The exact position of the hole (or of any other element in the multiset) is not important.

For the purpose of this section, the definitions take into account the model of a concurrent language employing the concept of a global state. The "all possible traces" version of the monad is given in appendix A. Here, we only present a monad that implements the "single trace" semantics. The monad is parameterized by an *oracle* that decides the alternative to be selected in nondeterministic choices. The oracle is implemented by using the concept of a random number generator. Different execution traces can be obtained at consecutive executions of a nondeterministic program; given different oracles *any* possible trace can be obtained.

For the experiments given in this paper we use the random number generator that is given in the Haskell library `Random.hs`. We define the domain R of random number generators as a type synonym of $Random.StdGen$.

$$\textbf{type } R = Random.StdGen$$

We also need an initial value $r0 :: R$, together with a function for obtaining a new random number. We put:

$$\begin{aligned} &r_0 :: R \\ &r_0 = Random.mkStdGen\ 42 \end{aligned}$$

The revised monad M is a composition of an environment monad, with environment of type S, and a state monad, with state of type R. Function *random* extracts one value from the random number generator; it can be defined easily, as the function *Random.next* from Haskell's library has precisely the appropriate type. The functions *put*, *display*, *rdState* and *inState* can easily be redefined.

$$\textbf{data } M\ a = InM\{\, unM :: S \to R \to (a, R)\,\}$$

$$\begin{aligned} &\textbf{instance } Monad\ M\ \textbf{where} \\ &\quad return\ a \qquad\quad = InM\ (\backslash s \to \backslash r \to (a, r)) \\ &\quad InM\ m >\!\!>\!= f = InM\ (\backslash s \to \backslash r \to \textbf{let } (a, r') = m\ s\ r\ \textbf{in } unM\ (f\ a)\ s\ r') \end{aligned}$$

$$\begin{aligned} &random :: M\ Int \\ &random = InM\ (\backslash s \to Random.next) \end{aligned}$$

$$\begin{aligned} &put\ u\ (InM\ m) \quad = InM\ (\backslash s \to \backslash r \to \textbf{let } (q, r') = m\ s\ r\ \textbf{in } (Observe\ u\ q, r')) \\ &display\ (InM\ m) = print\ (fst\ (m\ s_0\ r_0)) \end{aligned}$$

$$\begin{aligned} &rdState\ c \qquad\quad = InM\ (\backslash s \to \backslash r \to (s, r)) \\ &inState\ s\ f\ c \qquad = InM\ (\backslash s' \to \backslash r \to unM\ (f\ c)\ s\ r) \end{aligned}$$

The revised monad's most important operation is non-deterministic choice. This is implemented with the aid of two functions: *ned* and *bigned*. Both use the random number generator; the former chooses between two alternatives, whereas the latter chooses between a (finite) set of alternatives.

$$\begin{aligned} &ned :: M\ a \to M\ a \to M\ a \\ &ned\ m_1\ m_2 = bigned\ [m_1, m_2] \end{aligned}$$

$$\begin{aligned} &bigned :: [M\ a] \to M\ a \\ &bigned\ ml \ = random >\!\!>\!= \backslash r \to \\ &\qquad\qquad ml\ !!\ (r\ `mod`\ (length\ ml)) \end{aligned}$$

The definitions of the domains *Val* (of values), S (of states), D (of denotations) and *Env* (of semantic environments), as well as the definitions of the initial state s_0 and the initial

environment e_0, remain as in the section 2. Also, the semantic valuations *evN* (for numeric expressions) and *evB* (for boolean expressions), as well as the equations given in section 2 for *semA* (the semantics of elementary actions) and *sem* (the semantics of statements), remain unchanged. The same definitions will also be employed without modifications in sections 4 and 5.1. In addition, the definition of the continuation completion mapping *cc* remain as in section 2.2.

The function *test* given in section 2.1 can be used without modifications to test our semantic interpreters in "all possible traces" semantics; of course, for the purpose of this section *test* uses the current definition of *display*. To avoid any ambiguities, we repeat below the definition of *test*:

$$\begin{aligned}&test :: X \to IO\ ()\\&test\ x = display\ (sem\ x\ e_0\ c_0)\end{aligned}$$

The above definition can also be used to (test and) evaluate our semantic interpreters in "single trace" semantics, but it always returns the same execution trace. It is convenient to define a function *testR* for obtaining a different random trace at every new execution.

$$\begin{aligned}&testR :: X \to IO\ ()\\&testR\ x = \mathbf{do}\ r_0 \leftarrow Random.newStdGen\\&\qquad\qquad \mathbf{let}\ InM\ m = sem\ x\ e_0\ c_0\\&\qquad\qquad print\ (fst\ (m\ s_0\ r_0))\end{aligned}$$

We are finally prepared to present the CSC continuation structure, the evaluator, the normalization procedure and the scheduler for each of the five concurrent languages that we study in section 3.

3.1 Parallel Composition, Action Prefixing and Synchronous Communication

The language $\mathcal{L}_{ap}$ is the simplest concurrent language that we study in this paper. It replaces the unrestricted sequential computation of $\mathcal{L}_{seq}$ by *action prefixing* and adds an operator for *parallel composition*. The combination of these two can be used to build concurrent computations; however, parallel computations cannot *meet*. The last additional feature is CSP-like *synchronous* communication with the aid of a *non-deterministic choice* operator. Communication is performed through *channels*, each named with an element of $(\gamma \in) Ch$. The syntax of $\mathcal{L}_{ap}$, including the communication primitives $(g \in) C$ for send and receive, is given below.

$$\begin{aligned}g\ &::=\ \gamma\,!\,n\ \mid\ \gamma\,?\,v\\x\ &::=\ \mathbf{skip}\ \mid\ a \cdot x\ \mid\ \mathbf{if}\ b\ \mathbf{then}\ x\ \mathbf{else}\ x\ \mid\ x \parallel x\ \mid\ \mathbf{ned}\ [\ (g \to x)^*\ ]\\&\ \ \mid\ \mathbf{letrec}\ y = x\ \mathbf{in}\ x\ \mid\ \mathbf{call}\ y\end{aligned}$$

A non-deterministic statement is a construct that contains a set of statements, each guarded by a communication primitive g. Two such constructs can "communicate" if one of them has a clause of the form $\gamma\,!\,n \to x_1$ and the other has a clause of the form $\gamma\,?\,v \to x_2$. The result of this communication is that n is evaluated, its value is assigned to variable v and execution continues with x_1 and x_2 respectively.

The syntax of $\mathcal{L}_{ap}$ in Haskell is given below.

```
type PC = [Comp]

type Kont = PC
type Cont = Kont
```

Figure 2: Structure of continuations for $\mathcal{L}_{ap}$: a multiset of computations.

```
type Ch = String
data C  = Snd Ch N | Rcv Ch V

data X  = Skip | Prefix A X | If B X X | Par X X | Ned [(C, X)]
        | LetRec Y X X | Call Y
```

The first difference in the semantics of $\mathcal{L}_{ap}$ w.r.t. $\mathcal{L}_{seq}$ is that there are now two kinds of computations, waiting to be executed: normal computations, that will eventually perform an elementary action as in $\mathcal{L}_{seq}$, and *communication attempts*, i.e. computations guarded by communication primitives. The guards are semantically modeled by the data type *SemC*, which contains the name of the channel and either the meaning of the expression to be sent or the name of the variable in which the received value will be assigned.

```
data SemC = SemSnd Ch (S → Val) | SemRcv Ch V
data Comp = Den D | Sync [(SemC, D)]
```

The structure of continuations has to change, in order to model the semantics of $\mathcal{L}_{ap}$. Continuations are now parallel computations, i.e. multisets of simple computations as depicted in Figure 2. One element of the multiset is taken to be the *active computation*; in our implementation, this is the head of the list.

As already explained in section 2.2, a CSC-based semantic interpreter implements an "evaluate-normalize-schedule" loop. However, a more sophisticated *scheduler* is now necessary. The data type *Sched* models the options that the scheduler can select from. An element of the form *Scheda d c* is an *activation schedule*: a normal computation d with continuation c. On the other hand, an element of the form *Scheds v pe c* is a *synchronization schedule*: a successful communication attempt, where *pe* is the meaning of the expression whose value is sent, v is the variable in which the received value is assigned and c is the future continuation. Several auxiliary functions are needed by the scheduler, which is implemented in function *kc*:

- *actc* decomposes a closed continuation and finds all activable computations and the corresponding open continuations;
- *scheda* uses *actc* to find all possible activations schedules;
- *sync* synchronizes two independent closed continuations; and
- *scheds* uses *sync* to find all possible synchronization schedules.

The implementation of the scheduler and the auxiliary functions is given below.

```
data Sched = Scheda D Cont | Scheds V (S → Val) Kont

kc :: Kont → M Q
kc k = case scheda k ++ scheds k of
         []    → return Deadlock
```

```
        ws   → bigned (map exe ws)
  where exe (Scheda d c')    =  d c'
        exe (Scheds v pe k') =  rdState k >>= \s →
                                inState (upd v (pe s) s) kc k'

scheda :: Kont → [Sched]
scheda k = [Scheda d c | (Den d, c) ← actc k]

actc :: Kont → [(Comp, Cont)]
actc = ms

scheds :: Kont → [Sched]
scheds k = [w | (ps, pc) ← ms k, w ← sync [ps] pc]

sync :: Kont → Kont → [Sched]
sync k1 k2 = [Scheds v pe (re (addc (Den d1) c1) ++
                           re (addc (Den d2) c2))
             | (Sync snd, c1)        ← actc k1,
               (Sync rcv, c2)        ← actc k2,
               (SemSnd γs pe, d1)    ← snd,
               (SemRcv γr v, d2)     ← rcv,
               γs == γr]
```

The continuation completion function remains as in section 2.2. Also, the normalization function (which transforms an open continuation into a corresponding closed continuation) is again defined as the identity function on multisets.

```
re :: Cont → Kont
re = id
```

Finally, the semantics of $\mathcal{L}_{ap}$ concludes with the meanings of the new kinds of statements. The semantics of communication primitives is defined at the same time. The semantics of parallel composition is based on the non-deterministic choice between two alternative computations: one starting from the first statement and another starting from the second. It uses the control operator *addp* which adds a new computation to be executed in parallel with a given continuation.[15] The evaluator functions for $\mathcal{L}_{ap}$ are as follows:

```
addp :: Comp → Cont → Cont
addp p pc = p : pc

sem (Prefix a x) e c = semA a (addc (Den (sem x e)) c)
sem (Par x1 x2)  e c = sem x1 e (addp (Den (sem x2 e)) c) `ned`
                       sem x2 e (addp (Den (sem x1 e)) c)
sem (Ned gx)     e c = cc (addc p c)
  where p              = Sync [(semC g, sem x e) | (g, x) ← gx]
        semC           :: C → SemC
        semC (Snd γ e) = SemSnd γ (evN e)
        semC (Rcv γ v) = SemRcv γ v
```

[15] In the simple setting of $\mathcal{L}_{ap}$, where a general operator for sequential composition is missing, the control operator *addc* behaves the same as *addp* (*addc* remains as in section 2.2). In the languages that we study in the following sections we will encounter more complex combinations of sequential and parallel compositions.

All evaluator functions operate upon open continuations. Recall that the type of the semantic function is $sem :: X \to Env \to D$, where $D = Cont \to M\ Q$ and $Cont$ is the domain of open continuations in the semantic models designed with CSC.

As an example of $\mathcal{L}_{ap}$, consider the following program. Procedure y_1 is the producer, while procedure y_2 is the consumer. Variable v receives the communicated value and both y_1 and y_2 operate as long as v is positive. At each turn, y_1 decrements the value of v with probability $2/3$ or increments it with probability $1/3$. Under a fair implementation of non-deterministic choice, the program is bound to terminate.

$$
\begin{array}{lllll}
\textbf{letrec} & y_1 = \textbf{if}\ 0 < v\ \textbf{then} & \textbf{ned}\ [\ \gamma\,!\,(v-1) & \to & \textbf{call}\ y_1 \\
 & & \mid \gamma\,!\,(v+1) & \to & \textbf{call}\ y_1 \\
 & & \mid \gamma\,!\,(v-1) & \to & \textbf{call}\ y_1\ ] \\
 & \textbf{else} & \textbf{skip} & & \\
\textbf{in letrec} & y_2 = \textbf{if}\ 0 < v\ \textbf{then} & \textbf{ned}\ [\ \gamma\,?\,v & \to & \textbf{write}\ v \cdot \textbf{call}\ y_2\ ] \\
 & \textbf{else} & \textbf{skip} & & \\
\textbf{in} & v := 10 \cdot \textbf{write}\ v \cdot (\textbf{call}\ y_1 \parallel \textbf{call}\ y_2) & & &
\end{array}
$$

At consecutive executions, our semantic interpreter can produce different execution traces if the program is non-deterministic and the random number generator is initialized with different seeds. A possible execution trace for this program, obtained by taking $c_0 = [\,]$ is:

$[10, 9, 10, 11, 10, 9, 8, 7, 8, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0]$

By employing the powerdomain monad given in appendix A instead of the monad M defined in this section, we obtain a classic denotational model for $\mathcal{L}_{ap}$. Unfortunately, the meaning of the previous program in this model is an infinite set of different possible traces, produced one by one. Such a model is only useful to verify toy programs such as:

$(\textbf{write}\ 1 \cdot \textbf{write}\ 2 \cdot \textbf{skip}) \ \parallel\ (\textbf{write}\ 3 \cdot \textbf{skip})$

which produces the following set of traces:

$\{\!|\ [1,2,3], [3,1,2], [1,3,2]\ |\!\}$

As a second small example, consider the following program:

$$
\begin{array}{ll}
 & \textbf{ned}\ [\ \gamma\,!\,1 \to \textbf{ned}\ [\ \gamma\,?\,v \to \textbf{skip}\ ]\ \mid\ \gamma\,!\,2 \to \textbf{write}\ 3 \cdot \textbf{skip}\ ] \\
\parallel & \textbf{ned}\ [\ \gamma\,?\,v \to \textbf{write}\ v \cdot \textbf{skip}\ ]
\end{array}
$$

The set of all possible traces that is produced is given below.

$\{\!|\ [1, \mathit{deadlock}], [3,2], [2,3]\ |\!\}$

3.2 Process Creation

The language $\mathcal{L}_{new}$ takes a different approach to concurrency from $\mathcal{L}_{ap}$. Action prefixing is replaced by sequential composition in general, but parallel composition is replaced with a **new** construct, that allows the *creation of new processes*. Such processes can communicate with each other but, as in $\mathcal{L}_{ap}$, their execution cannot "meet". The full abstract syntax of $\mathcal{L}_{new}$ is given below:

```
type SC = [Comp]
type PC = [SC]

type Kont = PC
type Cont = Kont
```

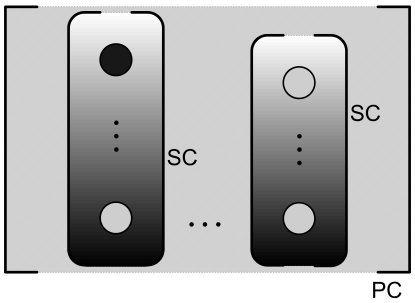


Figure 3: Structure of continuations for $\mathcal{L}_{new}$: a multiset of stacks of computations.

$$x \ ::= \ \textbf{skip} \mid a \mid \textbf{if}\ b\ \textbf{then}\ x\ \textbf{else}\ x \mid x\ ;\ x \mid \textbf{new}\ x \mid \textbf{ned}\ [\ (g \rightarrow x)^*\ ]$$
$$\mid \ \textbf{letrec}\ y = x\ \textbf{in}\ x \mid \textbf{call}\ y$$

and its implementation in Haskell is very straightforward.

```
data X = Skip | Action A | If B X X | Seq X X | New X | Ned [(C, X)]
       | LetRec Y X X | Call Y
```

Just one modification in the semantic domains of $\mathcal{L}_{ap}$ is necessary, to model the behavior of concurrent programs in $\mathcal{L}_{new}$. Continuations now can be structured as multisets of stacks of computations: each stack (SC) models a sequential composition that takes place in a single process, whereas the multiset (PC) contains all processes that are executed in parallel (see Figure 3). A specific element of this multiset is designed to be the *active process*; in our implementation of open continuations, we always assume that the active process is the head of the list representing PC. The top element in the stack representing the active process is the active computation.

The definition of *actc* (the auxiliary function used by the scheduler to find all activable computations) must change, as all processes of the multiset are candidates for activation.

```
actc k = [(p, sc : pc) | (p : sc, pc) ← ms k]
```

Furthermore, the normalization function *re* must make sure that a continuation does not contain any empty SC.[16]

```
re ([] : pc) = pc
re pc        = pc
```

In the evaluator, function *addc* also needs to be modified, so as to prepend a computation to the active process. A new function *addn* is defined, to add a computation as a new process to the multiset.

```
addc :: Comp → Cont → Cont
addc p (sc : pc) = (p : sc) : pc

addn :: Comp → Cont → Cont
addn p (sc : pc) = sc : [p] : pc
```

The semantics of sequential composition and process creation are easily defined in terms of *addc* and *addn*.

[16] An invariant of the new structure of continuations in our semantics for $\mathcal{L}_{new}$ is that an empty SC can only occur as the active process of a PC.

$sem\ (Seq\ x_1\ x_2)\ e\ c = sem\ x_1\ e\ (addc\ (Den\ (sem\ x_2\ e))\ c)$
$sem\ (New\ x)\quad e\ c = cc\ (addn\ (Den\ (sem\ x\ e))\ c)$

The following example program in $\mathcal{L}_{new}$ is very similar to that of section 3.1; the only difference is in the use of **new** instead of parallel composition for running the producer and the consumer in parallel. The empty continuation is appropriately redefined as $c_0 = [[]]$.)

$$\begin{array}{lllll}
\textbf{letrec} & y_1 = \textbf{if}\ 0 < v\ \textbf{then} & \textbf{ned}\ [\ \gamma\,!\,(v-1) & \rightarrow & \textbf{call}\ y_1 \\
 & & \mid \gamma\,!\,(v+1) & \rightarrow & \textbf{call}\ y_1 \\
 & & \mid \gamma\,!\,(v-1) & \rightarrow & \textbf{call}\ y_1\ ] \\
 & \textbf{else} & \textbf{skip} & & \\
\textbf{in letrec} & y_2 = \textbf{if}\ 0 < v\ \textbf{then} & \textbf{ned}\ [\ \gamma\,?\,v & \rightarrow & \textbf{write}\ v; \textbf{call}\ y_2\ ] \\
 & \textbf{else} & \textbf{skip} & & \\
\textbf{in} & v := 10; \textbf{write}\ v; \textbf{new}\ (\textbf{call}\ y_1); \textbf{call}\ y_2 & & &
\end{array}$$

It is little surprising that the produced trace is identical to that of section 3.1.

$[10, 9, 10, 11, 10, 9, 8, 7, 8, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0]$

3.3 Parallel and Sequential Composition

In a language with unrestricted parallel and sequential composition, like $\mathcal{L}_{ps}$ that we consider in this section, the major issue is the presence of statements such as $(x_1 \parallel x_2)\,; x_3$. In such a statement, x_3 can only execute after *both* x_1 and x_2 have terminated. This "meeting" of computations is not possible in the previous languages. Let us consider the language $\mathcal{L}_{ps}$, whose syntax contains again the non-deterministic choice and synchronous communication of $\mathcal{L}_{ap}$ and a combination of parallel and sequential composition.

$$\begin{array}{rcl}
x & ::= & \textbf{skip} \mid a \mid \textbf{if}\ b\ \textbf{then}\ x\ \textbf{else}\ x \mid x\,;x \mid x \parallel x \mid \textbf{ned}\ [\ (g \rightarrow x)^*\ ] \\
 & \mid & \textbf{letrec}\ y = x\ \textbf{in}\ x \mid \textbf{call}\ y
\end{array}$$

The same syntax in Haskell is given below.

$$\begin{array}{l}
\textbf{data}\ X = Skip \mid Action\ A \mid If\ B\ X\ X \mid Seq\ X\ X \mid Par\ X\ X \mid Ned\ [(C, X)] \\
\qquad \mid LetRec\ Y\ X\ X \mid Call\ Y
\end{array}$$

A richer structure for continuations is required for $\mathcal{L}_{ps}$. A combination of parallel and sequential computation is necessary, precisely in this order: a parallel computation is first executed ($x_1 \parallel x_2$ in our example) and, upon its termination, a sequence of computations that was blocked up to that point (x_3) is launched. This leads us to the structure of a *ps*-tree (see Figure 4). The shape of a *ps*-tree is inspired from the structure of a *cactus stack* [Bobr73].

Every node in a *ps*-tree consists of one *PC* and one *SC* with the aforementioned semantics. A *PC* is a multiset of *ps*-trees or, in other words, a *ps*-forest. In a *ps*-tree, parallel levels (i.e. *PC* multisets) alternate with sequential levels (i.e. *SC* stacks). The active computation is (conceptually) a *leaf* in a *ps*-forest. In our implementation of an open continuation, we take this leaf to be the leftmost one. The semantic operators on continuation structures are designed in such a way as to maintain the following *invariant* of *ps*-trees: every *SC* stack is non-empty, with the possible exception of the leftmost one (the active stack), which conceptually contains at its head the active computation. When the active *SC* becomes empty, it is removed by the normalization function.

```
type SC = [Comp]
data PS = Meet PC SC
type PC = [PS]

type Kont = PC
type Cont = Kont
```

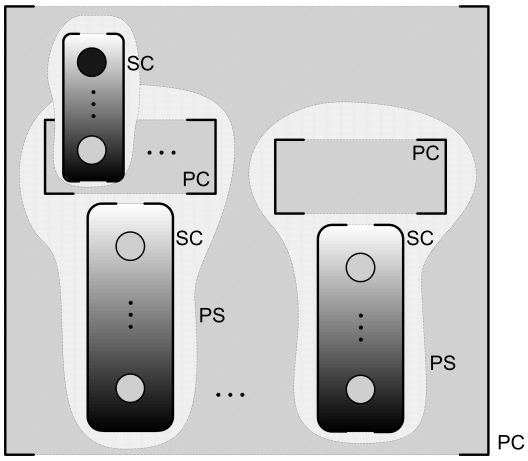


Figure 4: Structure of continuations for $\mathcal{L}_{ps}$: a *ps*-forest with active computations at the leaves.

$$
\begin{aligned}
re\ (Meet\ [\,]\ [\,] : pc) &= pc \\
re\ (Meet\ [\,]\ sc : pc) &= Meet\ [\,]\ sc : pc \\
re\ (Meet\ pc_0\ sc : pc) &= Meet\ (re\ pc_0)\ sc : pc
\end{aligned}
$$

Furthermore, part of the scheduler must be redefined. The two auxiliary functions *actc* and *scheds* now become:

$$
\begin{aligned}
&actc\ k = [w \mid (ps, pc) \leftarrow ms\ k, w \leftarrow aux\ ps\ pc] \\
&\quad \textbf{where}\ aux\ (Meet\ [\,]\ (p : sc))\ pc = [(p, Meet\ [\,]\ sc : pc)] \\
&\qquad\qquad aux\ (Meet\ pc_0\ sc)\ \ pc = [(p, Meet\ pc_0'\ sc : pc) \\
&\qquad\qquad\qquad\qquad\qquad\qquad \mid (p, pc_0') \leftarrow actc\ pc_0]
\end{aligned}
$$

$$
\begin{aligned}
&scheds\ k = [w \mid (ps, pc) \leftarrow ms\ k, w \leftarrow sync\ [ps]\ pc +\!\!+ aux\ ps\ pc] \\
&\quad \textbf{where}\ aux\ (Meet\ pc_0\ sc)\ pc = \\
&\qquad\qquad [Scheds\ v\ pe\ (Meet\ pc_0'\ sc : pc) \\
&\qquad\qquad\quad \mid Scheds\ v\ pe\ pc_0' \leftarrow scheds\ pc_0]
\end{aligned}
$$

Modifications are also required in the evaluator, in functions *addc* and *addp*. The former simply adds a computation on top of the active stack. The latter creates a branching point on top of the active stack; an example of its operation is shown in Figure 5.

$$
\begin{aligned}
addc\ p\ (Meet\ [\,]\ sc : pc) &= Meet\ [\,]\ (p : sc) : pc \\
addc\ p\ (Meet\ pc_0\ sc : pc) &= Meet\ (addc\ p\ pc_0)\ sc : pc
\end{aligned}
$$

$$
\begin{aligned}
addp\ p\ (Meet\ [\,]\ [\,] : pc) &= Meet\ [\,]\ [\,] : Meet\ [\,]\ [p] : pc \\
addp\ p\ (Meet\ [\,]\ sc : pc) &= Meet\ [Meet\ [\,]\ [\,], Meet\ [\,]\ [p]]\ sc : pc \\
addp\ p\ (Meet\ pc_0\ sc : pc) &= Meet\ (addp\ p\ pc_0)\ sc : pc
\end{aligned}
$$

As usual, the semantics of sequential and parallel composition are directly based on *addc* and *addp*.

$$
\begin{aligned}
sem\ (Seq\ x_1\ x_2)\ \ e\ c &= sem\ x_1\ e\ (addc\ (Den\ (sem\ x_2\ e))\ c) \\
sem\ (Par\ x_1\ x_2)\ e\ c &= sem\ x_1\ e\ (addp\ (Den\ (sem\ x_2\ e))\ c)\ `ned` \\
&\qquad\qquad sem\ x_2\ e\ (addp\ (Den\ (sem\ x_1\ e))\ c)
\end{aligned}
$$

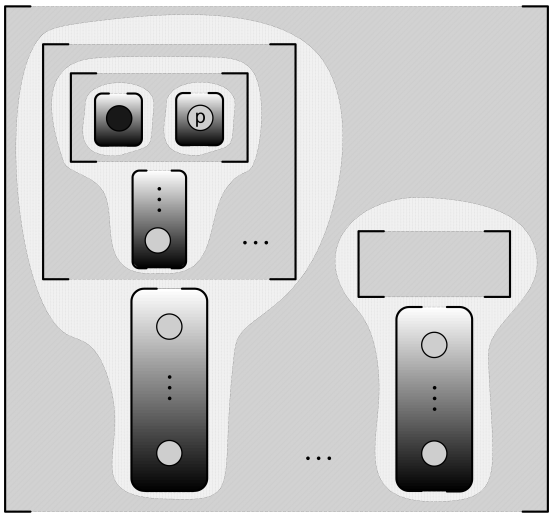


Figure 5: The result of applying *addp* to a process p and the open continuation depicted in Figure 4. Process p is added in parallel to the active computation.

The empty continuation must also change appropriately.

$c_0 = [\mathit{Meet}\ [\,]\ [\,]]$

As a first example, we consider a very simple program:

(**write** 1 $\parallel$ **write** 2); **write** 3

The set of all possible traces that is produced by employing the powerdomain monad given in appendix A contains only traces in which the observable 3 is found after both 1 and 2:

$\{\!\!\{\ [1,2,3],[2,1,3]\ \}\!\!\}$

We now convert the more involved example of section 3.1 using sequential and parallel composition. We add a final output primitive, which will produce 99 as the last observable.

$$
\begin{array}{lllllll}
\textbf{letrec} & y_1 = \textbf{if}\ 0 < v & \textbf{then} & \textbf{ned}\ [\ \gamma\,!\,(v-1) & \rightarrow & \textbf{call}\ y_1 \\
 & & & \quad |\ \gamma\,!\,(v+1) & \rightarrow & \textbf{call}\ y_1 \\
 & & & \quad |\ \gamma\,!\,(v-1) & \rightarrow & \textbf{call}\ y_1\ ] \\
 & & \textbf{else} & \textbf{skip} & & \\
\textbf{in letrec} & y_2 = \textbf{if}\ 0 < v & \textbf{then} & \textbf{ned}\ [\ \gamma\,?\,v & \rightarrow & \textbf{write}\ v; \textbf{call}\ y_2\ ] \\
 & & \textbf{else} & \textbf{skip} & & \\
\textbf{in} & v := 10; \textbf{write}\ v; (\textbf{call}\ y_1 \parallel \textbf{call}\ y_2); \textbf{write}\ 99 & & & &
\end{array}
$$

It is again little surprising that the produced trace in our prototype implementation is identical to that of sections 3.1 and 3.2, with the exception of the final observable.

$[10,9,10,11,10,9,8,7,8,9,8,7,6,5,4,3,2,1,0,99]$

As a last example, we present a program with two meeting points. The first group of parallel processes print a series of 1 and 2. Then a first meeting point is reached and 1000 is printed. Subsequently, the second group of parallel processes print a series of 3 and 4 and a second meeting point is reached. Finally, the number 2000 is printed.

$$
\begin{array}{ll}
\textbf{letrec} & y_1 = \textbf{if } 0 < v \textbf{ then } (\textbf{write } 1; \textbf{ call } y_1) \textbf{ else skip} \\
\textbf{in letrec} & y_2 = \textbf{if } 0 < v \textbf{ then } (\textbf{write } 2; \textbf{ call } y_2) \textbf{ else skip} \\
\textbf{in letrec} & y_3 = \textbf{if } 0 < v \textbf{ then } (\textbf{write } 3; \textbf{ call } y_3) \textbf{ else skip} \\
\textbf{in letrec} & y_4 = \textbf{if } 0 < v \textbf{ then } (\textbf{write } 4; \textbf{ call } y_4) \textbf{ else skip} \\
\textbf{in letrec} & y_c = \textbf{if } 0 < v \textbf{ then } (v := v - 1; \textbf{ call } y_c) \textbf{ else skip} \\
\textbf{in} & v := 10; (\textbf{call } y_1 \parallel \textbf{call } y_2 \parallel \textbf{call } y_c); \textbf{ write } 1000; \\
& v := 10; (\textbf{call } y_3 \parallel \textbf{call } y_4 \parallel \textbf{call } y_c); \textbf{ write } 2000;
\end{array}
$$

The produced trace shows precisely what is described above.

$$[1,1,2,2,1,2,1,1,1,1,2,1,2,1,2,1,1000,4,4,3,4,4,3,3,4,3,4,4,4,4,3,3,2000]$$

3.4 Rendez-vous

The language $\mathcal{L}_{rv}$ defined in this section replaces the CSP-like synchronous synchronization with a simplified version of the *rendez-vous* mechanism that is typical in Ada and POOL.

$$
\begin{array}{rcl}
x & ::= & \textbf{skip} \mid a \mid \textbf{if } b \textbf{ then } x \textbf{ else } x \mid x \,;\, x \mid \textbf{new } x \mid \textbf{ask } y \mid \textbf{answer } [\, y^* \,] \\
& \mid & \textbf{letrec } y = x \textbf{ in } x \mid \textbf{call } y
\end{array}
$$

The syntax in Haskell is given below.

$$
\begin{array}{l}
\textbf{data } X = \mathit{Skip} \mid \mathit{Action}\ A \mid \mathit{If}\ B\ X\ X \mid \mathit{Seq}\ X\ X \mid \mathit{New}\ X \mid \mathit{Ask}\ Y \mid \mathit{Answer}\ [\,Y\,] \\
\qquad\qquad \mid \mathit{LetRec}\ Y\ X\ X \mid \mathit{Call}\ Y
\end{array}
$$

Before we explain the semantics of rendez-vous, we provide a brief, intuitive description. In client-server terminology, **ask** y is executed by a client requesting a service, y is the name of the service, **answer** $[ys]$ is executed by a server providing a number of services and ys is the set of provided services.

A statement of the form **ask** y requests the invocation of procedure y. In POOL terminology, y is a *method* of some named *object*; in the simplified view of $\mathcal{L}_{rv}$, procedures are methods and processes are unnamed objects. The request **ask** y can be answered by any process that executes a statement of the form **answer** $[ys]$, where ys is a sequence of procedure variables and $y \in ys$. The sequence of processes ys determines what this process can invoke. A process executing **answer** $[ys]$ is blocked, until an **ask** request is made by some other process. When such a rendez-vous between two processes succeeds, the specified procedure is executed; both processes are suspended and only proceed when (and if) the procedure's execution finishes.

The first change that is required in the semantics of $\mathcal{L}_{rv}$ concerns the domain *Comp*. Two new kinds of computation must be added: $\mathit{Ask}\ y$ is a computation that requests invocation of procedure y and $\mathit{Answer}\ yds$ is a computation that waits for an invocation request, where yds is a list of pairs consisting of procedure names and procedure denotations.

$$\textbf{data } \mathit{Comp} = \mathit{Den}\ D \mid \mathit{Ask}\ Y \mid \mathit{Answer}\ [(Y, D)]$$

Furthermore, the structure of continuations must be adapted to the rendez-vous concept. A combination of sequential and parallel computation is again necessary, but a rendez-vous must be seen as the opposite of the "meeting" that we discussed in section 3.3. When a rendez-vous succeeds, a sequential computation is first executed (the invoked procedure) and, upon its termination, two processes (the asking and answering processes) that were blocked up to that point continue their execution in parallel. This leads again to a structure very

```
type SC = [Comp]
data PS = Fork SC PC
type PC = [PS]

type Kont = PC
type Cont = Kont
```

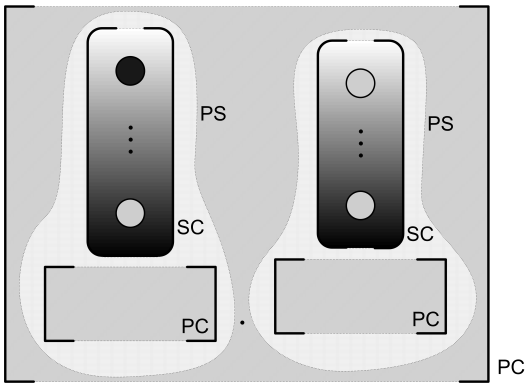


Figure 6: Structure of continuations for $\mathcal{L}_{rv}$: a *ps*-forest with active computations at the root.

similar to the *ps*-tree of section 3.3, but now the order of sequential and parallel levels of computation is reversed (see Figure 6). The active computation is now (conceptually) at the *root* of the the *ps*-tree.

In the semantics of $\mathcal{L}_{rv}$, the domain *Sched* is simplified but the scheduler functions become significantly more complex. The next step that is scheduled can be either an elementary action (scheduled by *scheda*) or a successful rendez-vous (scheduled by *scheds*).

```
data Sched = Sched D Cont

scheda :: Kont → [Sched]
scheda k = [Sched d c | (Den d, c) ← actc k]

scheds :: Kont → [Sched]
scheds k = [w | (ps, pc) ← ms k, w ← sync [ps] pc]

kc :: Kont → M Q
kc k = case scheda k ++ scheds k of
         []   → return Deadlock
         ws   → bigned (map exe ws)
  where exe (Sched d c') = d c'
```

The most interesting part in the semantics of $\mathcal{L}_{rv}$ is function *sync* which implements the rendez-vous synchronization.

```
sync k1 k2 = [Sched d (Fork [] (re [ps1] ++ re [ps2]) : pc1 ++ pc2)
               | (Ask y1, ps1 : pc1)        ← actc k1,
                 (Answer yds, ps2 : pc2)    ← actc k2,
                 (y2, d)                    ← yds,
                 y1 == y2]
```

The definition of *actc* is simpler again, as active computations are found at the root of the *ps*-tree.

```
actc :: Kont → [(Comp, Cont)]
actc k = [(p, Fork sc pc0 : pc) | (Fork (p : sc) pc0, pc) ← ms k]
```

The normalization function again makes sure that when the active SC stack becomes empty, it is removed from the *ps*-tree.

$re :: Cont \to Kont$
$re\ (Fork\ [\,]\ pc_0 : pc) = pc_0 +\!\!+ pc$
$re\ (Fork\ sc\ pc_0 : pc) = Fork\ sc\ pc_0 : pc$

The semantics of `ask` and `answer` simply use *addc* to add the corresponding rendez-vous attempts to the continuation. The control operator *addc* adds a computation on top of the active stack, which is at the root of the leftmost *ps*-tree in our implementation.

$addc\ p\ (Fork\ sc\ pc_0 : pc) = Fork\ (p : sc)\ pc_0 : pc$

$sem\ (Ask\ y)\qquad e\ c = cc\ (addc\ (Ask\ y)\ c)$
$sem\ (Answer\ ys)\ e\ c = cc\ (addc\ (Answer\ [(y, e\ y) \mid y \leftarrow ys])\ c)$

A straightforward modification is also required in the control operator *addn* of section 3.2.

$addn\ p\ (ps : pc) = ps : Fork\ [p]\ [\,] : pc$

The empty continuation c_0 must also change appropriately.

$c_0 = [Fork\ [\,]\ [\,]]$

A small program in $\mathcal{L}_{rv}$ will serve as our first example. In the following program, two processes synchronize upon the invocation of procedure m.

letrec $m =$ **write** 7; **write** 42
in **new** (**write** 1; **ask** m; **write** 3);
write 2; **answer** $[m]$; **write** 4

The set of all possible traces that is produced by employing the powerdomain monad given in appendix A is the following. All traces contain the subsequence $[7, 42]$ which is obtained when the two processes are synchronized and procedure m is called. Before this subsequence there are the observables 1 and 2, in any order, and after this subsequence there are the observables 3 and 4, again in any order.

$\{\!\!\{\ [2,1,7,42,3,4], [1,2,7,42,3,4], [2,1,7,42,4,3], [1,2,7,42,4,3]\ \}\!\!\}$

The following example program demonstrates the synchronization capabilities of $\mathcal{L}_{rv}$. It uses a semaphore, modeled by procedure *sema*, to implement mutual exclusion between two procedures, y_1 and y_2. Procedure *sema* acts as a monitoring process. Procedures *inc* and *dec* are the semaphore's basic methods and *done* is a method for signaling that the semaphore is no more necessary. Notice that when the value of variable v is zero, the semaphore does not answer to *dec*.

$$
\begin{array}{lll}
\textbf{letrec} & \mathit{inc} & = v := v + 1;\ \textbf{write}\ v \\
\textbf{in letrec} & \mathit{dec} & = v := v - 1; \textbf{write}\ v \\
\textbf{in letrec} & \mathit{done} & = m := m - 1;\ \textbf{write}\ 99 \\
\textbf{in letrec} & \mathit{sema} & = \textbf{if}\ 0 < v\ \ \textbf{then}\ \ \textbf{answer}\ [\mathit{done}, \mathit{inc}, \mathit{dec}] \\
 & & \qquad\qquad\ \ \textbf{else}\ \ \textbf{answer}\ [\mathit{done}, \mathit{inc}]; \\
 & & \ \ \textbf{if}\ 0 < m\ \textbf{then call}\ \mathit{sema}\ \textbf{else skip} \\
\textbf{in letrec} & y_1 & = \textbf{if}\ 0 < n\ \ \textbf{then}\ \ \textbf{ask}\ \mathit{dec};\ n := n - 1;\ \textbf{write}\ 42; \\
 & & \qquad\qquad\ \ \textbf{ask}\ \mathit{inc};\ \textbf{call}\ y_1 \\
 & & \qquad\qquad\ \ \textbf{else}\ \ \textbf{ask}\ \mathit{done} \\
\textbf{in letrec} & y_2 & = \textbf{if}\ 0 < n\ \ \textbf{then}\ \ \textbf{ask}\ \mathit{dec};\ n := n - 1;\ \textbf{write}\ 7; \\
 & & \qquad\qquad\ \ \textbf{ask}\ \mathit{inc};\ \textbf{call}\ y_2 \\
 & & \qquad\qquad\ \ \textbf{else}\ \ \textbf{ask}\ \mathit{done} \\
\textbf{in} & & v := 1;\ n := 10;\ \textbf{new}\ (\textbf{call}\ y_1);\ \textbf{new}\ (\textbf{call}\ y_2); \\
 & & m := 2;\ \textbf{call}\ \mathit{sema}
\end{array}
$$

In our prototype implementation, the produced trace is the following. It is not coincidental that y_1 and y_2 tend to alternate; given a fair implementation of non-deterministic choice, alternation is the highest probability scenario.

$$[0, 42, 1, 0, 7, 1, 0, 42, 1, 0, 42, 1, 0, 7, 1, 0, 42, 1, 0, 7, 1, 0, 42, 1, 0, 7, 1, 0, 7, 1, 0, 99, 42, 1, 99]$$

3.5 The Await Statement

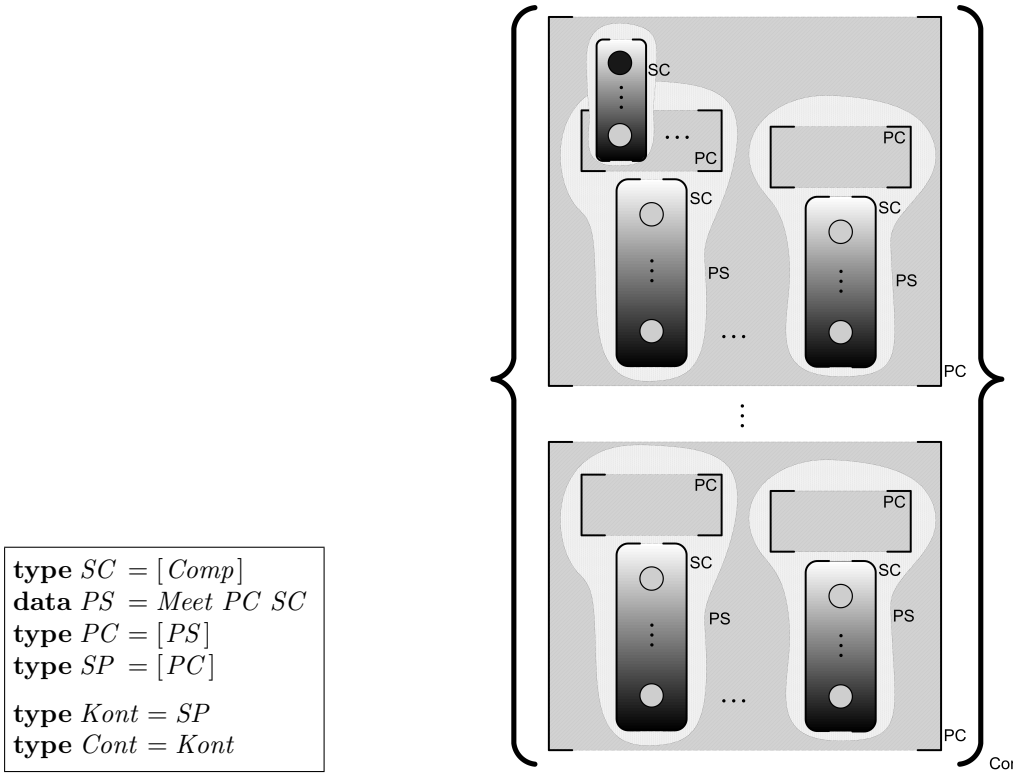
Let us now consider the language $\mathcal{L}_{aw}$ featuring parallel, sequential composition and the **await** statement as a means of suspension. This suspension mechanism may be considered as an instance of a general paradigm for asynchronous communication [dBoe93]. We expect that other instances of this paradigm can be handled similarly. We omit synchronous communication from the syntax of $\mathcal{L}_{aw}$, mostly for reasons of simplicity.

$$
\begin{array}{rcl}
x & ::= & \textbf{skip} \mid a \mid \textbf{if}\ b\ \textbf{then}\ x\ \textbf{else}\ x \mid x\ ;\ x \mid x \parallel x \mid \textbf{await}\ b\ \textbf{then}\ x \\
 & \mid & \textbf{letrec}\ y = x\ \textbf{in}\ x \mid \textbf{call}\ y
\end{array}
$$

According to the semantics of **await** b **then** x, as defined by Owicki and Gries [Owic76], execution is blocked until b is true. Subsequently, the body x is executed *indivisibly*, as if it were an elementary action. The syntax of $\mathcal{L}_{aw}$ in Haskell is given below.

```
data X = Skip | Action A | If B X X | Seq X X | Par X X | Await B X
       | LetRec Y X X | Call Y
```

The atomization of an **await** statement's body is what requires a redesign in the semantics of $\mathcal{L}_{aw}$, w.r.t. $\mathcal{L}_{ps}$. Yet a richer structure is necessary allowing computations to be executed indivisibly. In the work of Owicki and Gries, the body of **await** was restricted to a simple sequential computation that was bound to terminate. We have no reason to impose such a restriction: in $\mathcal{L}_{aw}$, the body of **await** is unrestricted. If the guarding condition is true, then all parallel processes are suspended and the computation of the body takes place *in isolation*, as if it was the only statement in the whole program. Only when (and if) it terminates do the suspended processes continue their execution. Moreover, $\mathcal{L}_{aw}$ allows **await** statements to be nested; in this case there may be several "layers" of suspended parallel processes, each layer waiting for the body of some nested **await** to finish.



type $SC = [Comp]$
data $PS = Meet\ PC\ SC$
type $PC = [PS]$
type $SP = [PC]$

type $Kont = SP$
type $Cont = Kont$

Figure 7: Structure of continuations for $\mathcal{L}_{aw}$: a stack of *ps*-forests with active computations at the leaves.

With all this in mind, the obvious model of continuations in $\mathcal{L}_{aw}$ is a *stack* of *ps*-forests. The top element of the stack is the multiset of parallel computations (*ps*-trees) that is currently executed. All other elements of the stack contain various "layers" of suspended processes, waiting their turn until the upper elements complete their execution. Figure 7 shows this structure. The active computation is found at the *leaves* of the stack's top element.

Apart from the different continuation structure, the semantics of $\mathcal{L}_{aw}$ is a direct adaptation of $\mathcal{L}_{ps}$, with one exception. A new kind of computation for **await** statements waiting for their guarding condition to become true is added in domain *Comp*: in $Await\ f\ d$, f is the meaning of the condition and d is the denotation of the body.

data $Comp = Den\ D \mid Await\ (S \to Bool)\ D$

The scheduler operation for synchronization has a dual rôle: to find all activation schedules that can resume their execution, including **await** statements with a true guarding condition, and to suspend all processes if the body of an **await** statement is scheduled for execution. In order to evaluate guarding conditions, *scheds* needs to access the state, which must be passed by the scheduler.

data $Sched = \ Sched\ D\ Cont$

$scheda :: Kont \to [Sched]$
$scheda\ k = [Sched\ d\ c \mid (Den\ d, c) \leftarrow actc\ k]$

$$
\begin{array}{l}
scheds :: Kont \to S \to [Sched] \\
scheds\ k\ s = [Sched\ d\ ([Meet\ [\,]\ [\,]] : re\ c) \\
\qquad\qquad\qquad\qquad \mid (Await\ f\ d, c) \leftarrow actc\ k, f\ s]
\end{array}
$$

$$
\begin{array}{l}
kc :: Kont \to M\ Q \\
kc\ k = rdState\ k >\!\!>\!\!= \backslash s \to \\
\qquad \textbf{case}\ scheda\ k +\!\!+ scheds\ k\ s\ \textbf{of} \\
\qquad\quad [\,] \quad \to return\ Deadlock \\
\qquad\quad ws \quad \to bigned\ (map\ exe\ ws) \\
\quad \textbf{where}\ exe\ (Sched\ d\ c) = d\ c
\end{array}
$$

$$
\begin{array}{l}
actc :: Kont \to [(Comp, Cont)] \\
actc\ (pc : c) = [(p, pco : c) \mid (p, pco) \leftarrow paux\ pc] \\
\quad \textbf{where}\ paux\ pc = [w \mid (ps, pc') \leftarrow ms\ pc, w \leftarrow aux\ ps\ pc'] \\
\qquad aux\ (Meet\ [\,]\ (p : sc))\ pc = [(p, Meet\ [\,]\ sc : pc)] \\
\qquad aux\ (Meet\ pc_0\ sc)\ pc = [(p, Meet\ pc_0'\ sc : pc) \\
\qquad\quad \mid (p, pc_0') \leftarrow paux\ pc_0]
\end{array}
$$

The remaining changes to the normalization procedure and the evaluator are necessary to restrict the focus to the stack's bottom layer. As a general rule, the revised functions apply their counterparts of $\mathcal{L}_{ps}$ to the stack's first element and leave the other elements unaffected.

The revised normalization procedure for $\mathcal{L}_{aw}$ is given below.

$$
\begin{array}{l}
re :: Cont \to Kont \\
re\ (pco : c) = \textbf{case}\ aux\ pco\ \textbf{of} \\
\quad [\,] \to c \\
\quad pc \to pc : c \\
\quad \textbf{where}\ aux\ (Meet\ [\,]\ [\,] : pc) \quad = pc \\
\qquad\qquad aux\ (Meet\ [\,]\ sc : pc) \quad = Meet\ [\,]\ sc : pc \\
\qquad\qquad aux\ (Meet\ pc_0\ sc : pc) = Meet\ (aux\ pc_0)\ sc : pc
\end{array}
$$

Furthermore, the evaluator's functions *addc* and *addp* must be revised as follows.

$$
\begin{array}{l}
addc\ p\ (pco : c) = aux\ p\ pco : c \\
\quad \textbf{where}\ aux\ p\ (Meet\ [\,]\ sc : pc) \quad = Meet\ [\,]\ (p : sc) : pc \\
\qquad\qquad aux\ p\ (Meet\ pc_0\ sc : pc) = Meet\ (aux\ p\ pc_0)\ sc : pc
\end{array}
$$

$$
\begin{array}{l}
addp :: Comp \to Cont \to Cont \\
addp\ p\ (pco : c) = aux\ p\ pco : c \\
\quad \textbf{where}\ aux\ p\ (Meet\ [\,]\ [\,] : pc) \quad = Meet\ [\,]\ [\,] : Meet\ [\,]\ [p] : pc \\
\qquad\qquad aux\ p\ (Meet\ [\,]\ sc : pc) \quad = Meet\ [Meet\ [\,]\ [\,], Meet\ [\,]\ [p]]\ sc : pc \\
\qquad\qquad aux\ p\ (Meet\ pc_0\ sc : pc) = Meet\ (aux\ p\ pc_0)\ sc : pc
\end{array}
$$

The semantics of the **await** statement is also straightforward.

$$sem\ (Await\ b\ x)\ e\ c = cc\ (addc\ (Await\ (evB\ b)\ (sem\ x\ e))\ c)$$

The last modification is the redefinition of the empty continuation.

$$
\begin{array}{l}
c_0 :: Cont \\
c_0 = [[Meet\ [\,]\ [\,]]]
\end{array}
$$

Our first example will be a simple program with two **await** statements.

$v := 0$; (($\mathbf{write}\ 0$; $v := 1$; $\mathbf{write}\ 3$) $\|$
($\mathbf{await}\ 2 == v\ \mathbf{then}\ \mathbf{write}\ 2$) $\|$
($\mathbf{await}\ 1 == v\ \mathbf{then}\ (\mathbf{write}\ 1$; $v := 2$; $\mathbf{write}\ 4)$))

The first process prints 0 and sets v to 1, thus releasing the guard of the second **await** statement. When this is executed, it prints 1 and 4 without interruption and sets v to 2, thus releasing the guard of the first **await** statement. When this is executed, it prints 2. Thus, the sequence of observables $[0, 1, 4, 2]$ will occur in this order. It is not known, however, at which point the first process will print 3: it will be after 0 and not between 1 and 4, so there are three possibilities. The set of all possible traces obtained by employing the powerdomain of appendix A shows this:

$\{\!|\ [0, 3, 1, 4, 2], [0, 1, 4, 3, 2], [0, 1, 4, 2, 3]\ |\!\}$

We will now rewrite the example of section 3.4, implementing mutual exclusion using a semaphore. In $\mathcal{L}_{aw}$ it is much more natural to implement the semaphore and it is not necessary to have a monitoring process. We use the always true condition $0 < 1$ in procedure *inc* to ensure that incrementing and output will be indivisible.

$$
\begin{array}{lllll}
\mathbf{letrec} & inc = \mathbf{await}\ 0 < 1\ \mathbf{then}\ (v := v + 1;\ \mathbf{write}\ v) & & & \\
\mathbf{in\ letrec} & dec = \mathbf{await}\ 0 < v\ \mathbf{then}\ (v := v - 1;\ \mathbf{write}\ v) & & & \\
\mathbf{in\ letrec} & y_1 = \mathbf{if}\ 0 < n & \mathbf{then} & \mathbf{call}\ dec;\ n := n - 1;\ \mathbf{write}\ 42; & \\
 & & & \mathbf{call}\ inc;\ \mathbf{call}\ y_1 & \\
 & & \mathbf{else} & \mathbf{skip} & \\
\mathbf{in\ letrec} & y_2 = \mathbf{if}\ 0 < n & \mathbf{then} & \mathbf{call}\ dec;\ n := n - 1;\ \mathbf{write}\ 7; & \\
 & & & \mathbf{call}\ inc;\ \mathbf{call}\ y_2 & \\
 & & \mathbf{else} & \mathbf{skip} & \\
\mathbf{in} & v := 1;\ n := 10;\ (\mathbf{call}\ y_1 \parallel \mathbf{call}\ y_2);\ \mathbf{write}\ 99 & & &
\end{array}
$$

In our prototype implementation, the produced trace is the following. Again, it is not purely coincidental that y_1 and y_2 alternate; different traces can be obtained by using different random number generators. In the appendix, we use this very example to demonstrate the ability of the powerdomain monad to compute all possible execution traces.

$[0, 42, 1, 0, 7, 1, 0, 42, 1, 0, 7, 1, 0, 42, 1, 0, 7, 1, 0, 42, 1, 0, 7, 1, 0, 42, 1, 0, 7, 1, 0, 42, 1, 0, 7, 1, 0, 42, 1, 99]$

4 Continuations for Distributed Computing

All the languages considered in sections 2 and 3 have one thing in common: a *global state*, shared by all concurrent processes. In this section, we explore the possibility of having a *local state* for each process. This is essential for defining the semantics of concurrent programming languages used in distributed computing. Furthermore, we investigate the notion of remote processes as *named objects* and allow object-to-object communication.

We first introduce support for distributed state, which is identical in structure to that of the previous languages. Each process created with **new** and the other object creation constructs introduced in the rest of this section will have its own local state, stored within its semantic representation. This provides a more elegant compositional solution to the problem

of distributed state than, e.g., the one proposed by America [Amer89] for the semantic description of POOL, or by de Bakker and de Vink [dBak96, ch. 13]. In the rest of this section, we refer to such stateful processes as *objects*.

The introduction of distributed state forces a redefinition of monad M. The environment monad component of M, supporting the global state, is now removed and only the state monad component for the random number generator is kept: $M\ a = R \to (a, R)$. The definition of the new M is standard.

$$\textbf{type}\ M\ a = S \to R \to (a, R)$$

$$\begin{aligned}&\textbf{instance}\ \mathit{Monad}\ M\ \textbf{where}\\ &\quad \mathit{return}\ a \qquad\quad = \mathit{InM}\ (\backslash r \to (a, r))\\ &\quad \mathit{InM}\ m >\!\!>\!\!= f = \mathit{InM}\ (\backslash r \to \textbf{let}\ (a, r') = m\ r\ \textbf{in}\ \mathit{unM}\ (f\ a)\ r')\end{aligned}$$

$$\begin{aligned}&\mathit{random} :: M\ \mathit{Int}\\ &\mathit{random} = \mathit{InM}\ (\mathit{Random.next})\end{aligned}$$

$$\begin{aligned}&\mathit{put}\ u\ (\mathit{InM}\ m) \quad = \mathit{InM}\ (\backslash r \to \textbf{let}\ (q, r') = m\ r\ \textbf{in}\ (\mathit{Observe}\ u\ q, r'))\\ &\mathit{display}\ (\mathit{InM}\ m) = \mathit{print}\ (\mathit{fst}\ (m\ r_0))\end{aligned}$$

What is interesting, though, is the redefinition of *rdState* and *inState*. These two operations until now ignored the continuation passed as a parameter and were defined as operations of the monad M; they now cannot be defined unless the structure of continuations is known and their definition is deferred until section 4.1.

4.1 Distributed State

We start with the language $\mathcal{L}_{new}$ and we redefine its semantics, so that a new local state is introduced for each process created with **new**. The structure of the state is identical to that of the previous languages. The resulting language $\mathcal{L}_{ds}$ has exactly the same syntax as $\mathcal{L}_{new}$:

$$\begin{aligned}x\ ::=\ &\textbf{skip}\ \mid\ a\ \mid\ \textbf{if}\ b\ \textbf{then}\ x\ \textbf{else}\ x\ \mid\ x\ ;\ x\ \mid\ \textbf{new}\ x\ \mid\ \textbf{ned}\ [\ (\ g \to x)^*\ ]\\ \mid\ &\textbf{letrec}\ y = x\ \textbf{in}\ x\ \mid\ \textbf{call}\ y\end{aligned}$$

and its implementation in Haskell is the following:

$$\begin{aligned}&\textbf{data}\ X = \mathit{Skip} \mid \mathit{Action}\ A \mid \mathit{If}\ B\ X\ X \mid \mathit{Seq}\ X\ X \mid \mathit{New}\ X \mid \mathit{Ned}\ [(C, X)]\\ &\qquad\quad \mid \mathit{LetRec}\ Y\ X\ X \mid \mathit{Call}\ Y\end{aligned}$$

The local state must be made part of each parallel process. Therefore, in the semantics of $\mathcal{L}_{ds}$ a parallel computation is a multiset of pairs, each having a sequential computation and its local state. Elements of *PS* (processes) can now be thought of as objects, although still unnamed. The structure of continuations is shown in Figure 8. Except for the local state, it is identical to that of Figure 3.

Functions *rdState* and *inState* must access the local state of *active computation*, i.e. the first element of the current continuation.

$$\begin{aligned}&\mathit{rdState}\ (\mathit{Obj}\ \mathit{sc}\ s : \mathit{pc}) \qquad = \mathit{InM}\ (\backslash r \to (s, r))\\ &\mathit{inState}\ s\ f\ (\mathit{Obj}\ \mathit{sc}\ s' : \mathit{pc}) = \mathit{InM}\ (\backslash r \to \mathit{unM}\ (f\ (\mathit{Obj}\ \mathit{sc}\ s : \mathit{pc}))\ r)\end{aligned}$$

The scheduler must also change. Apart from the straightforward changes, having to do with the propagation of the local state, it is worth noticing how function *sync* uses the state of the *sending object* to evaluate the expression whose value must be sent, and then assigns

$$
\begin{array}{l}
\textbf{type } SC = [Comp] \\
\textbf{data } PS = Obj\ SC\ S \\
\textbf{type } PC = [PS] \\
\\
\textbf{type } Kont = PC \\
\textbf{type } Cont = Kont
\end{array}
$$

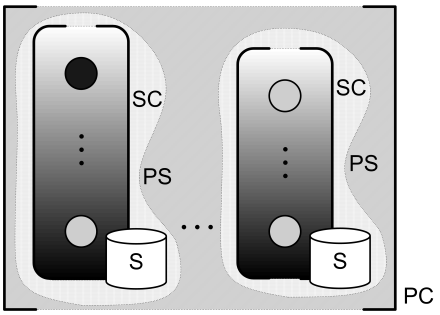


Figure 8: Structure of continuations for $\mathcal{L}_{ds}$: a multiset of stacks of computations, each with its local state.

this to a variable in the state of the *receiving object*. Synchronization schedules are simplified versions of those in section 3.2.

$$
\begin{array}{l}
\textbf{data } Sched = Scheda\ D\ Cont \mid Scheds\ Cont \\
\\
kc\ c = \textbf{case } scheda\ c +\!\!+ scheds\ c \textbf{ of} \\
\qquad\qquad [\,] \quad \rightarrow return\ Deadlock \\
\qquad\qquad sc \quad \rightarrow bigned\ (map\ exe\ sc) \\
\quad \textbf{where } exe\ (Scheda\ d\ c') = d\ c' \\
\qquad\qquad exe\ (Scheds\ c') \quad = kc\ c' \\
\\
actc\ k = [(p, Obj\ sc\ s : pc) \mid (Obj\ (p : sc)\ s, pc) \leftarrow ms\ k] \\
\\
sync\ k_1\ k_2 = \\
\quad [Scheds\ (re\ (addc\ (Den\ d_1)\ (Obj\ sc_1\ s_1 : pc_1)) +\!\!+ \\
\qquad\qquad re\ (addc\ (Den\ d_2)\ (Obj\ sc_2\ (upd\ v\ (pe\ s_1)\ s_2) : pc_2))) \\
\qquad \mid (Sync\ snd, Obj\ sc_1\ s_1 : pc_1) \leftarrow actc\ k_1, \\
\qquad\quad (Sync\ rcv, Obj\ sc_2\ s_2 : pc_2) \leftarrow actc\ k_2, \\
\qquad\quad (SemSnd\ \gamma_s\ pe, d_1) \qquad\quad \leftarrow snd, \\
\qquad\quad (SemRcv\ \gamma_r\ v, d_2) \qquad\quad \leftarrow rcv, \\
\qquad\quad \gamma_s == \gamma_r]
\end{array}
$$

The rest of required changes in the semantics is trivial.

$$
\begin{array}{l}
c_0 = [Obj\ [\,]\ s_0] \\
\\
re\ (Obj\ [\,]\ s : pc) = pc \\
re\ (Obj\ sc\ s : pc) = Obj\ sc\ s : pc \\
\\
addc\ p\ (Obj\ sc\ s : pc) = Obj\ (p : sc)\ s : pc \\
\\
addn\ p\ (ps : pc) = ps : Obj\ [p]\ s_0 : pc
\end{array}
$$

Notice that newly created processes (objects) are equipped with the *initial state* s_0. This does not agree with the behavior of Unix's `fork` system call for process creation, which makes a copy of the parent's local state and equips the child process with that copy. This behavior is possible by changing the last equation with:

$$
addn\ p\ (Obj\ sc\ s : pc) = Obj\ sc\ s : Obj\ [p]\ s : pc
$$

A trivial change is also required in the function *testR* that is used to execute a program and produce a random trace.

$$
\begin{array}{l}
\mathit{testR} :: X \to \mathit{IO}\ () \\
\mathit{testR}\ x = \mathbf{do}\ r_0 \leftarrow \mathit{Random.newStdGen} \\
\qquad\qquad\quad \mathbf{let}\ \mathit{InM}\ m = \mathit{sem}\ x\ e_0\ c_0 \\
\qquad\qquad\quad \mathit{print}\ (\mathit{fst}\ (m\ r_0))
\end{array}
$$

We now provide an example that shows the difference between the global state of $\mathcal{L}_{new}$ and the local state of $\mathcal{L}_{ds}$. Consider the following program:

$$
\begin{array}{l}
v := 7; \\
\mathbf{new}\ (\mathbf{ned}\ [\ \gamma\,!\,42 \to \mathbf{skip}\ ]); \\
\mathbf{new}\ (\mathbf{ned}\ [\ \gamma\,?\,v \to \mathbf{write}\ v\ ]); \\
\mathbf{write}\ v
\end{array}
$$

When we execute this program with the powerdomain monad for local state that is given in appendix A, the following set of traces is produced:

$$\{\!\!\{\ [7, 42], [42, 7]\ \}\!\!\}$$

Notice that the fourth line always produces the observable 7, as the value of its local variable v never changes, and that the third line produces the observable 42 when the processes created by the second and third lines synchronize. The first possible trace occurs when the fourth line is executed before the synchronization, whereas the second trace occurs when it is executed after the synchronization.

If we execute the same program with the semantics of $\mathcal{L}_{new}$, where the state is global, the second trace above is not possible. If the synchronization occurs first, the value of v in the global state becomes 42 and the fourth line produces the observable 42. The set of all possible traces produced by the powerdomain monad with a global state is:

$$\{\!\!\{\ [7, 42], [42, 42]\ \}\!\!\}$$

4.2 Distributed Objects

The language $\mathcal{L}_{obj}$ enhances $\mathcal{L}_{ds}$ with a mechanism for *naming* newly created objects (processes), so that other objects can later "refer" to them. A new class of variables, modeled by the data type W, is used for references to objects. A typical such variable is denoted by w. The syntax of the **new** statement is slightly changed, to support the naming of new objects. To demonstrate the ability of $\mathcal{L}_{obj}$ to refer to such named objects, we replace the communication mechanism through named channels of $\mathcal{L}_{new}$ and $\mathcal{L}_{ds}$ with a new mechanism for synchronous *object-to-object* communication. The revised syntax for $\mathcal{L}_{obj}$ is given below.

$$
\begin{array}{lcl}
g & ::= & w\,¡\,n \mid ¿\,v \\
x & ::= & \mathbf{skip} \mid a \mid \mathbf{if}\ b\ \mathbf{then}\ x\ \mathbf{else}\ x \mid x\ ;\ x \mid \mathbf{new}\ w\ \mathbf{is}\ x \mid \mathbf{ned}\ [\ (g \to x)^*\ ] \\
 & \mid & \mathbf{letrec}\ y = x\ \mathbf{in}\ x \mid \mathbf{call}\ y
\end{array}
$$

The new communication mechanism employs a new pair of communication primitives: $w\,¡\,n$ and $¿\,v$. The former evalutes the expression n and sends the value to the object refered by w. The latter suspends execution until a value is received (by a use of the former primitive);

```
type SC = [Comp]
data PS = Obj O SC S
type PC = [PS]

type Kont = PC
type Cont = Kont
```

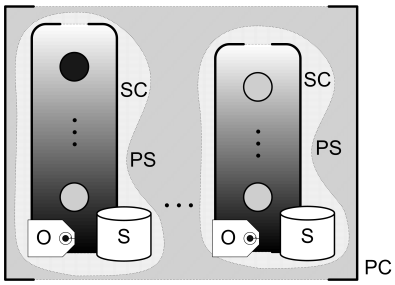


Figure 9: Structure of continuations for $\mathcal{L}_{obj}$: a multiset of stacks computations. Each stack represents an object and contains an object identifier and the local state.

the value is then assigned to variable w. Notice that the receiving object is willing to accept communication messages from *any* other object that knows how to refer to it. Therefore, the revised statement **new** not only creates a new object but also a new communication connection to it.

The syntax of $\mathcal{L}_{obj}$ is encoded in Haskell as follows.

```
data C = Snd W N | Rcv V
data X = Skip | Action A | If B X X | Seq X X | New W X | Ned [(C, X)]
       | LetRec Y X X | Call Y
```

References to objects necessitate a new semantic domain O of *object identifiers*. Object identifiers must be unique in any continuation; for simplicity, we choose to represent them by integer numbers. The mapping between an object variable (reference) w and an object identifier o has a local character and must be made part of the distributed state. Therefore, the data type S is extended with a mapping of type $W \to O$.

```
type O = Int
data S = St{ getv :: V → Val, getw :: W → O }

s0 = St{ getv = sv0, getw = sw0 }
  where sv0 v  = error "variable not initialized"
        sw0 w = error "object variable not initialized"
```

As a result of the modification of S, trivial changes are required in the semantics of expressions and elementary actions, where values are read from the state and stored in the state respectively:

```
evN (V v) s = getv s v

semA (Assign v n) c = rdState c >>= \s →
                      inState s{ getv = upd v (evN n s) (getv s) }cc c
```

The structure of continuations in $\mathcal{L}_{obj}$ is very similar to that of $\mathcal{L}_{ds}$. Continuations are again structured as multisets of stacks of computations, but each sequential composition (process) contains additionally an object identifier and its local state (see Figure 9). In this structure of continuations, functions *rdState* and *inState* can be simply defined as follows:

```
rdState     (Obj o sc s : pc) = InM (\r → (s, r))
inState s f (Obj o sc s' : pc) = InM (\r → unM (f (Obj o sc s : pc)) r)
```

The only interesting change to the scheduler for $\mathcal{L}_{obj}$, in comparison to that for $\mathcal{L}_{ds}$, concerns synchronization. First of all, the semantic domain *SemC* needs to change, to reflect the change in C.

```
data SemC = SemSnd W (S → Val) | SemRcv V
data Comp = Den D | Sync [(SemC, D)]
```

Synchronization schedules are similar to those in section 4.1, but function *sync* checks that the receiving object is indeed the one referred to by w in the sending object's local state.

```
actc k = [(p, Obj o sc s : pc) | (Obj o (p : sc) s, pc) ← ms k]

sync k1 k2 =
  [Scheds (re (addc (Den d1) (Obj o1 sc1 s1 : pc1)) ++
           re (addc (Den d2) (Obj o2 sc2 s2{getv = upd v (pe s1) (getv s2)} : pc2)))
   | (Sync snd, Obj o1 sc1 s1 : pc1) ← actc k1,
     (Sync rcv, Obj o2 sc2 s2 : pc2) ← actc k2,
     (SemSnd w pe, d1)               ← snd,
     (SemRcv v, d2)                  ← rcv,
     getw s1 w == o2]
```

The normalization function is very similar to that of $\mathcal{L}_{ds}$, with the trivial addition of object identifiers.

```
re (Obj o [] s : pc) = pc
re (Obj o sc s : pc) = Obj o sc s : pc
```

In the evaluator, a trivial change is also required in the semantics of **ned**, reflecting the changes in *SemC*.

```
sem (Ned gx) e c = cc (addc p c)
  where p = Sync [(semC g, sem x e) | (g, x) ← gx]
        semC :: C → SemC
        semC (Snd w e) = SemSnd w (evN e)
        semC (Rcv v)   = SemRcv v
```

For the semantics of **new**, there must be a way of allocating new unique object identifiers; this is achieved with function *newo* :: *Cont* → *O* which takes as parameter the complete current continuation, so as to avoid returning an object identifier that is already in use. Also, function *addn* must change, in order to update the new object's variable w in the state of the object that created it. Furthermore, trivial changes are required in several functions, to propagate object identifiers.

```
newo :: Cont → O
newo pc = maximum [o | Obj o sc s ← pc] + 1

addn :: Comp → Cont → W → Cont
addn p (Obj o sc s : pc) w = Obj o sc s' : Obj o' [p] s0 : pc
  where s' = s{getw = upd w o' (getw s)}
        o' = newo (Obj o sc s : pc)

addc :: Comp → Cont → Cont
addc p (Obj o sc s : pc) = Obj o (p : sc) s : pc
```

The semantics of **new** now becomes:

$sem\ (New\ w\ x)\ e\ c = cc\ (addn\ (Den\ (sem\ x\ e))\ c\ w)$

Furthermore, the definition of an empty continuation changes again. The initial object's identifier is 0.

$c_0 = [\,Obj\ 0\ [\,]\ s_0\,]$

The combination of object references and object-to-object communication suffices to encode a nice parallel implementation of the *sieve of Eratosthenes*. Although this implementation is not realistic in terms of resource usage (it creates one process for filtering out the multiples of each computed prime number), it is interesting to see how distributed state and object-to-object communication can be used to set up a pipeline of filters.

```
letrec      generate = ned [ first¡i  →  i := i + 2 ];
                       call generate
in letrec   filter   = ned [ ¿i  →  skip ];
                       if i%p == 0 then skip
                                   else  ned [ next¡i → skip ];
                       call filter
in letrec   sieve    = ned [ ¿p  →  write p ];
                       new next is (call sieve);
                       call filter
in          write 2; i := 3;
            new first is (call sieve);
            call generate
```

This program is deterministic but does not terminate. In our prototype implementation the produced trace begins with the sequence:

$[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73,$
$79, 83, 89, 97, 101, 103, 107, 109, 113, 127, 131, 137\ \mathit{Interrupted}\ .$

5 Advanced Control Concepts

In this section we study advanced control concepts which can be encountered in some distributed systems or in some parallel logic programming systems. In section 5.1 we present a compositional model designed with CSC for a language $\mathcal{L}_{kc}$ that extends $\mathcal{L}_{obj}$ with constructs for remote object destruction and cloning. The section 5.2.1 is dedicated to logic programming. The presentation focuses on a compositional semantics designed with CSC for a language $\mathcal{L}_{ao}$ that captures the control flow kernel of Warren's basic Andorra model. To the best of our knowledge, the remote control operations that we study in section 5.1 and the Andorra model that we study in section 5.2.1 have never been modeled denotationally by using only classic compositional techniques.

5.1 Remote Object Destruction and Cloning

The language $\mathcal{L}_{kc}$ that we consider in this section is an extension of $\mathcal{L}_{obj}$ with two new constructs: remote process (object) destruction and cloning. As in $\mathcal{L}_{obj}$, in $\mathcal{L}_{kc}$ objects are executed in parallel. Each object o has an unique identifier which can be used by any other object (that knows the identifier of o) to communicate a message to o (call one of its methods), as in $\mathcal{L}_{obj}$, but also to destroy or clone o. The syntax of $\mathcal{L}_{kc}$ is given below.

$$\begin{array}{rcl} x & ::= & \textbf{skip} \mid a \mid \textbf{if } b \textbf{ then } x \textbf{ else } x \mid x\,;\,x \mid \textbf{new } w \textbf{ is } x \mid \textbf{ned } [\,(g \rightarrow x)^*\,] \\ & \mid & \textbf{letrec } y = x \textbf{ in } x \mid \textbf{call } y \mid \textbf{kill } w \mid \textbf{clone } w \textbf{ is } w \end{array}$$

The syntax of $\mathcal{L}_{kc}$ in Haskell is given below.

$$\begin{array}{l} \textbf{data } X = Skip \mid Action\ A \mid If\ B\ X\ X \mid Seq\ X\ X \mid New\ W\ X \mid Ned\ [(C, X)] \\ \qquad\qquad \mid LetRec\ Y\ X\ X \mid Call\ Y \mid Kill\ W \mid Clone\ W\ W \end{array}$$

As in $\mathcal{L}_{obj}$, the construct **new** w **is** x creates a new object (with a new private state) to evaluate x; the identifier of the newly created object is assigned to the variable w. In $\mathcal{L}_{kc}$, object references can be used to control objects remotely. The statement **kill** destroys the object running in parallel to which the object variable w refers and the statement **clone** w_{new} **is** w_{old} clones an existing object running in parallel, referred to by w_{old}, and assigns the clone's identifier to the object variable w_{new}.

Object identifiers can be used for object to object communication, as in $\mathcal{L}_{obj}$. But the semantics of **kill** and **clone** can be studied in isolation from any communication mechanism. For simplicity, we disallow object suicide and self-cloning; these operations are less interesting because they can easily be modeled both with CSC and with classic denotational techniques.

Two new functions are required to model the semantics of **kill** and **clone**. They take as parameter the current continuation and one or two object references. They traverse the continuation until they locate the object to be destroyed or cloned; then they perform their job upon it.

$$\begin{array}{l} kill :: Cont \rightarrow W \rightarrow Cont \\ kill\ (Obj\ o\ sc\ s : pc)\ w = \\ \quad Obj\ o\ sc\ s : [Obj\ o'\ sc'\ s' \mid Obj\ o'\ sc'\ s' \leftarrow pc, o' /= getw\ s\ w] \end{array}$$

$$\begin{array}{l} clone :: Cont \rightarrow W \rightarrow W \rightarrow Cont \\ clone\ (Obj\ o\ sc\ s : pc)\ w_{new}\ w_{old} = \\ \quad \textbf{case } [Obj\ o_{new}\ sc'\ s' \mid Obj\ o'\ sc'\ s' \leftarrow pc, o' == getw\ s\ w_{old}] \textbf{ of} \\ \qquad [\,] \rightarrow Obj\ o\ sc\ s : pc \\ \qquad pc' \rightarrow Obj\ o\ sc\ s\{getw = upd\ w_{new}\ o_{new}\ (getw\ s)\} : pc' +\!\!+ pc \\ \quad \textbf{where } o_{new} = newo\ (Obj\ o\ sc\ s : pc) \end{array}$$

The semantics of **kill** and **clone** are directly based on the functions *kill* and *clone*:

$$\begin{array}{ll} sem\ (Clone\ w_{new}\ w_{old})\ e\ c = cc\ (clone\ c\ w_{new}\ w_{old}) \\ sem\ (Kill\ w) \qquad\qquad e\ c = cc\ (kill\ c\ w) \end{array}$$

As an illustration of programs that can be written in $\mathcal{L}_{kc}$ we present some examples. In our first example, a counting object w is created and left alone to do its job for a little while. Then, the object is cloned and there are two counters (w and z) working in parallel. After some time, w is killed and later on z is killed too.

letrec *count* = **write** v; $v := v + 1$; **call** *count*
in letrec *sleep* = **if** $0 < d$ **then** $d := d - 1$; **call** *sleep*
else skip
in **new** w **is** ($v := 100$; **call** *count*);
$d := 5$; **call** *sleep*; **write** 1; **clone** z **is** w;
$d := 10$; **call** *sleep*; **write** 2; **kill** w;
$d := 5$; **call** *sleep*; **write** 3; **kill** z

This program can only be tested in "single trace" semantics. Here is a possible execution trace produced by our semantic interpreter:

$[100, 1, 101, 102, 101, 103, 102, 104, 105, 103, 106, 107, 2, 108, 104,$
$109, 110, 111, 112, 3]$

In the sequel, we consider three toy programs that can be tested in the "all possible traces" semantics, using the powerdomain monad given in appendix A. The first one contains no recursive definitions.

new w_1 **is** (**write** 1; **write** 2); **clone** w_2 **is** w_1; **kill** w_1; **kill** w_2

This program can produce nine different traces. Running the program in "all possible traces" semantics, it produces the following result:

$\{\!\{\ [], [1,2], [1,2,1,2], [1], [1,1,2], [1,2,2], [1,2,1], [1,1], [1,1,2,2]\ \}\!\}$

It is not surprising that the following program, using a recursive definition, behaves exactly the same. Our semantic interpreter produces the same collection of nine different traces.

letrec y = **if** $v < 3$ **then** **write** v; $v := v + 1$; **call** y
else skip
in **new** w_1 **is** ($v := 1$; **call** y); **clone** w_2 **is** w_1; **kill** w_1; **kill** w_2

The last $\mathcal{L}_{kc}$ example is again a recursionless program. It avoids a deadlock state by appropriately using the **kill** primitive. An object is created which produces an observable and immediately suspends upon an unmatched communication attempt. The object is then cloned but the program terminates normally because both clones are killed.

new w **is** (**write** 42; **ned** [¿ $v \rightarrow$ **skip**]);
clone z **is** w; **kill** w; **kill** z

The program can produce three different traces, depending on the exact time when the object was cloned:

$\{\!\{\ [], [42], [42, 42]\ \}\!\}$

5.2 Nondeterministic Promotion in Andorra-like Languages

This section is dedicated to logic programming. In dealing with this paradigm we adopt the "logic programming without logic" approach [dBak91], which relies on the general idea of separating the logic from control [Kowa79]. In this approach, the elementary actions are modeled as simple (uninterpreted) symbols, thus ignoring specific logic programming concepts

such as unification or substitution generation.[17] Following [dBak96], we say that a language providing only uninterpreted elementary actions is *uniform*.[18] A semantic study conducted in the "logic programming without logic" style proceeds in two phases. In the first phase, a semantic model of an uniform language capturing the control flow kernel of the logic language under study is designed. Next, the atomic actions are interpreted as unifications, substitution generations, etc. As the second phase is similar for most logic programming languages, the study usually focuses on the first phase [dBak91, dBak90, dBoe93, dBak96].

The presentation in this section focuses on a semantic interpreter designed with CSC for a uniform language $\mathcal{L}_{ao}$ capturing the control kernel of Warren's basic Andorra model [Warr88]. The essence of this model can be explained by using the concepts of a *determinate goal* and a *nondeterminate goal*. A goal is determinate if at most one of its alternatives (clauses) succeeds; a goal is nondeterminate if it has more than one non-failing alternatives. In the Andorra model the goals in a conjunction can be executed in parallel. According to the Andorra principle, the executive of the language gives priority to the determinate goals over the nondeterminate goals, as the execution of a nondeterminate goal could possibly (unnecessarily) multiply the number of inference steps. When a state is reached where only nondeterminate goals remain, the system selects one such goal and performs a *nondeterministic promotion*: it makes a copy of all AND-parallel goals for each alternative of the selected nondeterminate goal. Next, these groups of AND-parallel processes can be executed either in sequence, giving rise to a backtracking mechanism (a sequential OR), or in parallel, giving rise to OR parallelism. The logical disjunction is implemented by using the concept of search or *don't know nondeterminism*. The operations of failure and nondeterministic promotion in Andorra may be seen as generalizations of the **kill** and **clone** operations that we studied in section 5.1.

Following [Todo00c], in the semantic treatment of the Andorra model we employ the following domain of observables:

type $Obs = [Act]$

where Act is the type of the (uninterpreted) atomic actions (which will be introduced in section 5.2.1). A list containing exactly one atomic action models a determinate reduction step. A list containing more than one atomic actions models a nondeterminate reduction step, which marks the moment of a nondeterministic promotion operation. By recording the nondeterminate promotion steps we simplify the interpretation of the final yield of the semantic interpreter of an Andorra-like language.

In [Todo00c], a combination of CSC and the classic direct approach to concurrency was employed in designing a full compositional semantics for Andorra and this domain of observables was essential for the implementation of the Andorra priority mechanism. In the present paper we offer a "pure" CSC-based semantic interpreter for Andorra. The above definition of *Obs* is employed only to simplify the interpretation of the final yield of the semantic function. The Andorra priority mechanism is implemented by the scheduler of the semantic interpreter which does not depend on this particular definition of *Obs* to model the suspension of the nondeterminate goals.

[17] In the concurrent contraint programming paradigm the elementary actions are executions of **ask** or **tell** primitives [Sara93].

[18] By contrast, a language where the semantics of an elementary action depends on an appropriate notion of a state is called *nonuniform* in [dBak96]. Following this terminology, the imperative languages of the previous sections or a "real" logic programming language are called nonuniform.

The domain Q of lists of observables remains as in section 2; termination is modeled by using the constant *Epsilon*. In the context of logic programming it makes sense to distinguish formally between successful termination and failure; this specific treatment of termination is explored in appendix B.2.

The CSC continuation structure that we employ in section 5.2.1 in designing the semantic interpreter of $\mathcal{L}_{ao}$ is a stack of multisets of computations (each computation modeling a goal). In section 5.2.2 we study a language $\mathcal{L}_{aop}$ that extends $\mathcal{L}_{ao}$ with OR parallelism (and don't care nondeterminism); in this case the CSC continuation structure is a *ps*-forest having in nodes multisets of computations. To complete the picture of logic programming, in appendix B.4 we sketch a CSC-based semantic interpreter for a language $\mathcal{L}_{pro}$ capturing the control flow kernel of pure sequential Prolog; in this particular case the CSC continuation structure is a stack of stacks of computations.

Finally, the program behavior monads that facilitates the switch between the "single trace" and the "all possible traces" semantic interpretations of the Andorra-like languages $\mathcal{L}_{ao}$ and $\mathcal{L}_{aop}$ are given in appendix B.1. The domains D of denotations and *Env* of semantic environments (including the initial environment e_0) are defined as usual (see section 2.1).

5.2.1 Combining AND parallelism with backtracking

The language $\mathcal{L}_{ao}$ combines the following features: failure, atomic actions, recursion, backtracking and parallel composition (parallel AND). We assume given a set $(\alpha \in) Act$ of *atomic actions* whose elements are simple (uninterpreted) symbols; as in the previous sections, we also use the set $(y \in) Y$ of *procedure variables*. An elementary statement $a(\in A)$ can be either an atomic action or the symbol **fail** (which is *not* an element of *Act*). The backtracking operator $\langle\cdot\rangle$ takes a (possibly empty) list of operands, i.e. an element of the syntactic class $(g \in) G$, each operand being a statement guarded by an elementary statement (which can model head unification). This definition is in the spirit of the basic Andorra model [Warr88], which is based on flat guards.[19] The symbol ? can be viewed as a simple sequential composition operator prefixing a guard to a statement. If the guard succeeds then the statement is executed; upon backtracking other alternatives of a g list may follow. In AKL [Hari90], the guard operator ? is called *wait*. The syntactic class $(x \in) X$ of the $\mathcal{L}_{ao}$ statements is given below.

$$
\begin{array}{lcl}
a & ::= & \alpha \mid \textbf{fail} \\
g & ::= & \epsilon \mid a\,?\,x\ (+\ a\,?\,x\ )^* \\
x & ::= & a \mid \langle g \rangle \mid x \parallel x \mid \textbf{letrec}\ y = x\ \textbf{in}\ x \mid \textbf{call}\ y
\end{array}
$$

We implement the abstract syntax of $\mathcal{L}_{ao}$ as follows:

```
type Act = String
type Y   = String

data A   = Fail | Atom Act
type G   = [(A, X)]
data X   = Action A | Pand X X | Sor G | LetRec Y X X | Call Y
```

[19] The so-called extended Andorra model [Hari90, Ware90] —which is based on deep guards— is not formally studied in this paper.

type $PA = [Comp]$
type $SO = [PA]$

type $Kont = SO$
type $Cont = Kont$

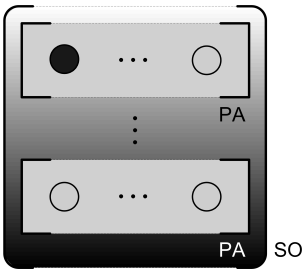


Figure 10: Structure of continuations for $\mathcal{L}_{ao}$: a stack of multisets of computations.

The CSC-based semantic interpreter of $\mathcal{L}_{ao}$ employs the domain of computations defined below. A computation of the form *Den d* models the semantics of a determinate goal; a computation of the form *SemSor gs* models the semantics a nondeterminate goal.

data $Comp = Den\ D \mid SemSor\ [(Act, D)]$

The continuations for $\mathcal{L}_{ao}$ can be structured as stacks of multisets of computations (see Figure 10). Each *PA* multiset models a parallel conjunction (a parallel AND). The *SO* stack is used to model the backtracking mechanism (a sequential OR). The initial continuation is: $c_0 = [[\,]]$.

The evaluator of the semantic interpreter of $\mathcal{L}_{ao}$ comprises the definitions of the semantic functions *semA*, *sem* and the control operators *addc*, *addp* and *fails*. To avoid any ambiguity we give below the complete definitions of all these mappings.

$$
\begin{array}{l}
semA :: A \to D \\
semA\ Fail \qquad\quad c = cc\ (fails\ c) \\
semA\ (Atom\ act)\ c = put\ [act]\ (cc\ c)
\end{array}
$$

$$
\begin{array}{l}
sem :: X \to Env \to D \\
sem\ (Action\ a) \qquad e \quad c = semA\ a\ c \\
sem\ (Call\ y) \qquad\ e \quad c = e\ y\ c \\
sem\ (LetRec\ y\ x_1\ x_2)\ e \quad c = sem\ x_2\ e'\ c \\
\quad \textbf{where}\ e' = upd\ y\ (fix\ (\backslash d \to sem\ x_1\ (upd\ y\ d\ e)))\ e \\
\qquad\quad fix \quad :: \quad (a \to a) \to a \\
\qquad\quad fix\ f = \quad f\ (fix\ f) \\
sem\ (Sor\ g) \qquad\ e \quad c = \\
\quad \textbf{case}\ [(act, sem\ x\ e) \mid (Atom\ act, x) \leftarrow g]\ \textbf{of} \\
\qquad [\,] \qquad\ \to cc\ (fails\ c) \\
\qquad [(act, d)] \to put\ [act]\ (cc\ (addc\ (Den\ d)\ c)) \\
\qquad sg \qquad\ \to cc\ (addc\ (SemSor\ sg)\ c) \\
sem\ (Pand\ x_1\ x_2) \qquad e \quad c = sem\ x_1\ e\ (addp\ (Den\ (sem\ x_2\ e))\ c)\ `ned` \\
\qquad\qquad\qquad\qquad\qquad\quad sem\ x_2\ e\ (addp\ (Den\ (sem\ x_1\ e))\ c)
\end{array}
$$

$$
\begin{array}{l}
addc :: Comp \to Cont \to Cont \\
addc\ p\ (pa : so) = (p : pa) : so
\end{array}
$$

$$
\begin{array}{l}
addp :: Comp \to Cont \to Cont \\
addp\ p\ (pa : so) = (p : pa) : so
\end{array}
$$

$$
fails :: Cont \to Cont
$$

$fails\ (pa : so) = [\,] : so$

The semantics of an elementary action and the semantics of a (failing or a non-failing) determinate goal are easily defined with the aid of the operators *put* (which is given in appendix B.1) and *fails*. For a determinate goal *put* is always used to prefix an observable (of the type *Obs*) representing a determinate step (a list of length one). The operator *fails* implements the semantics of failure by voiding the multiset (the parallel conjunction) of goals at the top of the backtracking stack. The semantics of a nondeterminate goal (an n-ary disjunction with more than one non-failing guards) is added to the continuation to be handled by the scheduler; the evaluation of a nondeterminate goal is delayed as much as possible in Andorra. As usual, the continuation semantics of parallel composition (interpreted here as a parallel AND) is based on the combination of two alternative computations: one starting from the first statement and one starting from the second. The control operators *addc* and *addp* simply add a computation to the multiset at the top of the backtracking stack.

The continuation completion mapping *cc* remains as in section 2.2. The normalization procedure removes an eventual empty multiset from the top of the backtracking stack.

$re :: Cont \to Kont$
$re\ ([\,] : so) = so$
$re\ so = so$

According to the Andorra principle [Warr88], the scheduler of $\mathcal{L}_{ao}$ must give priority to computations modeling determinate goals ($Den\ d$) over computations modeling nondeterminate goals ($SemSor\ gs$). The scheduler function *kc* calls *scheda* to compute all possible activation schedules of the form $Scheda\ d\ c$, where d is an activable computation (a denotation implementing a determinate goal) and c is a corresponding open continuation. If *scheda* produces an empty list, i.e. if the continuation (is not empty but it) contains no determinate goals, then the scheduler calls *scheds* which computes all possible schedules of the form $Scheds\ obs\ k$, where k is a closed continuation obtained by a nondeterministic promotion and *obs* is a corresponding observable.

data $Sched = Scheda\ D\ Cont \mid Scheds\ Obs\ Kont$

$kc :: Kont \to M\ Q$
$kc\ k =$ **case** $scheda\ k$ **of**
$\quad [\,] \to bigned\ (map\ exe\ (scheds\ k))$
$\quad ws \to bigned\ (map\ exe\ ws)$
where $exe\ (Scheda\ d\ c) = d\ c$
$\quad exe\ (Scheds\ obs\ k') = put\ obs\ (kc\ k')$

$scheda :: Kont \to [Sched]$
$scheda\ k = [Scheda\ d\ c \mid (Den\ d, c) \leftarrow actc\ k]$

$actc :: Kont \to [(Comp, Cont)]$
$actc\ k = [(p, pa : tail\ k) \mid (p, pa) \leftarrow ms\ (head\ k)]$

The computations (that model nondeterminate goals) of the form $SemSor\ sg$ are handled by *scheds*, which uses the auxiliary mapping *semSor*. Function *semSor* implements a nondeterministic promotion: it replicates the multiset from the top of the backtracking stack for each alternative of a nondeterminate goal. It is designed to produce a closed continuation which can immediately be passed to the scheduler. The effect of this operation is represented in figure 11.

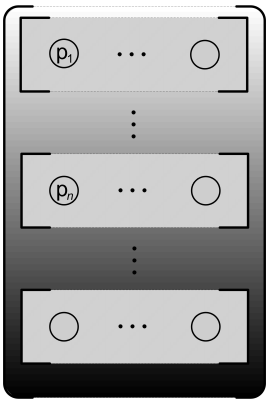


Figure 11: The result of applying *semSor* to a list of computations [p$_1$,...,p$_n$] and the open continuation depicted in Figure 10; *semSor* implements a nondeterministic promotion step and yields a closed continuation.

$$
\begin{array}{l}
scheds :: Kont \to [Sched] \\
scheds\ k = [Scheds\ [act \mid (act, \_) \leftarrow sg] \\
\quad (semSor\ [Den\ d \mid (\_, d) \leftarrow sg]\ c) \\
\quad \mid (SemSor\ sg, c) \leftarrow actc\ k]
\end{array}
$$

$$
\begin{array}{l}
semSor :: [Comp] \to Cont \to Kont \\
semSor\ lp\ (pa : so) = [p : pa \mid p \leftarrow lp] +\!\!+ so
\end{array}
$$

To test this semantic interpreter we consider three $\mathcal{L}_{ao}$ example programs. The first program executes two determinate goals in AND parallel.

$$c \parallel \langle a \,?\, b \rangle$$

Throughout this paper we prefer to give the abstract syntax of the test programs. In this section the results are produced on the assumption that the symbols representing atomic actions are implemented in Haskell by using corresponding strings. For example, we implement the syntax of this $\mathcal{L}_{ao}$ program as follows:

$$Pand\ (A\ (Atom\ \texttt{"c"}))\ (Sor\ [(Atom\ \texttt{"a"}, A\ (Atom\ \texttt{"b"}))])$$

The program can produce three different traces. Here is the result produced by our semantic interpreter in "all possible traces" semantics:

$$
\begin{array}{l}
\{\!|\ [[\texttt{"c"}], [\texttt{"a"}], [\texttt{"b"}]], \\
\quad [[\texttt{"a"}], [\texttt{"b"}], [\texttt{"c"}]], \\
\quad [[\texttt{"a"}], [\texttt{"c"}], [\texttt{"b"}]]\ |\!\}
\end{array}
$$

The second program executes two determinate goals in parallel with a nondeterminate goal, which is given the least priority.

$$
\begin{array}{ll}
\textbf{letrec} & y = \langle A \,?\, a + \textbf{fail} \,?\, b + C \,?\, c \rangle \\
\textbf{in} & d \parallel \textbf{call}\ y \parallel e
\end{array}
$$

This program can produce two different traces. The evaluation of the nondeterminate goal ($\langle A\,?\,a + \mathbf{fail}\,?\,b + C\,?\,c\rangle$) begins only after the completion of the interleaved execution of the elementary goals d and e. The evaluation of the nondeterminate goal begins by outputing the observable `["A","C"]` (the list of non-failing guards of the nondeterminate goal), which marks the moment when the nondeterministic promotion occurs. Next, the (bodies of the) two alternatives of the the nondeterminate goal are tried in sequence according to the implicit backtracking mechanism of $\mathcal{L}_{ao}$.

```
{| [["d"],["e"],["A","C"],["a"],["c"]],
   [["e"],["d"],["A","C"],["a"],["c"]] |}
```

The next $\mathcal{L}_{ao}$ program launches four goals in parallel: two determinate goals, and two nondeterminate goals. When the execution of the determinate goals terminates the parallel execution of the two nondeterminate goals begins. In this state an arbitrary nondeterminate goal is selected and execution can only proceed by a nondeterministic promotion, i.e. by making copies of the other goal for each alternative of the selected one. This example is taken from [Todo00c].

$$a \parallel \langle C\,?\,c + D\,?\,d\rangle \parallel b \parallel \langle E\,?\,e + F\,?\,f\rangle$$

The program can produce four different execution traces. Each trace is marked by three nondeterminate reduction steps. Running this program in "all possible traces" semantics produces the following output:[20]

```
{| [["a"],["b"],["E","F"],["e"],["C","D"],["c"],["d"],["f"],["C","D"],["c"],["d"]],
   [["b"],["a"],["C","D"],["c"],["E","F"],["e"],["f"],["d"],["E","F"],["e"],["f"]],
   [["a"],["b"],["C","D"],["c"],["E","F"],["e"],["f"],["d"],["E","F"],["e"],["f"]],
   [["b"],["a"],["E","F"],["e"],["C","D"],["c"],["d"],["f"],["C","D"],["c"],["d"]] |}
```

By running the above program in "single trace" semantics one can obtain each of the four possible traces at subsequent executions.

Such experiments are, of course, less interesting as long as one can run the semantic interpreter to obtain all possible traces. But let's consider the following $\mathcal{L}_{ao}$ program.

letrec $y = \langle a\,?\,(\mathbf{call}\ y \parallel \mathbf{fail})\rangle$
in $b \parallel \mathbf{call}\ y \parallel c$

This program cannot be tested in "all possible traces" semantics because it can produce an infinite number of different traces. The length of each trace depends on the moment when failure occurs. However, the program *can* be tested in "single trace" semantics. We reproduce below four possible traces obtained with our semantic interpreter in "single trace" semantics.

```
[["b"],["c"],["a"],["a"],["a"]]

[["c"],["b"],["a"]]

[["a"],["a"],["b"]]

[["a"],["c"],["b"],["a"]]
```

At each execution, the program terminates in failure. In the present form, our semantic interpreter does not make any formal distinction between normal termination and failure. However, it is easy to modify the semantic interpreter to obtain this effect; the required modifications are described in appendix B.2.

[20] This output is equivalent to the one that we have obtained in [Todo00c]; only the order of the execution traces is different but that is semantically irrelevant.

5.2.2 AND-OR Parallelism

The language $\mathcal{L}_{aop}$ extends $\mathcal{L}_{ao}$ with OR parallelism and don't care nondeterminism. In the grammar for $\mathcal{L}_{aop}$ the rule for statements has been extended as follows:

$$h \ ::= \ \epsilon \mid a : x \ (+ \ a : x \ )^*$$
$$x \ ::= \ a \mid \langle g \rangle \mid \#\langle g \rangle \mid \ll h \gg \mid x \parallel x \mid \textbf{letrec } y = x \textbf{ in } x \mid \textbf{call } y$$

where, in addition to the features already present in $\mathcal{L}_{ao}$, $\#\langle g \rangle$ is the construct for OR parallelism and $\ll h \gg$ is the construct for don't care nondeterministic choice between the alternatives of the list h. The guard operator : is called *commit*.

We emphasize that the constructs $\langle g \rangle$ and $\#\langle g \rangle$ are interpreted by using the concept of search or *don't know nondeterminism*. There is a fundamental distinction between don't know and don't care nondeterminism in logic programming. The alternatives of a $\langle g \rangle$ construct are tried in sequence and the alternatives of a $\#\langle g \rangle$ construct are tried in parallel, but in the both cases *all* the alternatives are inspected. The construct $\ll h \gg$ is interpreted by using the concept of (committed choice or) *don't care nondeterminism*. In this case only *one*, arbitrarily selected, alternative of h is tried by the executive of the language; this leads to a more efficient evaluation strategy at the expense of sacrificing the logical completeness. Notice that only a don't know goal can be nondeterminate; a don't care goal is always determinate.

We implement the syntax of $\mathcal{L}_{aop}$ in Haskell as follows:

```
type H = [(A, X)]
data X = Action A | Pand X X | Sor G | Por G | Ned H
       | Call Y | LetRec Y X X
```

$\mathcal{L}_{aop}$ is identical to the Andorra-like language that we studied in [Todo00c], but in this paper we follow a different language design approach. In [Todo00c], we employed the CSC technique only in the implementation of the AND parallelism by using the simple structure of a multiset of computations that are executed in parallel; for the sequential OR, the parallel OR and the Andorra priority mechanism we used the classic direct approach to concurrency semantics. In this section we offer a "pure" CSC-based semantic interpreter for $\mathcal{L}_{aop}$; consequently, the structure of CSC continuations is more complex.

The implementation of the evaluator, the normalization procedure and the scheduler of $\mathcal{L}_{aop}$ are given in appendix B.3; in the main text we only describe the structure of the CSC continuations that we employ in its design. We extend the definition of the domain of computations with a new alternative (with data constructor *SemPor*) that we need in the semantic design of the OR parallelism.

```
data Comp = Den D | SemSor [(Act, D)] | SemPor [(Act, D)]
```

Continuations in $\mathcal{L}_{aop}$ can be structured as *ps*-forests having in nodes multisets of computations. The active computations are selected from the multisets at the leaves of (any of the *ps*-trees in) the *ps*-forest. This continuation structure is depicted in figure 12. The *ps*-structure is needed for modeling the general combination of the sequential OR and the parallel OR connectives in $\mathcal{L}_{aop}$. The multiset structure is needed for the implementation of the AND parallelism. In the case of $\mathcal{L}_{aop}$ the initial continuation $c_0 :: Cont$ is defined as follows: $c_0 = [Meet \ [] \ [[]]]$. We also mention that it is possible to extend $\mathcal{L}_{aop}$ with an operator for sequential conjunction, i.e. a sequential AND operator. In this case, the CSC continuation structure becomes a *ps*-forest having as nodes *ps*-forests.

type $PA = [Comp]$
type $SO = [PA]$
data $PS = Meet\ PO\ SO$
type $PO = [PS]$

type $Kont = PO$
type $Cont = Kont$

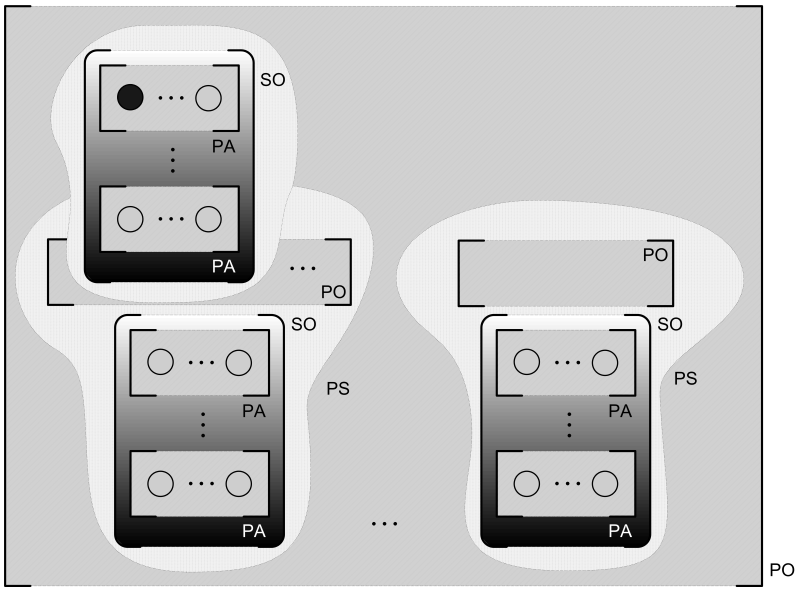


Figure 12: Structure of continuations for $\mathcal{L}_{aop}$: a *ps*-forest having as nodes multisets of computations; the active computations are selected from the leaves.

In the final part of this section we present a number of $\mathcal{L}_{aop}$ example programs together with their interpretation in "all possible traces" semantics. The examples are taken from [Todo00c]. This gives us the opportunity to compare the semantic interpreter designed in this section in a "pure" CSC style, with the semantic model given in [Todo00c]. All our experiments show that the two semantic models behave the same.[21]

The first test program given here illustrates the behaviour of a parallel composition in interleaving semantics.

$a \parallel b \parallel c$

This program can produce six different interleavings.

$\{\!|$ [["a"], ["b"], ["c"]], [["c"], ["a"], ["b"]], [["b"], ["a"], ["c"]]
[["c"], ["b"], ["a"]], [["a"], ["c"], ["b"]], [["b"], ["c"], ["a"]] $|\!\}$

The next program illustrates the semantics of don't care nondeterminism.

$\ll a : b + \mathbf{fail} : c + d : e \gg$

The two possible execution traces correspond to the two non-failing guards.

$\{\!|$ [["a"], ["b"]], [["d"], ["e"]] $|\!\}\ cb$

The next program illustrates a combination of don't care nondeterministic choice and parallel composition.

$a \parallel \ll \mathbf{fail} : b + c : d \gg$

[21] The order of the execution traces may be different for some tests in the two interpretations, but this is semantically irrelevant.

This program can produce three different execution traces.

$\{\!|$ [["a"], ["c"], ["d"]], [["c"], ["d"], ["a"]], [["c"], ["a"], ["d"]] $|\!\}$

The last two examples illustrate the biased execution behavior towards determinate goals. In the first case the don't know goal is a sequential OR.

$a \parallel \langle c : d + e : f \rangle \parallel b$

The don't know goal is reduced only after the completion of the interleaved execution of the elementary goals a and b. The observable ["c", "e"] marks the moment when the nondeterministic promotion occurs. Next, the (bodies of the) alternatives of the don't know goal are tried in sequence.

$\{\!|$ [["a"], ["b"], ["c", "e"], ["d"], ["f"]],
[["b"], ["a"], ["c", "e"], ["d"], ["f"]] $|\!\}$

The next example is similar, only the don't know goal is a parallel OR.

$a \parallel \#\langle c : d + e : f \rangle \parallel b$

Our semantic interpreter produces four different execution traces. Notice that after the nondeterministic promotion step the alternatives of the don't know goal are tried in parallel (their execution is interleaved).

$\{\!|$ [["a"], ["b"], ["c", "e"], ["d"], ["f"]],
[["b"], ["a"], ["c", "e"], ["d"], ["f"]],
[["a"], ["b"], ["c", "e"], ["f"], ["d"]],
[["b"], ["a"], ["c", "e"], ["f"], ["d"]] $|\!\}$

6 Conclusions and Future Research

In the CSC approach, a continuation is a language-specific configuration of partially evaluated meaning functions (denotations), which can be accessed and manipulated separately, and can be executed either in some specific order or in an interleaved fashion. The continuation-based approach to communication and concurrency given in this paper is general. The CSC technique provides an excellent flexibility in the compositional modeling of concurrent control flow concepts.

Based on our experiments, we believe the remote control operations and the nondeterministic promotion operation in the Andorra-like languages that are modeled compositionally with CSC continuations in section 5 are beyond the expressive power of the classic denotational techniques. In this paper we showed that, by using CSC continuations denotational semantics can be used both as a method for formal specification and design, and as a general method for implementing compositional prototypes of concurrent programming languages. To the best of our knowledge, denotational semantics has never been used systematically for concurrent languages prototyping and all our attempts to get a general solution to this problem by using only classic compositional techniques have failed. In a mathematical model a CSC continuation is an element of a semantic domain defined as the solution of a domain equation where the domain variable occurs in the left-hand side of a function space construction [Todo00a, Todo04]; in this sense the CSC computation model is rather complex.

However, in this paper we have shown that the computations contained in CSC continuations can be assembled in simple structures which can provide operational intuition in the strict compositional setting of denotational semantics.

In this paper we have focused on the significance of CSC as a language design tool. Instead of using mathematical notation we used the lazy functional programming language Haskell, which proved to provide a nice expressive support as a prototyping environment for denotational semantics. In the future, we would like to completely formalize our semantic interpreters by using an appropriate mathematical apparatus - either classic order-theoretic (cpo-based) structures or metric spaces. Also, we plan to evaluate our prototyping approach on more complex languages, including POOL [Amer89, ABKR89] and concurrent extensions of Algol-like languages [OHea97, Broo02]. We expect that this will give us the opportunity to try various new, and possibly innovative, combinations of CSC with other (more classical) techniques used in denotational semantics, including traditional continuations, possible worlds semantics and monads.

References

[Abra96] Abramsky, S. Semantics of Interaction. In H. Kirchner, editor, *Trees in Algebra and Programming: Proceedings of the 21st International Colloquium (CAAP'96)*, Linköping, Sweden. LNCS 1095:1, Springer, 1996.

[Ada83] U. S. Department of Defense. *Reference Manual for the Ada Programming Language*. ANSI/MIL-STD-1815A, 1983.

[Aglets] IBM Aglets website: `http://www.trl.ibm.com/aglets/`.

[Amer89] America, P. Issues in the Design of a Parallel Object-Oriented Language. *Formal Aspects of Computing*, 1(4):366–411, 1989.

[Appe92] Appel, A. W. *Compiling with Continuations*. Cambridge University Press, 1992.

[Bobr73] Bobrow, D. G. and B. Wegbreit. A Model and Stack Implementation of Multiple Environments. *Communications of the ACM*, 16(10):591–603, 1973.

[Card95] Cardelli, L. A Language with Distributed Scope. In *Proceedings of the 22nd Annual ACM Symposium on Principles of Programming Languages*, pages 286–297, ACM Press, 1995.

[Danv92] Danvy, O. and A. Filinski. Representing Control: a Study of the CPS Transformation. *Mathematical Structures in Computer Science*, 2(4):361–391, 1992.

[Danv04] Danvy, O. On Evaluation Contexts, Continuations, and the Rest of the Computation. In H. Thielecke, editor, *Proceedings of the 4th ACM SIGPLAN Continuations Workshop (CW'04)*, pages 13–23, 2004.

[dBak90] de Bakker, J. W. and J. N. Kok. Comparative metric semantics for Concurrent Prolog. *Theoretical Computer Science* 75:15–43, 1990.

[dBak91] de Bakker, J. W. Comparative Semantics for Flow of Control in Logic Programming without Logic. *Information and Computation*, 94:123–179, 1991.

[dBak96] de Bakker, J. W. and E. P. de Vink. *Control Flow Semantics*. MIT Press, 1996.

[dBoe93] de Boer, F. S., J. N. Kok, C. Palamidessi, and J. J. M. M. Rutten. A Paradigm for Asynchronous Communication and its Application to Concurrent Constraint Programming. In K. R. Apt, J. W. de Bakker, and J. J. M. M. Rutten, editors, *Logic Programming Languages: Constraints, Functions and Objects*, pages 82–114, MIT Press, 1993.

[Dybv89] Dybvig, R. K. and R. Hieb. Engines from Continuations. *Computer Languages*, 14(2):109–123, 1989.

[Feel93] Feeley, M. *An Efficient and General Implementation of Futures on Large Scale Shared-Memory Multiprocessors*. Ph.D. thesis, Massachusetts Institute of Technology, 1993.

[Fell88a] Felleisen, M. The Theory and Practice of First-Class Prompts. In *Procedings of 15th Annual ACM Symposium on Principles of Programming Languages*, pages 180–190, ACM Press, 1988.

[Fell88b] Felleisen, M., M. Wand, D. P. Friedman and B. Duba. Abstract Continuations: a Mathematical Semantics for Handling Full Functional Jumps. In *Proceedings of the 1988 ACM Conference on Lisp and Functional Programming*, pages 52–62, ACM Press, 1988.

[Fell06] Felleisen, M. and M. Flat. *Programming Languages and Lambda Calculi*. Unpublished lecture notes. available from `http://www.cs.utah.edu/plt/publications/pllc.pdf`, 1989–2006.

[Frie86] Friedman, D. P., C. T. Haynes and M. Wand. Obtaining Coroutines with Continuations. *Computer Languages*, 11(3/4):143–153, 1986.

[Hals85] Halstead, R. MultiLisp: a Language for Concurrent Symbolic Computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–538, 1985.

[Hari90] Haridi, S. and S. Janson. Kernel Andorra Prolog and its computation model. In *Proc. of International Conference on Logic Programming (ICLP'90)*, MIT Press, 1990.

[Hewi73] Hewitt, C., P. Bishop, I. Greif, B. Smith, T. Matson, and R. Steiger. Actor induction and meta-evaluation. In *Proceedings of the 1st Annual ACM Symposium on Principles of Programming Languages*, pages 153–168, ACM Press, 1973.

[Hewi77a] Hewitt, C. Viewing Control Structures as Patterns of Passing Messages. *Journal of Artificial Intelligence*, 8(3):323-364, 1977.

[Hewi77b] Hewitt, C. and H. Bakker. Laws for Communicating Parallel Processes. In *Proceedings of the IFIP Congress 77*, pages 987-992, North-Holland, 1977.

[Hieb94] Hieb, R., R. K. Dybvig and C. W. Anderson. Subcontinuations. *Lisp and Symbolic Computation*, 7(1):83–110, 1994.

[Hoar78] Hoare, C. A. R. Communicating Sequential Processes. *Communications of the ACM*, 21(8):666–677, 1978.

[Hoar85] Hoare, C. A. R. *Communicating Sequential Processes.* Prentice-Hall, 1985.

[Holz96] Holzbacher, A. A. A Software Environment for Concurrent Coordinated Programming. In *Proceedings of the 1st International Conference on Coordination Languages and Models*, LNCS 1061:249–266, Springer, 1996.

[Katz90] Katz, M., D. Weise. Continuing into the Future: On the Interactions of Futures and First-Class Continuations. In *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, pages 176–184, ACM Press, 1990.

[Kels98] Kelsey, R., W. Clinger and J. Rees, editors, *Revised Report on the Algorithmic Language Scheme*, 1998. Available from `http://swiss.csail.mit.edu/projects/scheme/`.

[Kowa79] R. Kowalski. Algorithm = logic + control. *Communications of the ACM* 22:424–435, 1979.

[Lang98] Lange, D. B and M. Oshima. *Programming and Deploying Java Mobile Agents with Aglets.* Addison Wesley, 1998.

[Lian95] Liang, S., P. Hudak and M. Jones. Monad Transformers and Modular Interpreters. In *Proceedings of the 22nd Annual ACM Symposium on Principles of Programming Languages*, pages 333–344, ACM Press, 1995.

[Lian96] Liang, S. and P. Hudak. Modular Denotational Semantics for Compiler Construction. In *Proceedings of the 6th European Symposium on Programming*, LNCS 1058:219–234, Springer, 1996.

[Lian98] Liang, S. *Modular Monadic Semantics and Compilation.* Ph.D. thesis, Yale University, 1998.

[Mill87] Miller, J. *MultiScheme: A Parallel Processing System Based on MIT Scheme.* Ph.D. thesis, Massachusetts Institute of Technology, 1987.

[Miln89] Milner, R. *Communication and concurrency.* Prentice Hall, 1989.

[Miln99] Milner, R. *Communicating and mobile systems: the π-calculus.* Cambridge University Press, 1999.

[Mitc96] Mitchell, J. C. *Foundations for Programming Languages.* MIT Press, 1996.

[Mogg90] Moggi, E. An Abstract View of Programming Languages. Technical Report ECS-LFCS-90-113, University of Edinburgh, Laboratory for Foundations of Computer Science, 1990.

[Mogg91] Moggi, E. Notions of computation and monads. *Information and Computation*, 93(1):55–92, 1991.

[More94] Moreau, L. and C. Queinnec. Partial Continuations as the Difference of Continuations: A Duumvirate of Control Operators. In *Proceedings of the 6th International Symposium on Programming Languages Implementation and Logic Programming*, LNCS 844:182–197, Springer, 1994.

[Moss75] Mosses, P. D. *Mathematical Semantics and Compiler Generation.* Ph.D. thesis, Oxford University, 1975.

[Moss79] Mosses, P. D. SIS, Semantics Implementation System: Reference manual and user guide. Technical Monograph MD-30, Computer Science Department, Aarhus University, 1979.

[Moss90] Mosses, P. D. Denotational semantics. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, chapter 11, pages 577–631, Elsevier Science, 1990.

[Moss96] Mosses, P. D. Theory and practice of action semantics. Research Report RS-96-53, Laboratory of Basic Research in Computer Science (BRICS), Aarhus University, 1996.

[Occa84] INMOS Ltd. *Occam Programming Manual.* Prentice-Hall, 1984.

[OHea97] O'Hearn, P. W. and R. D. Tennent. *Algol-like Languages.* Birkhauser, 1997.

[Owic76] Owicki, S. and D. Gries. An Axiomatic Proof Technique for Parallel Programs. *Acta Informatica*, 6:319–340, 1976.

[Papa00] Papaspyrou, N. S. and D. Maćoš. A Study of Evaluation Order Semantics in Expressions with Side Effects. *Journal of Functional Programming*, 10(3):227–244, 2000.

[Paul82] Paulson, L. A Semantics-Directed Compiler Generator. In *Proceedings of the 9th Annual ACM Symposium on Principles of Programming Languages*, pages 224–233, ACM Press, 1982.

[Peyt99] Peyton Jones, S. and J. Hughes, editors. *Report on the Programming Language Haskell 98: A Non-Strict Purely Functional Language*, 1999. Available from `http://www.haskell.org/`.

[Plot76] Plotkin, G. D. A Powerdomain Construction. *SIAM Journal of Computing*, 5(3):452–487, 1976.

[Plot78] Plotkin, G. D. *The Category of Complete Partial Orders: A Tool for Making Meanings.* Lecture Notes for the Summer School on Foundations of Artificial Intelligence and Computer Science, Pisa, June 1978.

[Quei90] Queinnec, C. PolyScheme: A Semantics for a Concurrent Scheme. In *Proceedings of BCS Workshop on High Performance and Parallel Computing in Lisp*, Europal Ltd., Twickenham, UK, 1990.

[Quei91] Queinnec C. and B. Serpette. A Dynamic Extent Conrol Operator for Partial Continuations. In *Proceedings of the 18th Annual ACM Symposium on Principles of Programming Languages*, pages 174–184, ACM Press, 1991.

[Quei92a] Queinnec, C. A Concurrent and Distributed Extension of Scheme. In *Proceedings of the 4th International PARLE Conference on Parallel Architectures and Languages Europe*, LNCS 605:431–446, Springer, 1992.

[Quei92b] Queinnec C. and D. de Roure. Design of a Concurrent and Distributed Language. In R. H. Halstead, Jr. and T. Ito, editors, *Proceedings of the US/Japan Workshop on Parallel Symbolic Computing: Languages, Systems, and Applications*, LNCS 748:234–259, Springer, 1992.

[Repp92] Reppy, J. H. *Higher-Order Concurrency.* Ph.D. thesis, Cornell University, 1992.

[Repp99] Reppy, J. H. *Concurrent Programming in ML.* Cambridge University Press, 1999.

[Reyn93] Reynolds, J. C. The discoveries of continuations. *LISP and Symbolic Computation*, 6:233–247, 1993.

[Sara93] Saraswat, V. A. *Concurrent constraint programming.* MIT Press, 1993.

[Schm86] Schmidt, D. A. *Denotational Semantics: A Methodology for Language Development.* Allyn & Bacon, Newton, MA, 1986.

[Sita90] Sitaram D. and M. Felleisen. Control Delimiters and their Hierarchies. *Lisp and Symbolic Computation*, 3(1):67–99, 1990.

[SMLNJ] Standard ML of New Jersey web page: `http://www.smlnj.org`.

[Somm06] Sommerville, *Software engineering* (8th edition). Addison-Wesley, 2006.

[Stra74] Strachey C. and C. P. Wadsworth. Continuations: A Mathematical Semantics for Handling Full Jumps. *Higher-Order and Symbolic Computation*, 13(1/2):135–152, 2000. Reprint of the technical monograph PRG-11, Oxford University Computing Laboratory (1974), with a foreword.

[Tenn91a] Tennent, R. D. *Semantics of Programming Languages.* Prentice Hall, 1991.

[Tenn91b] Tennent, R. D. and J. K. Tobin. Continuations in Possible World Semantics. *Theoretical Computer Science*, 85(2):283–303, 1991.

[Todo00a] Todoran, E. Metric Semantics for Synchronous and Asynchronous Communication: A Continuation-Based Approach. In *Proceedings of FCT'99 Workshop on Distributed Systems*, *Electronic Notes in Theoretical Computer Science*, 28:119–146, Elsevier, 2000.

[Todo00b] E. Todoran. *Semantic techniques in concurrent systems development.* Ph.D. Thesis, Technical University of Cluj-Napoca, Romania, 2000.

[Todo00c] Todoran E. and N. S. Papaspyrou. Continuations for Parallel Logic Programming, In *Proceedings of the 2nd International ACM-SIGPLAN Conference on Principles and Practice of Declarative Programming*, pages 257–267, ACM Press, 2000.

[Todo04] Todoran E. and N. S. Papaspyrou. Denotational Prototype Semantics for a Simple Concurrent Language with Synchronous Communication. Technical Report CSD-SW-TR-1-04, National Technical University of Athens, School of Electrical and Computer Engineering, Software Engineering Laboratory, 2004.

[Todo06] Todoran E. and N. S. Papaspyrou. Continuations for Prototyping Concurrent Languages. Technical Report CSD-SW-TR-1-06, National Technical University of Athens, School of Electrical and Computer Engineering, Software Engineering Laboratory, 2006.

[Wadl92] Wadler, P. The Essence of Functional Programming. In *Proceedings of the 19th Annual ACM Symposium on Principles of Programming Languages*, pages 1–14, ACM Press, 1992.

[Wand80] Wand, M. Continuation-Based Multiprocessing. *Higher-Order and Symbolic Computation*, 12(3):285–299, 1999. Reprint from the Proceedings of the 1980 ACM Conference on LISP and Functional Programming, with a foreword.

[Wand84] Wand, M. A Semantic Prototyping System. In *Proceedings of the 1984 ACM SIGPLAN Symposium on Compiler Construction*, pages 213–221, ACM Press, 1984. Also published in *ACM SIGPLAN Notices*, 19(6).

[Warr88] Warren, D. H. D. *The Andorra Principle.* Talk given at Gigalips workshop, SICS, Sweden, 1988.

[Ware90] Warren, D. H. D. Extended Andorra model with implicit control. Talk given at Logic Programming Symposium, Eilat, Israel, 1990.

[Watt86] Watt, D. A. Executable semantic descriptions. *Software Practice and Experience* 16(1):13–43, 1986.

[dBru86] de Bruin, A. *Experiments with continuation semantics: jumps, backtracking, dyn amic networks.* Ph.D. thesis, Vrije Universiteit, Amsterdam, 1986.

[ABKR89] America P., de Bakker J. W., Kok J. N., and Rutten J. J. N. N. Denotational semantics of a parallel object-oriented language. *Information and Computation* 83(2):152–205, 1989.

[Broo02] Brookes, S. The essence of parallel Algol. *Information and Computation* 179(1):118–149, 2002.

A Computing all possible traces

It is possible to define a monad M for computing all possible traces of a program's execution. This monad implements a continuation semantics using powerdomains. It is useful in concurrent language design, as it allows one to detect whether a particular execution trace is possible.

A.1 Languages with global state

The monad given below can be used with all languages of sections 2 (although not very useful there) and 3. M is the composition of an environment monad, with the global state S as the environment, and Haskell's list monad.

type $M\ a = S \rightarrow [a]$

```
instance Monad M where
  return a          = InM (\s -> [a])
  InM m >>= f = InM (\s -> concatMap (\a -> unM (f a) s) (m s))

put u (InM m) = InM (\s -> map (Observe u) (m s))

display (InM m) = print (m s0)

ned :: Eq a => M a -> M a -> M a
ned (InM m1) (InM m2) = InM (\s -> aux (m1 s) (m2 s))
  where aux []       l2 = l2
        aux (h : t) l2 = h : aux (filter (/= h) l2) t

bigned :: Eq a => [M a] -> M a
bigned []          = InM (\s -> [])
bigned (m : ml)    = ned m (bigned ml)

rdState c       = InM (\s -> [s])
inState s f c   = InM (\s' -> unM (f c) s)
```

Two things are worth noticing in the implementation of function *ned*. First, identical results obtained more than once are discarded. Second, the parameters to *aux* are alternated, in order to mix two lists of alternative execution traces. This roughly corresponds to a breadth-first traversal of the space of execution traces, which behaves much better than a depth-first traversal in the presence of non-termination.

A.2 Languages with distributed (local) state

The monad M defined in Appendix A.1 is particular to a global state. Similarly, Haskell's list monad together with simple appropriate definitions of *ned*, *bigned*, *rdState* and *inState* can be used as a monad computing all execution traces in languages with local state. The following monad is appropriate for the languages $\mathcal{L}_{obj}$ and $\mathcal{L}_{kc}$ that are studied in sections 4.2 and 5.1 respectively.

```
type M a = [a]

put u m = map (Observe u) m

display m = putStr "{| " >> aux m >> putStr " |}\n"
  where aux []       = return ()
        aux [x]      = putStr (show x)
        aux (x : xs) = putStr (show x) >> putStr ", " >> aux xs

ned :: Eq a => M a -> M a -> M a
ned m1 m2 = aux m1 m2
  where aux []       l2   = l2
        aux (h : t) l2   = h : aux (filter (/= h) l2) t

bigned :: Eq a => [M a] -> M a
bigned []          = []
bigned (m : ml) = ned m (bigned ml)
```

For the language $\mathcal{L}_{ds}$, the definitions of *rdState* and *inState* are as follows,

$rdState\ (Obj\ sc\ s : pc) \quad = [s]$
$inState\ s\ f\ (Obj\ sc\ s' : pc) = f\ (Obj\ sc\ s : pc)$

whereas for $\mathcal{L}_{obj}$ and $\mathcal{L}_{kc}$ they must also take into account the object identifiers.

$rdState\ (Obj\ o\ sc\ s : pc) \quad = [s]$
$inState\ s\ f\ (Obj\ o\ sc\ s' : pc) = f\ (Obj\ o\ sc\ s : pc)$

B The control flow kernel of logic programming

This appendix pertains to section 5.2. In section B.1 we present the program behavior monads that are appropriate for the Andorra-like languages $\mathcal{L}_{ao}$ and $\mathcal{L}_{aop}$ of section 5.2. Next, in section B.2 we show how to modify the semantic interpreters of $\mathcal{L}_{ao}$ and $\mathcal{L}_{aop}$ to distinguish formally between successful termination and failure. In section B.3 we present the definitions that are lacking from the main text (of section 5.2.2) for the semantic interpreter of $\mathcal{L}_{aop}$. Finally, in appendix B.4 we offer a CSC-based semantic interpreter for a language $\mathcal{L}_{pro}$ capturing the control flow kernel of pure sequential Prolog.

B.1 Program behavior monad

B.1.1 "Single trace" semantics

$$\textbf{data}\ M\ a = InM\{\, unM :: R \to (a, R)\}$$

$$\begin{aligned}&\textbf{instance}\ Monad\ M\ \textbf{where}\\ &\quad return\ a \quad\quad = InM\ (\backslash r \to (a, r))\\ &\quad InM\ m >\!\!>\!= f = InM\ (\backslash r \to \textbf{let}\ (a, r') = m\ r\ \textbf{in}\ unM\ (f\ a)\ r')\end{aligned}$$

$$\begin{aligned}&random :: M\ Int\\ &random = InM\ (Random.next)\end{aligned}$$

$$\begin{aligned}&put :: Obs \to M\ Q \to M\ Q\\ &put\ u\ (InM\ m) \quad = InM\ (\backslash r \to \textbf{let}\ (q, r') = m\ r\ \textbf{in}\ (Observe\ u\ q, r'))\end{aligned}$$

$$\begin{aligned}&display :: M\ Q \to IO\ ()\\ &display\ (InM\ m) = print\ (fst\ (m\ r_0))\end{aligned}$$

$$\begin{aligned}&ned :: M\ a \to M\ a \to M\ a\\ &ned\ m_1\ m_2 \quad\quad = bigned\ [m_1, m_2]\end{aligned}$$

$$\begin{aligned}&bigned :: [M\ a] \to M\ a\\ &bigned\ ml \quad\quad = random >\!\!>\!= \backslash r \to\\ &\qquad\qquad\quad ml\ !!\ (r\ `mod`\ (length\ ml))\end{aligned}$$

B.1.2 "All possible traces" semantics

$$\textbf{data}\ M\ a = InM\{\, unM :: [a]\}$$

$$\begin{aligned}&\textbf{instance}\ Monad\ M\ \textbf{where}\\ &\quad return\ a \quad\quad = InM\ [a]\end{aligned}$$

```
  InM m >>= f = InM (concatMap (\a → unM (f a)) m)

put :: Obs → M Q → M Q
put u (InM m) = InM (map (Observe u) m)

display :: M Q → IO ()
display (InM m) = print m

ned :: Eq a ⇒ M a → M a → M a
ned (InM m1) (InM m2) = InM (aux m1 m2)
  where aux []     l2 = l2
        aux (h : t) l2 = h : aux (filter (/= h) l2) t

bigned :: Eq a ⇒ [M a] → M a
bigned []       = InM []
bigned (m : ml) = ned m (bigned ml)
```

B.2 Success and failure detection

It is easy to modify the semantic interpreters given in Section 5.2.1 to distinguish formally between successful termination and failure. For this purpose we redefine the domain of sequences of observables as follows:

```
data Q = Success | Failure | Observe Obs Q
```

We employ a program behavior monad that can carry a success or a failure condition. The program behavior monad is obtained from the one that we have used in section 3 simply by replacing the domain S (of states) with the type *Bool*. The definition of *display* becomes:

```
display (M m) = print (m b0)
  where b0 :: Bool
        b0 = False
```

The continuation completion mapping *cc* must be adapted to the new definition of Q.[22]

```
cc :: Cont → M Q
cc c = case re c of
         [] → rdState c >>= \b →
                if b then return Success
                     else return Failure
         k → kc k
```

Recall that we implement a logical conjunction (of goals) by using the concept of a multiset (of computations). Therefore, our semantics should signal a successful termination in case the evaluation of at least one of the multisets terminates without a failure; otherwise they should signal a failure. In order to obtain the desired effect we redefine the semantic equation for the (successful) elementary action (the equation for failure remains unchanged).

```
semA (Act act) c =
  put [act] (rdState c >>= \b →
               inState (success c || b) c cc)
```

[22] This is the only place in the paper where we modify the definition of *cc* given in the section 2.2.

The auxiliary operator *success* is only called upon the execution of a successful elementary action. It takes as parameter an open continuation and it returns *True* in case the open (active) multiset remains empty; otherwise it returns *False*. For the semantic interpreter of $\mathcal{L}_{ao}$ this function is defined as follows:

$$\begin{aligned} &success :: Cont \rightarrow Bool \\ &success\ ([\,] : so) = True \\ &success\ so \quad\ = False \end{aligned}$$

For the semantic interpreter of $\mathcal{L}_{aop}$ the definition of *success* is:

$$\begin{aligned} &success :: Cont \rightarrow Bool \\ &success\ (Meet\ [\,]\ ([\,] : so) : po) = True \\ &success\ (Meet\ [\,]\ so : po) \quad\ = False \\ &success\ (Meet\ c\ so : po) \quad\ = success\ c \end{aligned}$$

Let's consider two simple $\mathcal{L}_{ao}$ test programs. In the experiments given below we employed a *Show* instance declaration for Q that uses the strings `"success"` and `"failure"` to visualize the difference between a sequence of observables that terminates with *Success* and a sequence of observables that terminates with *Failure*, respectively.

The first example program executes three goals in parallel.

$$a \parallel \langle C\,?\,c + \mathbf{fail}\,?\,d + E\,?\,e\rangle \parallel b$$

The don't know goal is nondeterminate, therefore its evaluation is delayed according to the Andorra principle. When its evaluation begins, it produces a nondeterminate step and next it tries the bodies of its non-failing alternatives (in sequence). This program can produce two different traces.

```
{| [["a"],["b"],["C","E"],["c"],["e"],"success"],
   [["b"],["a"],["C","E"],["c"],["e"],"success"] |}
```

The don't know goal of the next program fails immediately after the nondeterministic promotion.

$$a \parallel \langle C\,?\,\mathbf{fail} + \mathbf{fail}\,?\,d + E\,?\,\mathbf{fail}\rangle \parallel b$$

Here is the result produced by our semantic interpreter:

```
{| [["a"],["b"],["C","E"],"failure"],
   [["b"],["a"],["C","E"],"failure"] |}
```

B.3 AND-OR parallelism

In section 5.2.2 we have introduced the language $\mathcal{L}_{aop}$ and the CSC continuation structure that is appropriate for modeling its semantics. In this section of the appendix we present the evaluator, the normalization procedure and the scheduler for the semantic interpreter of $\mathcal{L}_{aop}$. $\mathcal{L}_{aop}$ extends $\mathcal{L}_{ao}$ with OR parallelism and don't care nondeterminism, therefore many definitions remain as in section 5.2.1. In the sequel we only present the differences.

The evaluator comprises the definitions of the semantic functions *semA* and *sem*, and the control operators *addc*, *addp* and *fails*. The equations given in section 5.2.1 for *semA* and *sem* remain unchanged. In addition, the definition of *sem* extends with the clauses for OR

parallelism and don't care nondeterminism given below. The control operators *addc* and *addp* simply add a computation to the active (open) multiset. The operator *fails* implements the semantics of failure by voiding the active multiset.

```
sem (Por g) e              c =
  case [(act, sem x e) | (Act act, x) ← g] of
    [] → cc (fails c)
    [(act, d)] → put [act] (cc (addc (Den d) c))
    sg → cc (addc (SemPor sg) c)
sem (Ned h) e              c =
  case [(act, sem x e) | (Act act, x) ← h] of
    [] → cc (fails c)
    sh → bigned [put [act] (cc (addc (Den d) c)) | (act, d) ← sh]

addc :: Comp → Cont → Cont
addc p (Meet [] (pa : so) : po) = Meet [] ((p : pa) : so) : po
addc p (Meet po0 so : po) =       Meet (addc p po0) so : po

addp :: Comp → Cont → Cont
addp p (Meet [] (pa : so) : po) = Meet [] ((p : pa) : so) : po
addp p (Meet po0 so : po) =       Meet (addp p po0) so : po

fails :: Cont → Cont
fails (Meet [] (pa : so) : po) = Meet [] ([] : so) : po
fails (Meet po0 so : po) =       Meet (fails po0) so : po
```

When the active multiset becomes empty it is removed by the normalization procedure.

```
re :: Cont → Kont
re (Meet [] [[]] : po) =       po
re (Meet [] ([] : so) : po) = Meet [] so : po
re (Meet [] so : po)   =       Meet [] so : po
re (Meet c so : po)    =       Meet (re c) so : po
```

The scheduler function *kc*, the domain *Sched* of schedules and the auxiliary function *scheda* remain as in section 5.2.1. The definitions that change are given below. The functions *semSor* and *semPor* implement the nondeterministic promotion operations for backtracking (sequential OR) and OR parallelism, respectively.

```
actc :: Kont → [(Comp, Cont)]
actc k = [w | (ps, po) ← ms k, w ← aux ps po]
  where aux (Meet [] (pa : so)) po = [(p, Meet [] (pa′ : so) : po)
                                       | (p, pa′) ← ms pa]
        aux (Meet po0 so)       po = [(p, Meet c so : po)
                                       | (p, c) ← actc po0]

scheds :: Kont → [Sched]
scheds k = [Scheds [act | (act, _) ← sg]
     (semSor [Den d | (_, d) ← sg] c)
     | (SemSor sg, c) ← actc k]
   ++ [Scheds [act | (act, _) ← sg]
     (semPor [Den d | (_, d) ← sg] c)
```

```
      | (SemPor sg, c) ← actc k]

semSor :: [Comp] → Cont → Kont
semSor lp (Meet [] (pa : so) : po) =
  Meet [] ([p : pa | p ← lp] ++ so) : po
semSor lp (Meet c so : po)     =
  Meet (semSor lp c) so : po

semPor :: [Comp] → Cont → Kont
semPor lp (Meet [] [pa] : po) =
  [Meet [] [(p : pa)] | p ← lp] ++ po
semPor lp (Meet [] (pa : so) : po) =
  (Meet [Meet [] [(p : pa)] | p ← lp] so) : po
semPor lp (Meet c so : po)     =
  Meet (semPor lp c) so : po
```

B.4 Sequential logic programming

This section offers a CSC-based semantic interpreter for a language $\mathcal{L}_{pro}$ capturing the control flow kernel of pure sequential Prolog. The syntax of $\mathcal{L}_{pro}$ is formally defined as follows:

$$x \ ::= \ a \mid x \,;\, x \mid x + x \mid \textbf{letrec}\ y = x\ \textbf{in}\ x \mid \textbf{call}\ y$$

where the class of elementary actions with typical variable a remains as in section 5.2.1. $\mathcal{L}_{pro}$ provides an operator ; for sequential composition, and an operator + for backtracking (don't know nondeterminism). The operator ; is a sequential conjunction (sequential AND), and the operator + is a sequential disjunction (sequential OR). Notice that $\mathcal{L}_{pro}$ does not need a mechanism for detecting the determinacy of a goal; the language lacks the notion of a guarded statement and the n-ary disjunctions of $\mathcal{L}_{ao}$ or $\mathcal{L}_{aop}$ are replaced with the binary disjunction operator +. We implement the abstract syntax of $\mathcal{L}_{pro}$ in Haskell as follows:

```
data X = A A | Call Y | LetRec Y X X | Sor X X | Sand X X
```

where the types *Act*, *A* and *Y* remain as in the section 5.2.1.

In $\mathcal{L}_{pro}$ there is no reason to record the moments when the system performs nondeterministic promotions. Therefore, we define the domain *Obs* of observables as being a synonym of the type *Act* of atomic actions.

```
type Obs = Act
```

The domain of sequences of observables also remain as in the section 5.2 (or 2.1).

In the case of $\mathcal{L}_{pro}$, the program behavior monad is equivalent to the identity monad.

```
data M a = M{unM :: a}

instance Monad M where
  return a      = M a
  M m >>= f = M (unM (f m))
```

The operations *put* and *display* become:

```
put :: Obs → M Q → M Q
put u (M m)   = M (Observe u m)
```

type $SA = [Comp]$
type $SO = [SA]$

type $Kont = SO$
type $Cont = Kont$

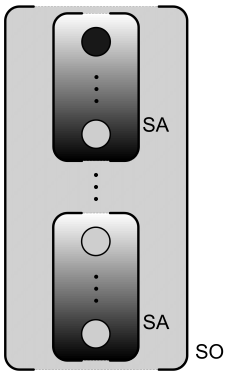


Figure 13: Structure of continuations for $\mathcal{L}_{pro}$: a stack of stacks of computations.

$display :: M\ Q \rightarrow IO\ ()$
$display\ (M\ m) = print\ m$

The domain of computations is again as in section 2.1.

data $Comp = Den\ D$

To model the semantics of $\mathcal{L}_{pro}$, continuations can be structured as stacks of stacks of computations. This continuation structure is depicted in figure 13. Each (inner) SA stack models a sequential conjunction (a sequential AND). The (outer) SO stack is needed to model the backtracking mechanism (a sequential OR). The initial continuation $c_0 :: Cont$ is defined as follows: $c_0 = [[\,]]$.

The semantics of an elementary action and the semantics of recursion remain as in section 5.2. The semantics of the sequential OR and the sequential AND connectives are directly based on the control operators *addo* and *adda*. *addo* implements the semantics of don't know nondeterminism in $\mathcal{L}_{pro}$. *fails* implements the semantics of failure by voiding the active stack (from the top of the backtracking stack).

$sem :: X \rightarrow Env \rightarrow D$
$sem\ (Sor\ x_1\ x_2)\quad e\ c = sem\ x_1\ e\ (addo\ (Den\ (sem\ x_2\ e))\ c)$
$sem\ (Sand\ x_1\ x_2)\ e\ c = sem\ x_1\ e\ (adda\ (Den\ (sem\ x_2\ e))\ c)$

$addo :: Comp \rightarrow Cont \rightarrow Cont$
$addo\ p\ (sa : so) = sa : (p : sa) : so$

$adda :: Comp \rightarrow Cont \rightarrow Cont$
$adda\ p\ (sa : so) = (p : sa) : so$

$fails :: Cont \rightarrow Cont$
$fails\ (sa : so) = [\,] : so$

Notice that in section 5.2 - where we have studied the semantics of the the Andorra-like languages $\mathcal{L}_{ao}$ and $\mathcal{L}_{aop}$- each nondeterministic promotion operation was implemented as a function of a scheduler. The scheduler of an Andorra-like language must delay the evaluation of a nondeterminate goal as much as possible. By contrast, in $\mathcal{L}_{pro}$ there is no priority mechanism and any don't know goal can immediately be evaluated. Therefore it is more convenient to implement the operator *addo* that models the semantics of don't know nondeterminism as a function of the evaluator (rather than the scheduler).

The normalization procedure simply removes the active stack when it becomes empty.

$re :: Cont \to Kont$
$re\ ([\,] : so) = so$
$re\ so = so$

The language $\mathcal{L}_{pro}$ is sequential and deterministic. The scheduler function kc produces exactly one decomposition of a closed continuation into an active computation and a corresponding open continuation, and then it executes the former with the latter as a continuation.

$kc :: Kont \to M\ Q$
$kc\ ((Den\ d : sa) : so) = d\ (sa : so)$

We only consider one $\mathcal{L}_{pro}$ test program.

$a;\ b;\ (c + \mathbf{fail} + d);\ e$

Our semantic interpreter produces the following result:

$[\texttt{"a"}, \texttt{"b"}, \texttt{"c"}, \texttt{"e"}, \texttt{"d"}, \texttt{"e"}]$